# Artificial Intelligence (CS30002)

## Module 3: Problem Solving by Searching

Dr. Jaydeep Das

# Contents

- Problem Solving Agents
- Example Problems
- Searching for Solutions

# Problem Solving Agents

# Problem Solving Agent Components

- A problem can be defined or formulated formally by 4 components

1. **Initial State:** Describes possible initial state from where problem solving agent starts in

2. **Set of actions and Successor Function:** Given a particular state x, Successor Function (SF(X)) returns a set of <action, successor> ordered pairs, where each action is one of the legal actions in state x and each successor is a state that can be reached from x by applying the action

- Together, the initial state and successor function implicitly define the state space of the problem, i.e., the set of all states reachable from the initial state

**Initial State + SF => State Space**

# Problem Solving Agent Components

• State space forms a graph in which the nodes are states and the arcs between nodes are actions

**3. Goal Test:** It determines whether a given state is a goal state

**4. Path Cost:** A path in the state space is a sequence of states connected by a sequence of actions A path cost function assigns a numeric cost to each path The problem solving agent chooses a cost function that reflects its own performance measure

$$\text{Path Cost} = \sum \text{Step Cost}$$

Path Cost is sum of the costs of the individual actions along the path

(Step cost of taking action a to go from state x to state y is denoted by $c(x, a, y)$

# Problem Solving Agent Components

- These four components that define a problem can be gathered together into a single data structure that is given as input to a problem solving algorithm
- A solution to a problem is a path from the initial state to a goal state
- Solution quality is measured by the path cost function and an optimal solution has the lowest path cost among all solutions

# Problem Solving Agent Function

```
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 do
        goal ← FORMULATE-GOAL(state)
        problem ← FORMULATE-PROBLEM(state, goal)
        seq ← SEARCH(problem)
    action ← FIRST(seq)
    seq ← REST(seq)
    return action
```

# Problem Solving Agent Example

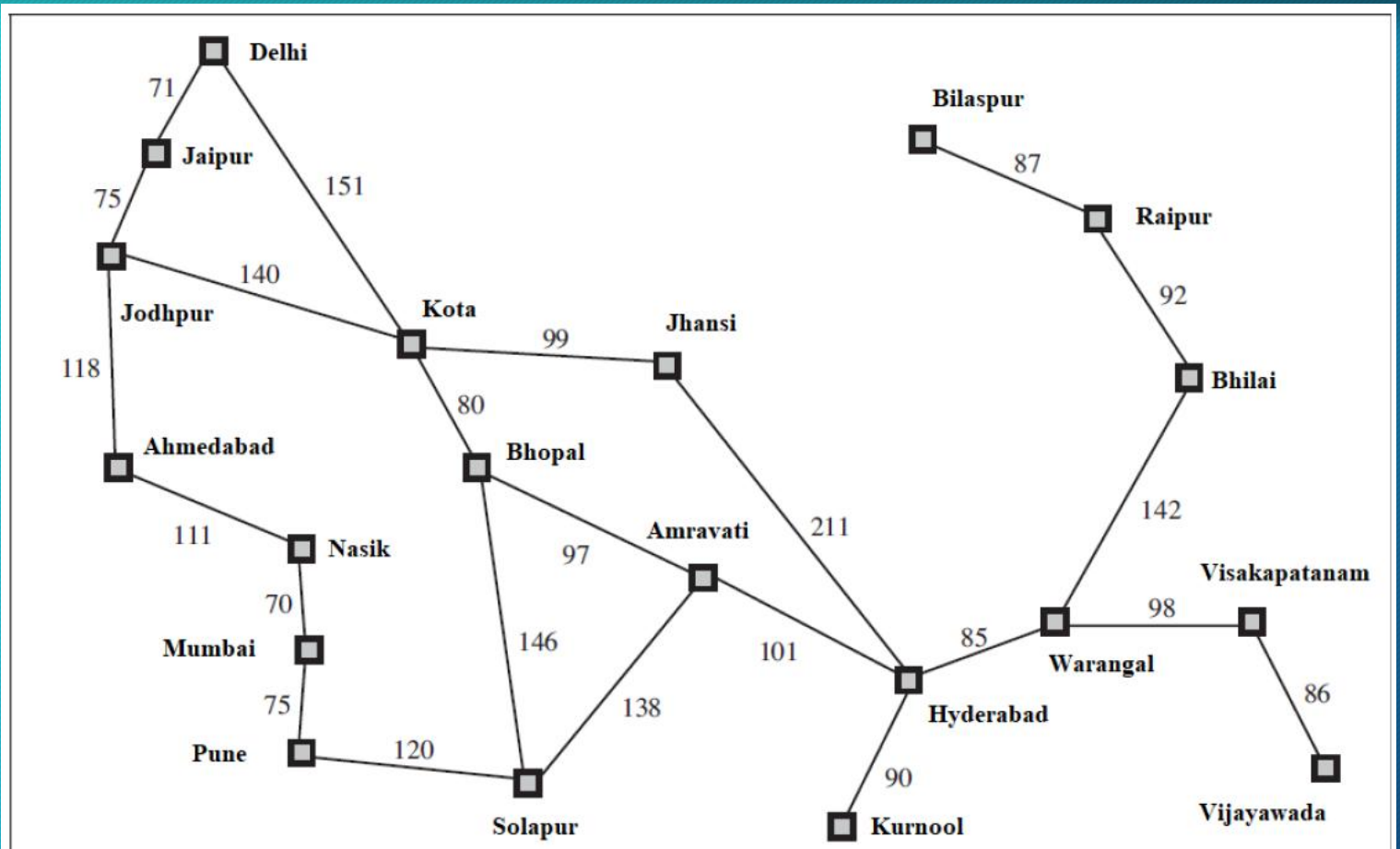



**Figure:** A simplified road map of part of India

# Problem Solving Agent Example

- On holiday in Central India.
- Currently Location: **Jodhpur**
- Formulate Goal:

  To be in **Hyderabad**

- Formulate Problem:
  - States: various cities
  - Actions: drive between cities

- Find solution:
  - sequence of cities, e.g., Jodhpur -> Kota -> Jhansi -> Hyderabad

# Selecting A State Space

- Real world is absurdly complex

  → state space must be abstracted for problem solving

- (Abstract) state = set of real states
- (Abstract) action = complex combination of real actions
  - Example: “Jodhpur →Jaipur" represents a complex set of possible routes, detours, rest stops, etc.
- For guaranteed realizability, any real state "in Jodhpur“ must get to some real state "in Jaipur"
- (Abstract) solution = set of real paths that are solutions in the real world
- Each abstract action should be "easier" than the original problem

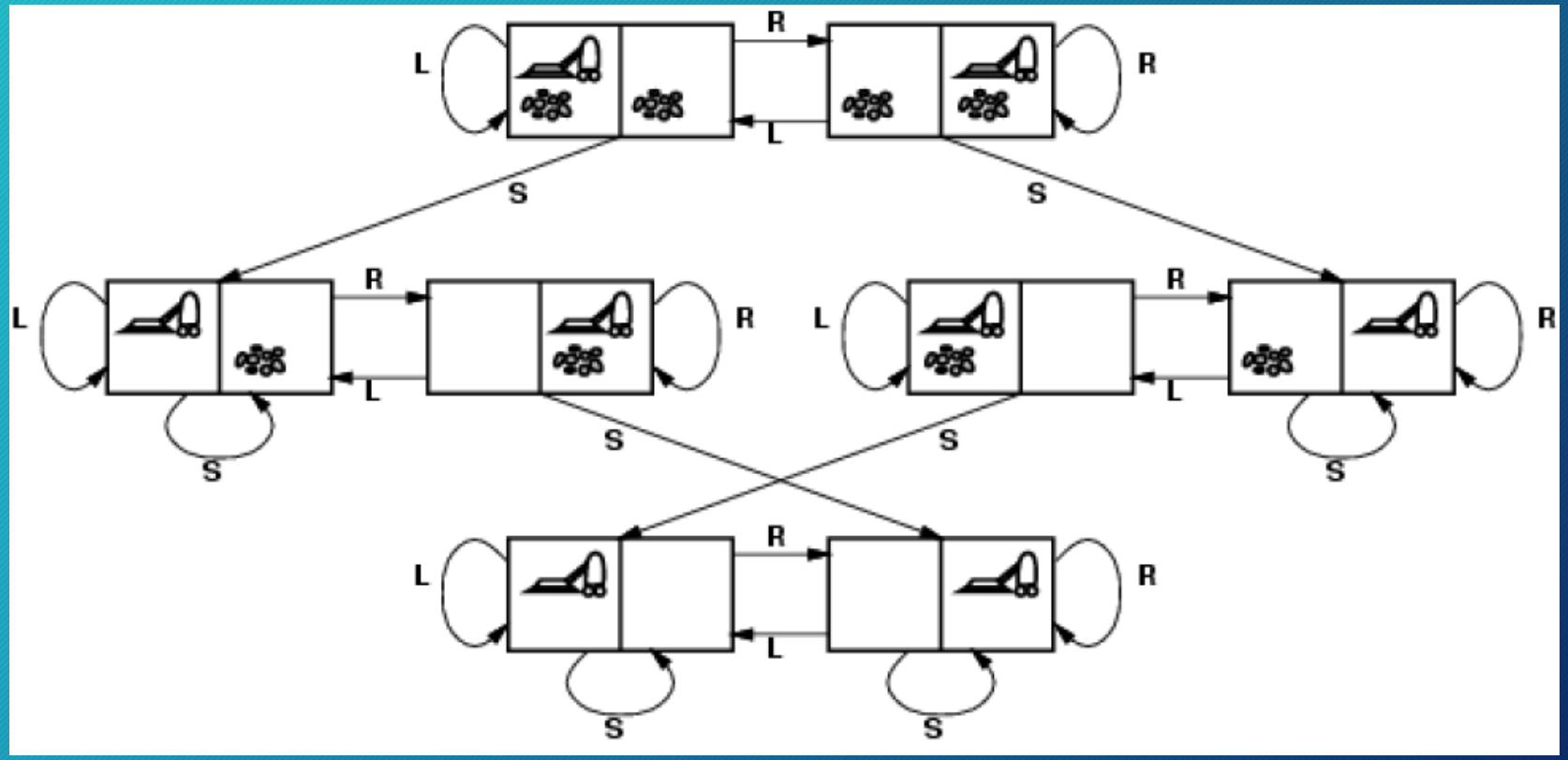
# Example Problems

# Problem Formulation for Toy Problems

- Simple Vacuum World Problem
- 8-Puzzle Problem (or Sliding Block Problem)
- 8-Queen Problem
- Knuth Number Generation Problem
- Airline Travel Problem

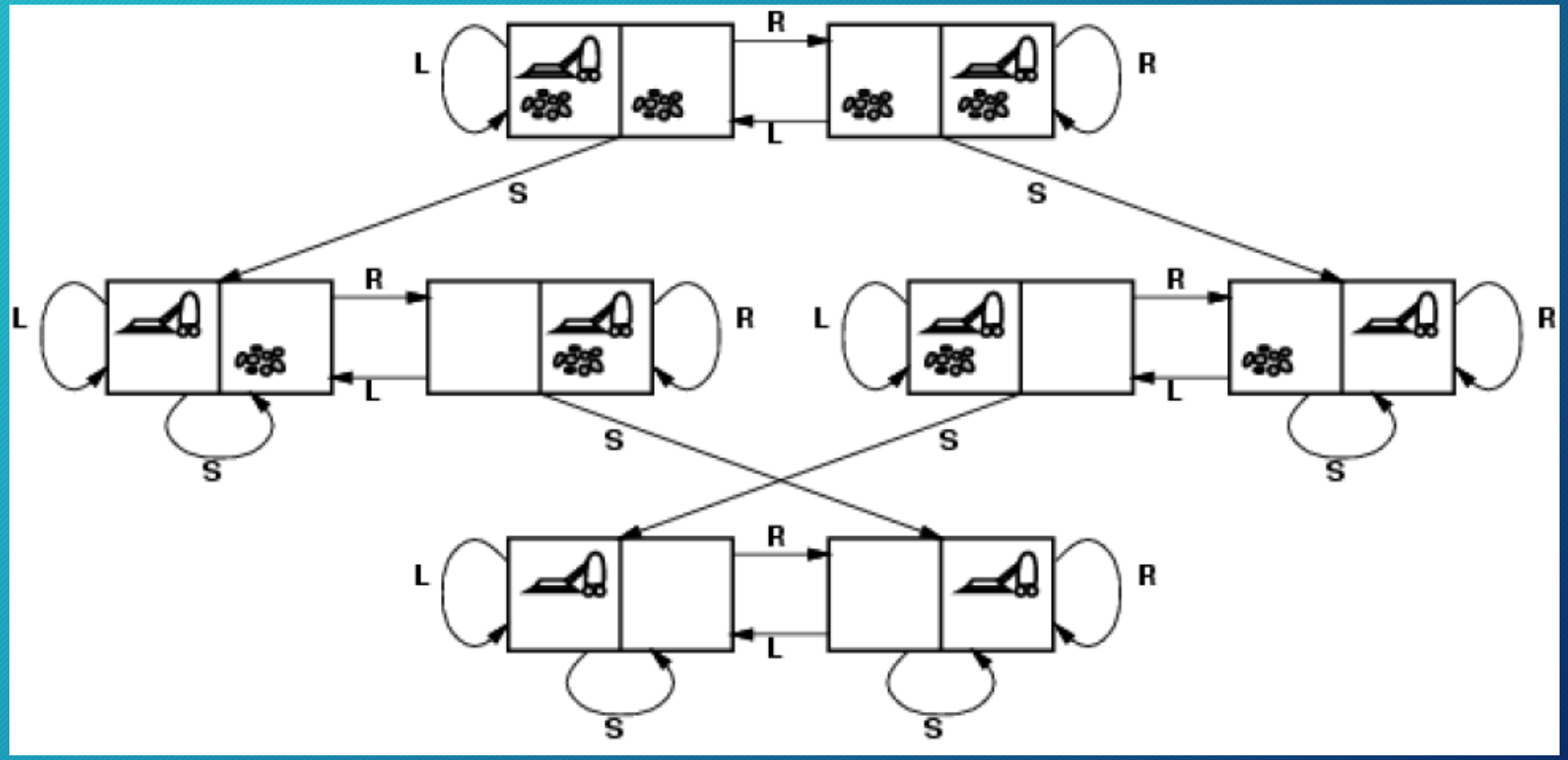
# Simple Vacuum World Problem (State Space)

- States?
- Actions?
- Goal test?
- Path cost?



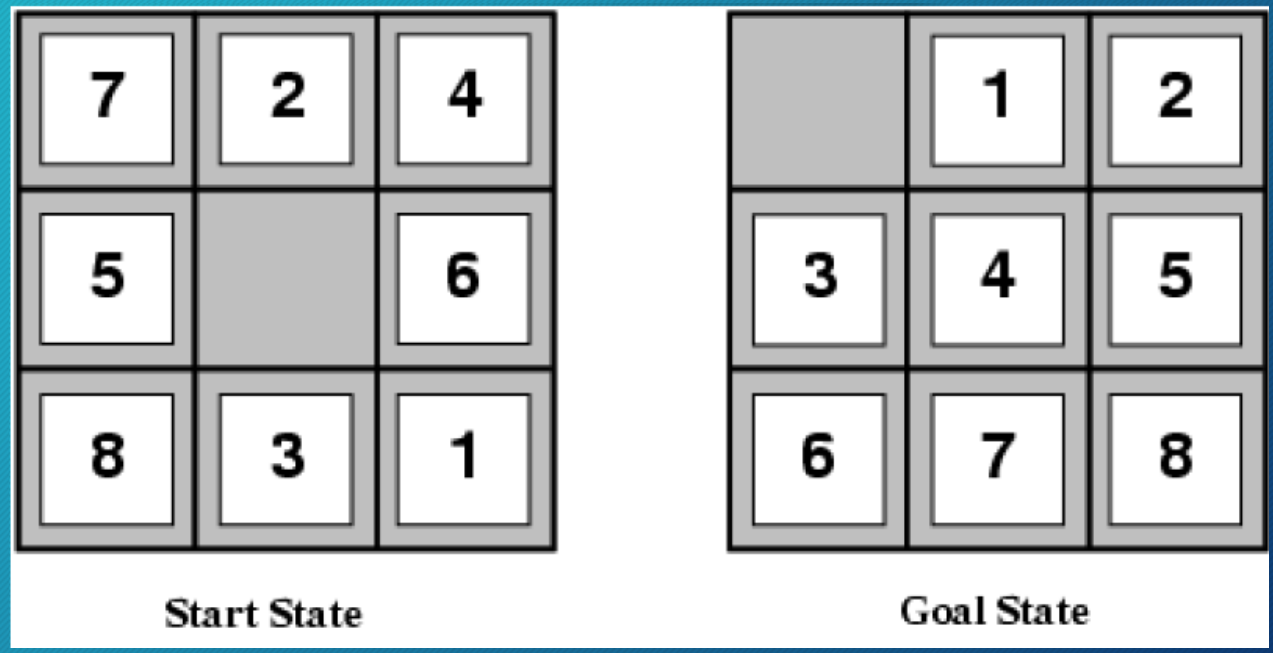
# Simple Vacuum World Problem (State Space)

- States: Dirt and Location
- Actions: *Left*, *Right*, *Suck*
- Goal test: No dirt at all locations
- Path cost: 1 per action



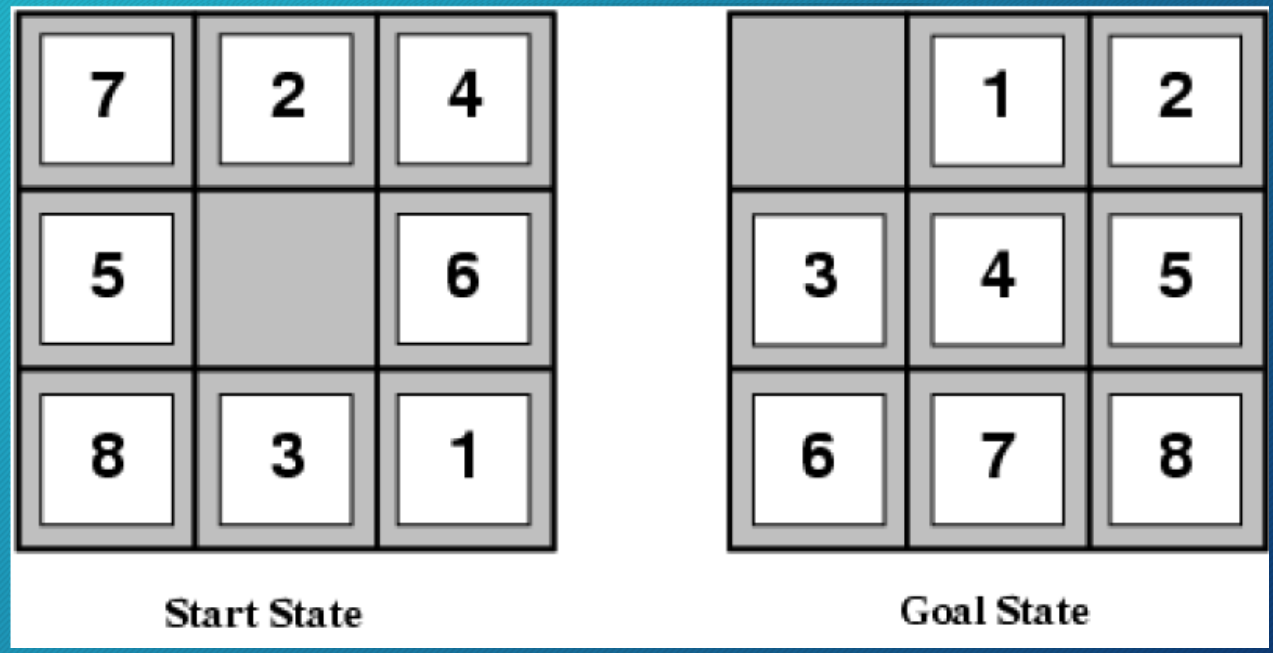
# 8-Puzzle Problem

- States?
- Actions?
- Goal test?
- Path cost?



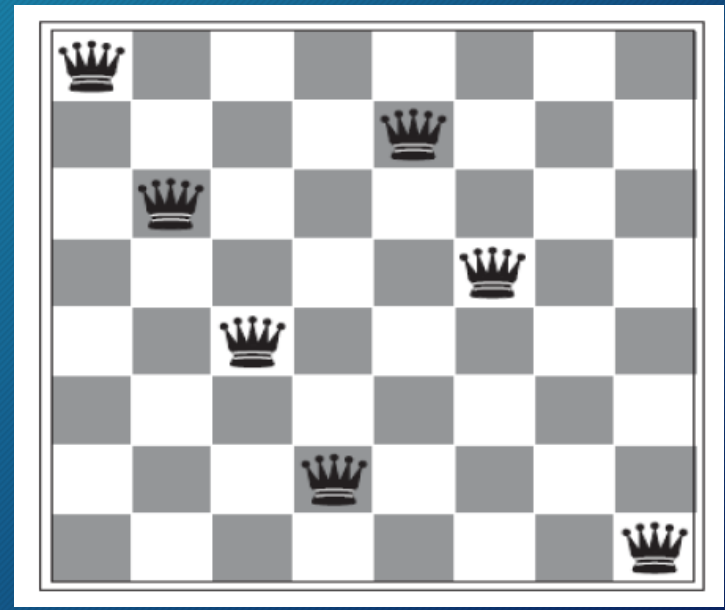
# 8-Puzzle Problem

- States: Location of Tiles
- Actions: Blank moves Left, Right, Up, Down
- Goal test: Goal state (Given)
- Path cost: 1 per move



# 8-Queens Problem

- There are two main kinds of formulation.
- An **incremental formulation** involves operators that *augment* the state description, starting with an empty state; for the 8-queens problem, this means that each action adds a queen to the state.
- A **complete-state formulation** starts with all 8 queens on the board and moves them around. In either case, the path cost is of no interest because only the final state counts. The first incremental formulation one might try is the following:



- **States**: Any arrangements of 0 to 8 queens on the board is a state
- **Initial State**: No queens on the board
- **Actions**: Add a queen to any empty square
- **Transition Model**: Returns the board with a queen added to the specified square
- **Goal Test**: 8 queens are on the board, none attacked

# 8-Queens Problem

In this formulation, we have $64 \cdot 63 \cdots 57 \approx 1.8 \times 10^{14}$ possible sequences to investigate. A better formulation would prohibit placing a queen in any square that is already attacked:

- **States**: All possible arrangements of $n$ queens ($0 \leq n \leq 8$), one per column in the leftmost $n$ columns, with no queen attacking another.
- **Actions**: Add a queen to any square in the leftmost empty column such that it is not attacked by any other queen.

This formulation reduces the 8-queens state space from $1.8 \times 10^{14}$ to just 2,057, and solutions are easy to find. On the other hand, for 100 queens the reduction is from roughly $10^{400}$ states to about $10^{52}$ states (Exercise 3.5)—a big improvement, but not enough to make the problem tractable. Section 4.1 describes the complete-state formulation, and Chapter 6 gives a simple algorithm that solves even the million-queens problem with ease.

# Knuth’ s Number Generation

Our final toy problem was devised by Donald Knuth (1964) and illustrates how infinite state spaces can arise. Knuth conjectured that, starting with the number 4, a sequence of factorial, square root, and floor operations will reach any desired positive integer. For example, we can reach 5 from 4 as follows:

$$\left\lfloor \sqrt{\sqrt{\sqrt{\sqrt{\sqrt{(4!)!}}}}} \right\rfloor = 5 \ .$$

The problem definition is very simple:

- **States**: Positive numbers.
- **Initial state**: 4.
- **Actions**: Apply factorial, square root, or floor operation (factorial for integers only).
- **Transition model**: As given by the mathematical definitions of the operations.
- **Goal test**: State is the desired positive integer.

# Problem formulation for Real world Problems

- The route finding problem is defined in terms of specified locations and transitions along links between them

- Route finding algorithms are used in a variety of applications.

- Some, such as Web sites and in car systems that provide driving directions, are relatively straightforward extensions of the Indian Cities example.

- Others, such as routing video streams in computer networks, military operations planning, and airline travel planning systems, involve much more complex specifications

# Airline Travel Problem

- Consider the airline travel problem that must be solved by a travel planning Web site:

- **States**: Each state obviously includes a location (e.g., an airport) and the current time. Furthermore, because the cost of an action (a flight segment) may depend on previous segments, their fare bases, and their status as domestic or international, the state must record extra information about these “historical” aspects

- **Initial State**: This is specified by the user’s query

- **Actions**: Take any flight from the current location, in any seat class, leaving after the current time, leaving enough time for within airport transfer if needed.

# Airline Travel Problem

- **Transition Model:** The state resulting from taking a flight will have the flight's destination as the current location and the flight's arrival time as the current time
- **Goal Test:** Are we at the final destination specified by the user?
- **Path Cost:** This depends on monetary cost, waiting time, flight time, customs and immigration procedures, seat quality, time of day, type of airplane, frequent flyer mileage awards, and so on.

# Some Real World Problems

- Touring Problems
- Traveling Salesperson Problem(TSP)
- VLSI Layout Problem
- Robot Navigation
- Automatic Assembly Sequencing
- Protein Design

# Searching for Solutions

# Basic AI Problem Solving Techniques

- Problem solving by Searching
- Problem solving by Reasoning Inference
- Problem solving by Matching

# Searching for Solutions

- State Space (Initial State + Successor Function)
- Here search techniques use an explicit search tree that is generated by initial state and successor function that together define the state space.
- **Example:** Search Tree for finding a route from Jodhpur to Hyderabad.
- The root of the search tree is a **search node** corresponding to the initial state *In(Jodhpur)*.

# Informal Tree Search Algorithm

- **Basic Idea:**
  - offline, simulated exploration of state space by generating successors of already-explored states (expanding states)

```
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
```

# Search Tree Example

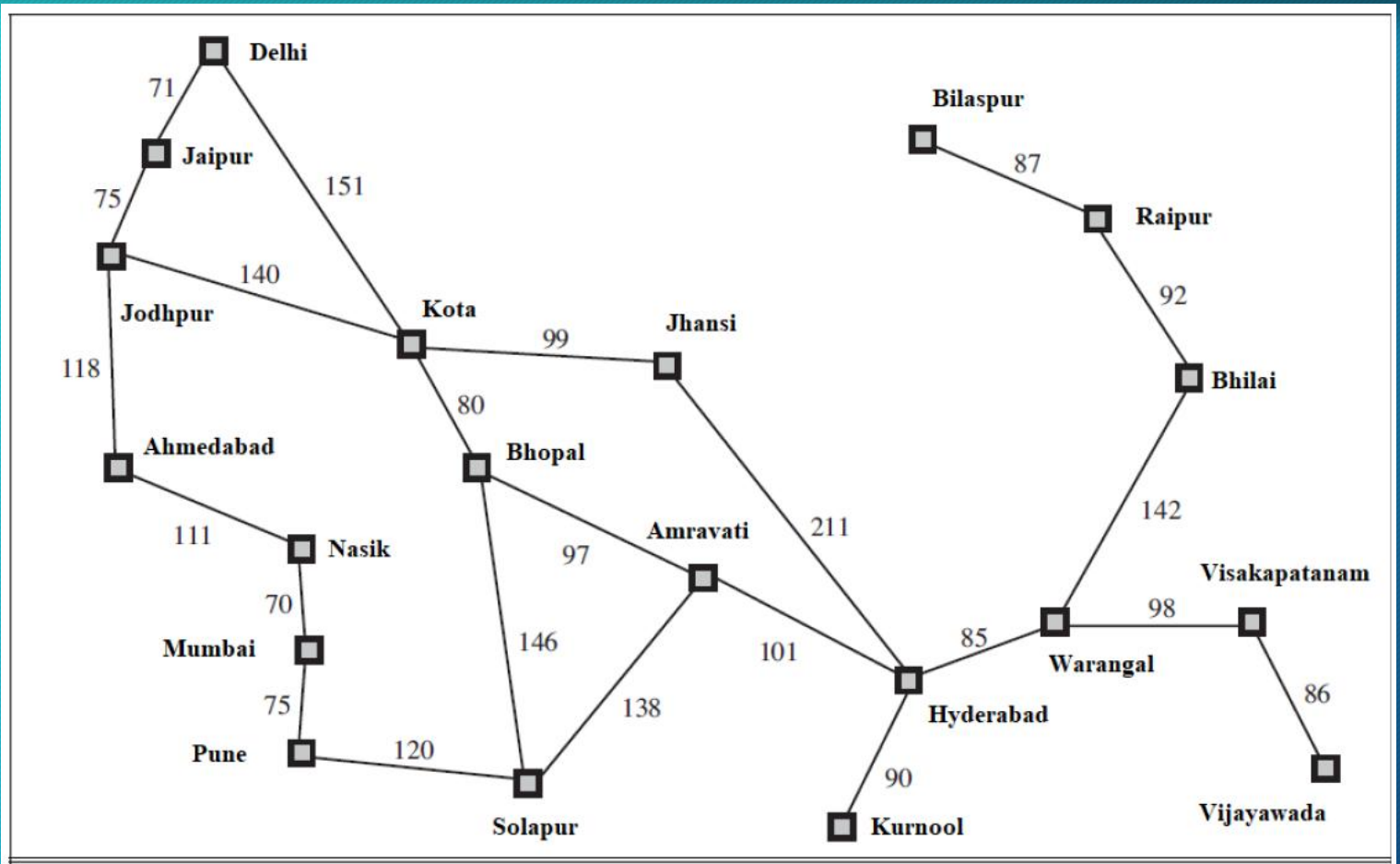
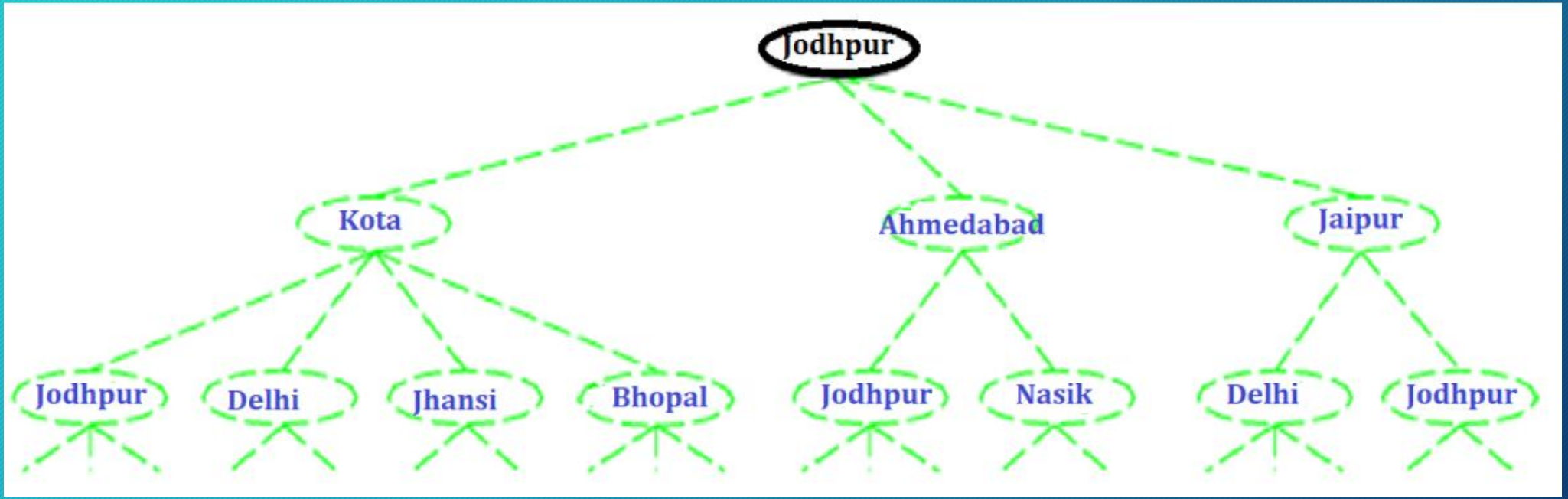



**Figure:** A simplified road map of part of India

# Search Tree Example (Contd…)

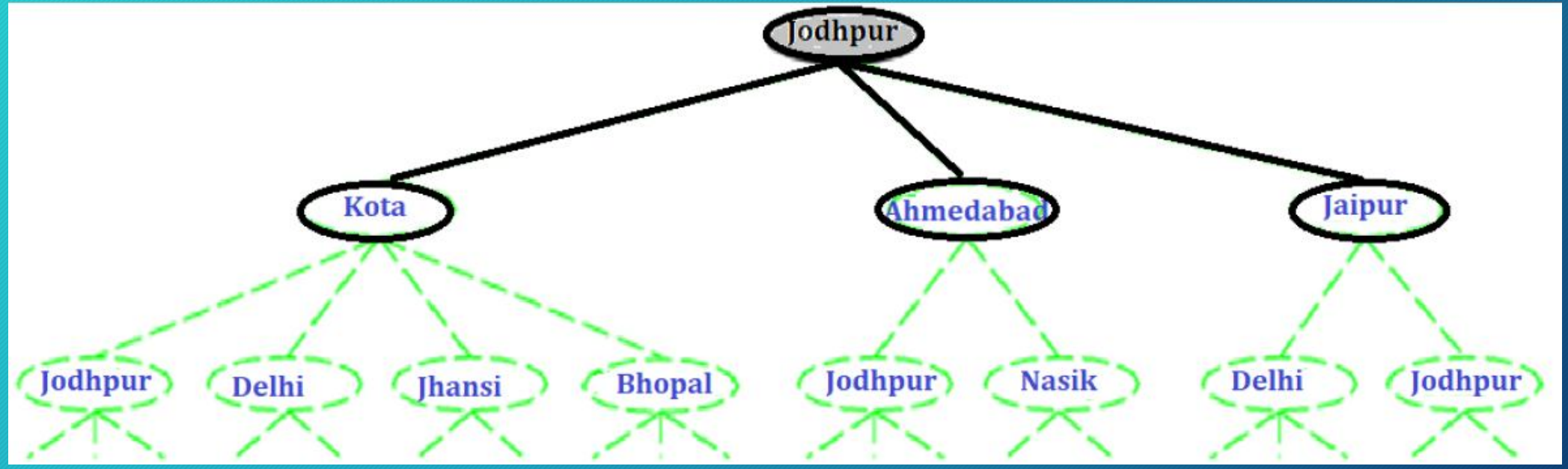
- **Initial Stage**




- Partial search trees for finding a route from *Jodhpur* to *Hyderabad*. 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.

# Search Tree Example (Contd…)

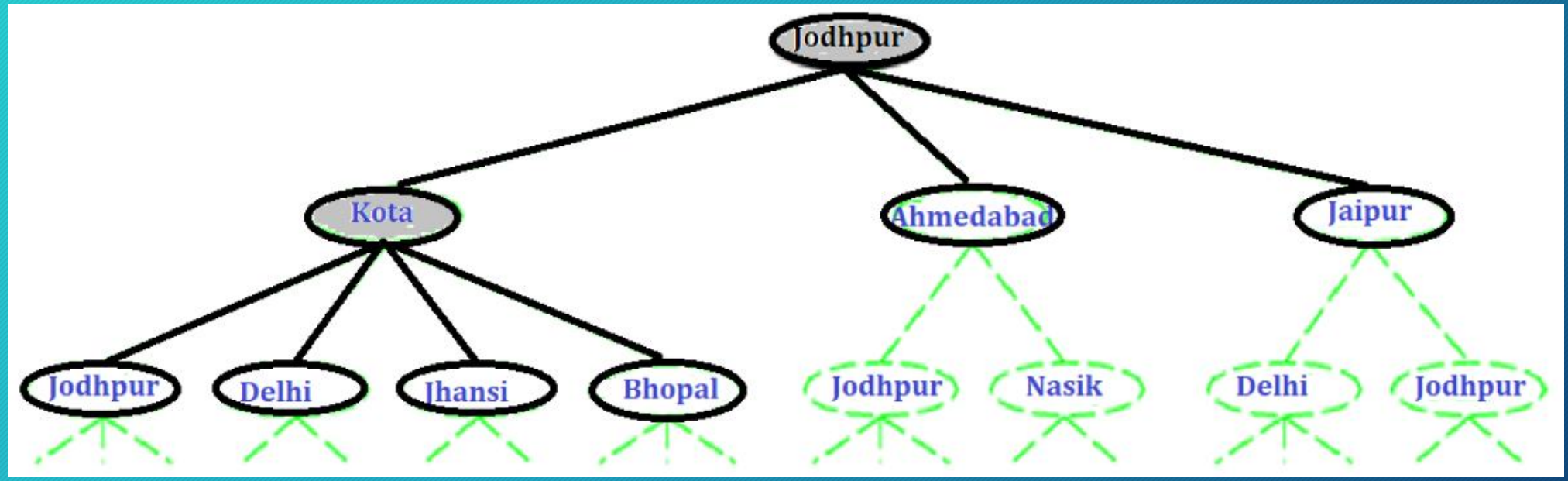
- **After Expanding Jodhpur**




- Partial search trees for finding a route from *Jodhpur* to *Hyderabad*. 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.

# Search Tree Example (Contd…)

- **After Expanding Kota**




- Partial search trees for finding a route from *Jodhpur* to *Hyderabad*. 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.

# Informal Graph Search Algorithm

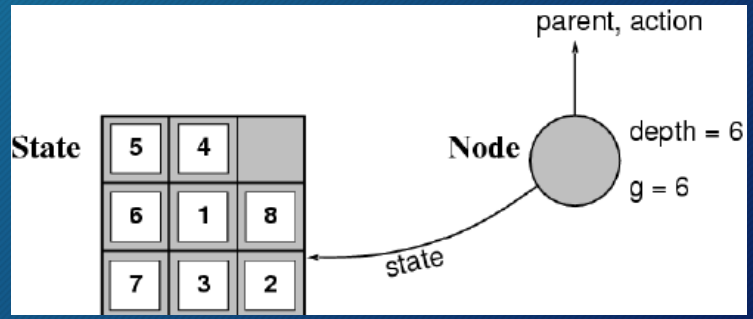
**function** GRAPH-SEARCH(*problem*) **returns** a solution, or failure
  initialize the frontier using the initial state of *problem*
  ***initialize the explored set to be empty***
  **loop do**
    **if** the frontier is empty **then return** failure
    choose a leaf node and remove it from the frontier
    **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***

- The set of all leaf nodes available for expansion at any given FRONTIER point is called the frontier. (Many authors call it the *open list*)

# Node Data Structure

**Node is a data structure with five components**

- **STATE:** The state in the space to which the node corresponds A state is a (representation of) a physical configuration
- **PARENT-NODE:** The node in the search tree that generated a node
- **ACTION:** The action that was applied to the parent to generate the node
- **PATH-COST:** The cost, traditionally denoted by g(n) of the path from the initial state to the node, as indicated by the parent pointers
- **DEPTH:** The number of steps along the path from the initial state



# Node Data Structure (Contd…)

The Expand function creates new nodes, filling in the various fields and using the Successor-Fn of the problem to create the corresponding states

```
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
```

# Implementation of Nodes in Search Algorithm

- A node is a bookkeeping data structure used to represent the search tree A state corresponds to a configuration of the world. Thus, nodes are on particular paths, as defined by PARENT pointers, whereas states are not
- Two different nodes can contain the same world state if that state is generated via two different search paths
- The frontier needs to be stored in such a way that the search algorithm can easily choose the next node to expand according to its preferred strategy. The appropriate data structure for this is a queue

# Implementation of Nodes in Search Algorithm (Contd…)

**The operations on a queue are as follows:**

- **EMPTY(queue)** returns true only if there are no more elements in the queue.
- **POP(queue)** removes the first element of the queue and returns it.
- **INSERT(element, queue)** inserts an element and returns the resulting queue.
- Queues are characterized by the *order* in which they store the inserted nodes.

- Three common variants are:
- **FIFO queue:** which pops the *oldest* element of the queue
- **LIFO queue:** (also known as a **stack**), which pops the *newest* element of the queue
- **Priority queue:** pop the element of the queue with highest priority according to some ordering function

# Evaluating Search Strategies

- Performance of search 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 complexities are measured in terms of
  - ✓ ***b*:branching factor** or maximum number of successors of any node
  - ✓ ***d*: *depth*** of the shallowest goal node (i.e., the number of steps along the path from the root)
  - ✓ ***m*: maximum length** of any path in the state space (may be ∞)

# Evaluating Search Strategies (Contd…)

- **Time** is often measured in terms of the number of nodes generated during the search, and **Space** in terms of the maximum number of nodes stored in memory.
- For the most part, we describe time and space complexities for search on a tree; for a graph, the answer depends on how "redundant" the paths in the state space are.
- To assess the effectiveness of a search algorithm, consider the search cost which typically depends on the time complexity but can also include a term for memory usage.
- Sometimes use the total cost, which combines the search cost and the path cost of the solution found (also sometimes called solution cost).

# Contents

- Search Strategies
- Uninformed Search Techniques
  - Breadth First Search
  - Uniform Cost Search
  - Depth First Search
  - Depth Limited Search
  - Iterative Deepening Search
  - Bi-Directional Search
- Repeated State

# Search Strategies

They are of 2 types:

- Uninformed search / Blind search

  Blind search has no additional information about the states beyond that provided in the problem definition. All they can do is *generate successor and distinguish a goal state from a non-goal state*.

- Informed search / Heuristic search

  Informed search identifies whether one non-goal state is “more promising” than another one.

***All search strategies are distinguished by the order in which nodes are expanded.***

# Uninformed Search Strategies

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

Popular ones are:

- Breadth-first search (BFS)
- Uniform-cost search (UCS)
- Depth-first search (DFS)
- Depth-limited search (DLS)
- Iterative deepening search (IDS)
- Bidirectional search (BDS)

# Breadth First Search (BFS)

- It is a strategy in which the root node is expanded first.
- Then all the successors of the root node are expanded, then their successors.
- At a given depth in the search tree, all the nodes are expanded before any node at the next level is expanded.
- A FIFO queue is used, i.e., new successors join the tail of the queue.

# BFS Function

```
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)
```

# BFS Algorithm

- Create a variable NODE-LIST (FIFO queue) and set it to initial state.

- Until a goal state is found or the NODE-LIST is empty,

  Do

  A. If NODE-LIST is empty then quit.
  else remove the first element from NODE-LIST and call it V(visited)

  B. For each rule ( from the rule set ) whose L.H.S matches with the state described in V

  Do

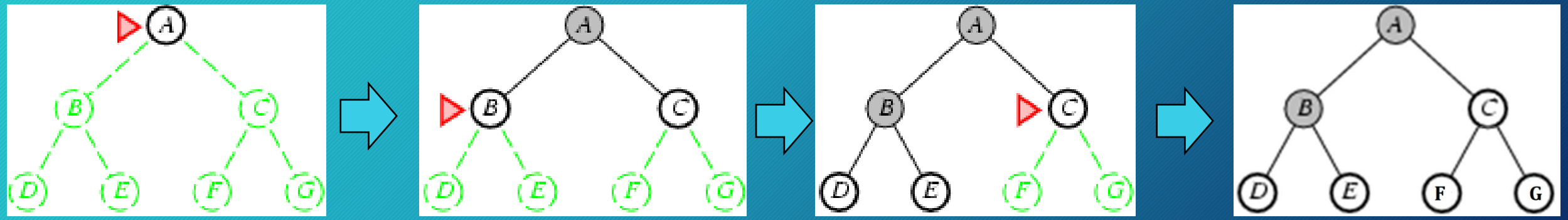
  I. Apply the rule to generate a new state.
  II. If the new state is a goal state then quit and return the state.
  III. Otherwise add the new state to the end of the NODE-LIST.

# Properties of BFS

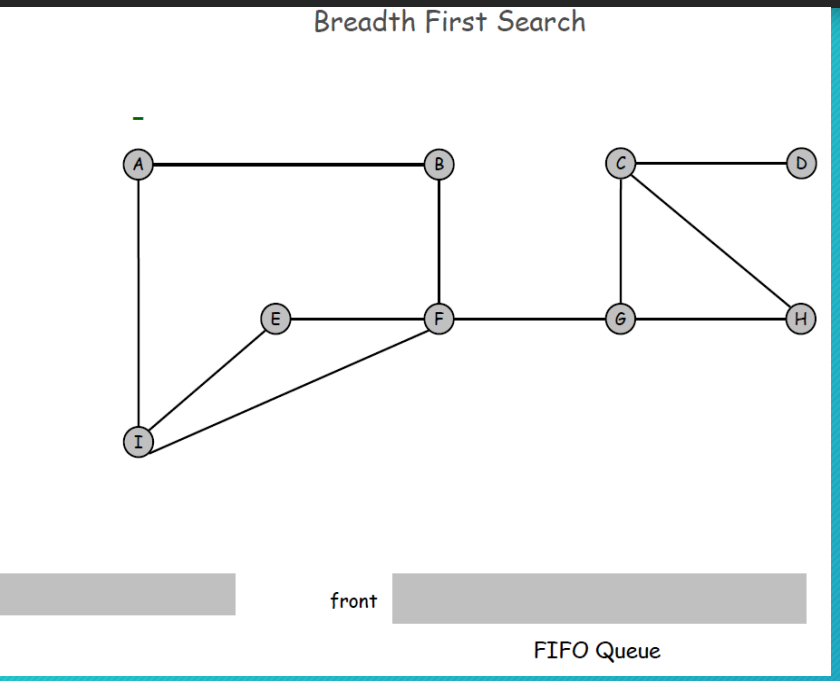
- Complete? **Yes** (if *b* is finite)
- Time? $1+b+b^2+b^3+\ldots+b^d$ = $\mathbf{O(b^d)}$
- Space? $\boldsymbol{O(b^d)}$ (keeps every node in memory)
- Optimal? **Yes** ( if path cost is a non-decreasing function of depth, i.e., path cost = 1 per step)
- Space is the bigger problem (more than time)

# BFS on Tree

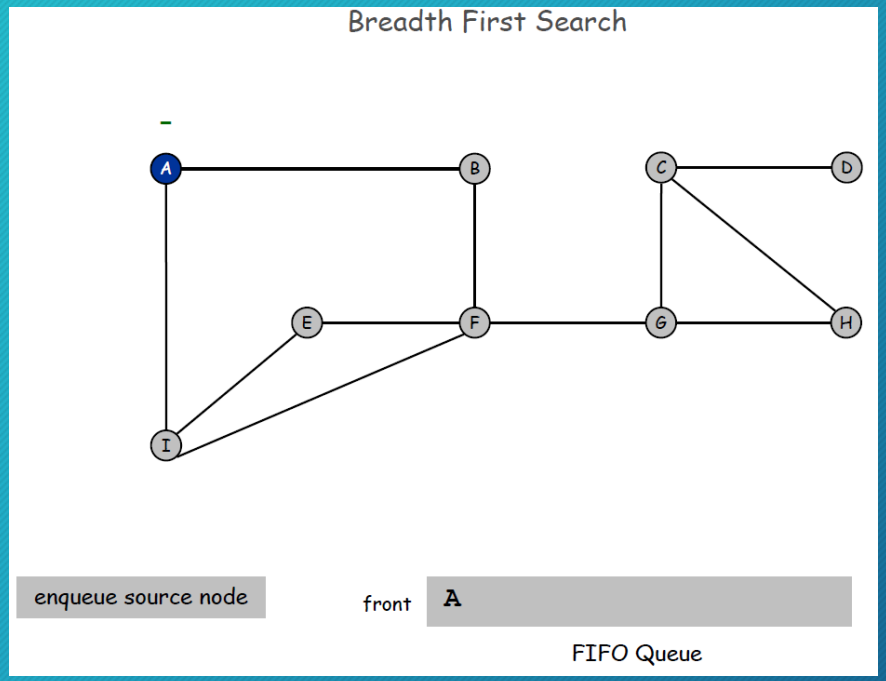
- Expand shallowest unexpanded node
- Implementation:
  - *fringe* is a FIFO queue, i.e., new successors go at end



# Example: BFS on Graph

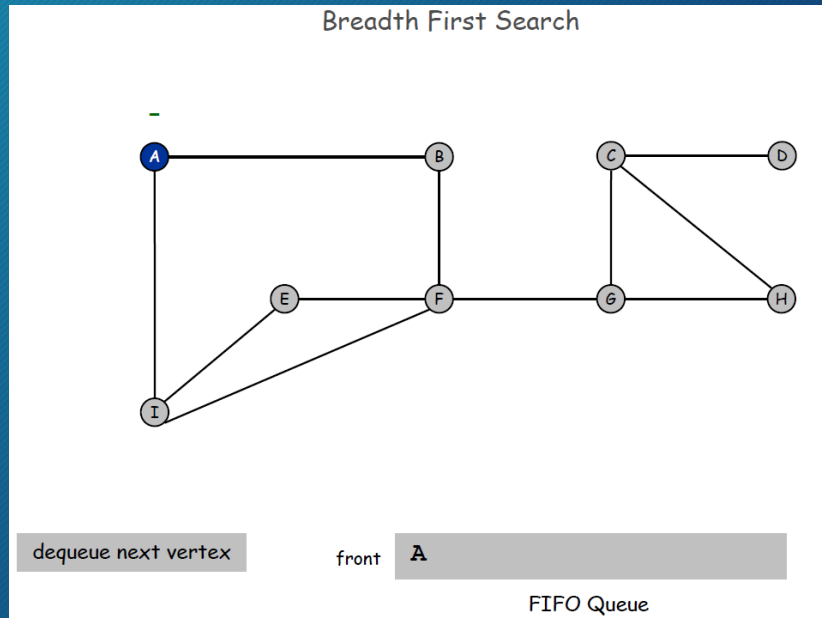



Step 1

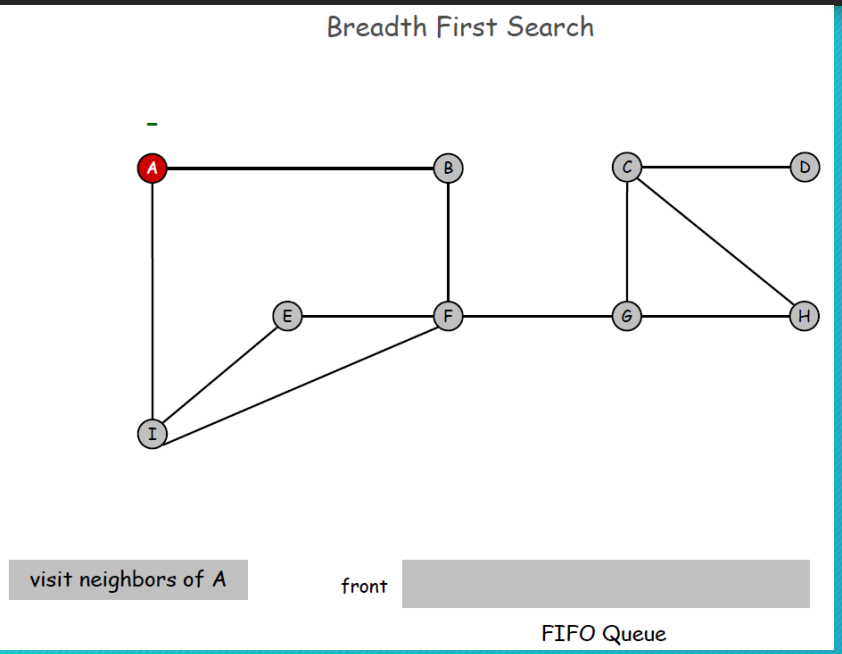



Step 2

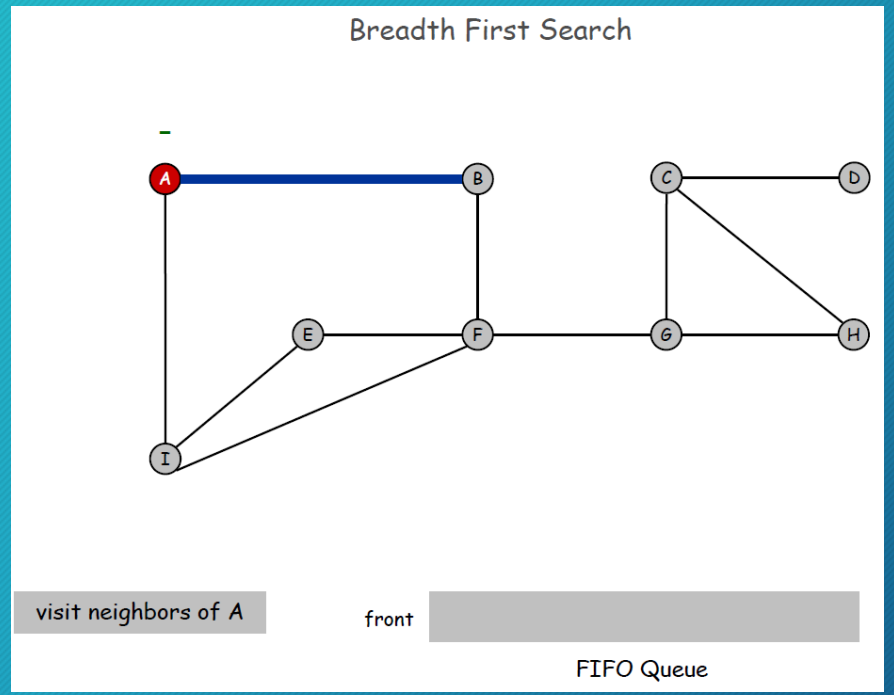
Step 3



# Example: BFS on Graph (contd…)

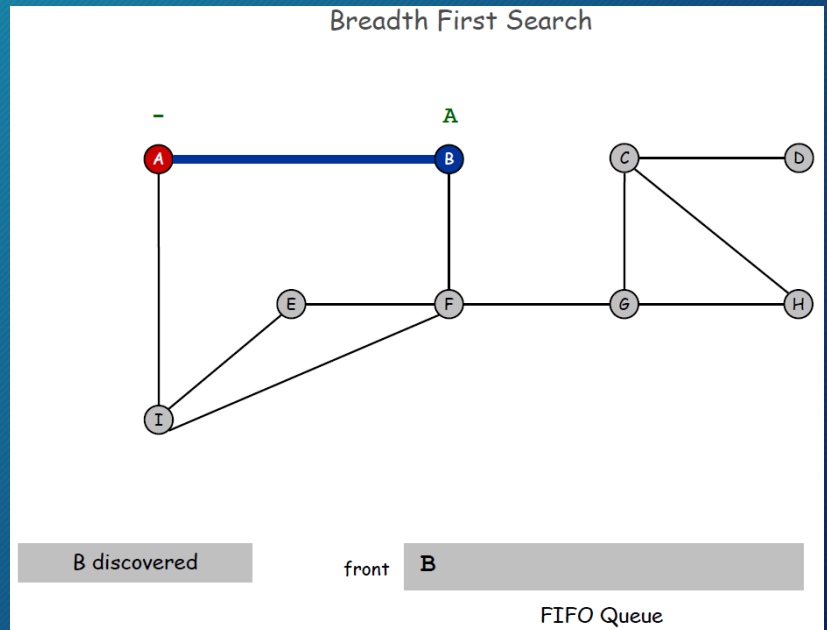



Step 4

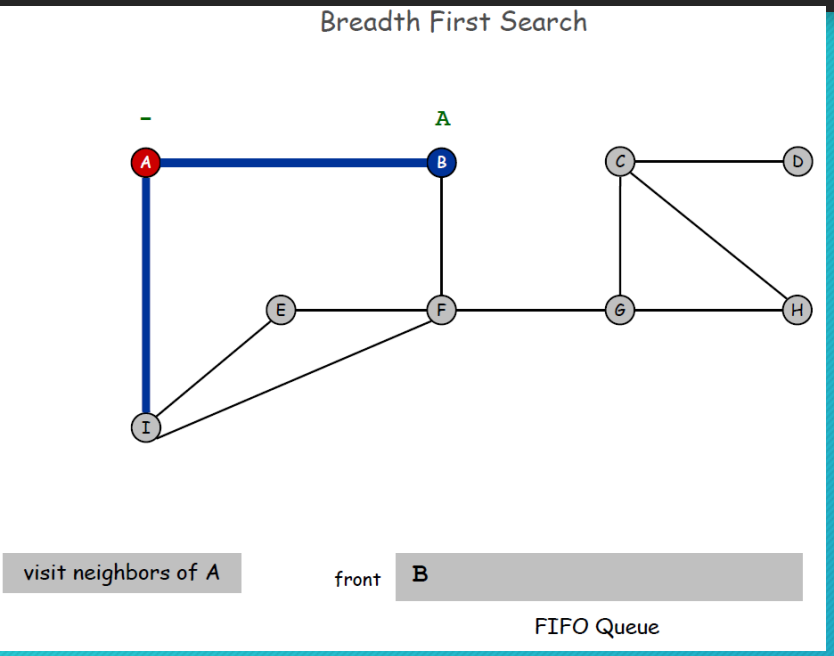



Step 5

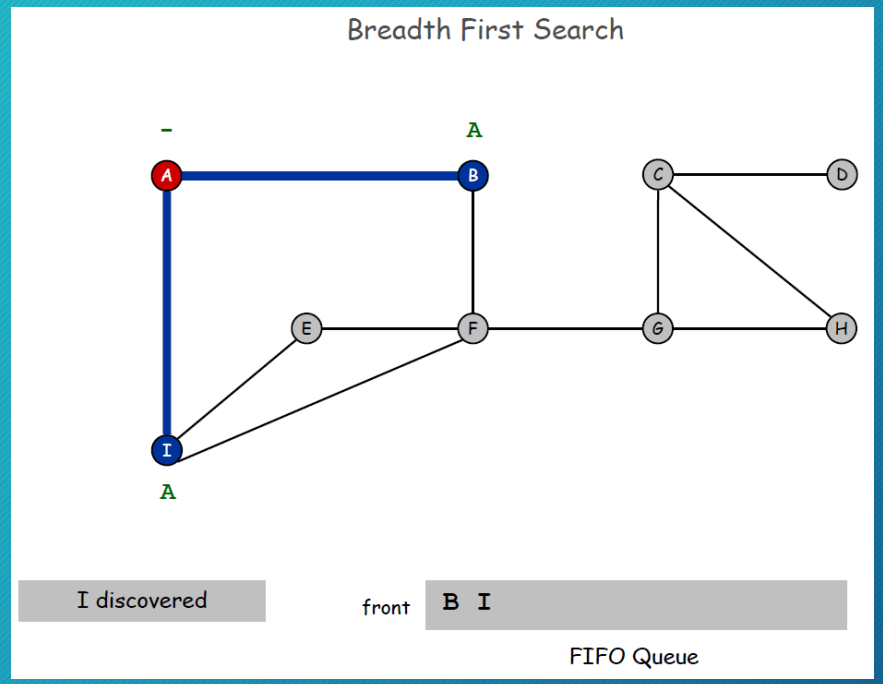
Step 6



# Example: BFS on Graph (contd…)

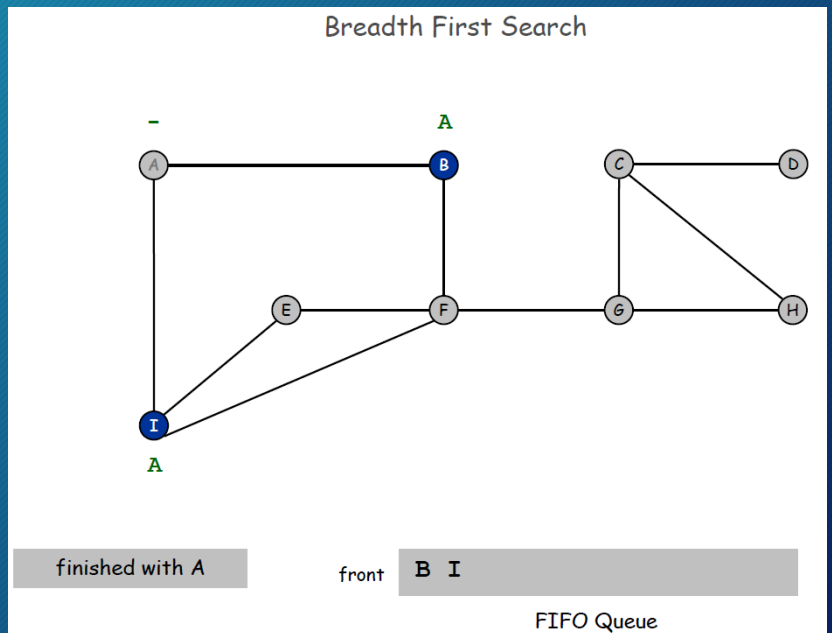



Step 7

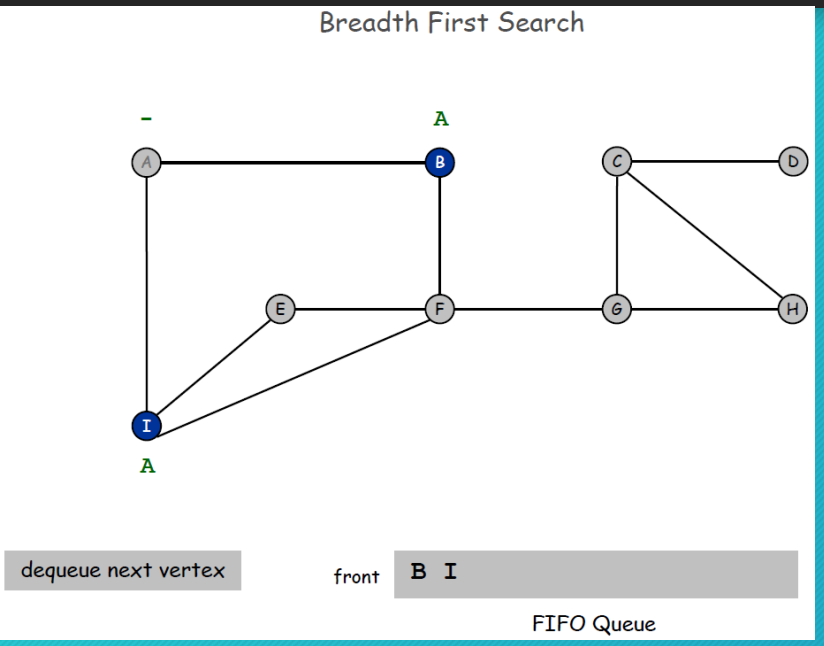



Step 8

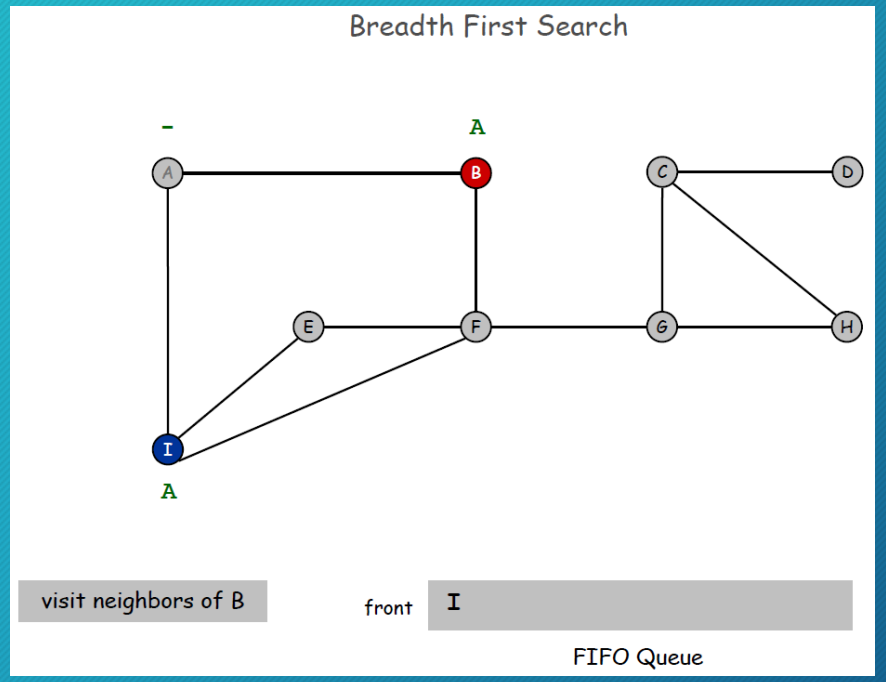
Step 9



# Example: BFS on Graph (contd…)

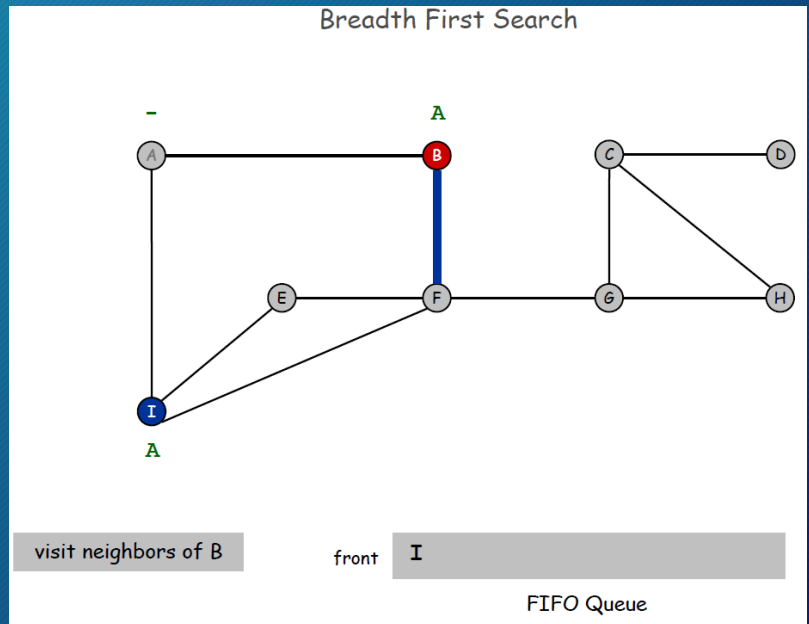



Step 10

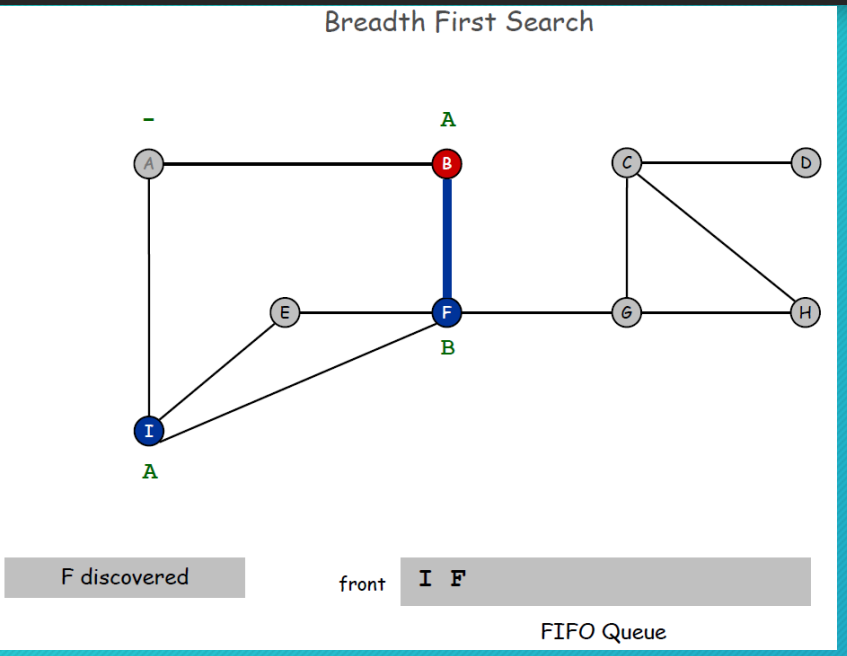



Step 11

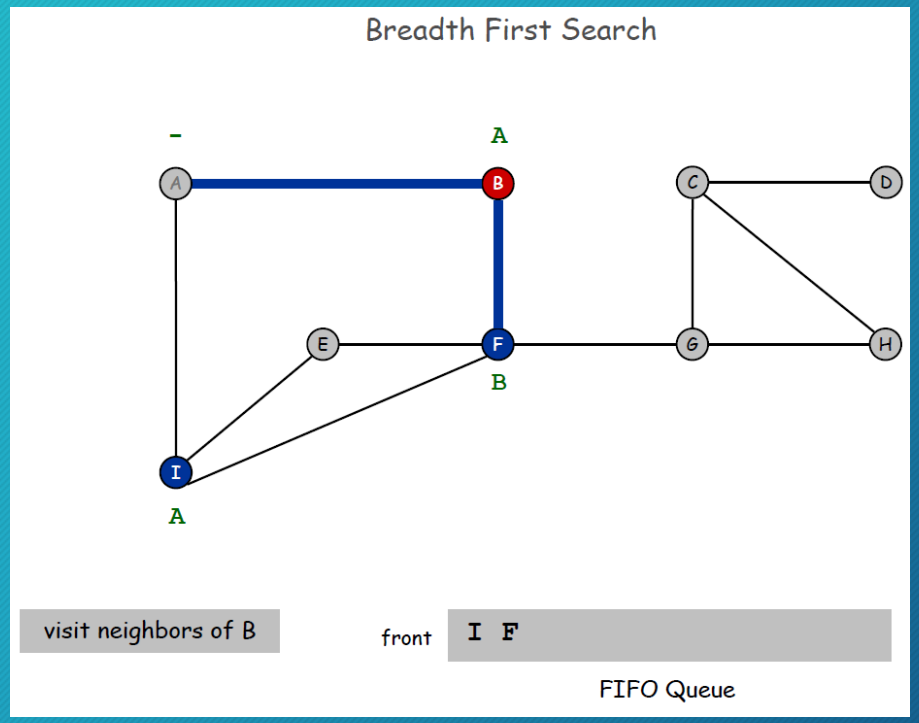
Step 12



# Example: BFS on Graph (contd…)

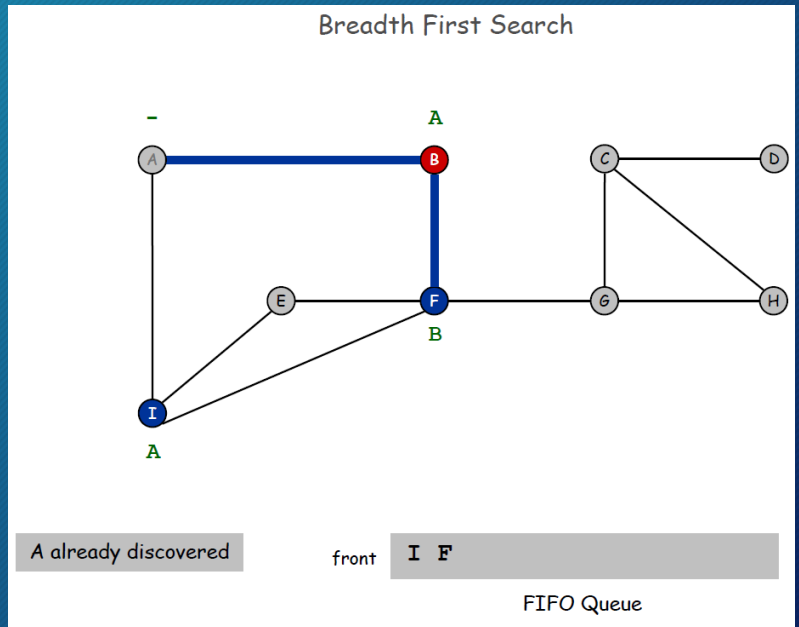



Step 13

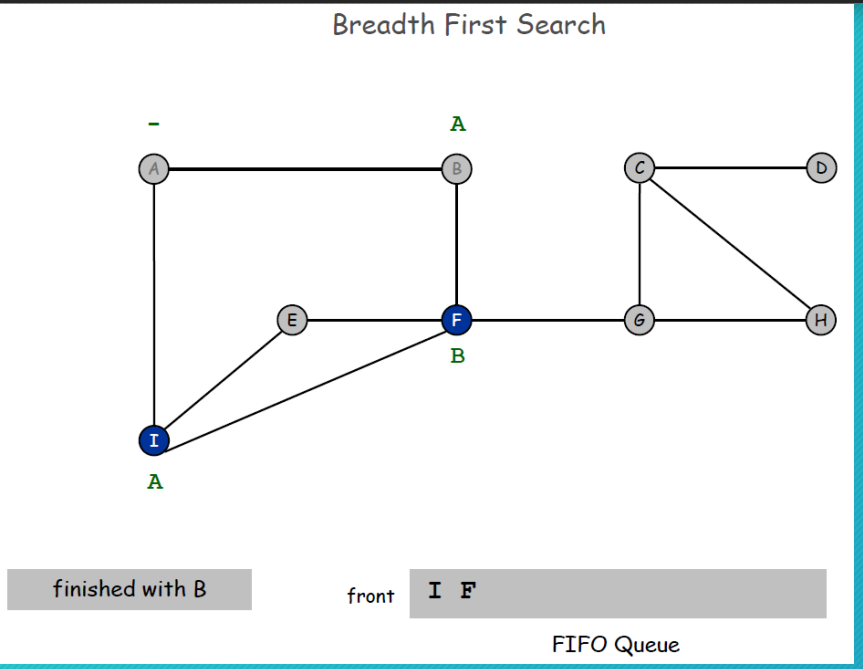



Step 14

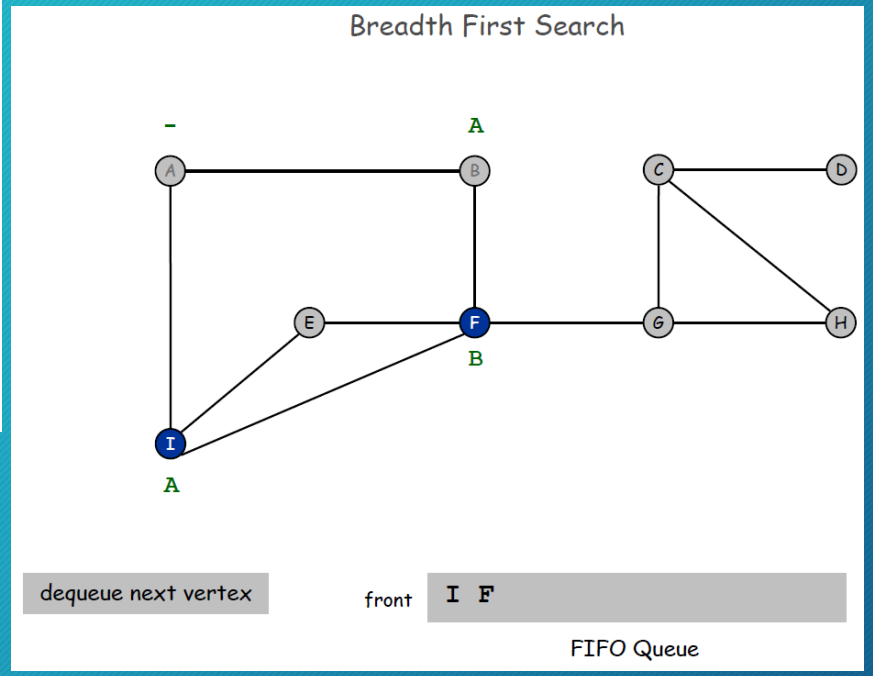
Step 15



# Example: BFS on Graph (contd…)

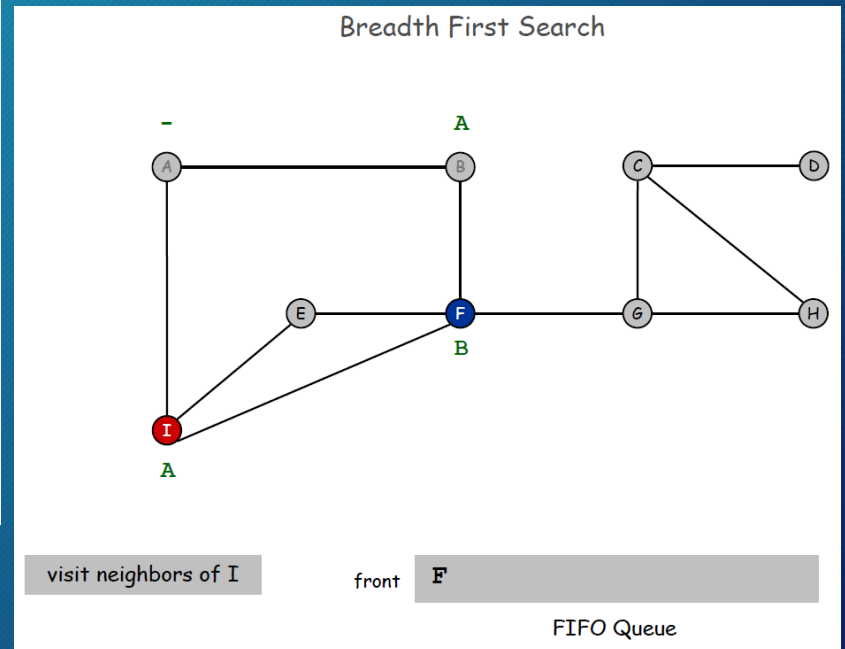



Step 16

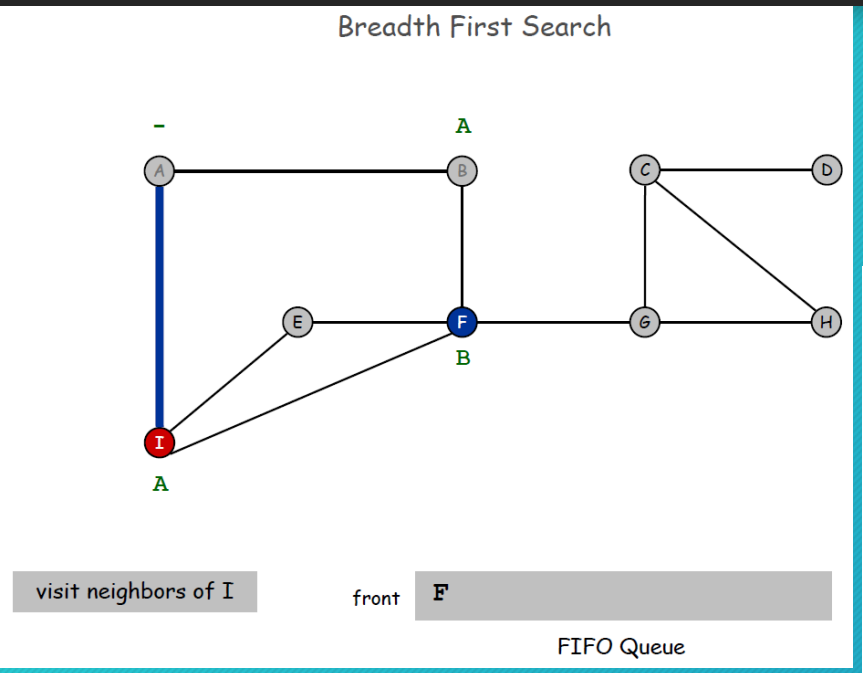



Step 17

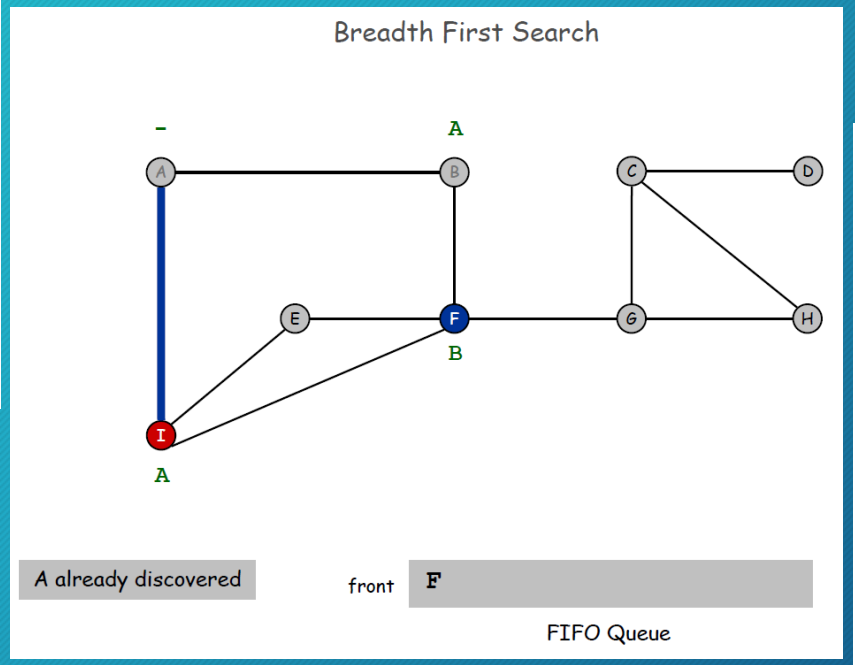
Step 18



# Example: BFS on Graph (contd…)

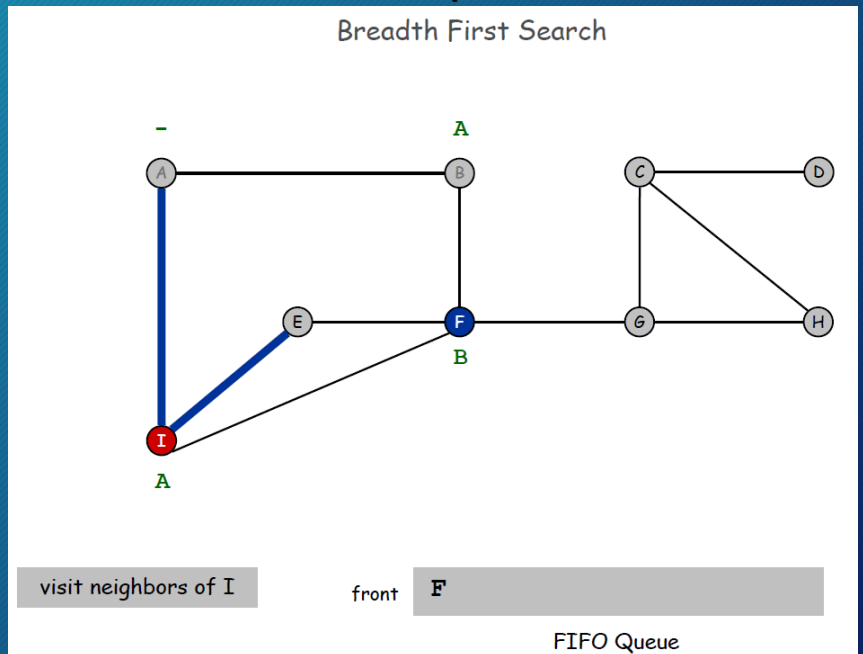



Step 19

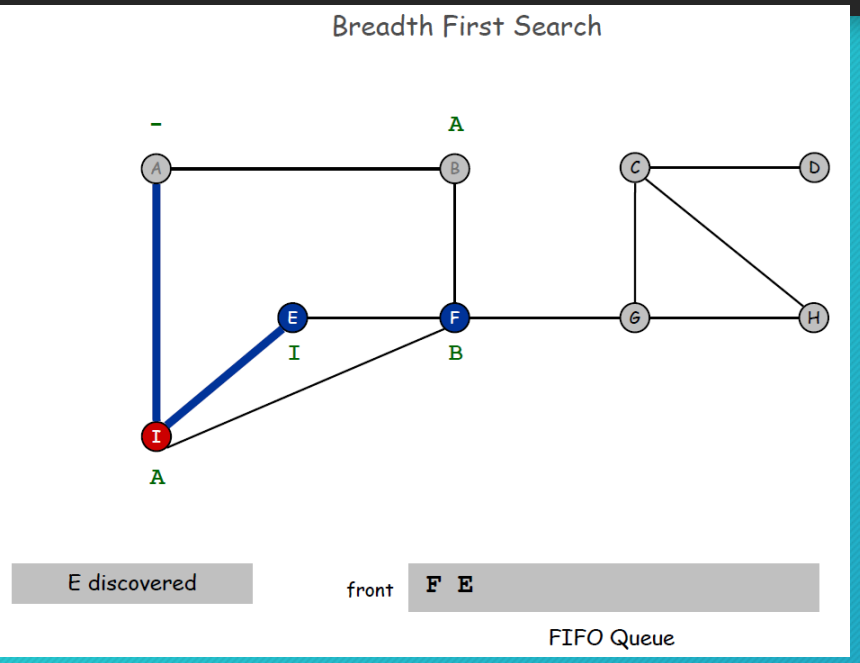



Step 20

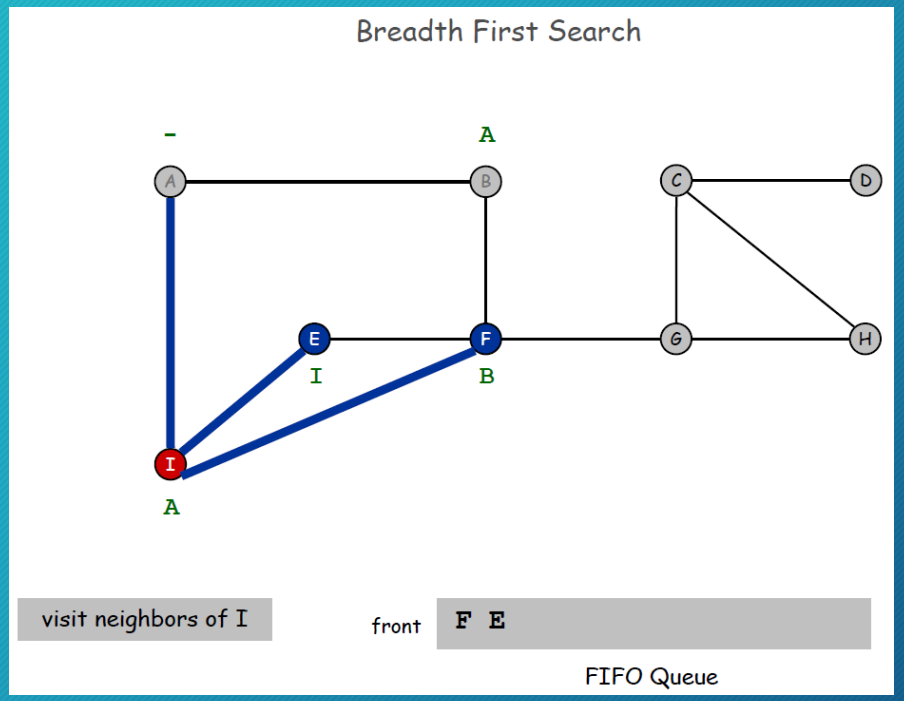
Step 21



# Example: BFS on Graph (contd…)

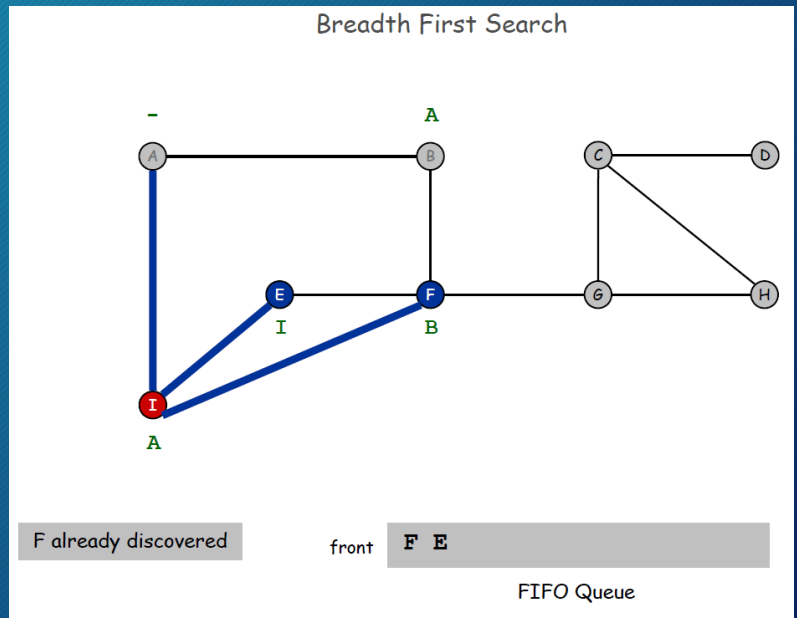



Step 22

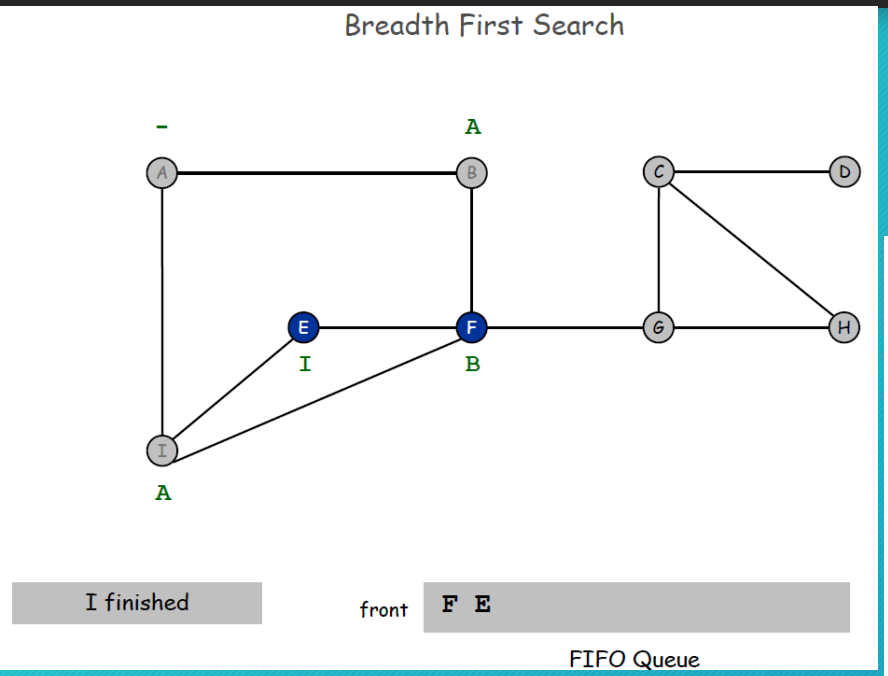



Step 23

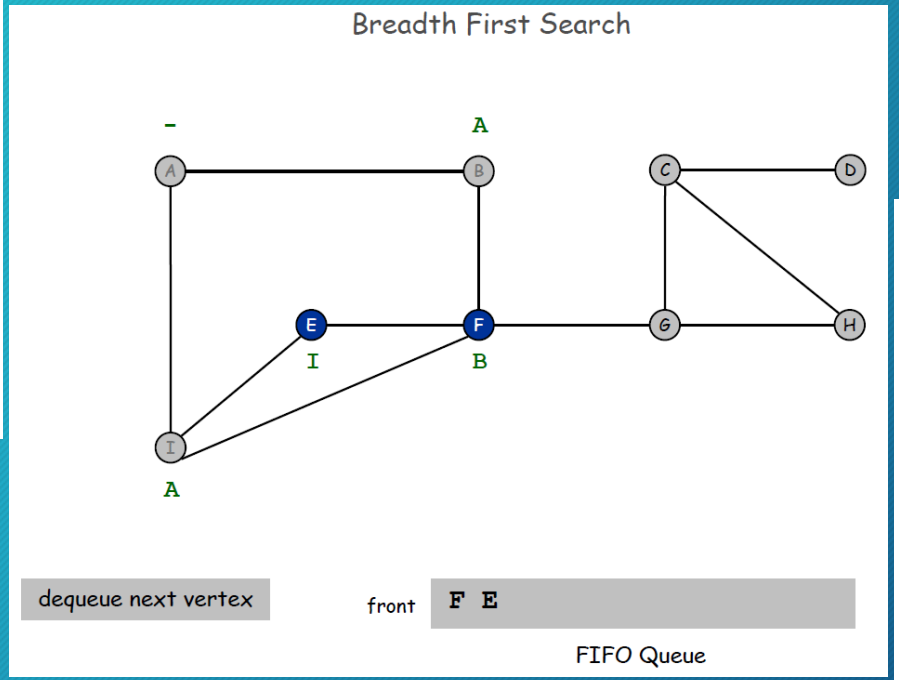
Step 24



# Example: BFS on Graph (contd…)

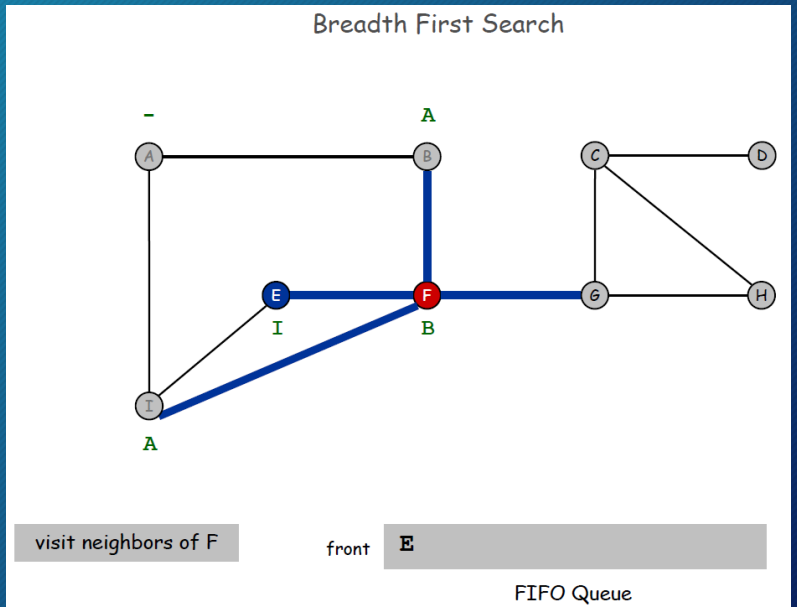



Step 25

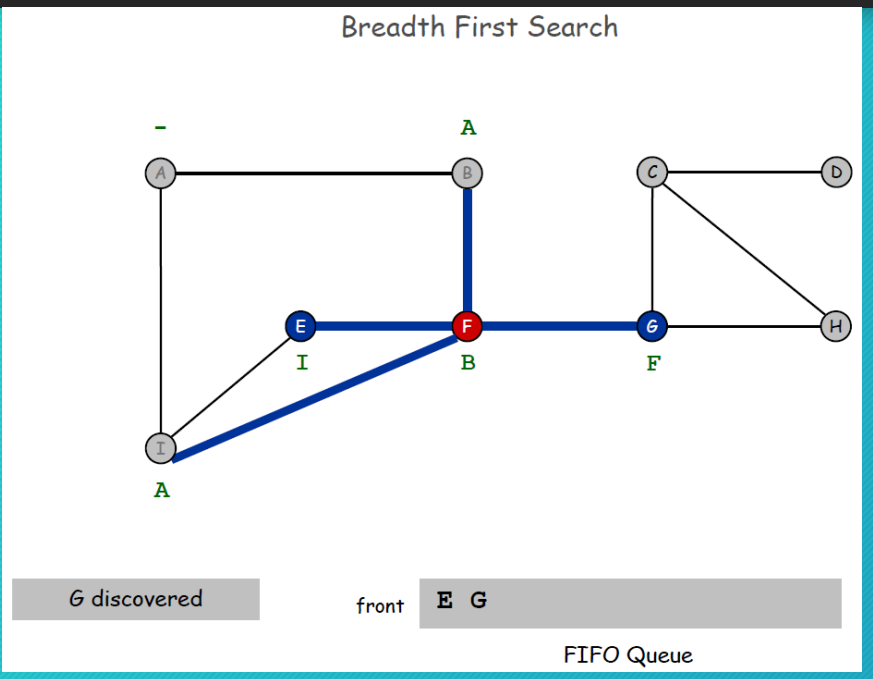



Step 26

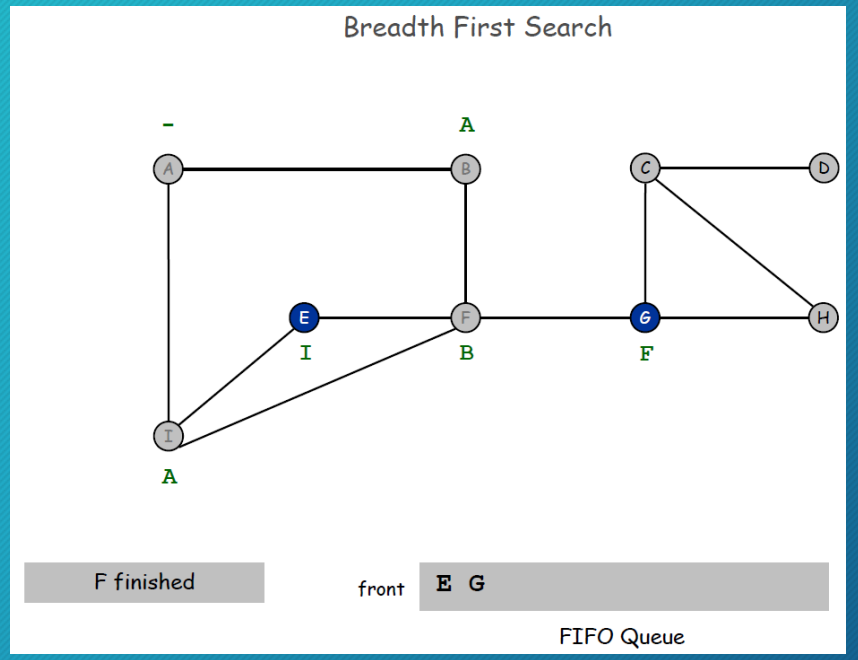
Step 27



# Example: BFS on Graph (contd…)

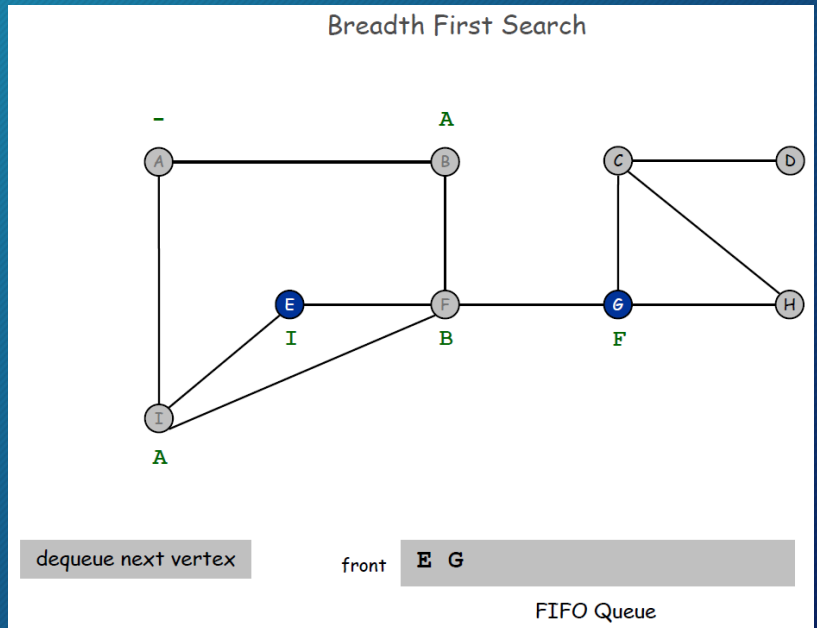



Step 28

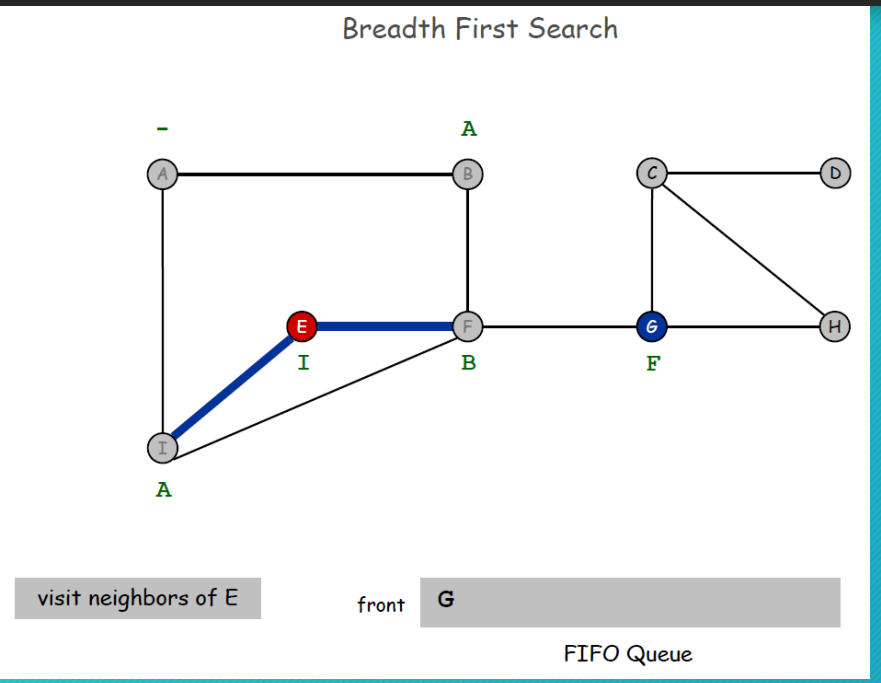



Step 29

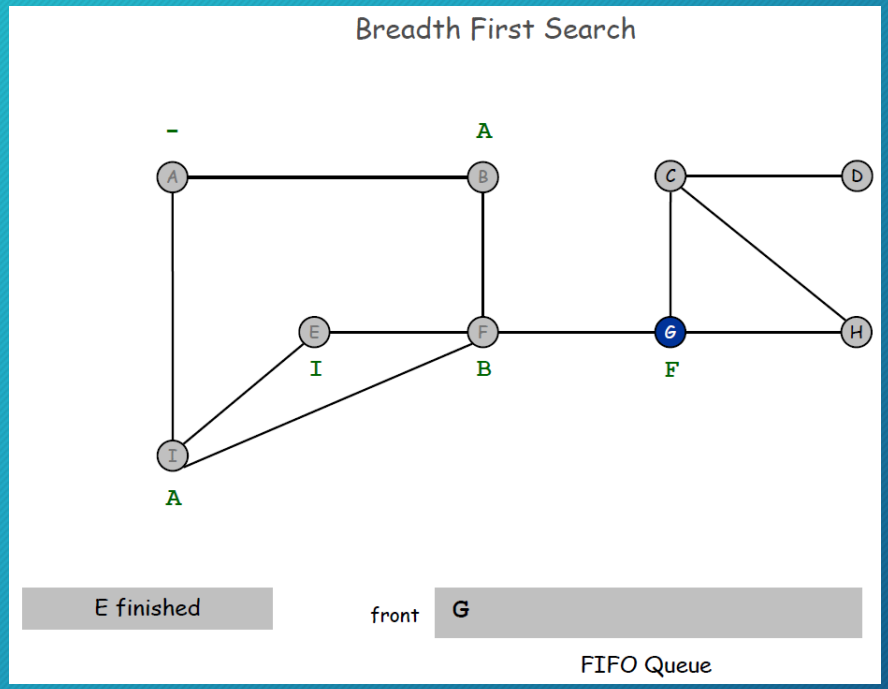
Step 30



# Example: BFS on Graph (contd…)

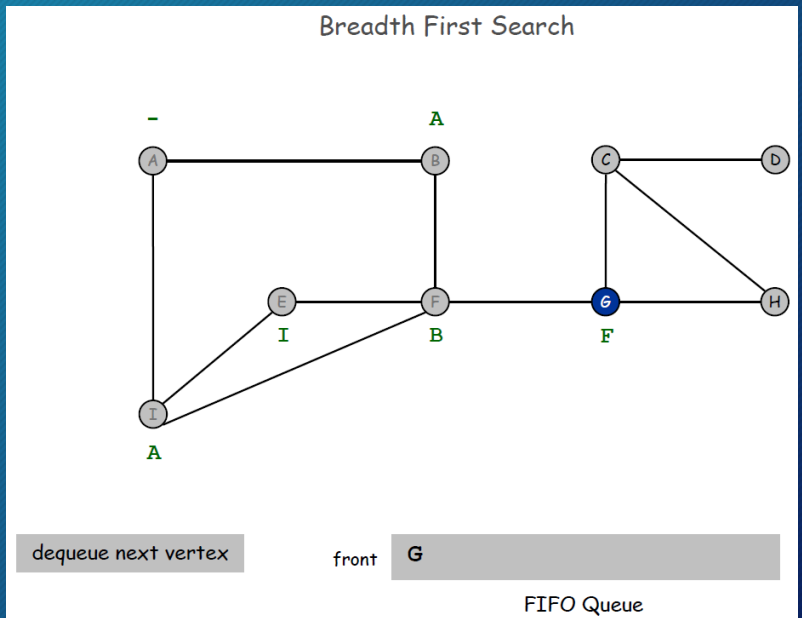



Step 31

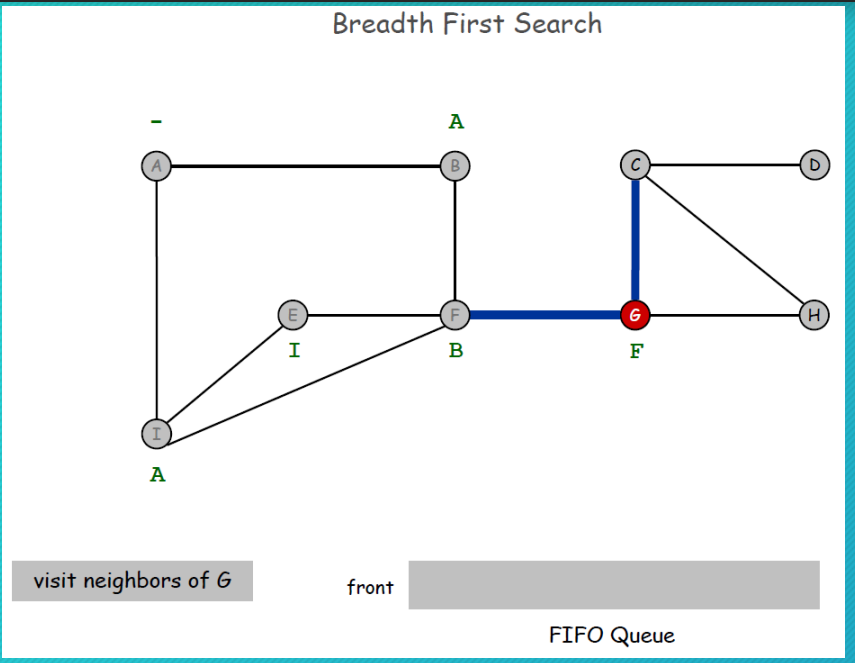



Step 32

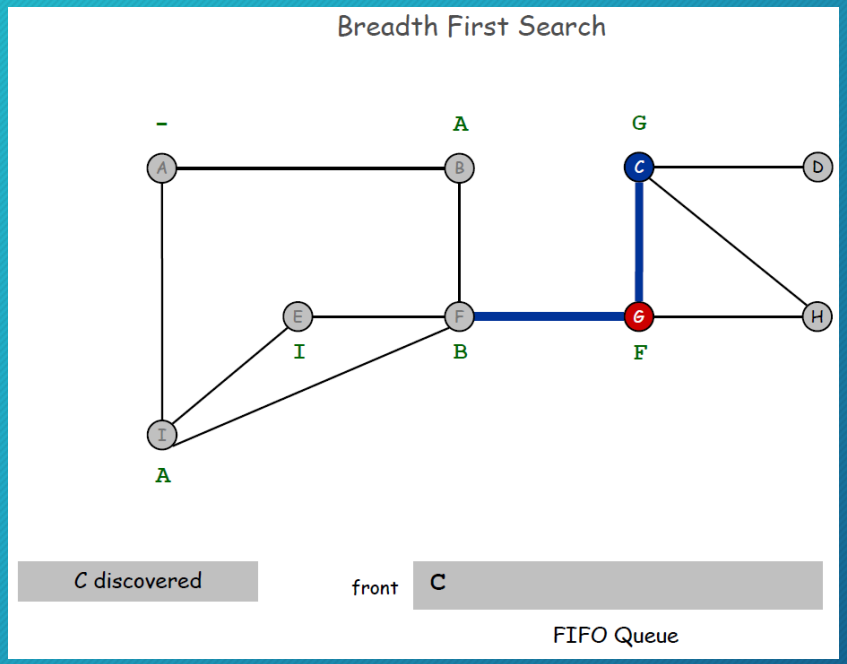
Step 33



# Example: BFS on Graph (contd…)

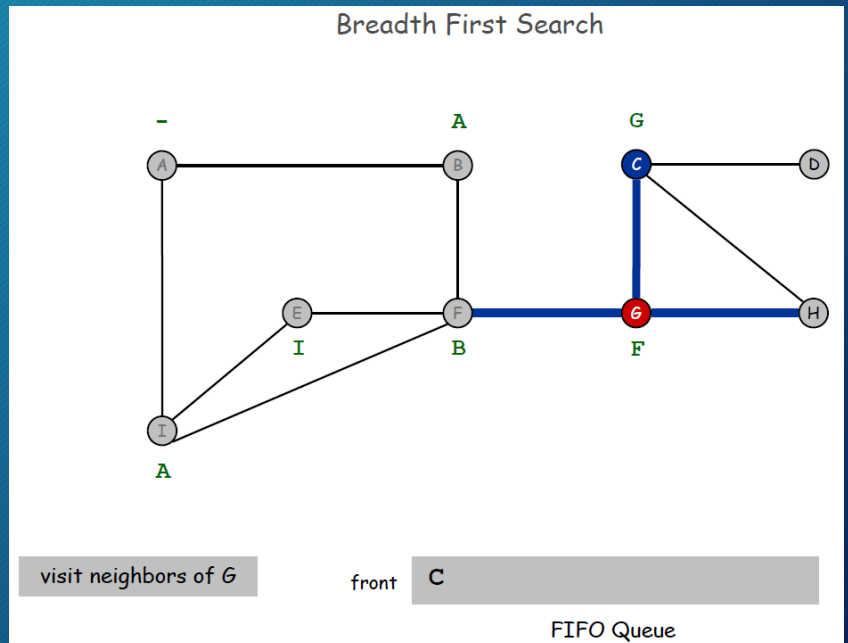



Step 34

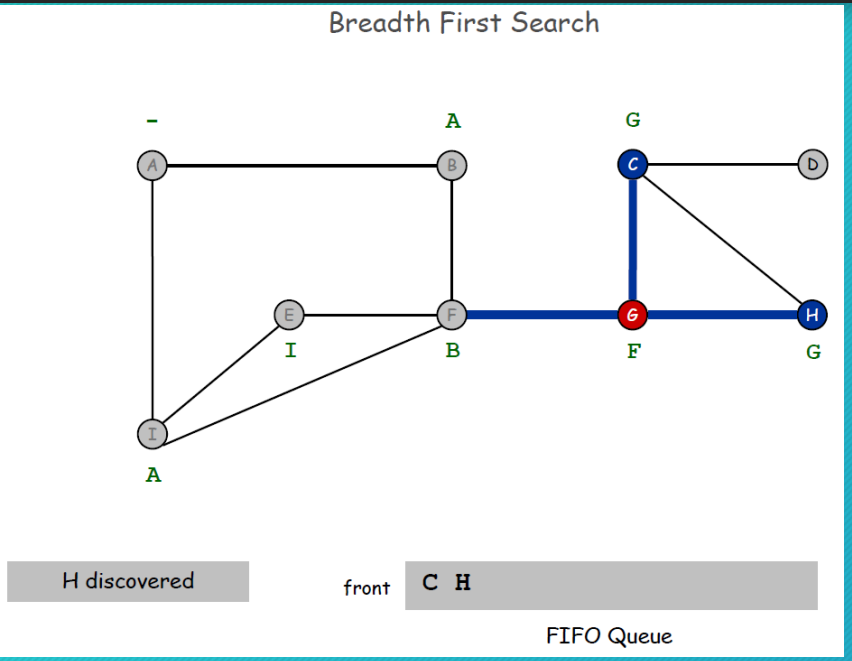



Step 35

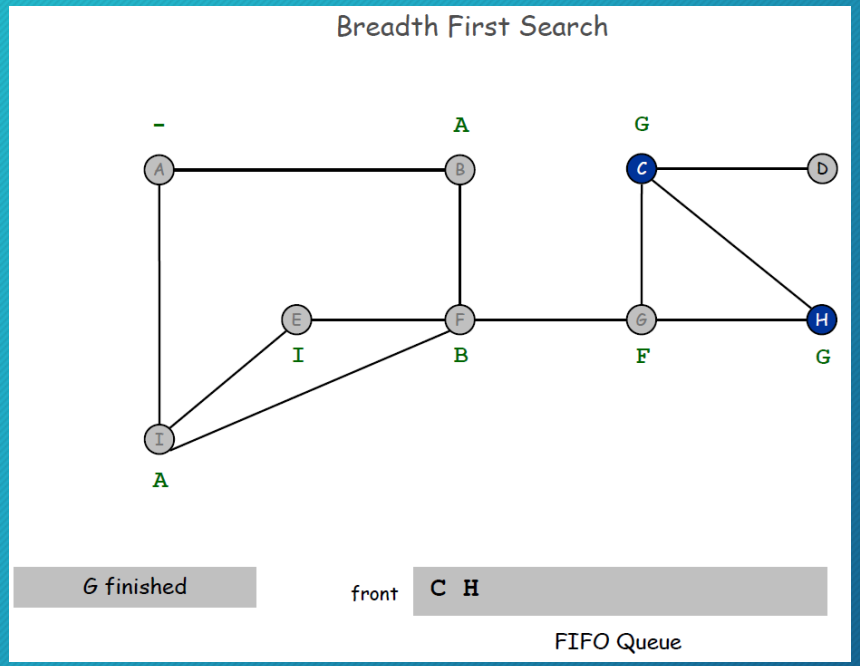
Step 36



# Example: BFS on Graph (contd…)

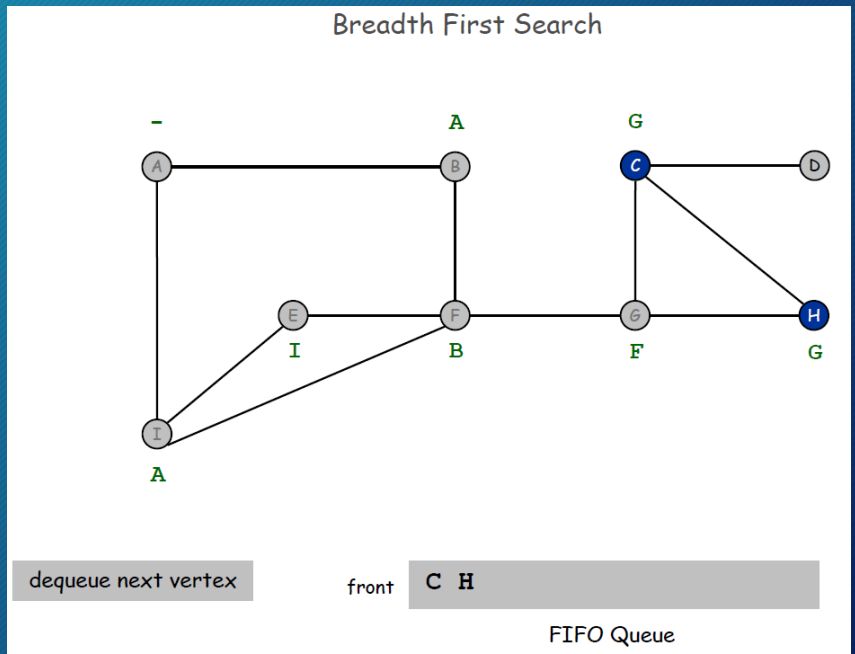



Step 37

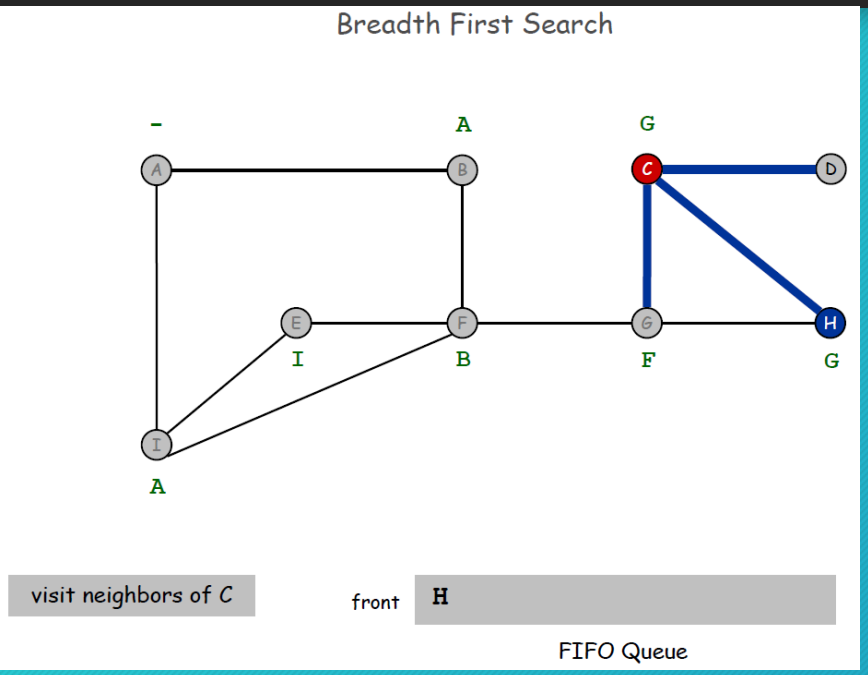



Step 38

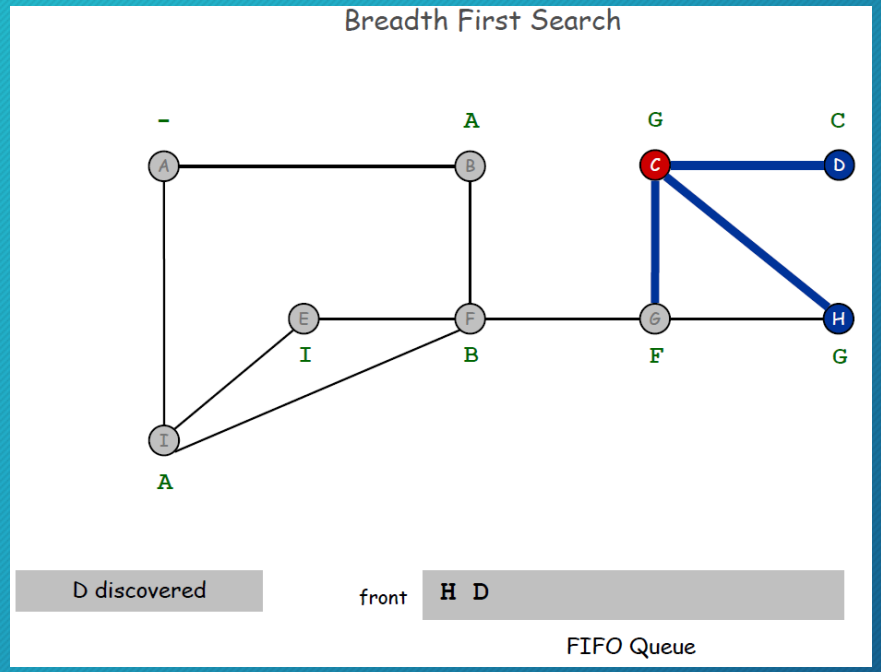
Step 39



# Example: BFS on Graph (contd…)

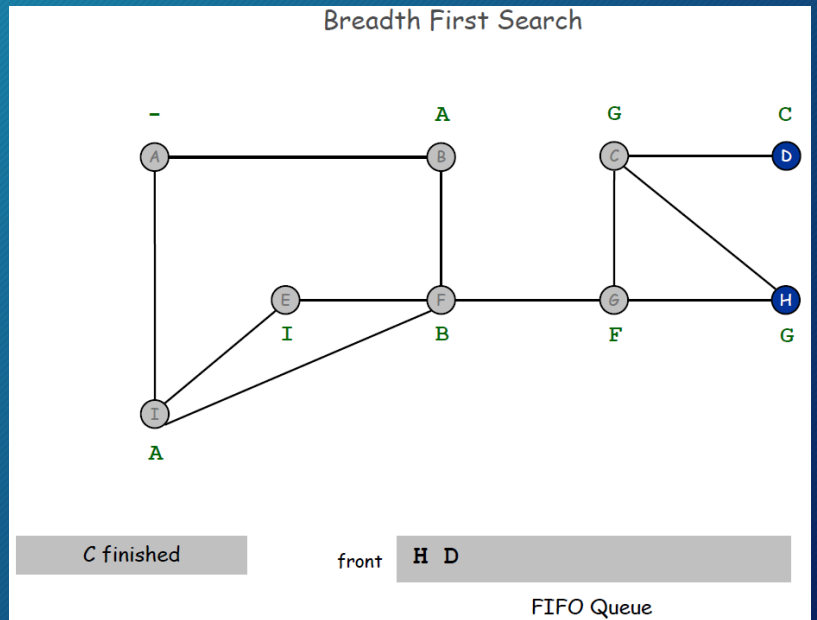



Step 40

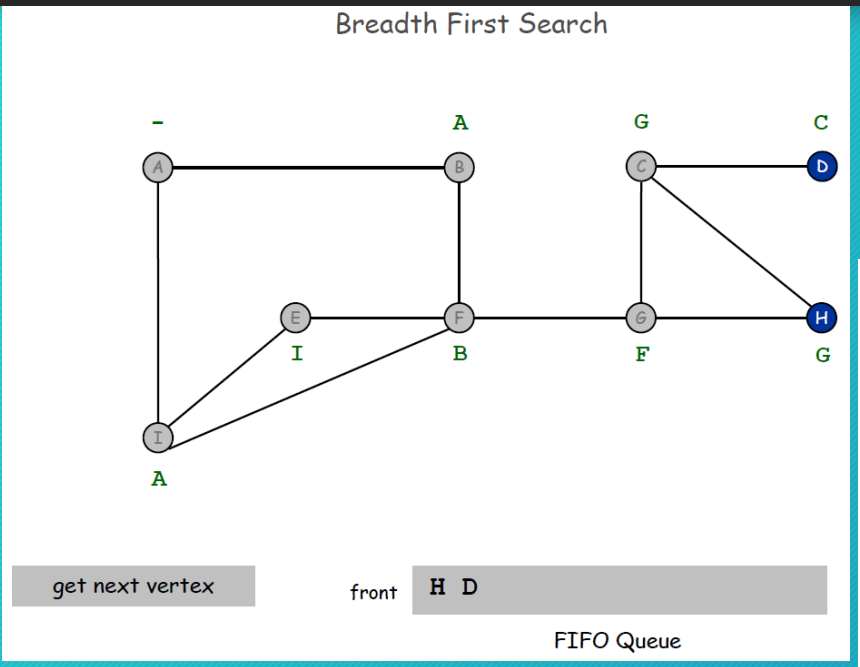



Step 41

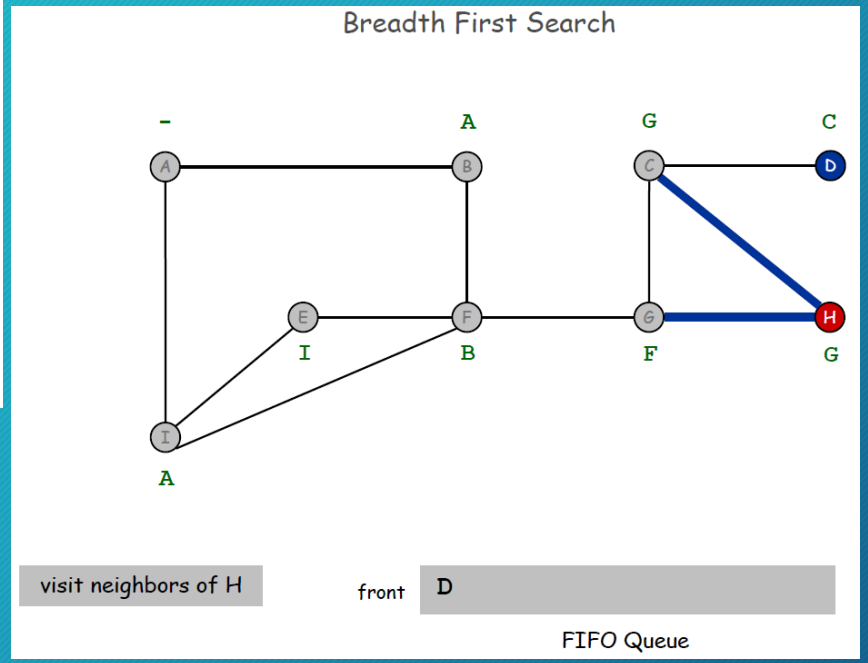
Step 42



# Example: BFS on Graph (contd…)

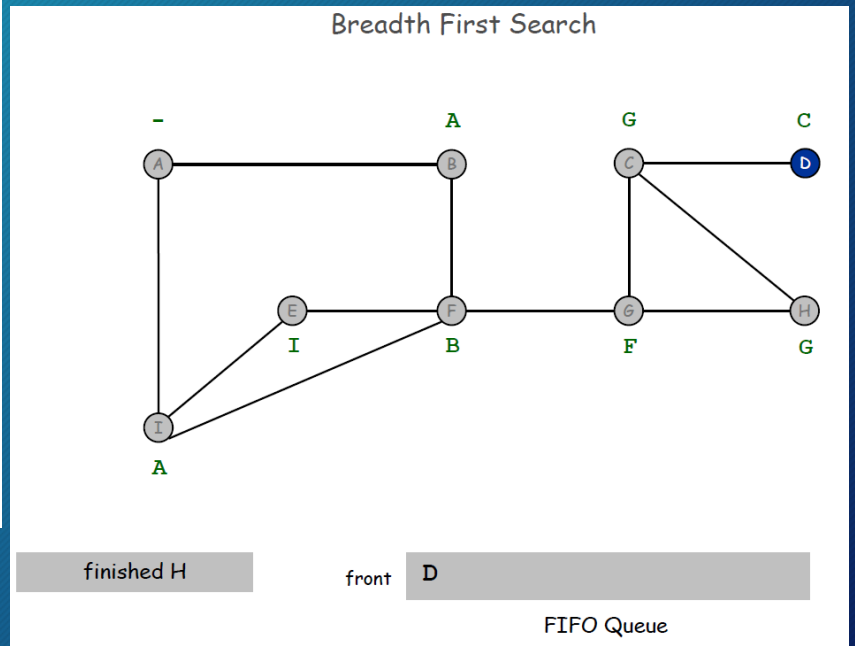



Step 43

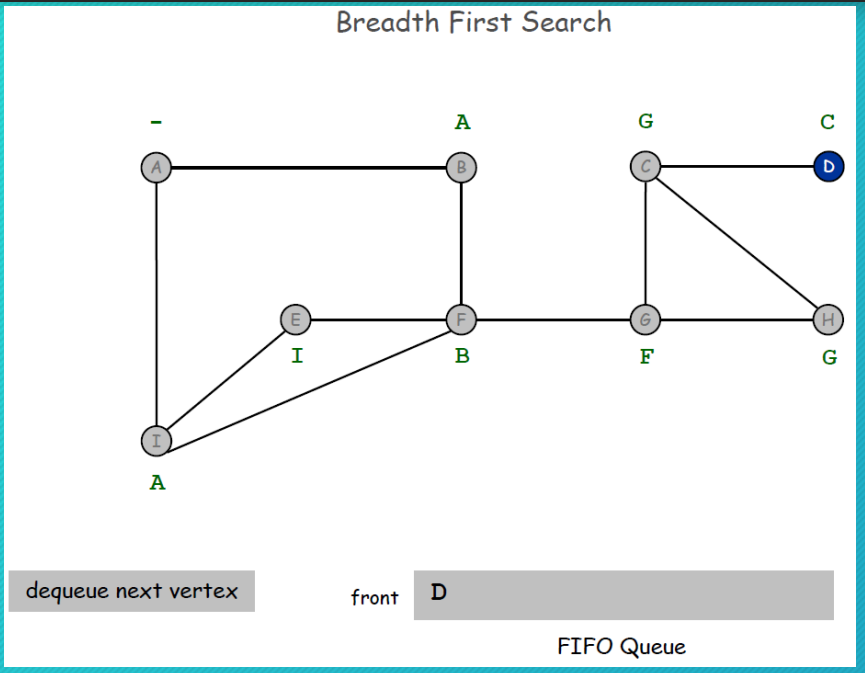



Step 44

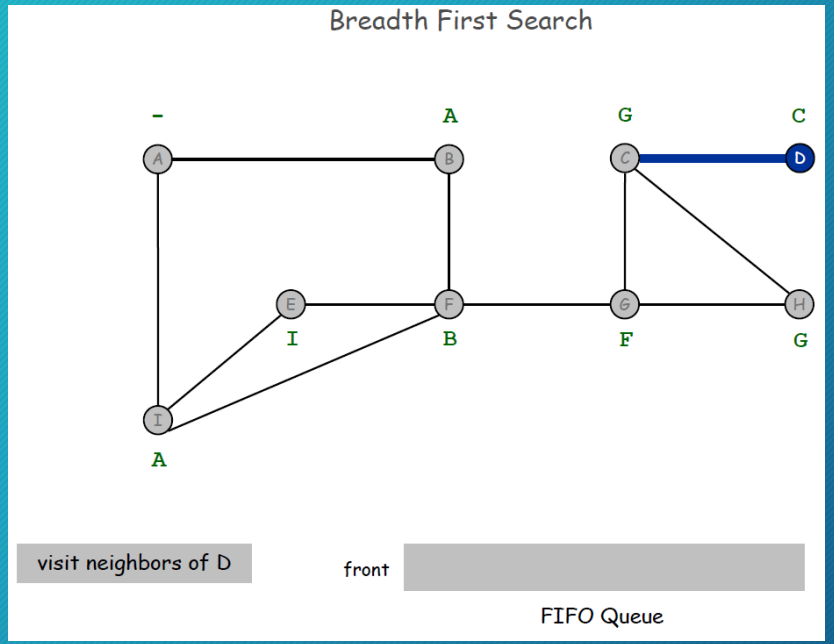
Step 45



# Example: BFS on Graph (contd…)

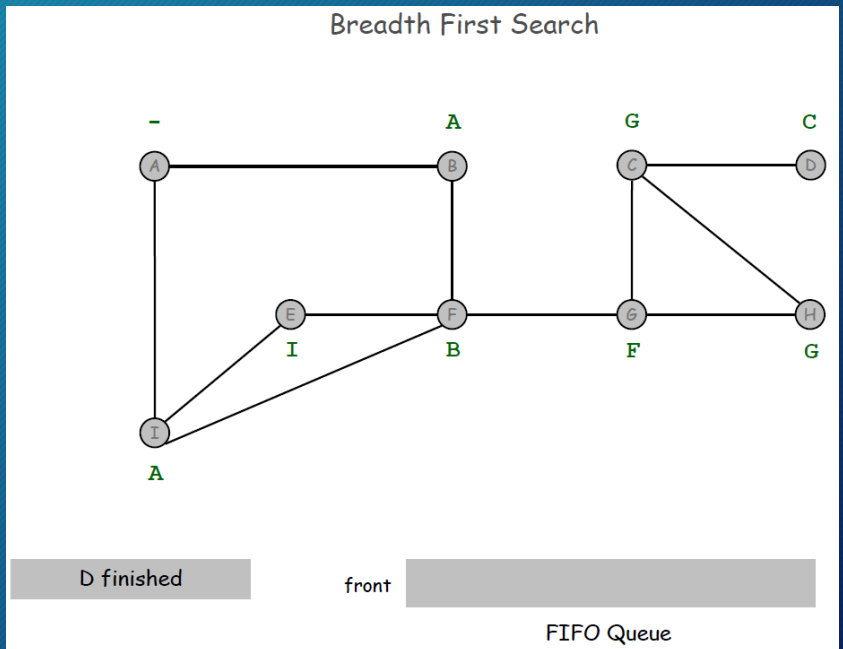



Step 46

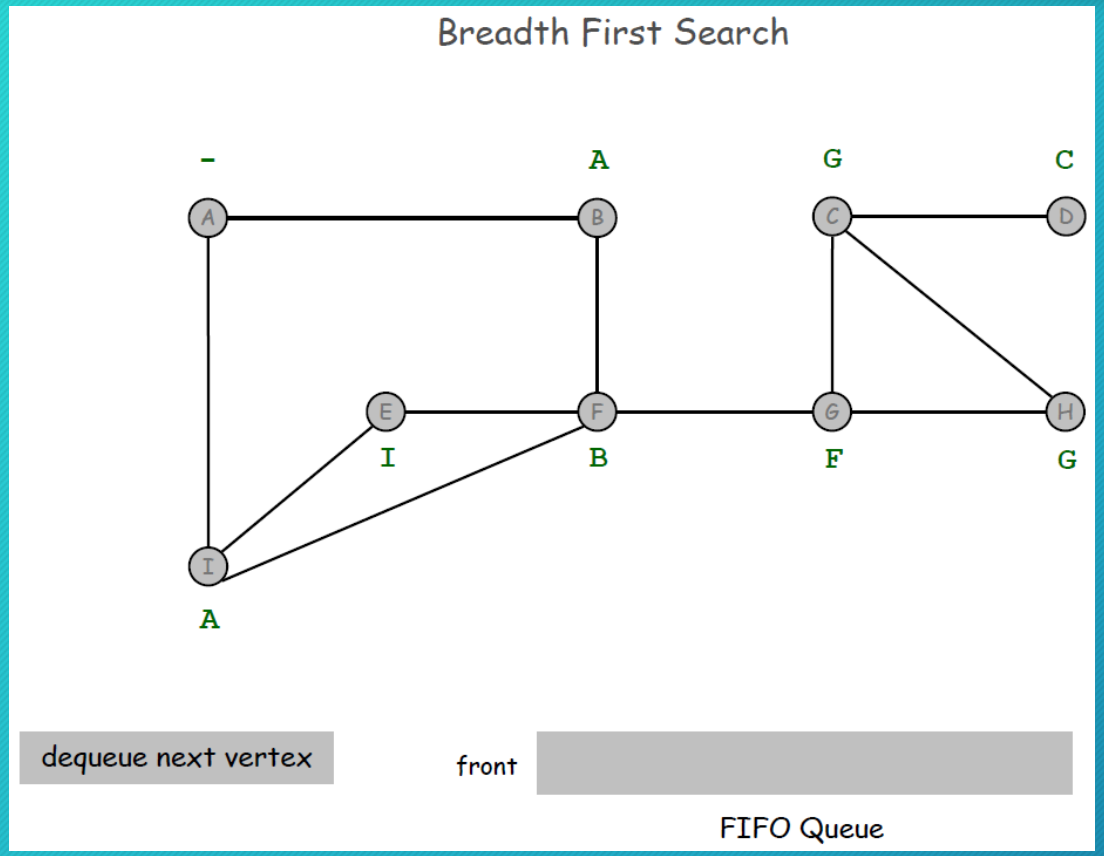



Step 47

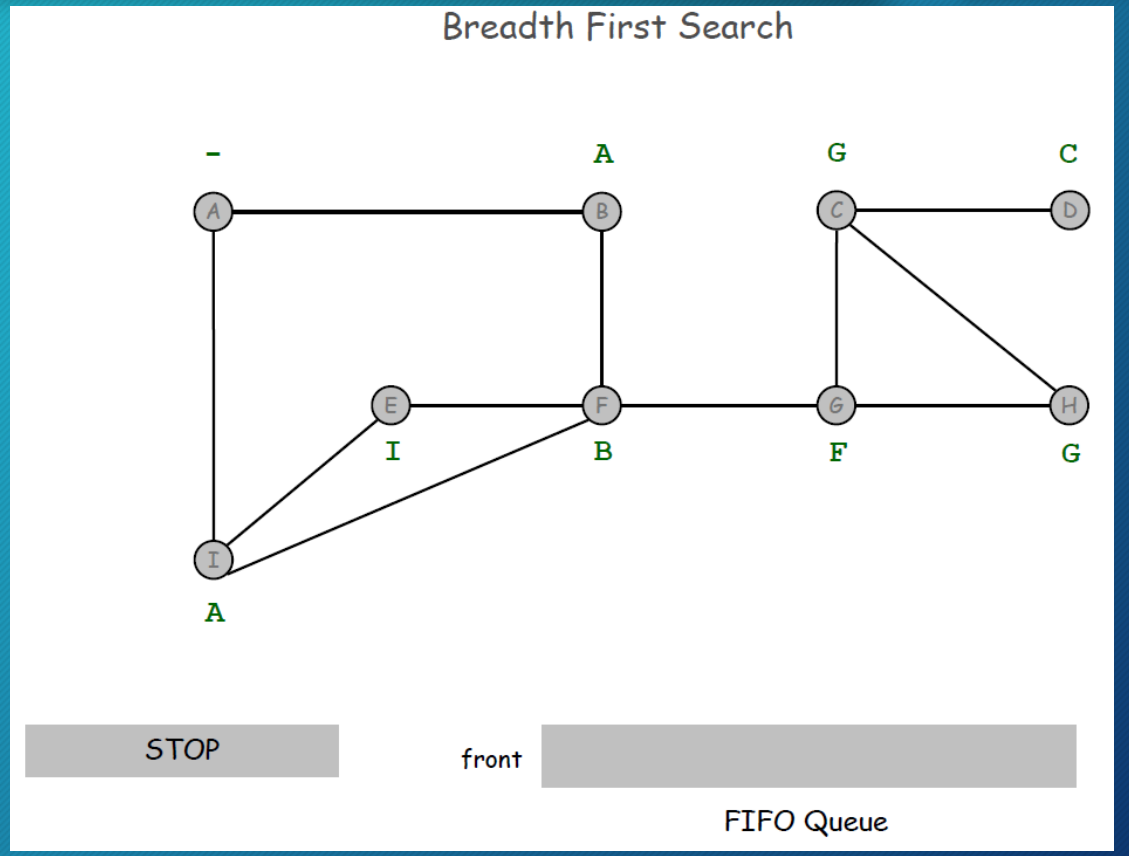
Step 48



# Example: BFS on Graph (contd…)




Step 49




Step 50

# Uniform Cost Search (UCS)

- When all step costs are equal, breadth-first search is optimal because it always expands the *shallowest* unexpanded node.
- By a simple extension, we can find an algorithm that is optimal with any step-cost function. Instead of expanding the shallowest node, **uniform-cost search** expands the node n with the *lowest path cost* g(n).
- This is done by storing the frontier as a priority queue ordered by g.
- Backtracking is allowed on weighted graph.

# UCS Function

```
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
```

# BFS Vs UCS

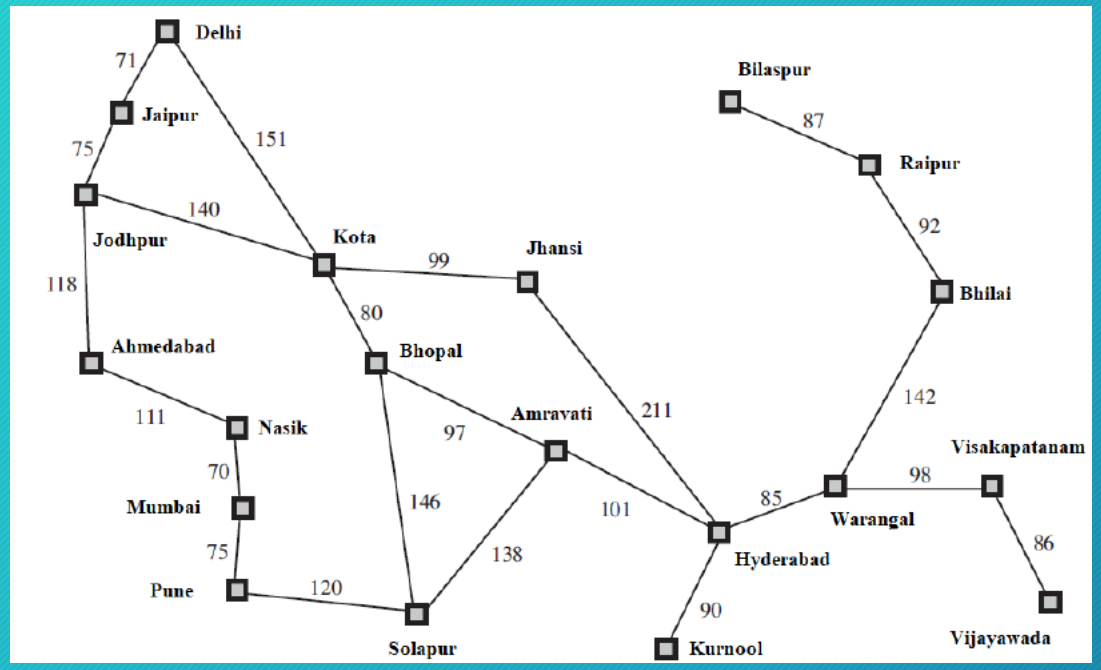
- In addition to the ordering of the queue by path cost, there are **two other significant differences** from breadth-first search.

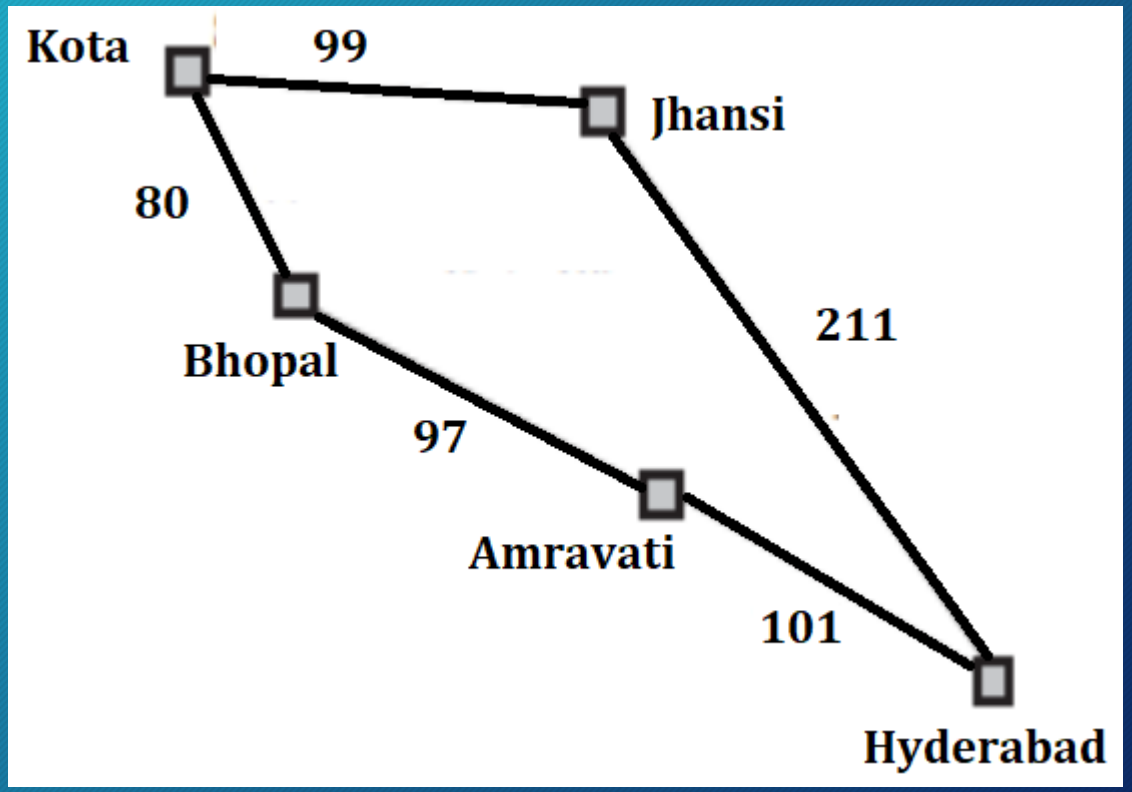
1) The first is that the goal test is applied to a node when it is *selected for expansion* rather than when it is first generated. The reason is that the first goal node that is *generated* may be on a suboptimal path.

2) The second difference is that a test is added in case a better path is found to a node currently on the frontier.

- Both of these modifications come into play in the next example where the problem is to get from Kota to Hyderabad in the India Touring problem.

# UCS Example


Delhi
71
Jaipur
75
151
140
Jodhpur
Kota
99
Jhansi
118
80
Ahmedabad
Bhopal
111
Nasik
97
Amravati
211
70
Mumbai
146
101
75
138
Pune
120
Solapur
Bilaspur
87
Raipur
92
Bhilai
142
Visakapatanam
98
85
Warangal
Hyderabad
86
90
Kurnool
Vijayawada


Kota
99
Jhansi
80
211
Bhopal
97
Amravati
101
Hyderabad

# UCS Example (contd…)

- The successors of Kota are Bhopal and Jhansi, with costs 80 and 99, respectively.
- The least-cost node, Bhopal, is expanded next, adding Amravati with cost 80 + 97=177.
- The least-cost node is now Jhansi, so it is expanded, adding Hyderabad with cost 99+211=310.
- Now a goal node has been generated, but uniform-cost search keeps going, choosing Amravati for expansion and adding a second path to Hyderabad 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. Hyderabad, now with g-cost 278, is selected for expansion and the solution is returned.
- It is easy to see that uniform-cost search is optimal in general. Uniform-cost search expands nodes in order of their optimal path cost. (It is because step costs are non-negative and paths never get shorter as nodes are added.)
- Hence, the first goal node selected for expansion must be the optimal solution.

# Properties of UCS

- Expand least-cost unexpanded node
- Implementation:
  - *fringe* = queue ordered by path cost
- Equivalent to breadth-first if step costs all equal
- Complete? **Yes**, if step cost ≥ ε
- Time? Number of nodes with $g \leq$ cost of optimal solution, $O(b^{ceiling(C^*/\ \varepsilon)})$ where $C^*$ is the cost of the optimal solution
- Space? Number of nodes with $g \leq$ cost of optimal solution, $O(b^{ceiling(C^*/\ \varepsilon)})$
- Optimal? **Yes** – nodes expanded in increasing order of $g(n)$

- **Disadvantage:** It may stuck in infinite loop.

# Depth First Search (DFS)

- It is a strategy that expands the deepest node in the search tree.
- Stack is used for DFS.

1. Form a one element stack consisting of the root node.
2. Until the stack is empty or the goal node is found out, repeat the following:-
    1. Remove the first element from the stack. If it is the goal node announce success, return.
    2. If not, add the first element's children if any, to the top of the stack.
3. If the goal node is not found announce failure.

# Properties of DFS

- Complete? **No**. fails in infinite-depth spaces, spaces with loops
  - We may modify the algorithm to avoid repeated states along the path
  - DFS is complete in finite spaces.

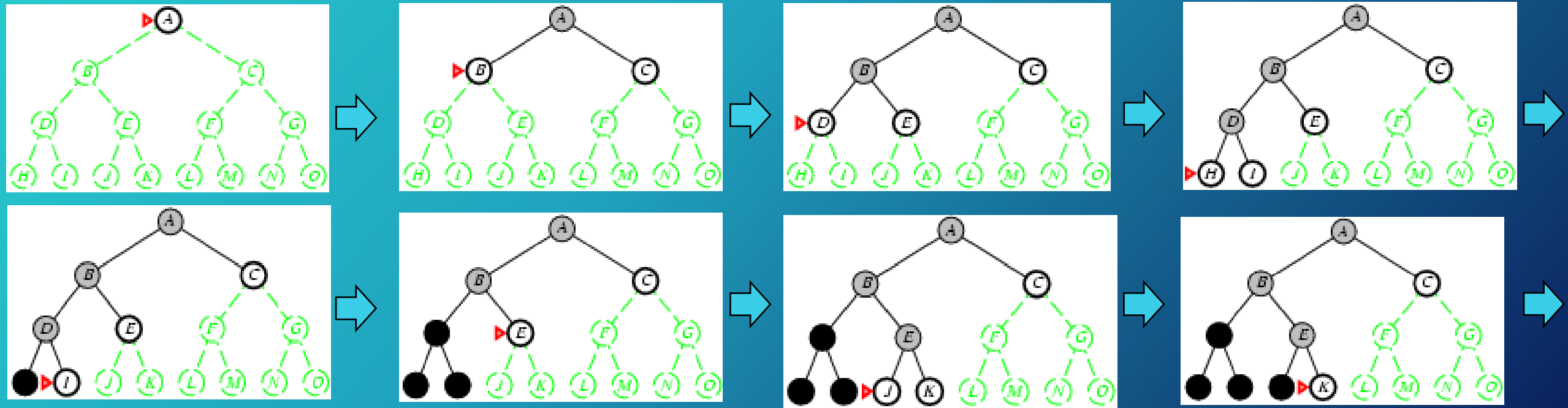
- Time? The number of nodes generated is of the order of $O(b^m)$ . It is terrible if $m$ is much larger than $d$
  - but if solutions are dense, may be much faster than breadth-first

- Space? It only stores the unexpanded nodes. So, it requires 1+b+b+b ….. m times = $O(bm)$, i.e., linear space!

- Optimal? **No**. If it makes a wrong choice, it may go down a very long path and finds a solution, where as there may be a better solution at a higher level in the tree.

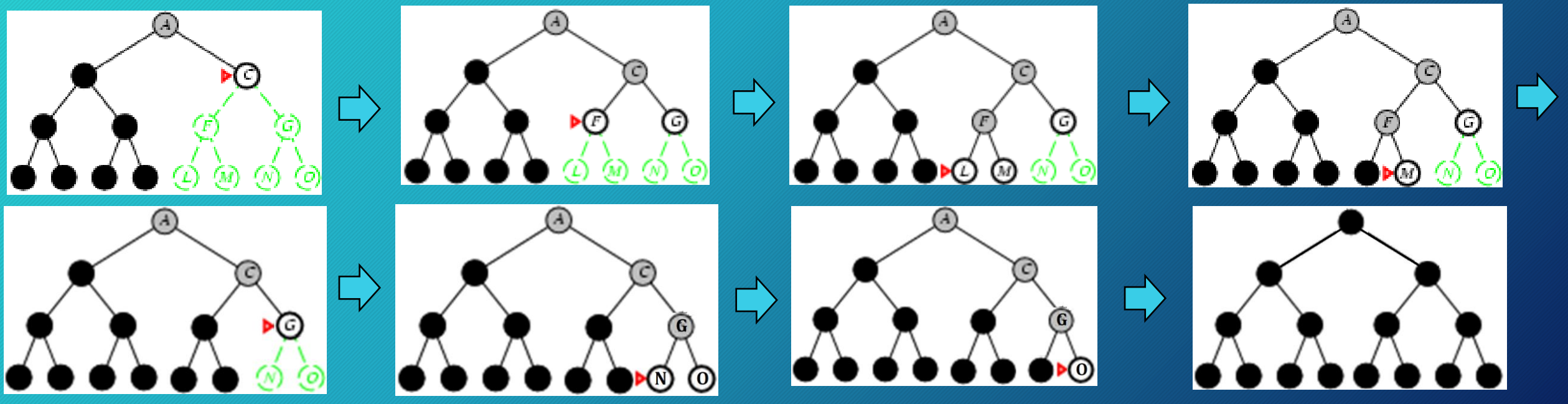
# DFS on Tree

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

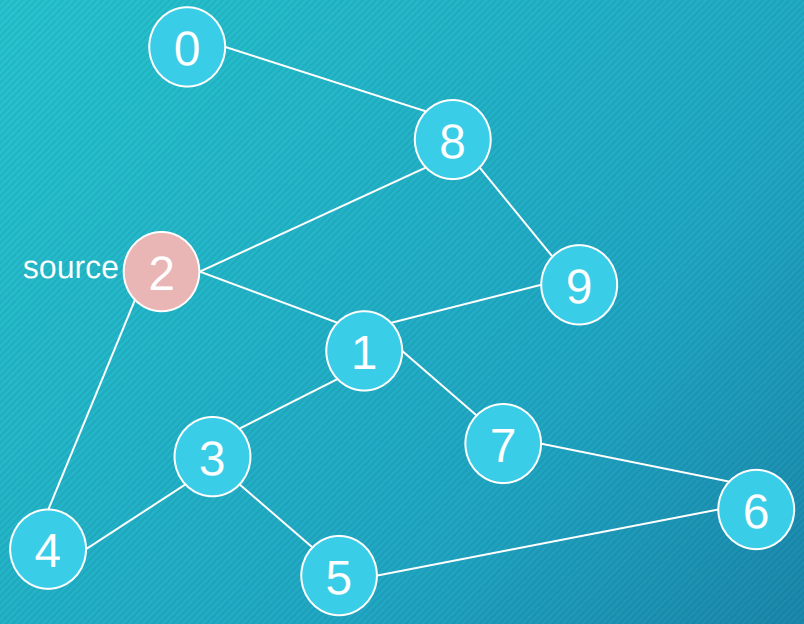


# DFS on Tree (contd…)

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



# Example: DFS on Graph




Adjacency List

| | |
|---|---|
| 0 | 8 |
| 1 | 3 7 9 2 |
| 2 | 8 1 4 |
| 3 | 4 5 1 |
| 4 | 2 3 |
| 5 | 3 6 |
| 6 | 7 5 |
| 7 | 1 6 |
| 8 | 2 0 9 |
| 9 | 1 8 |

Visited Table (T/F)

| | |
|---|---|
| 0 | F |
| 1 | F |
| 2 | F |
| 3 | F |
| 4 | F |
| 5 | F |
| 6 | F |
| 7 | F |
| 8 | F |
| 9 | F |

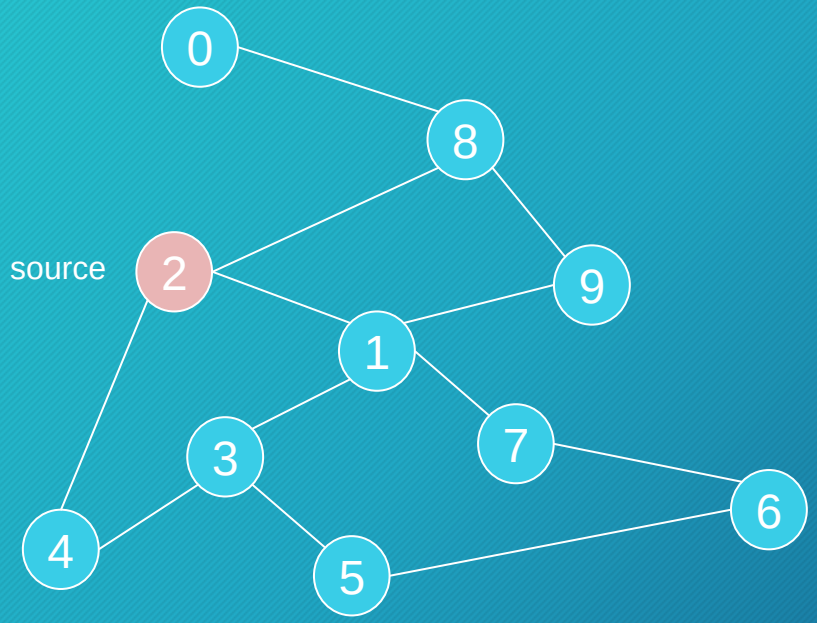
| |
|---|
| - |
| - |
| - |
| - |
| - |
| - |
| - |
| - |
| - |
| - |

*Pred*

Initialize visited table (all False)

Initialize Pred to -1

# Example: DFS on Graph (Contd…)

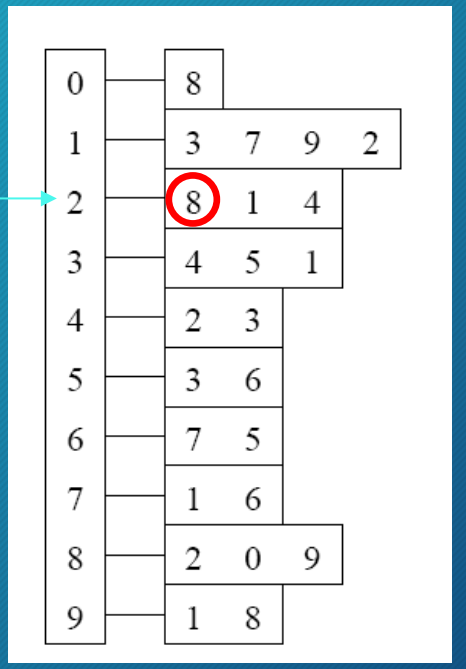



Recursive calls

RDFS( 2 )

Now visit RDFS(8)

Adjacency List



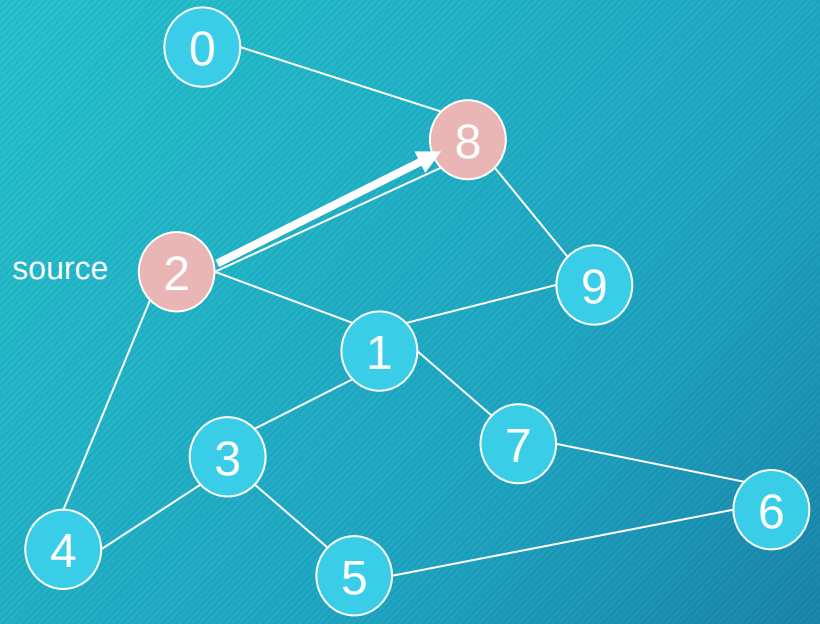

Visited Table (T/F)

| | |
|---|---|
| 0 | F |
| 1 | F |
| 2 | T |
| 3 | F |
| 4 | F |
| 5 | F |
| 6 | F |
| 7 | F |
| 8 | F |
| 9 | F |

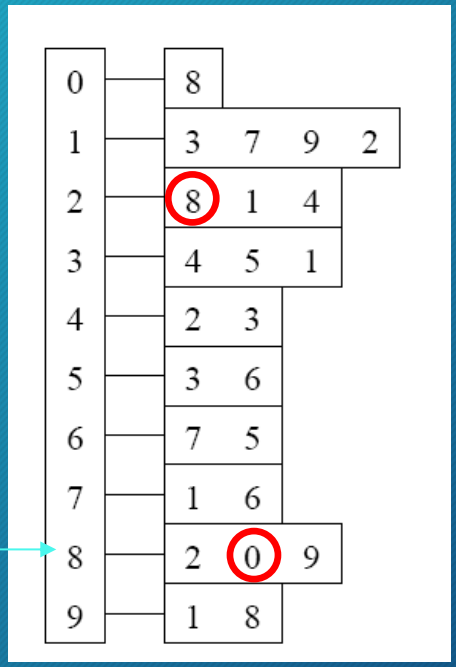
| *Pred* |
|---|
| - |
| - |
| - |
| - |
| - |
| - |
| - |
| - |
| - |
| - |

Mark 2 as visited

# Example: DFS on Graph (Contd…)




Adjacency List




Visited Table (T/F)

| | |
|---|---|
| 0 | F |
| 1 | F |
| 2 | T |
| 3 | F |
| 4 | F |
| 5 | F |
| 6 | F |
| 7 | F |
| 8 | T |
| 9 | F |

| |
|---|
| - |
| - |
| - |
| - |
| - |
| - |
| - |
| - |
| 2 |
| - |

*Pred*

Recursive calls

RDFS( 2 )

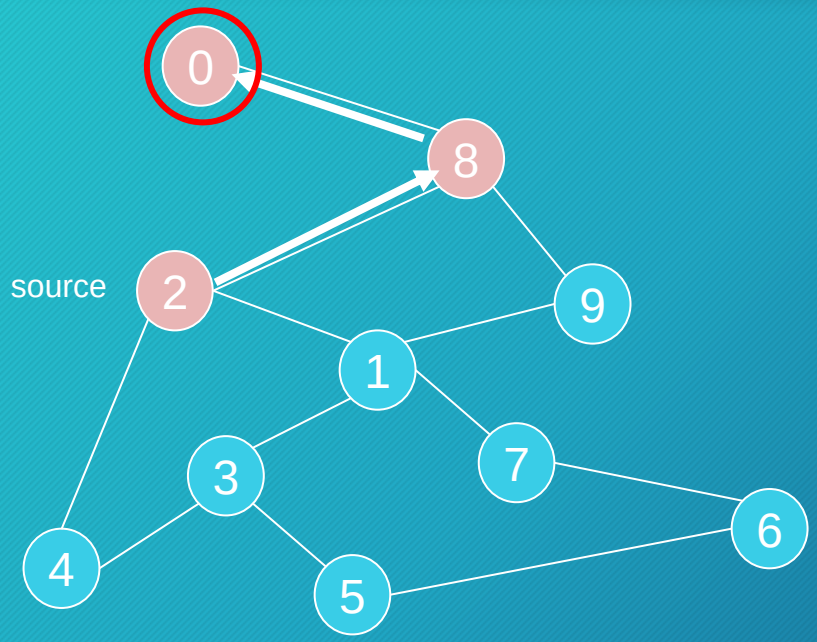
RDFS(8)

2 is already visited, so visit RDFS(0)

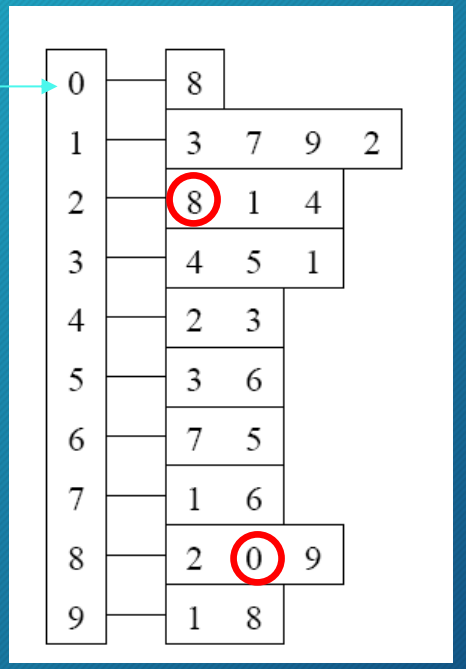
Mark 8 as visited

mark Pred[8]

# Example: DFS on Graph (Contd…)




Adjacency List




Visited Table (T/F)

| | |
|---|---|
| 0 | T |
| 1 | F |
| 2 | T |
| 3 | F |
| 4 | F |
| 5 | F |
| 6 | F |
| 7 | F |
| 8 | T |
| 9 | F |

| *Pred* |
|---|
| 8 |
| - |
| - |
| - |
| - |
| - |
| - |
| - |
| 2 |
| - |

Recursive calls

RDFS( 2 )

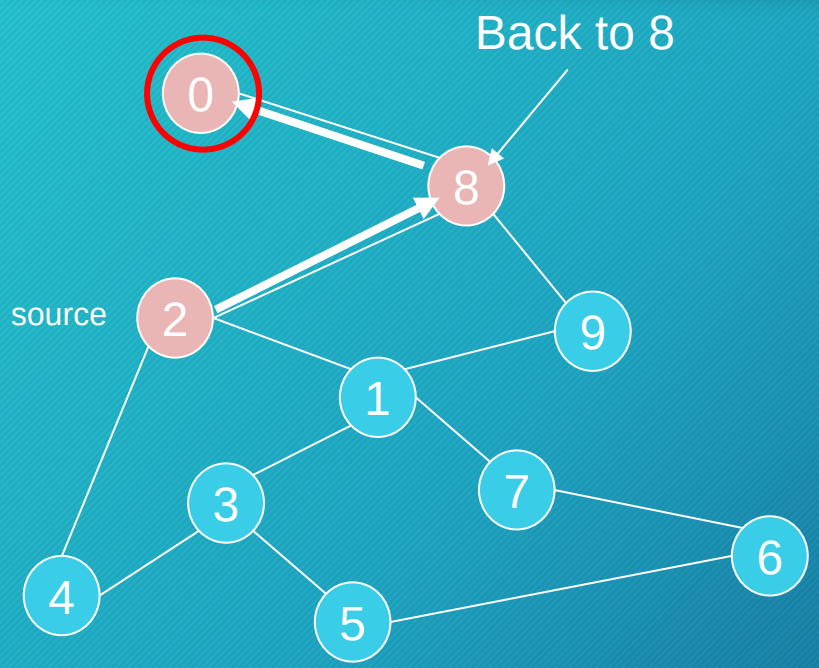
RDFS(8)

RDFS(0) -> no unvisited neighbors, return to call RDFS(8)

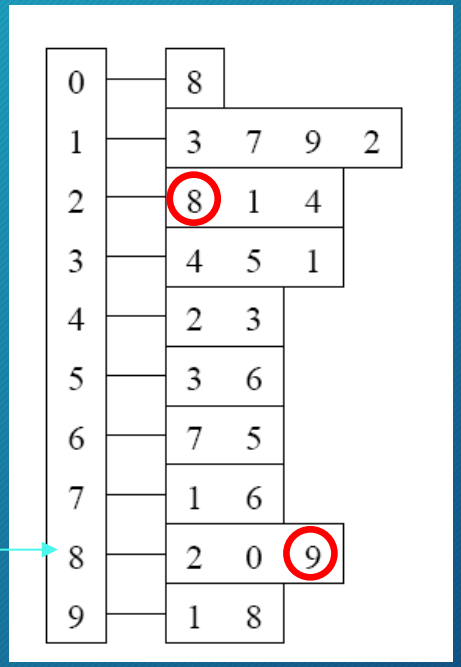
Mark 0 as visited

Mark Pred[0]

# Example: DFS on Graph (Contd…)




Adjacency List




Visited Table (T/F)

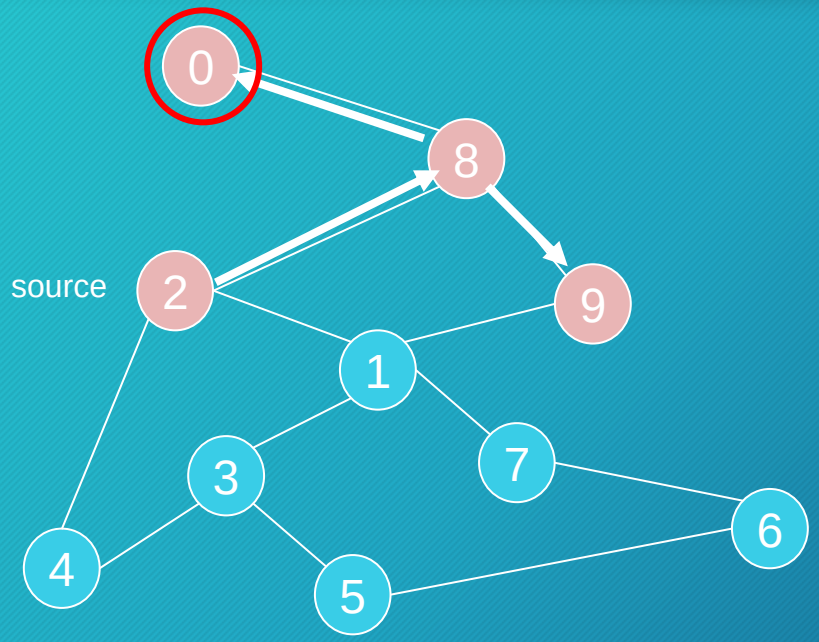
| | |
|---|---|
| 0 | T |
| 1 | F |
| 2 | T |
| 3 | F |
| 4 | F |
| 5 | F |
| 6 | F |
| 7 | F |
| 8 | T |
| 9 | F |

| Pred |
|---|
| 8 |
| - |
| - |
| - |
| - |
| - |
| - |
| - |
| 2 |
| - |

Recursive calls

RDFS( 2 )
RDFS(8)
Now visit 9 -> RDFS(9)

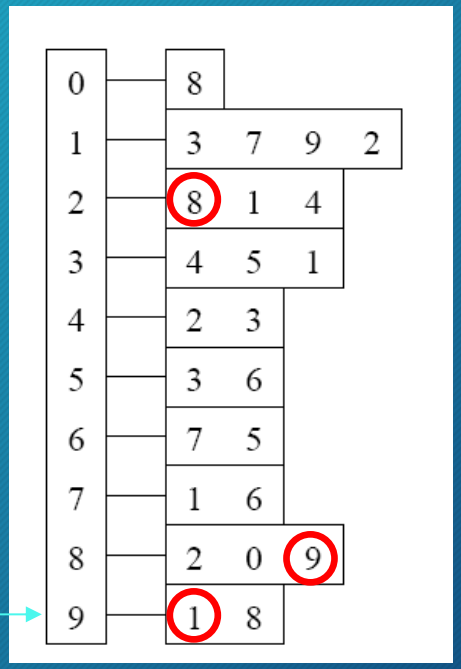
# Example: DFS on Graph (Contd…)




Recursive calls

RDFS( 2 )
RDFS(8)
RDFS(9)
-> visit 1, RDFS(1)

Adjacency List




Visited Table (T/F)

| | |
|---|---|
| 0 | T |
| 1 | F |
| 2 | T |
| 3 | F |
| 4 | F |
| 5 | F |
| 6 | F |
| 7 | F |
| 8 | T |
| 9 | T |

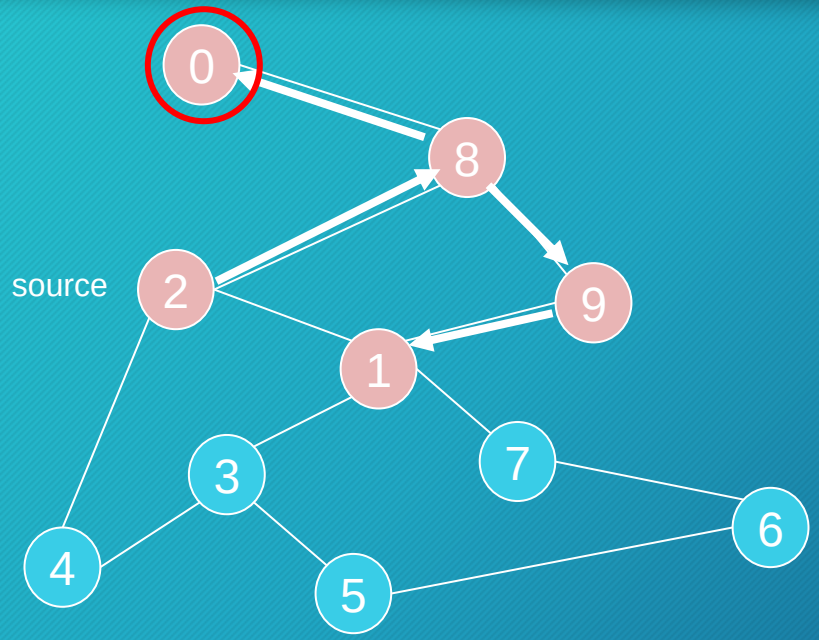
| |
|---|
| 8 |
| - |
| - |
| - |
| - |
| - |
| - |
| - |
| 2 |
| 8 |

*Pred*

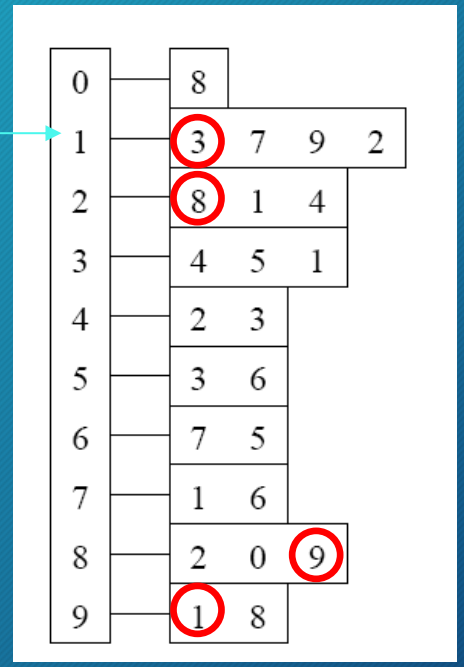
Mark 9 as visited

Mark Pred[9]

# Example: DFS on Graph (Contd…)




Adjacency List




Visited Table (T/F)

| | |
|---|---|
| 0 | T |
| 1 | T |
| 2 | T |
| 3 | F |
| 4 | F |
| 5 | F |
| 6 | F |
| 7 | F |
| 8 | T |
| 9 | T |

| |
|---|
| 8 |
| 9 |
| - |
| - |
| - |
| - |
| - |
| - |
| 2 |
| 8 |

*Pred*

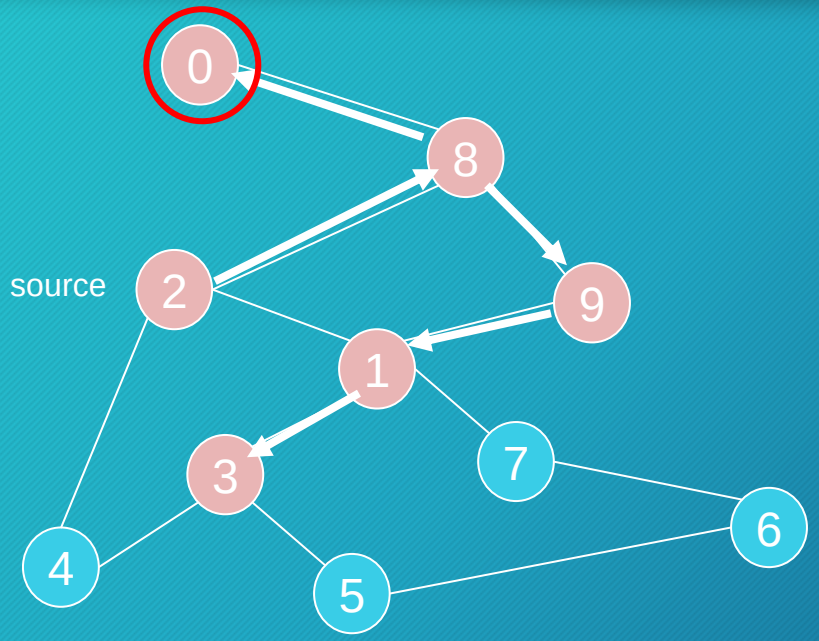
Recursive calls

RDFS( 2 )
RDFS(8)
RDFS(9)
RDFS(1)
visit RDFS(3)

Mark 1 as visited

Mark Pred[1]

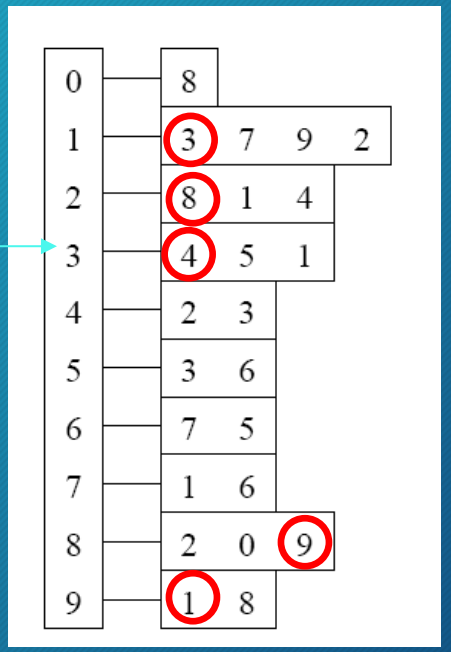
# Example: DFS on Graph (Contd…)




Recursive calls

RDFS( 2 )
RDFS(8)
RDFS(9)
RDFS(1)
RDFS(3)
visit RDFS(4)

Adjacency List




Visited Table (T/F)

| | |
|---|---|
| 0 | T |
| 1 | T |
| 2 | T |
| 3 | T |
| 4 | F |
| 5 | F |
| 6 | F |
| 7 | F |
| 8 | T |
| 9 | T |

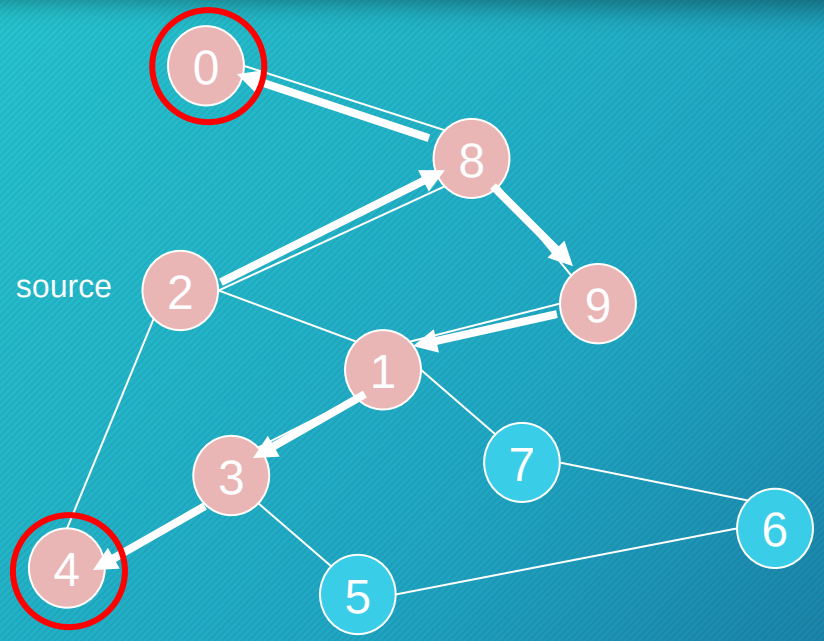
| |
|---|
| 8 |
| 9 |
| - |
| 1 |
| - |
| - |
| - |
| - |
| 2 |
| 8 |

*Pred*

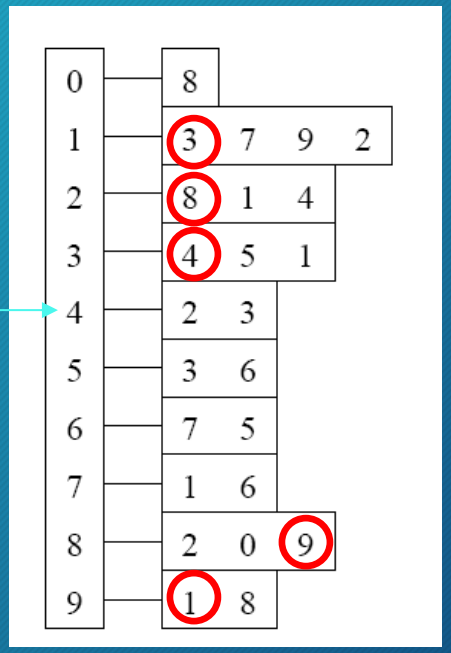
Mark 3 as visited

Mark Pred[3]

# Example: DFS on Graph (Contd…)




Adjacency List




Visited Table (T/F)

| | |
|---|---|
| 0 | T |
| 1 | T |
| 2 | T |
| 3 | T |
| 4 | T |
| 5 | F |
| 6 | F |
| 7 | F |
| 8 | T |
| 9 | T |

| |
|---|
| 8 |
| 9 |
| - |
| 1 |
| 3 |
| - |
| - |
| - |
| 2 |
| 8 |

*Pred*

Recursive calls

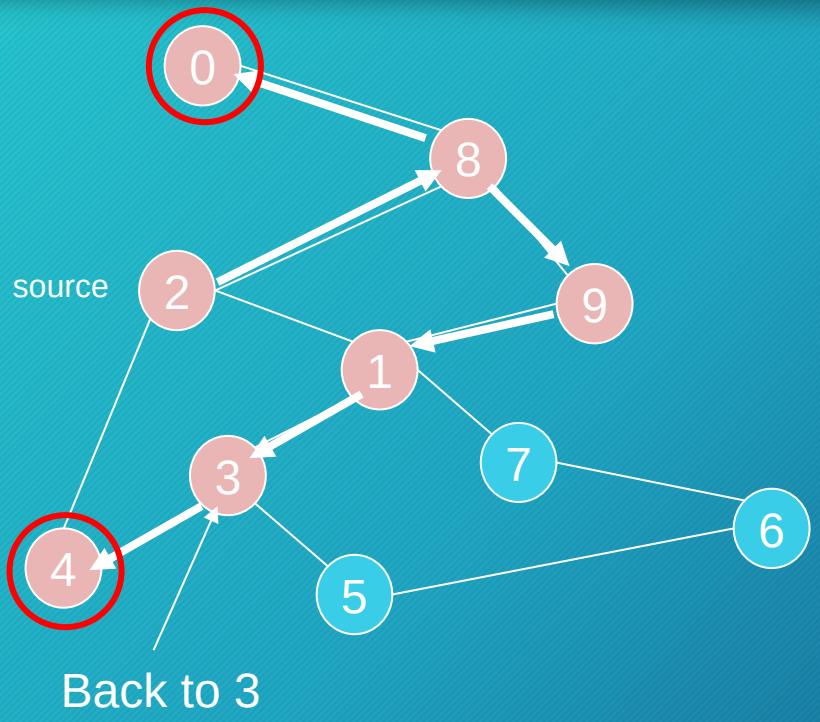
RDFS( 2 )
RDFS(8)
RDFS(9)
RDFS(1)
RDFS(3)

RDFS(4) → STOP all of 4's neighbors have been visited
return back to call RDFS(3)

Mark 4 as visited

Mark Pred[4]

# Example: DFS on Graph (Contd…)

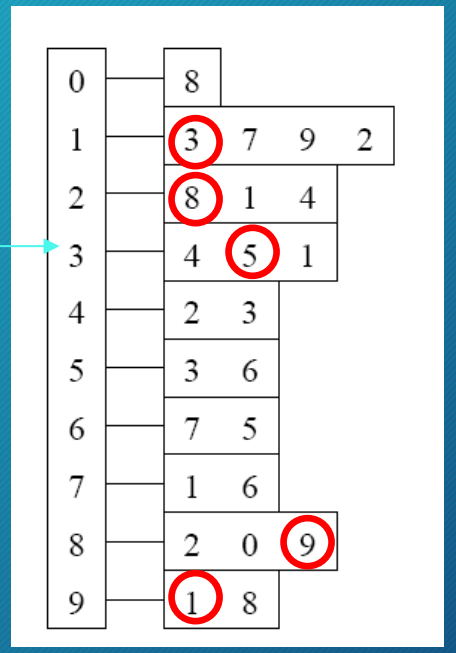



Recursive calls

RDFS( 2 )
RDFS(8)
RDFS(9)
RDFS(1)
RDFS(3)

visit 5 -> RDFS(5)

Adjacency List



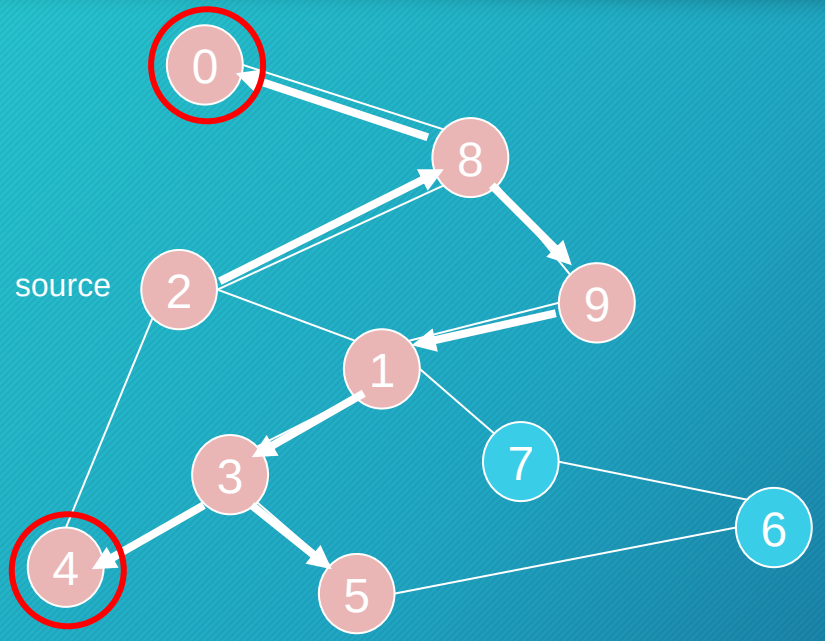

Visited Table (T/F)

| | |
|---|---|
| 0 | T |
| 1 | T |
| 2 | T |
| 3 | T |
| 4 | T |
| 5 | F |
| 6 | F |
| 7 | F |
| 8 | T |
| 9 | T |

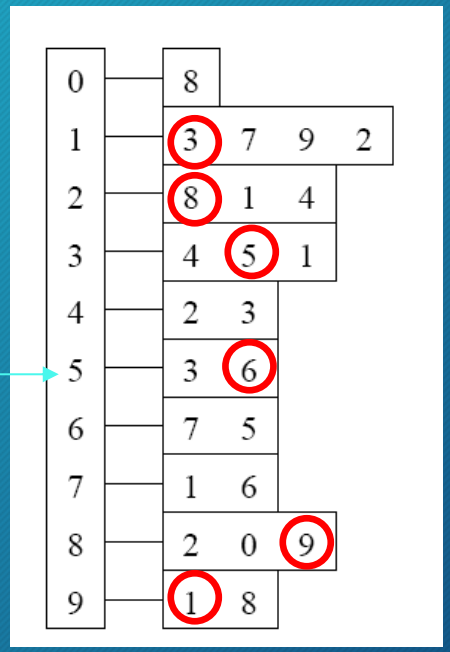
| |
|---|
| 8 |
| 9 |
| - |
| 1 |
| 3 |
| - |
| - |
| - |
| 2 |
| 8 |

*Pred*

# Example: DFS on Graph (Contd…)




Adjacency List




Visited Table (T/F)

| | |
|---|---|
| 0 | T |
| 1 | T |
| 2 | T |
| 3 | T |
| 4 | T |
| 5 | T |
| 6 | F |
| 7 | F |
| 8 | T |
| 9 | T |

| |
|---|
| 8 |
| 9 |
| - |
| 1 |
| 3 |
| 3 |
| - |
| - |
| 2 |
| 8 |

*Pred*

Recursive calls

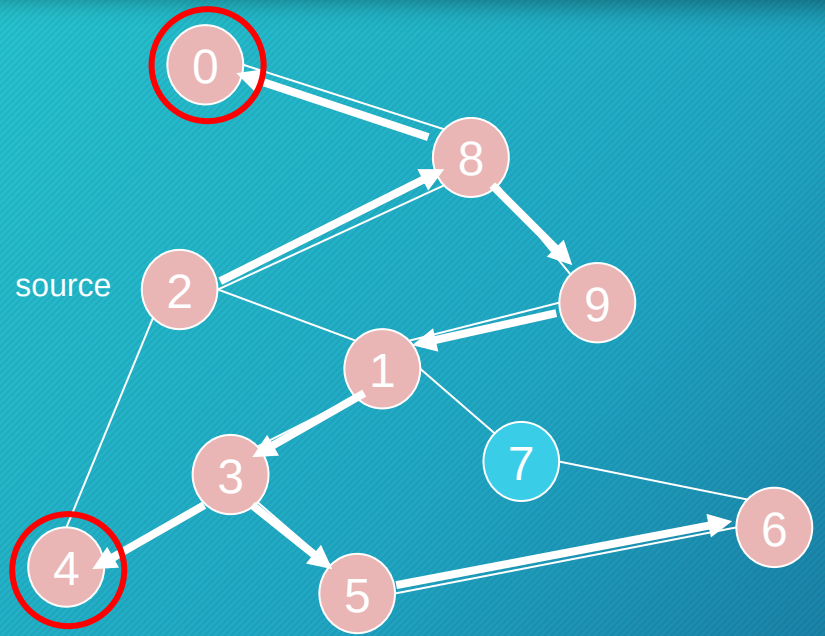
RDFS( 2 )
RDFS(8)
RDFS(9)
RDFS(1)
RDFS(3)

RDFS(5)
3 is already visited, so visit 6 -> RDFS(6)

Mark 5 as visited

Mark Pred[5]

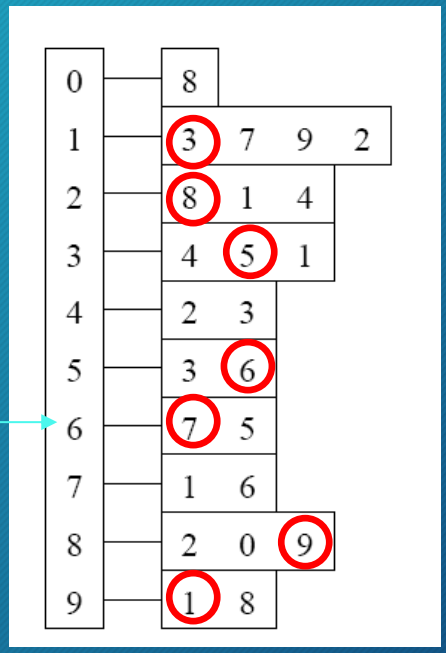
# Example: DFS on Graph (Contd…)




Recursive calls

RDFS( 2 )
RDFS(8)
RDFS(9)
RDFS(1)
RDFS(3)
RDFS(5)
RDFS(6)
visit 7 -> RDFS(7)

Adjacency List




Visited Table (T/F)

| | |
|---|---|
| 0 | T |
| 1 | T |
| 2 | T |
| 3 | T |
| 4 | T |
| 5 | T |
| 6 | T |
| 7 | F |
| 8 | T |
| 9 | T |

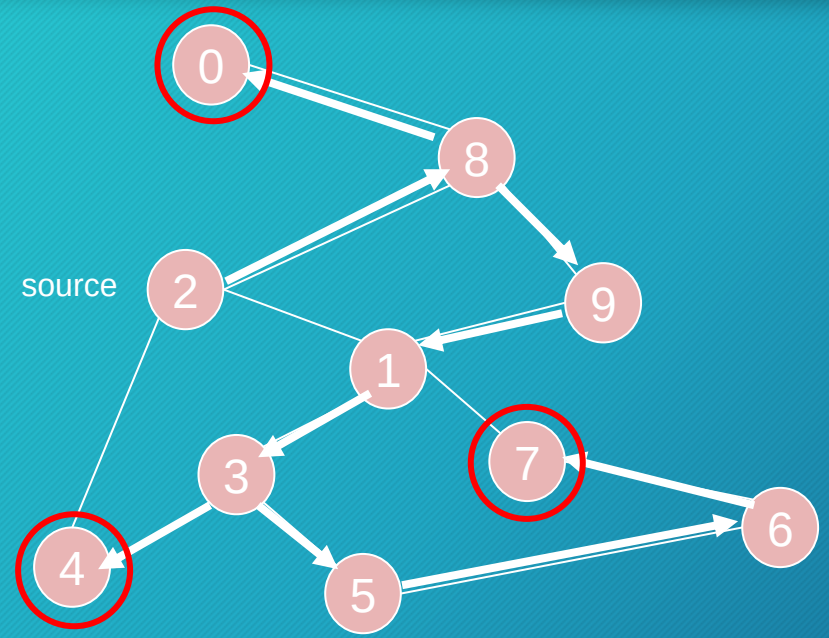
| |
|---|
| 8 |
| 9 |
| - |
| 1 |
| 3 |
| 3 |
| 5 |
| - |
| 2 |
| 8 |

*Pred*

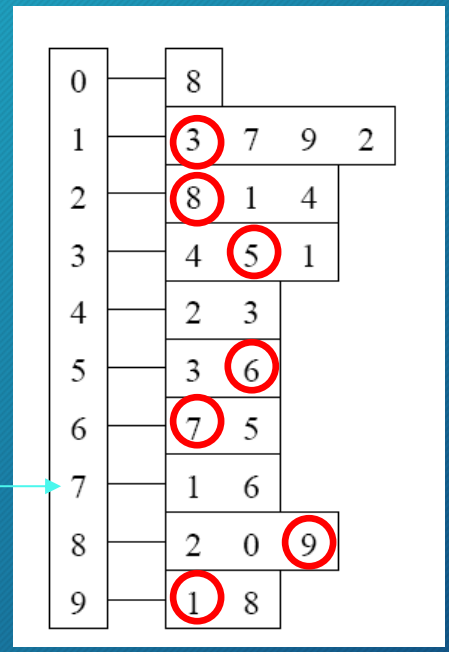
Mark 6 as visited

Mark Pred[6]

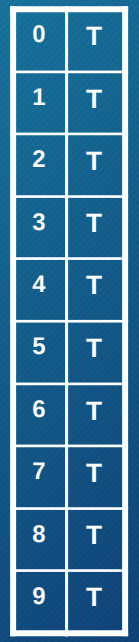
# Example: DFS on Graph (Contd…)




Adjacency List




Visited Table (T/F)





*Pred*

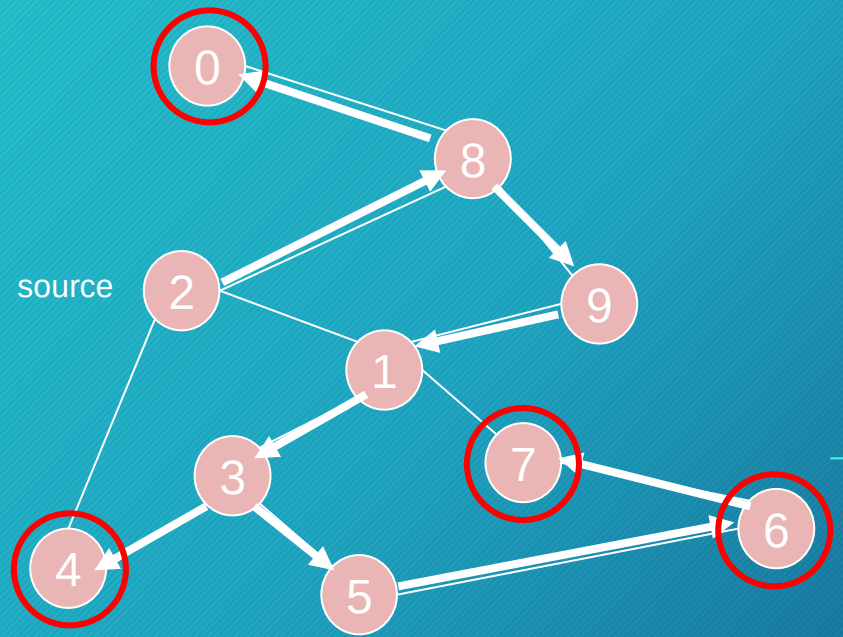
Recursive calls

RDFS( 2 )
RDFS(8)
RDFS(9)
RDFS(1)
RDFS(3)
RDFS(5)
RDFS(6)
RDFS(7) -> Stop no more unvisited neighbors

Mark 7 as visited

Mark Pred[7]

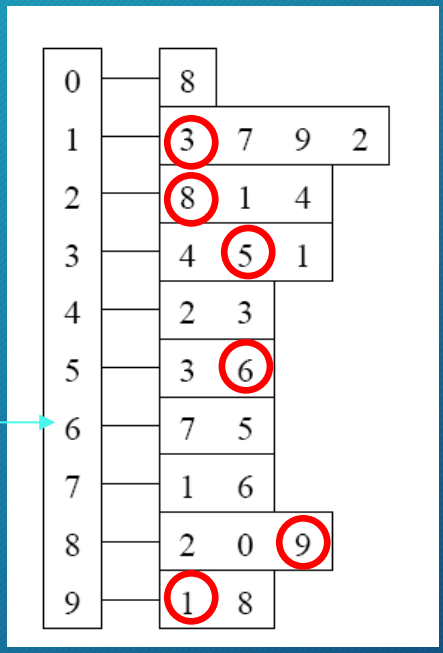
# Example: DFS on Graph (Contd…)




Recursive calls

RDFS( 2 )
RDFS(8)
RDFS(9)
RDFS(1)
RDFS(3)
RDFS(5)
RDFS(6) -> Stop

Adjacency List



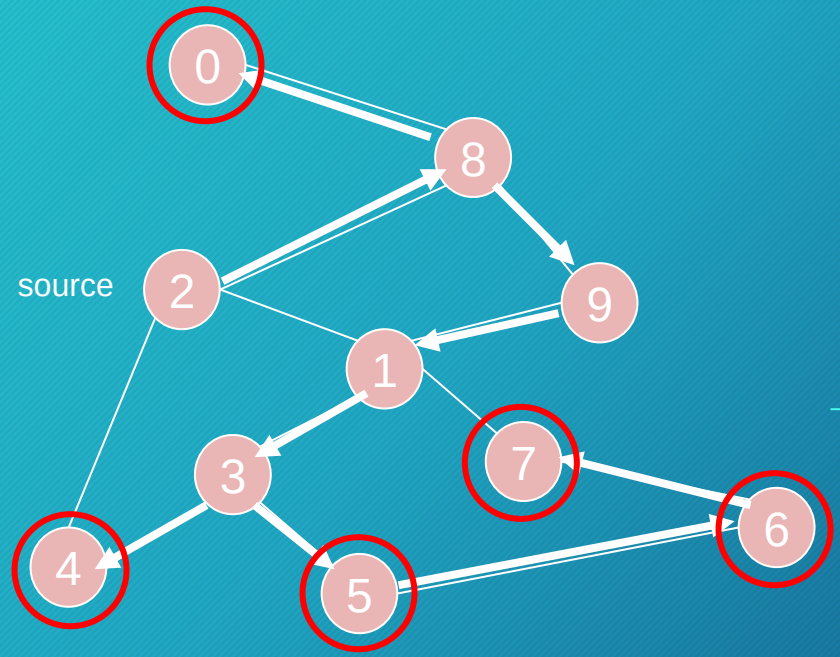

Visited Table (T/F)

| | |
|---|---|
| 0 | T |
| 1 | T |
| 2 | T |
| 3 | T |
| 4 | T |
| 5 | T |
| 6 | T |
| 7 | T |
| 8 | T |
| 9 | T |

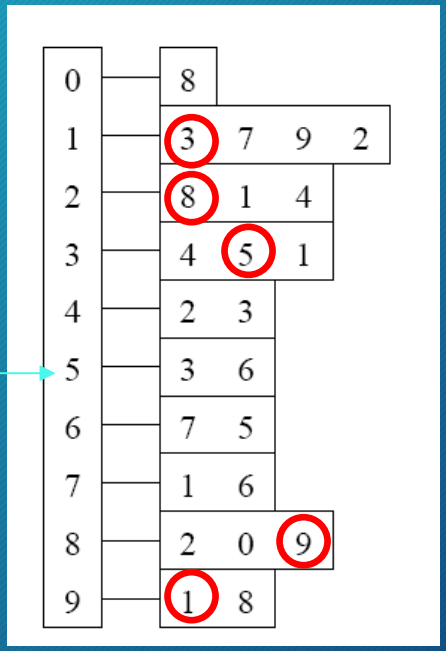
| |
|---|
| 8 |
| 9 |
| - |
| 1 |
| 3 |
| 3 |
| 5 |
| 6 |
| 2 |
| 8 |

*Pred*

# Example: DFS on Graph (Contd…)




Adjacency List




Visited Table (T/F)

| | |
|---|---|
| 0 | T |
| 1 | T |
| 2 | T |
| 3 | T |
| 4 | T |
| 5 | T |
| 6 | T |
| 7 | T |
| 8 | T |
| 9 | T |

| |
|---|
| 8 |
| 9 |
| - |
| 1 |
| 3 |
| 3 |
| 5 |
| 6 |
| 2 |
| 8 |

*Pred*

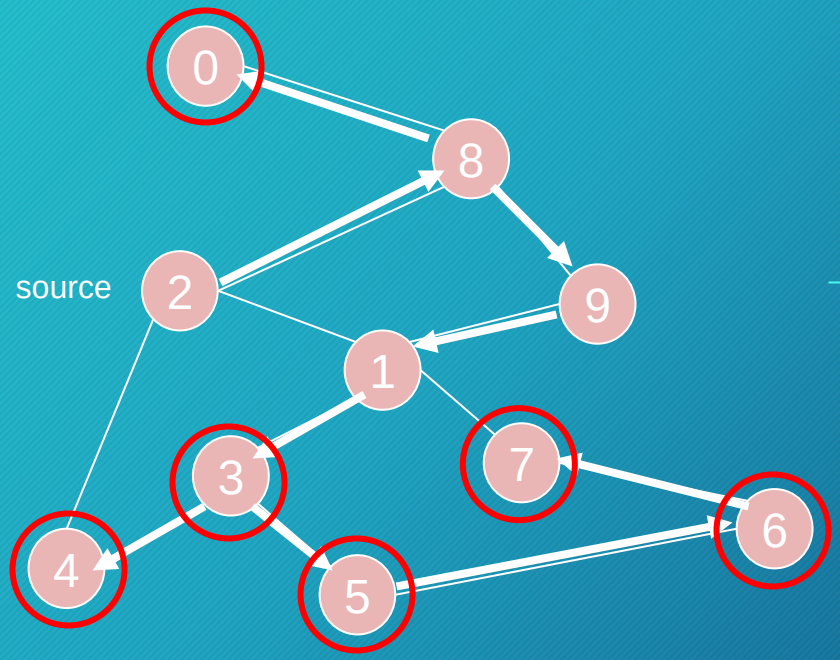
Recursive calls

RDFS( 2 )
RDFS(8)
RDFS(9)
RDFS(1)
RDFS(3)
RDFS(5) -> Stop

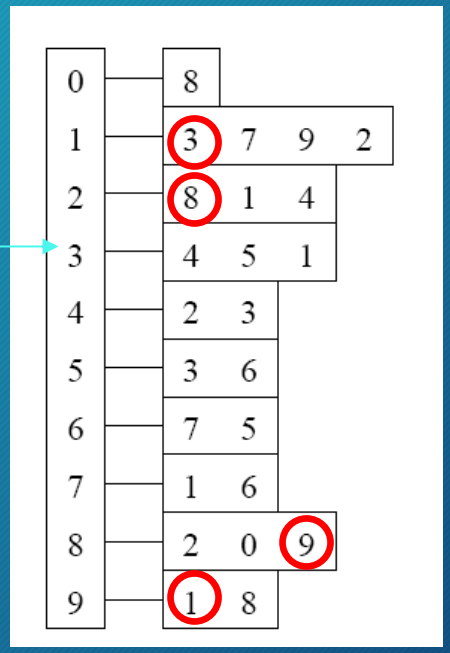
# Example: DFS on Graph (Contd…)

Recursive calls

RDFS( 2 )
RDFS(8)
RDFS(9)
RDFS(1)
RDFS(3) -> Stop




Adjacency List




Visited Table (T/F)

| | |
|---|---|
| 0 | T |
| 1 | T |
| 2 | T |
| 3 | T |
| 4 | T |
| 5 | T |
| 6 | T |
| 7 | T |
| 8 | T |
| 9 | T |

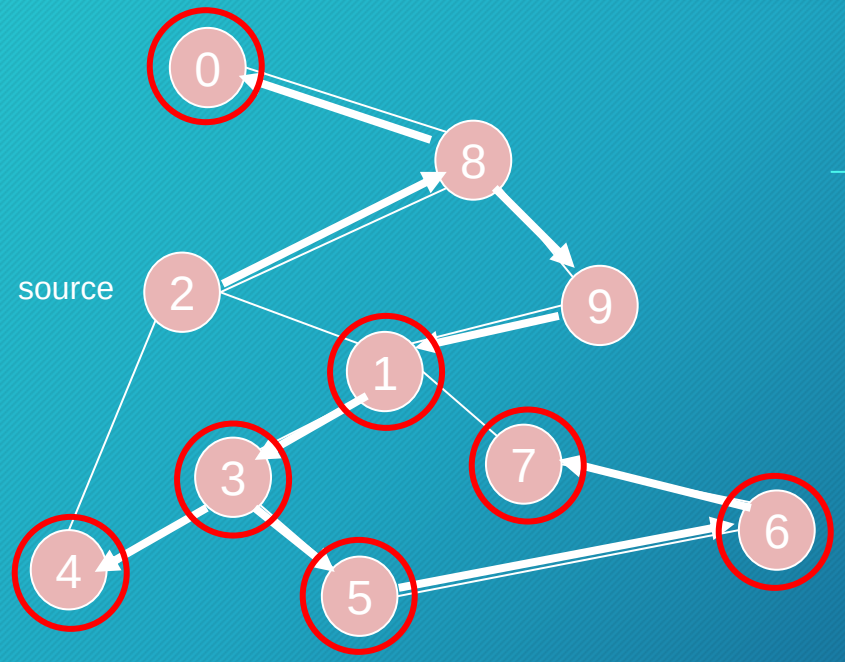
| |
|---|
| 8 |
| 9 |
| - |
| 1 |
| 3 |
| 3 |
| 5 |
| 6 |
| 2 |
| 8 |

*Pred*

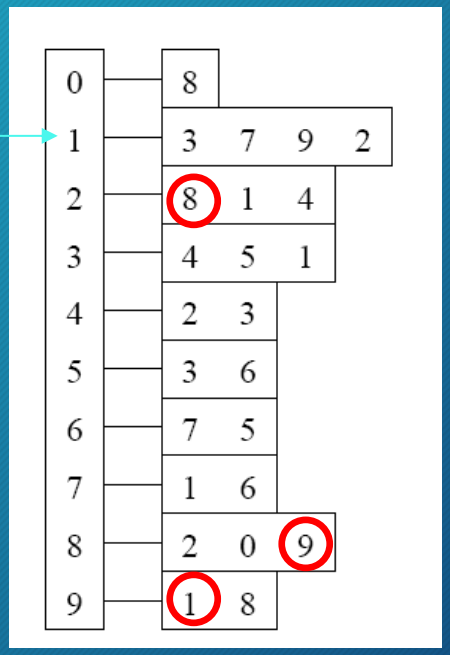
# Example: DFS on Graph (Contd…)

Recursive calls

RDFS( 2 )
RDFS(8)
RDFS(9)
RDFS(1) -> Stop




Adjacency List




Visited Table (T/F)

| | |
|---|---|
| 0 | T |
| 1 | T |
| 2 | T |
| 3 | T |
| 4 | T |
| 5 | T |
| 6 | T |
| 7 | T |
| 8 | T |
| 9 | T |

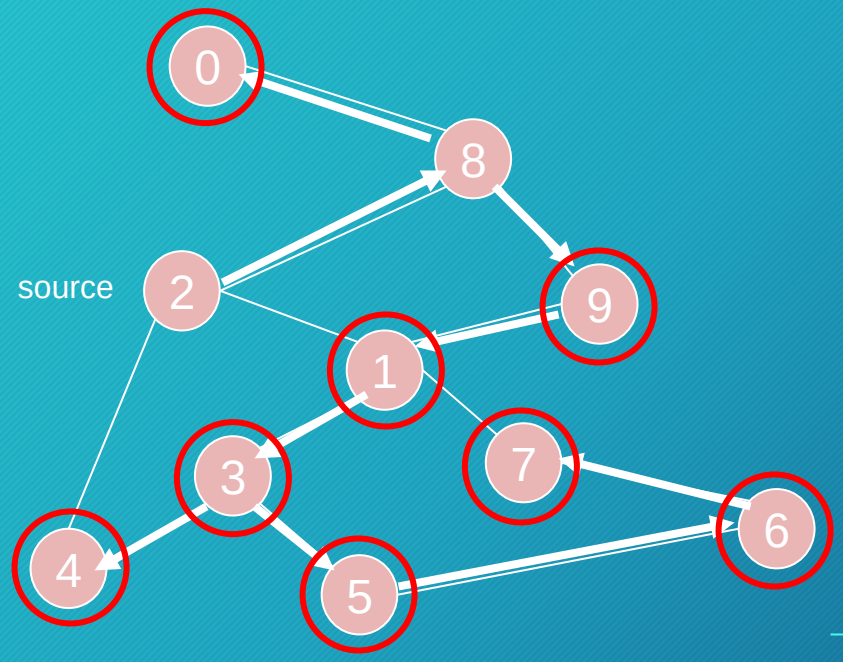
| |
|---|
| 8 |
| 9 |
| - |
| 1 |
| 3 |
| 3 |
| 5 |
| 6 |
| 2 |
| 8 |

*Pred*

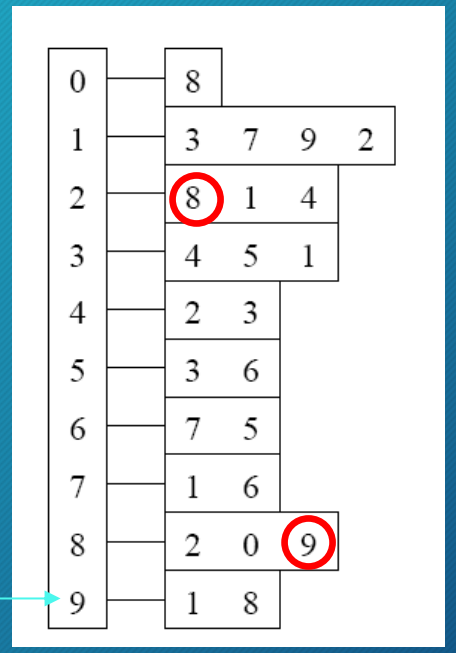
# Example: DFS on Graph (Contd…)

Recursive calls

RDFS( 2 )
RDFS(8)
RDFS(9) -> Stop




Adjacency List




Visited Table (T/F)



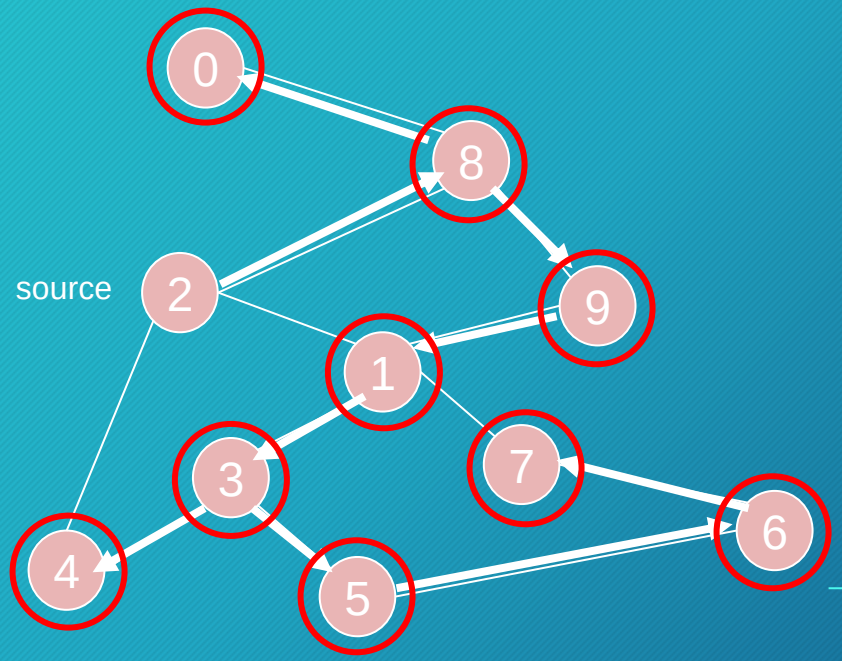
*Pred*

# Example: DFS on Graph (Contd…)

Recursive calls

RDFS( 2 )

RDFS(8) -> Stop




Adjacency List

| | |
|---|---|
| 0 | 8 |
| 1 | 3 7 9 2 |
| 2 | 8 1 4 |
| 3 | 4 5 1 |
| 4 | 2 3 |
| 5 | 3 6 |
| 6 | 7 5 |
| 7 | 1 6 |
| 8 | 2 0 9 |
| 9 | 1 8 |

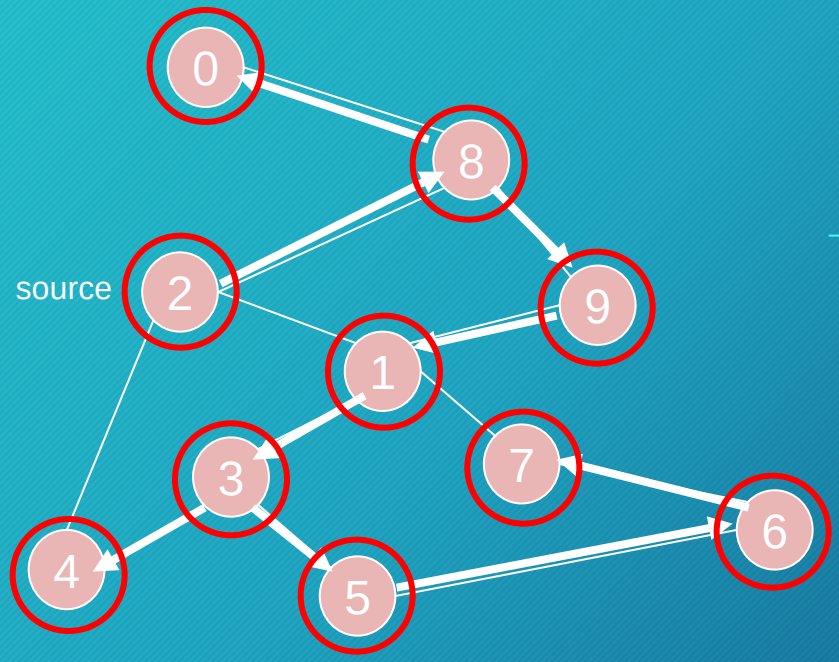
Visited Table (T/F)

| | |
|---|---|
| 0 | T |
| 1 | T |
| 2 | T |
| 3 | T |
| 4 | T |
| 5 | T |
| 6 | T |
| 7 | T |
| 8 | T |
| 9 | T |

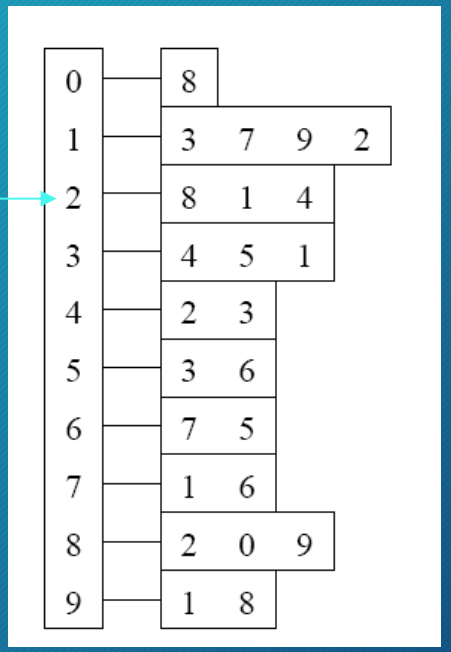
| |
|---|
| 8 |
| 9 |
| - |
| 1 |
| 3 |
| 3 |
| 5 |
| 6 |
| 2 |
| 8 |

*Pred*

# Example: DFS on Graph (Contd…)




Adjacency List




Visited Table (T/F)

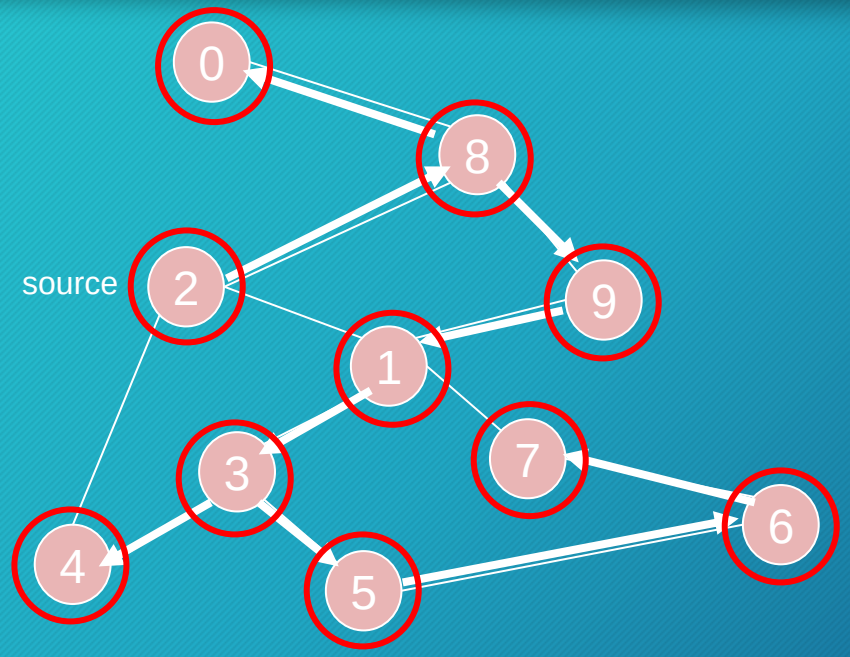
| | |
|---|---|
| 0 | T |
| 1 | T |
| 2 | T |
| 3 | T |
| 4 | T |
| 5 | T |
| 6 | T |
| 7 | T |
| 8 | T |
| 9 | T |

| *Pred* |
|---|
| 8 |
| 9 |
| - |
| 1 |
| 3 |
| 3 |
| 5 |
| 6 |
| 2 |
| 8 |

Recursive calls

RDFS( 2 ) -> Stop

# Example: DFS on Graph (Contd…)




Adjacency List

| | |
|---|---|
| 0 | 8 |
| 1 | 3 7 9 2 |
| 2 | 8 1 4 |
| 3 | 4 5 1 |
| 4 | 2 3 |
| 5 | 3 6 |
| 6 | 7 5 |
| 7 | 1 6 |
| 8 | 2 0 9 |
| 9 | 1 8 |

Visited Table (T/F)

| | |
|---|---|
| 0 | T |
| 1 | T |
| 2 | T |
| 3 | T |
| 4 | T |
| 5 | T |
| 6 | T |
| 7 | T |
| 8 | T |
| 9 | T |

| Pred |
|---|
| 8 |
| 9 |
| - |
| 1 |
| 3 |
| 3 |
| 5 |
| 6 |
| 2 |
| 8 |

Check our paths, does DFS find valid paths?
Yes.

**Algorithm** *Path*$(w)$

```
1.  if pred[w] ≠ −1
2.     then
3.           Path(pred[w]);
4.  output w
```

Try some examples.
Path(0) ->
Path(6) ->
Path(7) ->

# Depth Limited Search (DLS)

- The problem with DFS is that the  search can go down an infinite branch and thus never return.
- Depth limited search avoids this problem by imposing a depth limit (l) which effectively terminates the search at that depth. That is, nodes at depth l are treated as if they have no successors.
- The choice of depth parameter l is an important factor.
- If l is too deep , it is wasteful in terms of both time and space.
- But if **l < d** (the depth at which solution exists) i.e., the shallowest goal is beyond the depth limit, then this algorithm will **never reach a goal state**.

# DLS Function

Depth-first search with depth limit *l*, i.e., nodes at depth *l* have no successors

Recursive implementation:

```
function DEPTH-LIMITED-SEARCH(problem, limit) returns soln/fail/cutoff
    RECURSIVE-DLS(MAKE-NODE(INITIAL-STATE[problem]), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns soln/fail/cutoff
    cutoff-occurred? ← false
    if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
    else if DEPTH[node] = limit then return cutoff
    else for each successor in EXPAND(node, problem) do
        result ← RECURSIVE-DLS(successor, problem, limit)
        if result = cutoff then cutoff-occurred? ← true
        else if result ≠ failure then return result
    if cutoff-occurred? then return cutoff else return failure
```

# Properties of DLS

- Complete? If l <= d , then it is.
- Time? $O(b^l)$.
- Space? $O(bl)$, i.e., linear space.
- Optimal? No.

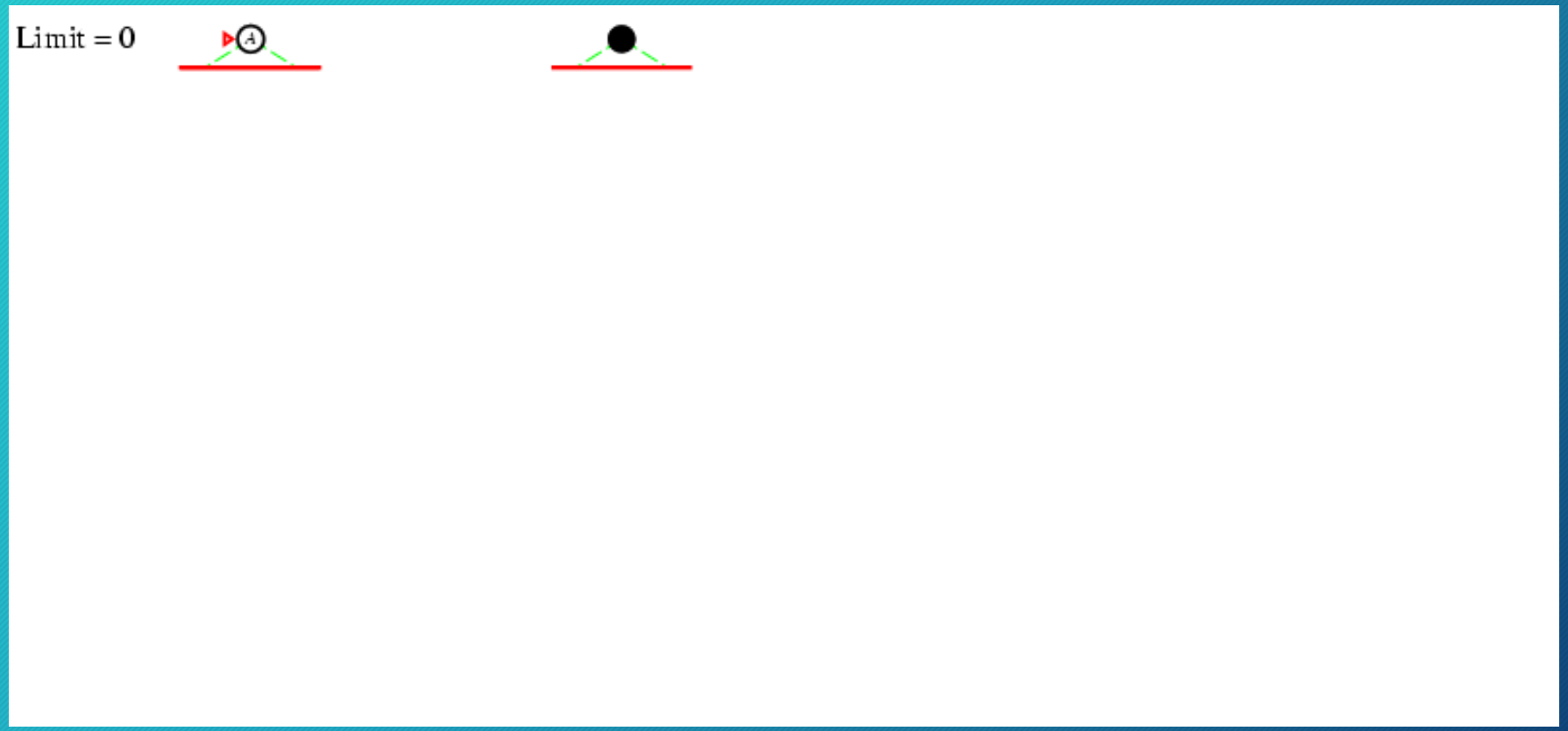
# Iterative Deepening Search (IDS)

- The problem with depth-limited search is deciding on a suitable depth parameter, which is not always easy.
- To overcome this problem there is another search called iterative deepening search.
- This search method simply tries all possible depth limits; first 0, then 1, then 2 etc. until a solution is found.
- Iterative deepening combines the benefits of DFS and BFS.
- It may appear wasteful as it is expanding nodes multiple times.
- But the overhead is small in comparison to the growth of an exponential search tree.
- It also named as Iterative deepening depth-first or Depth-First Iterative Deepening (DFID)

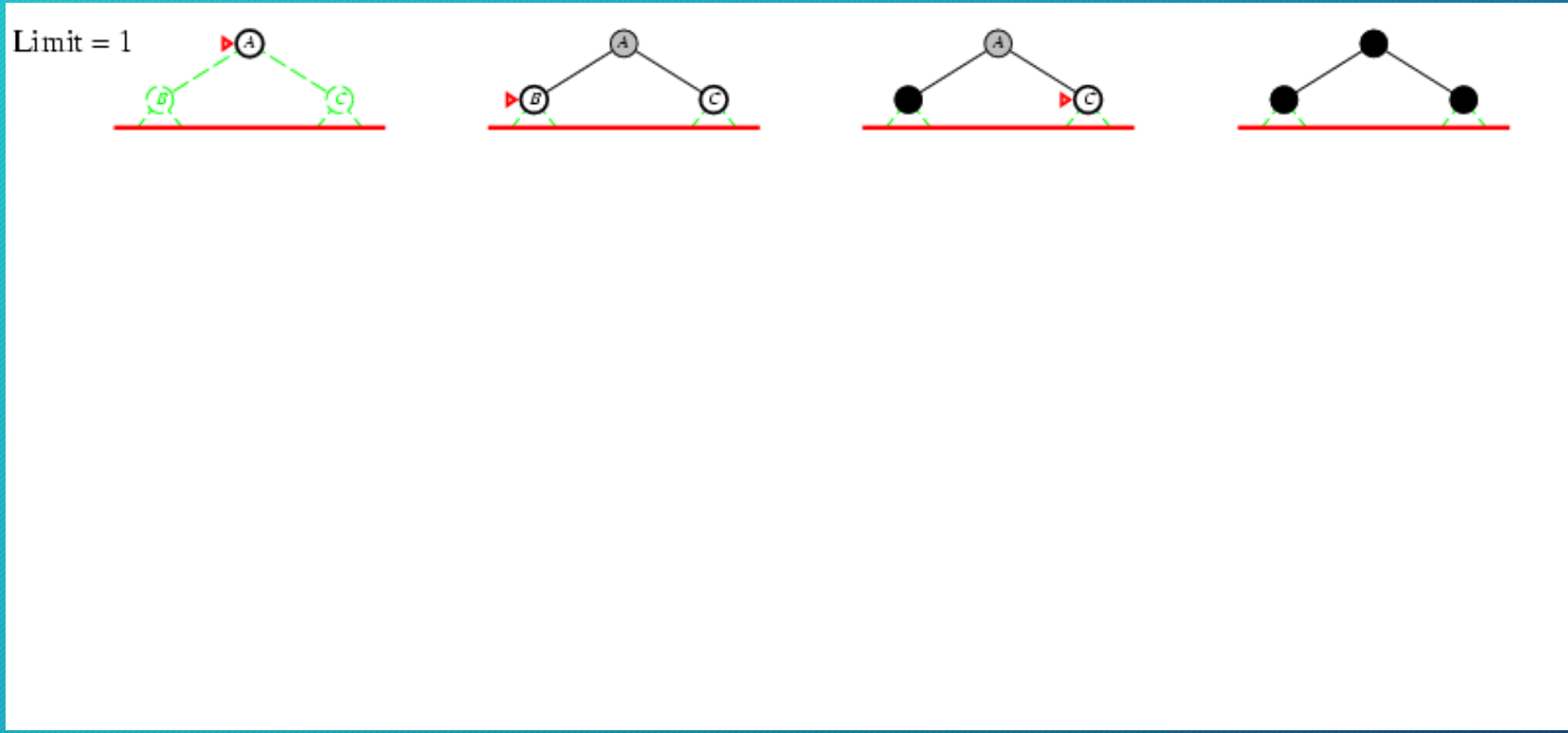
# IDS Function

```
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
```

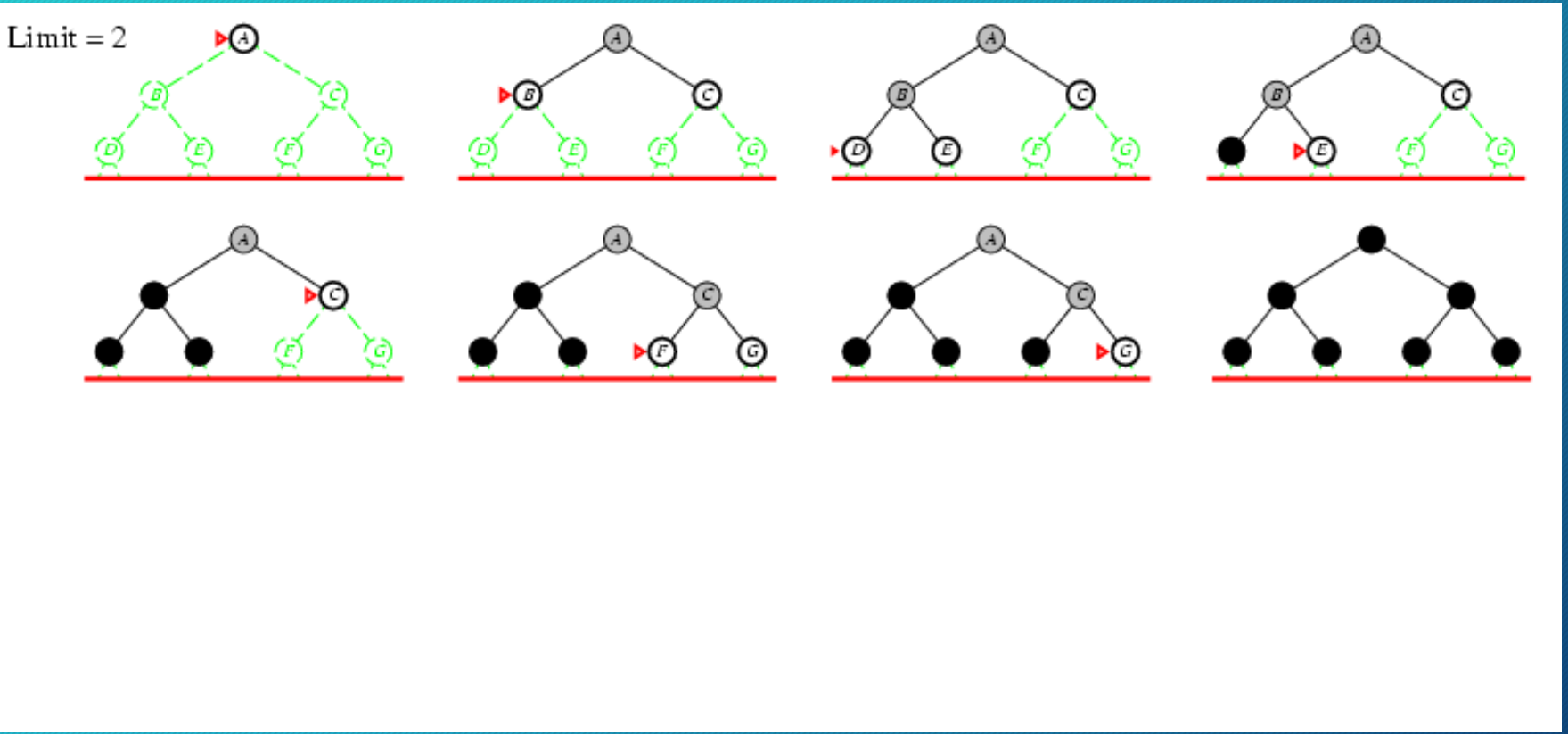
# IDS on Tree (at l= 0)



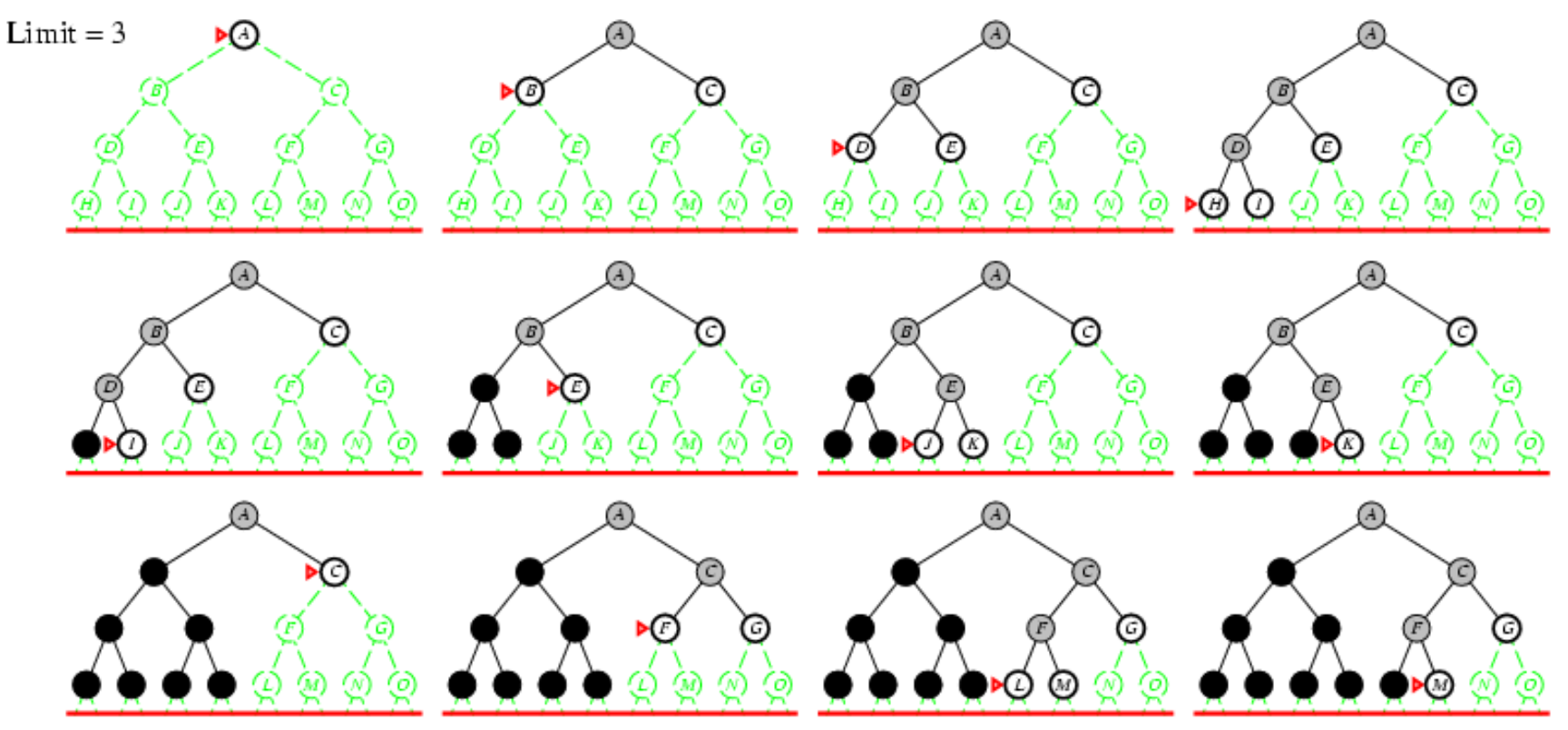
# IDS on Tree (at l= 1)



# IDS on Tree (at l= 2)



# IDS on Tree (at l= 3)



# Number of Nodes: BFS vs IDS

- Number of nodes generated in a breadth first search to depth *d* with branching factor *b*:

$$N_{BFS} = b^0 + b^1 + b^2 + \ldots + b^{d-2} + b^{d-1} + b^d$$

- Number of nodes generated in an iterative deepening search to depth *d* with branching factor *b*:

$$N_{IDS} = (d+1)b^0 + d\,b^1 + (d-1)b^2 + \ldots + 3b^{d-2} + 2b^{d-1} + 1b^d$$

as the nodes at the bottom level d are expanded once, the nodes at (d-1) are expanded twice, those at (d–3) are expanded three times and so on back to the root.

# Number of Nodes: BFS vs IDS Example

- For $b = 10$, $d = 5$,
  - $N_{BFS} = 1 + 10 + 100 + 1{,}000 + 10{,}000 + 100{,}000 = 111{,}111$
  - $N_{IDS} = 6 + 50 + 400 + 3{,}000 + 20{,}000 + 100{,}000 = 123{,}456$

- Overhead = (123,456 - 111,111)/111,111 = 11%

- We can see that compared to the overall number of expansions , the total is not substantially increased.

# Properties of IDS

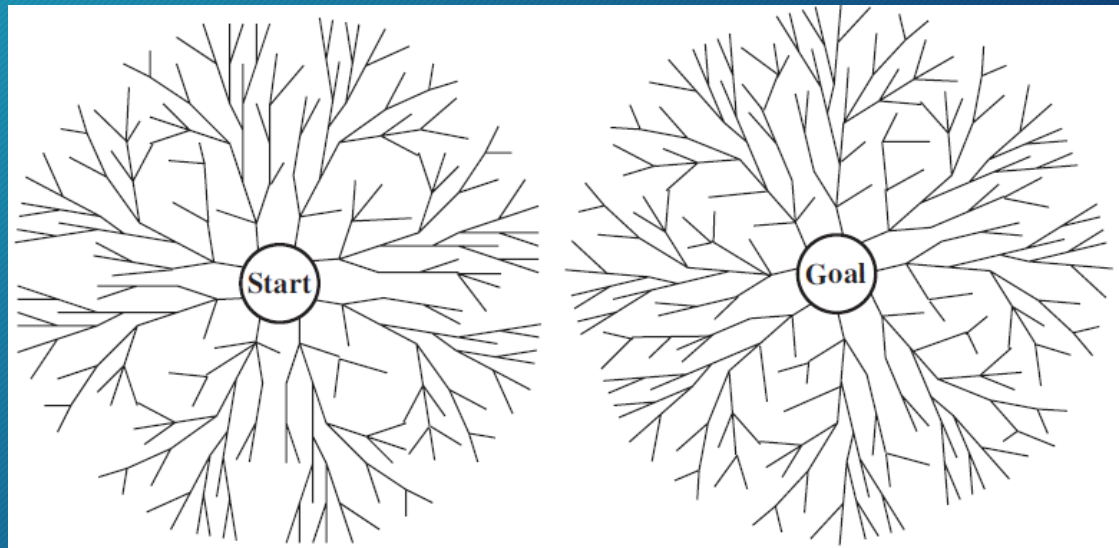
- Complete? **Yes** , like BFS it is complete when b is finite.
- Time? $(d+1)b^0 + d\,b^1 + (d-1)b^2 + \ldots + b^d = \boldsymbol{O(b^d)}$
- Space? **O(bd)** , like DFS memory requirement is linear.
- Optimal? **Yes**, if like BFS path cost is a non-decreasing function of the depth of the node, it is optimal.

In general iterative deepening is the preferred uninformed search method when there is a large search space and the depth of the solution is not known.

# Bi-Directional Search (BDS)

- Suppose that the search problem is such that the arcs are bidirectional. That is, if there is an operator that maps from state A to state B, there is another operator that maps from state B to state A. Many search problems have reversible arcs. 8-puzzle, 15-puzzle, path planning etc. are examples of search problems. However there are other state space search formulations which do not have this property. But if the arcs are reversible, you can see that instead of starting from the start state and searching for the goal, one may start from a goal state and try reaching the start state. If there is a single state that satisfies the goal property, the search problems are identical.

- Requires an explicit Goal State
- Simultaneously search forward from initial state and backwards from the Goal State until the two meet
- Concatenate the path from initial state to inverse of the path from Goal State to form a complete solution
- Use 2 Queues to implement for the halves



# Complexities of BDS

- Time complexity of bidirectional search using breadth-first searches in both directions is $O(b^{d/2})$.
- Space complexity is also $O(b^{d/2})$. This is because search stops in the midway as soon as any one of the search paths from initial state (forward search) meets with any one of the search paths from goal state (backward search).

- For the similar condition, the time complexity of breadth first search from initial state to goal state is $O(b^{d})$. The space complexity is also $O(b^{d})$. It is because searching proceeds only in one direction i.e. from initial state to goal state (only forward search).

- The motivation is that $b^{d/2} + b^{d/2}$ is much less than $b^{d}$, or in the figure, the area of the two small circles is less than the area of one big circle centered on the start and reaching to the goal.

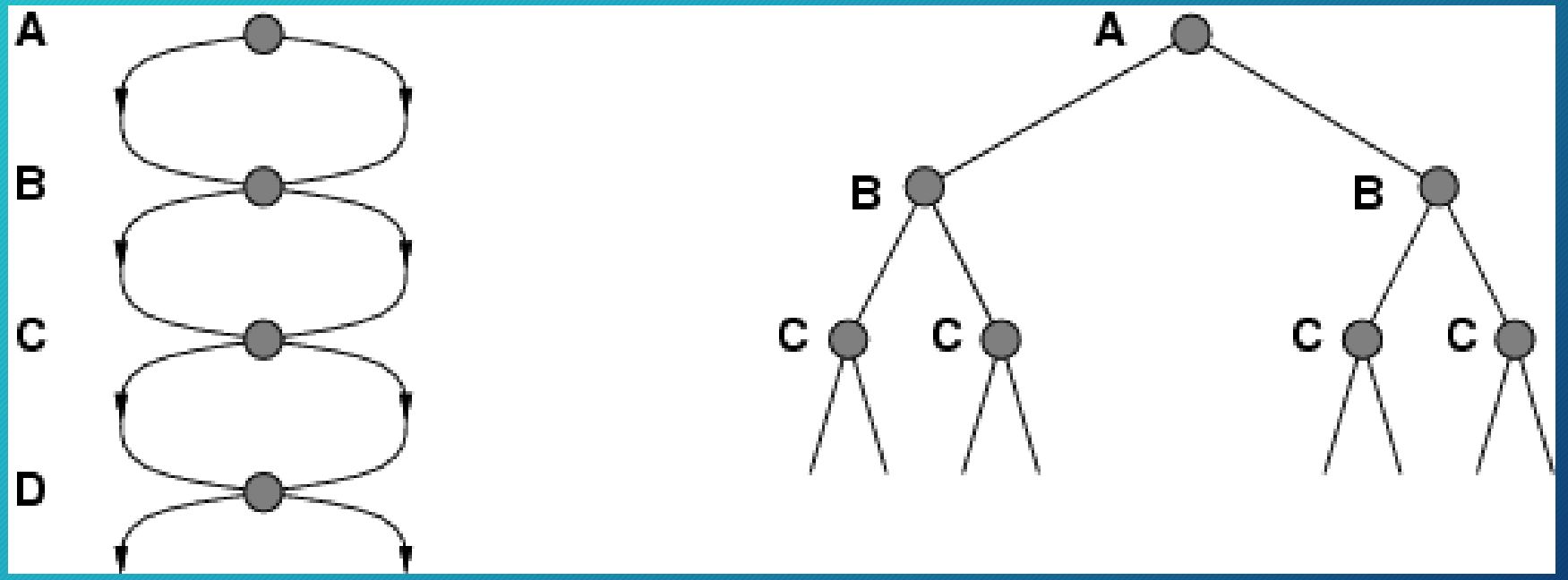
# Comparison among Uninformed Search Strategies

| 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$^{c,d}$ |

- ✓ 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 ≥ ϵ for positive ϵ; **c** optimal if step costs are all identical; **d** if both directions use breadth-first search.

# Repeated State

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



# Repeated State

- In most searching algorithm, the state space generates an exponentially larger search tree.
- This occurs mainly because of repeated nodes.
- If we can avoid the generation of repeated nodes we can limit the number of nodes that are created and stop the expansion of repeated nodes.

# Methods to Avoid Repeated State

- There are three methods having increasing order of computational overhead to control the generation of repeated nodes.

1. Don’t generate a node that is the same as the parent node.

2. Don’t create paths with cycles in them. To do this we can check each ancestor node and refuse to create a state that is the same as this set of nodes.

3. Don’t generate any state that is the same as any state generated before. This requires that every state is kept in memory. (space complexity is $O(b^d)$ )

# Summary

- Problem formulation usually requires abstracting away real-world details to define a state space that can feasibly be explored
- Variety of uninformed search strategies : BFS, UCS, DFS, DLS, IDS, BDS
- Iterative deepening search uses only linear space and not much more time than other uninformed algorithms

# Content

- Informed Search Strategies
  - Best First Search
  - A* search

# Informed Search Strategies

- In uninformed search methods, systematically explore the state space and find the goal. They are inefficient in most cases.
- Informed search methods use problem specific knowledge, and may be more efficient.
- The idea behind the heuristic search is that we explore the node that is most likely to be nearest to a goal state.
- At the heart of such algorithms there is the concept of a heuristic function.
- A heuristic function has some knowledge about the problem so that it can judge how close the current state is to the goal state.
- A heuristic function h(n) = estimated cost of the cheapest path from the state at node n to a goal state.

# Heuristic Function

- Heuristic functions are the most common form in which additional knowledge of the problem is imparted to search algorithm.

- For the present context, we consider these heuristic function to be **arbitrary, nonnegative, problem-specific functions**, with one constraint :

  If n is a goal state, then h(n) = 0.

- h(n) takes a node as input, unlike g(n) in UCS, it depends only on the state at that node.

- For example, in Romania, one might estimate the cost of the cheapest path from Arad to Bucharest via the straight-line distance from Arad to Bucharest.

# Best First Search

- Idea: use an evaluation function *f(n)* for each node
  - estimate of "desirability"
  - →Expand most desirable unexpanded node
- Evaluation function is constructed as a cost estimate, so the node with the *lowest* evaluation is expanded first.
- Implementation:
- Order the nodes in fringe in decreasing order of desirability
- Special cases:
  - Greedy best-first search
  - $A^*$ search

# Greedy Best First Search

- It tries to expand the node that is closest to the goal state.
- It follows a single path but switch over to a different path if it appears to be more promising at any stage.
- A promising node is selected by applying a suitable *Heuristic Function (HF)* to each competing node.

# Data Structures used for Best First Search

- OPEN
  - It is a priority queue of nodes which have been generated and have had the heuristic function applied to them but which have not yet been expanded. (priority is evaluated by a HF value).

- CLOSED
  - It contains nodes that have already been expanded. (it is used to ensure that the same node is not expanded more than once.)

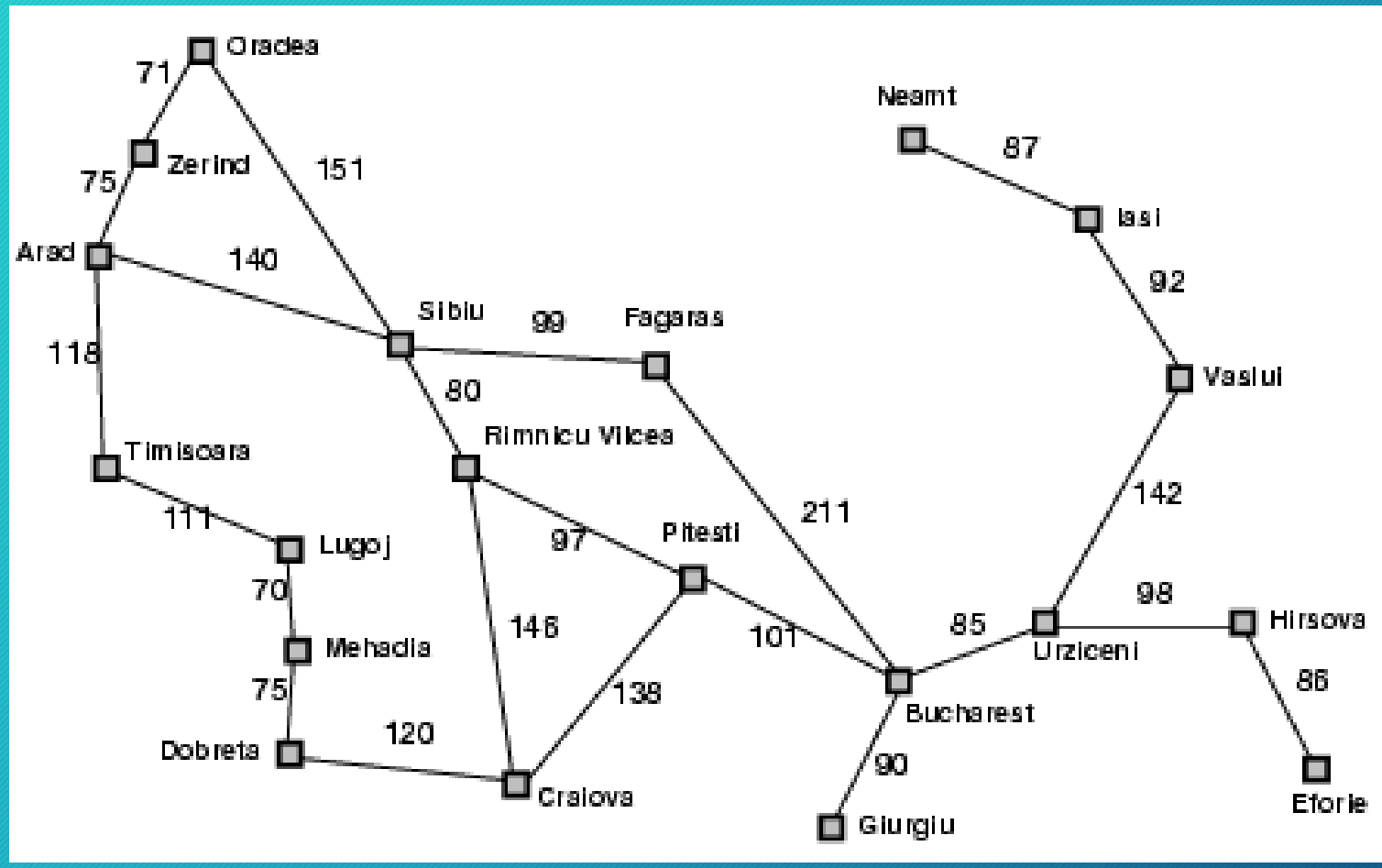
# Greedy Best First Search

- Evaluation function $f(n) = h(n)$ (heuristic)
  = estimate of cost from $n$ to *goal*

e.g., $h_{SLD}(n)$ = straight-line distance from $n$ to Bucharest

- Greedy Best First Search expands the node that appears to be closest to goal

# Romania with step costs in km



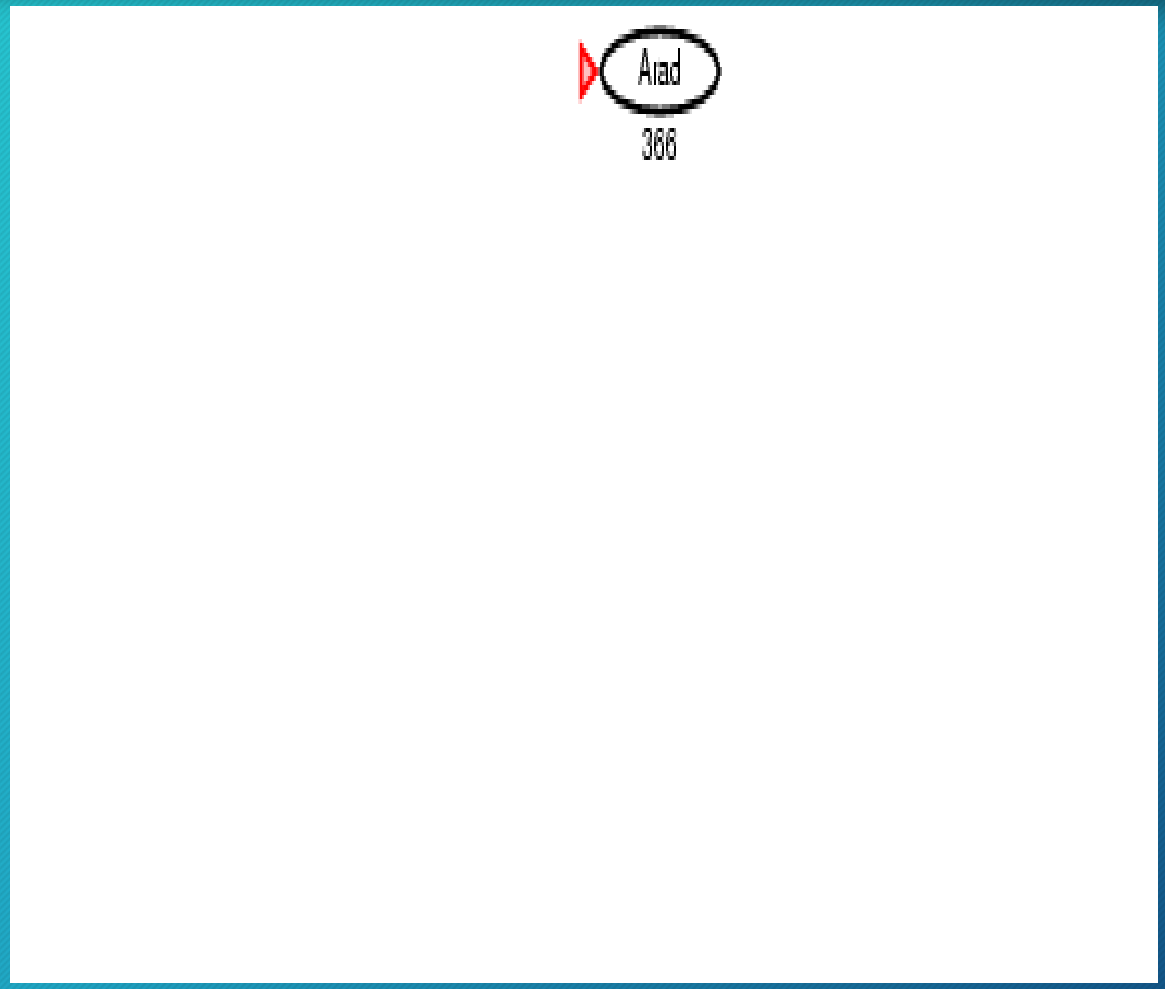

| Straight–line distance to Bucharest | |
|---|---|
| Arad | 366 |
| Bucharest | 0 |
| Craiova | 160 |
| Dobreta | 242 |
| Eforie | 161 |
| Fagaras | 176 |
| Giurgiu | 77 |
| Hirsova | 151 |
| Iasi | 226 |
| Lugoj | 244 |
| Mehadia | 241 |
| Neamt | 234 |
| Oradea | 380 |
| Pitesti | 10 |
| Rimnicu Vilcea | 193 |
| Sibiu | 253 |
| Timisoara | 329 |
| Urziceni | 80 |
| Vaslui | 199 |
| Zerind | 374 |

# Best First Search Algorithm

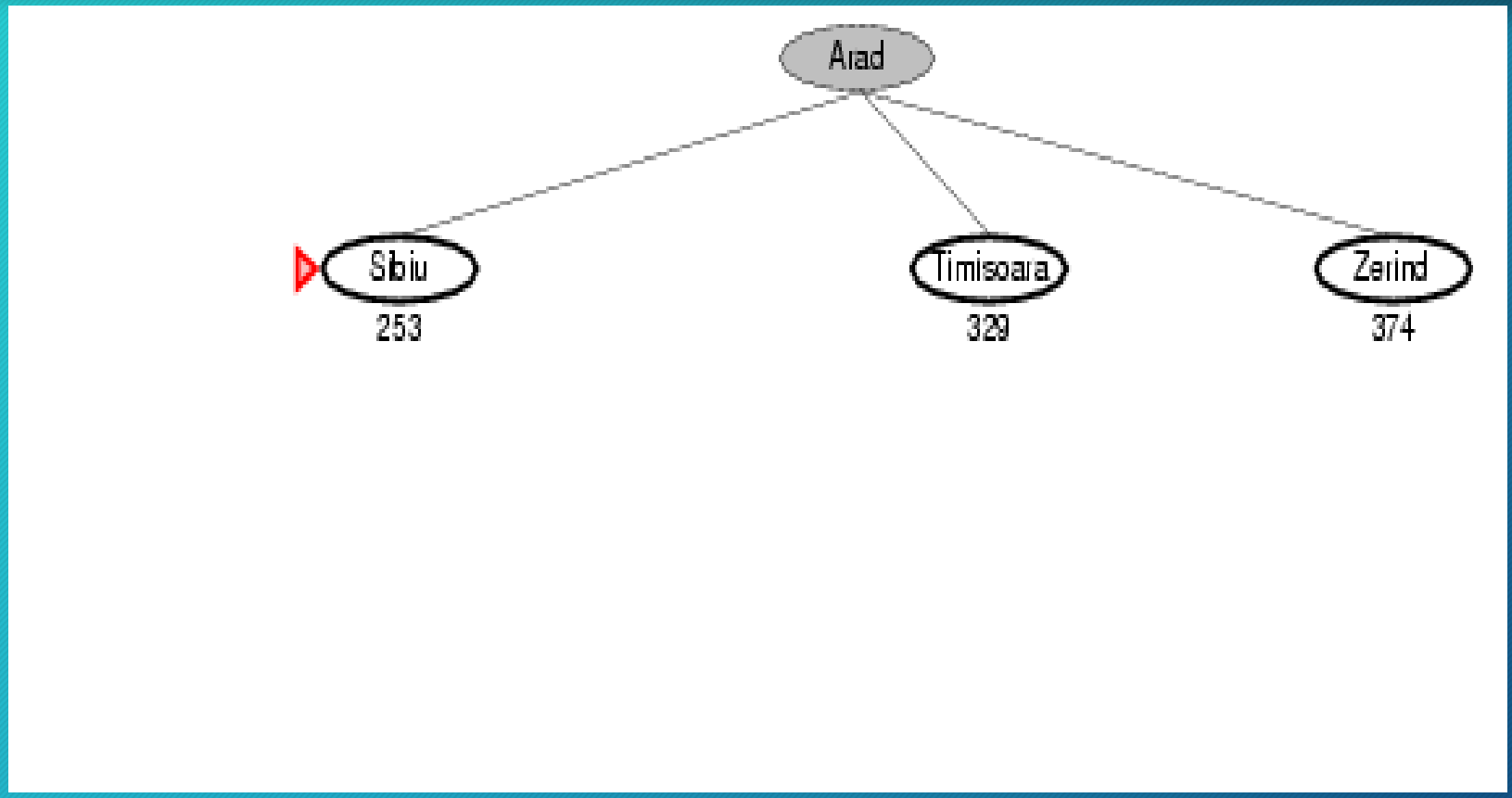
1. Start with OPEN containing just the initial state.
2. Until a goal is found  or there are no nodes left on OPEN do:
   a) Pick the best node from OPEN.
   b) Generate its successors.
   c) For each successor do:
      i) If it has not been generated before , evaluate it, add it to OPEN, and record its parent.

      ii) If it has been generated before, change the parent if this new path is better than the previous one. In that case , update the cost of getting to this node and to any successors that this node may already have.

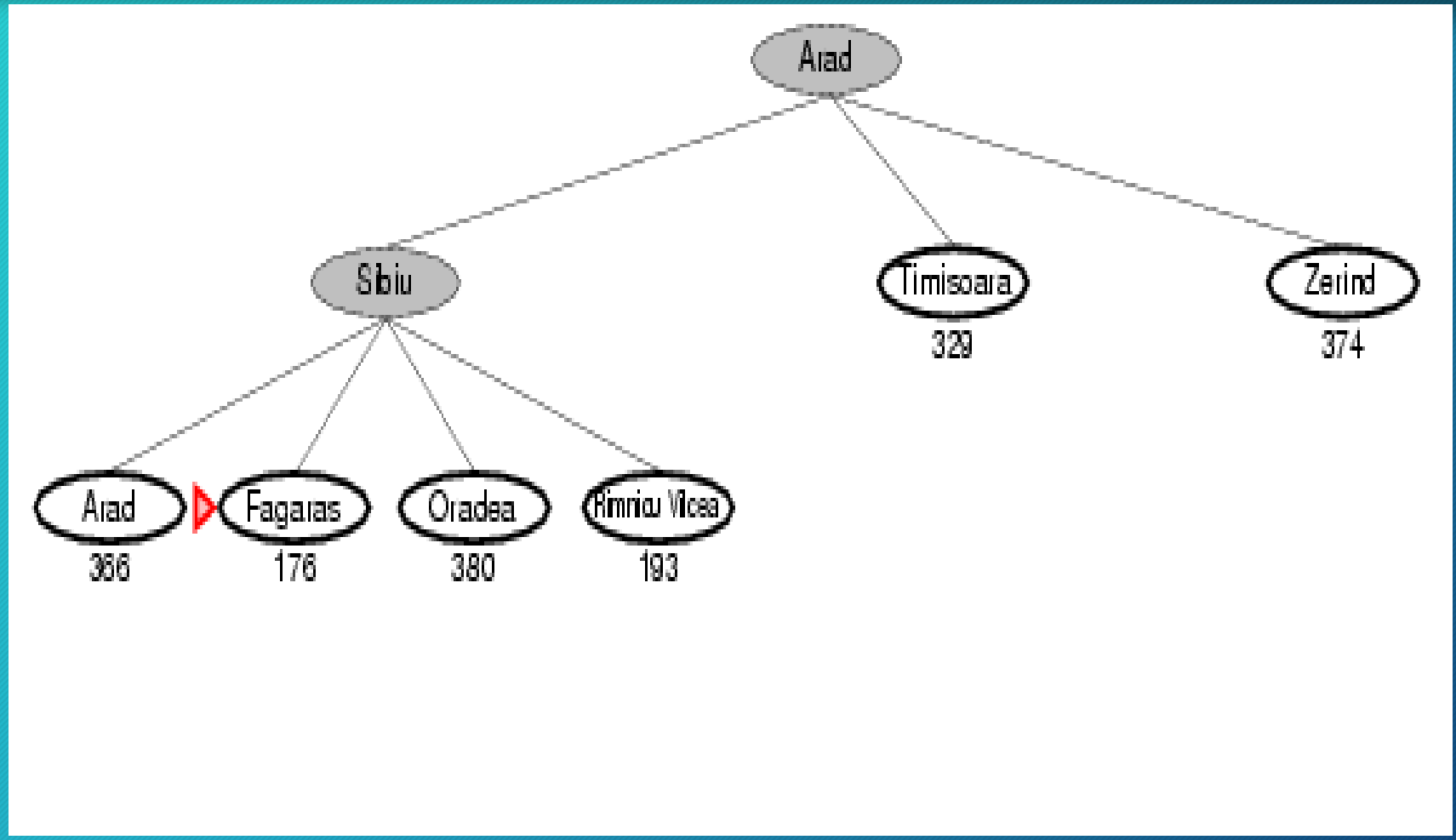
# Best First Search Example



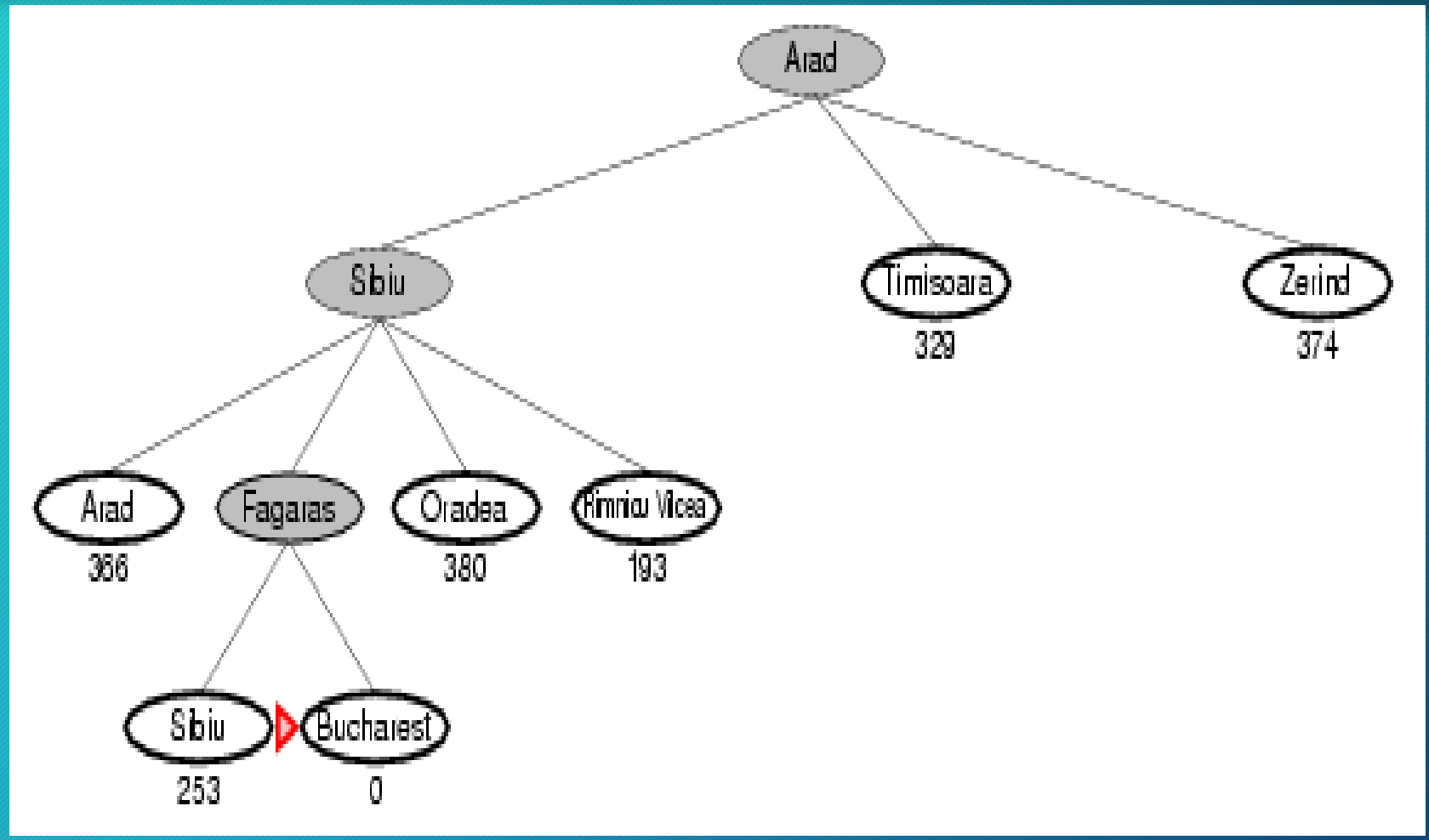
# Best First Search Example



# Best First Search Example



# Best First Search Example



# Properties of Greedy Best First Search

- Complete? No – can get stuck in loops, e.g., Iasi → Neamt → Iasi → Neamt → …
- Time? $O(b^m)$, but a good heuristic can give dramatic improvement
- Space? $O(b^m)$, keeps all nodes in memory
- Optimal? No

# A* Search

- **Idea:** avoid expanding paths that are already expensive
- Best First Search algorithm is a simplified version of A* algorithm .
- A* uses the same f, g ,h functions as well as the lists OPEN and CLOSED.

- Evaluation function $f(n) = g(n) + h(n)$ = *Actual Cost + Heuristic Value*

- $g(n)$ = cost so far to reach $n$
- $h(n)$ = estimated cost from $n$ to goal
- $f(n)$ = estimated total cost of path through $n$ to goal

# A* Search Algorithm

1. start with OPEN containing only the initial node.
   set that node's g value to zero, its h value to whatever it is and its f value to h + 0 = h.
   initially CLOSED is empty.
2. until a goal node is found, repeat  the procedure:
   if there are no nodes on OPEN, report failure.
   otherwise, pick the node from OPEN with the lowest f value.
   call it BESTNODE.
   remove it from OPEN , place it on CLOSED .
   if BESTNODE is a goal node, then exit and report a solution;
   else generate the successors of BESTNODE.
   for each such successor, do the following:

# A* Search Algorithm (contd…)

a) set SUCCESSOR to point back to BESTNODE.

b) compute g(SUCCESSOR) = g( BESTNODE) + the cost of getting from BESTNODE to SUCCESSOR.

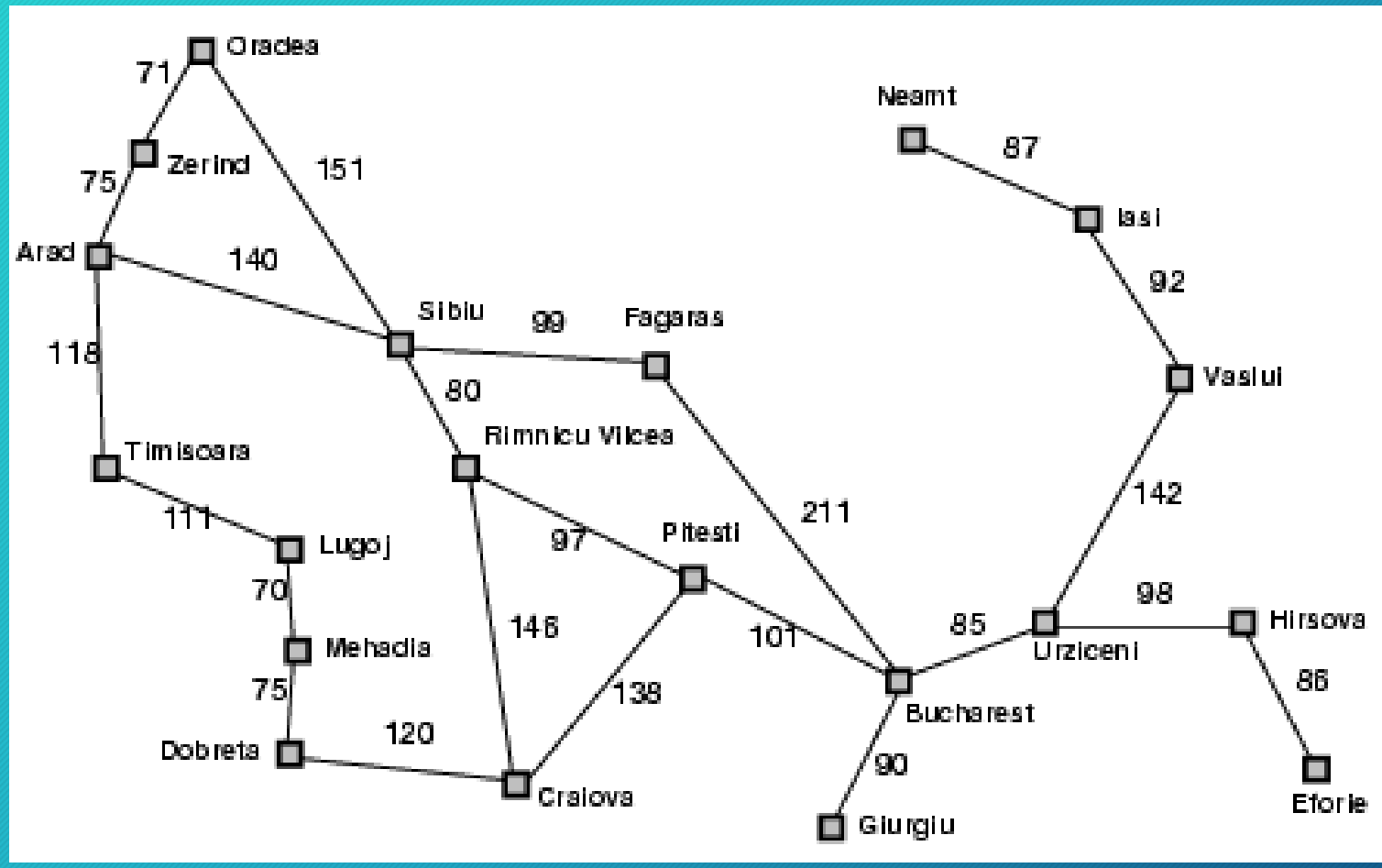
c) see if SUCCESSOR is the same as any node on OPEN. If so, call that node OLD. Since this node already exists in the graph, we can throw SUCCESSOR away and add OLD to the list of BESTNODES successors.

Determine the parent link: If the path just found to SUCCESSOR is cheaper than the current best path to OLD, then reset OLD's parent – link to BESTNODE; else, don't update anything.

d) if SUCCESSOR is not in OPEN but in CLOSED, call it OLD and add OLD to the list of BESTNODE's successors. Check to see if the new path is better as in step (c) . If it is, then set the parent link and g and f value appropriately. Propagate the new cost downward and determine the better path. If required set the parent link accordingly.

e) if SUCCESSOR is neither in OPEN nor in CLOSED , then put it on OPEN and add it to the list of BESTNODE's successors. Compute f(SUCCESSOR) = g(SUCCESSOR) + h( SUCCESSOR)

# Romania with step costs in km

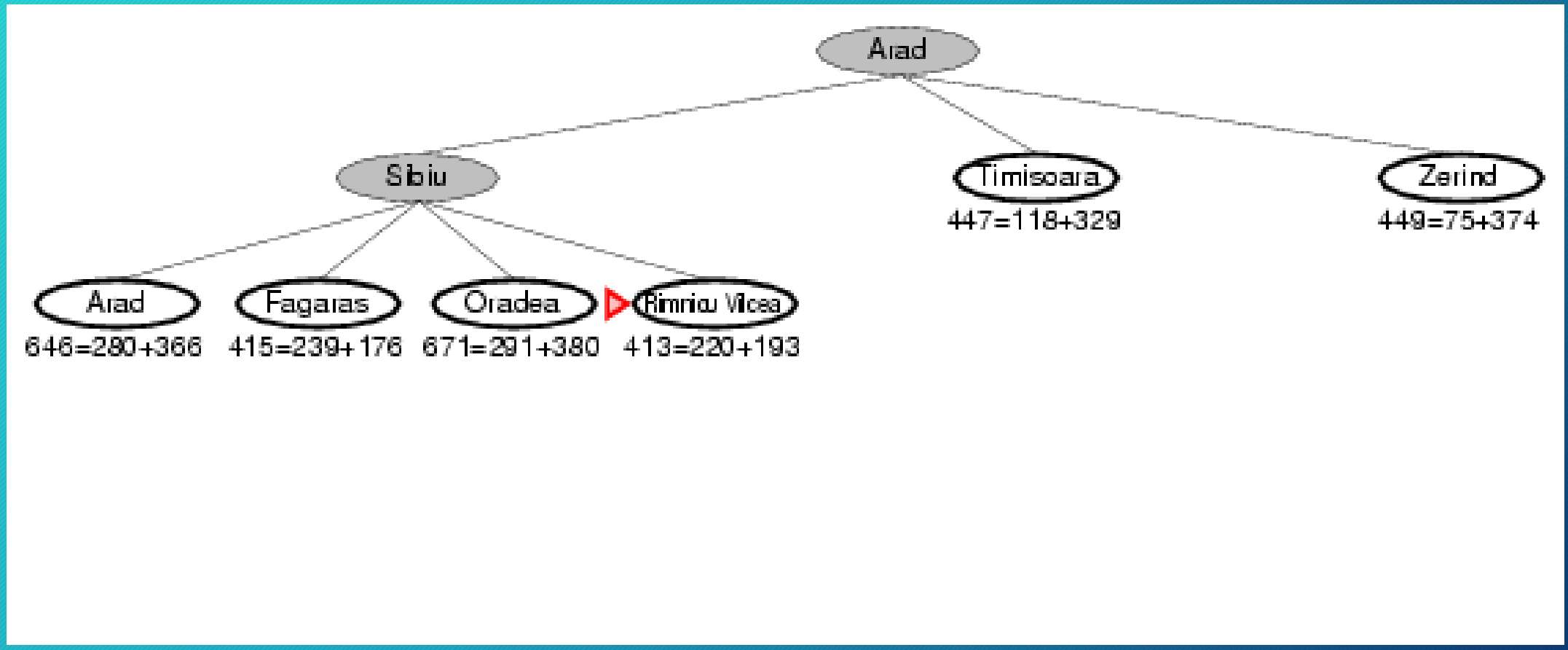



| Straight–line distance to Bucharest | |
|---|---|
| **Arad** | 366 |
| **Bucharest** | 0 |
| **Craiova** | 160 |
| **Dobreta** | 242 |
| **Eforie** | 161 |
| **Fagaras** | 176 |
| **Giurgiu** | 77 |
| **Hirsova** | 151 |
| **Iasi** | 226 |
| **Lugoj** | 244 |
| **Mehadia** | 241 |
| **Neamt** | 234 |
| **Oradea** | 380 |
| **Pitesti** | 10 |
| **Rimnicu Vilcea** | 193 |
| **Sibiu** | 253 |
| **Timisoara** | 329 |
| **Urziceni** | 80 |
| **Vaslui** | 199 |
| **Zerind** | 374 |

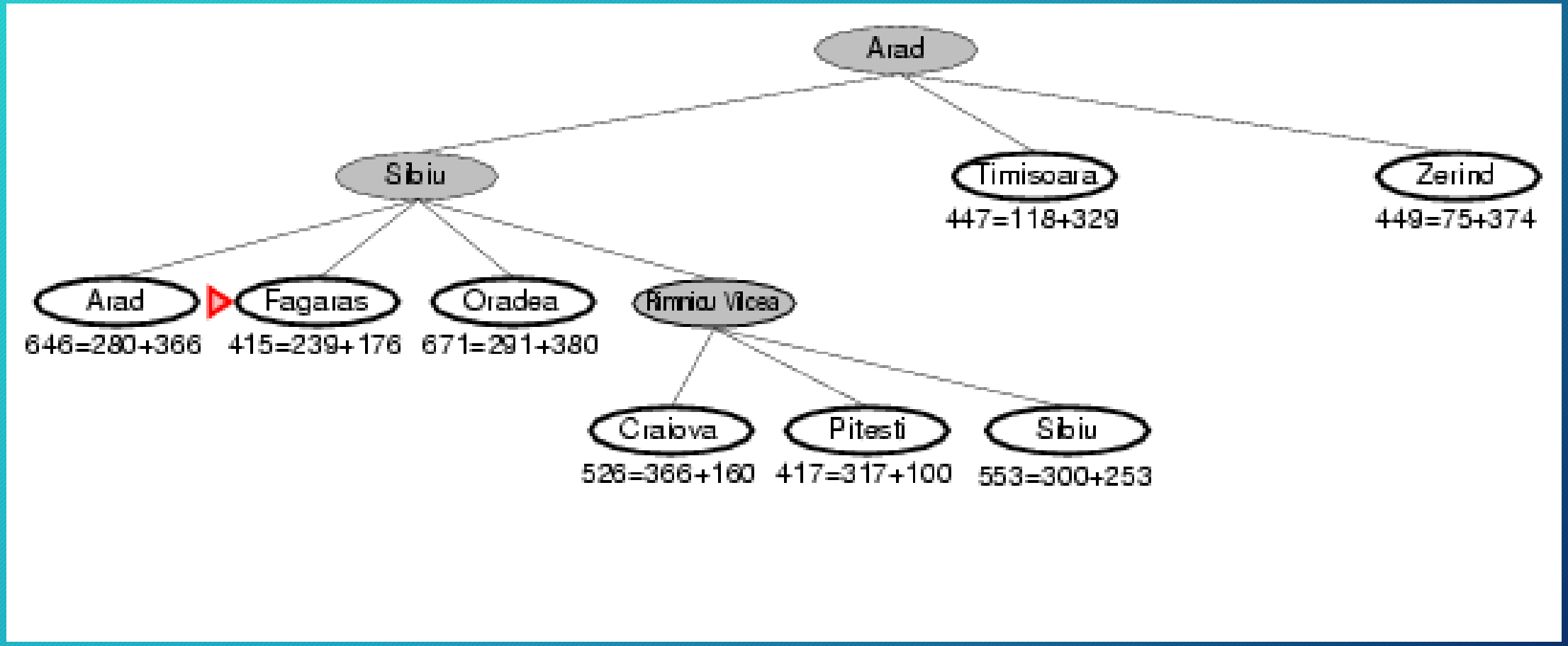
# A* Search Example

# A* Search Example

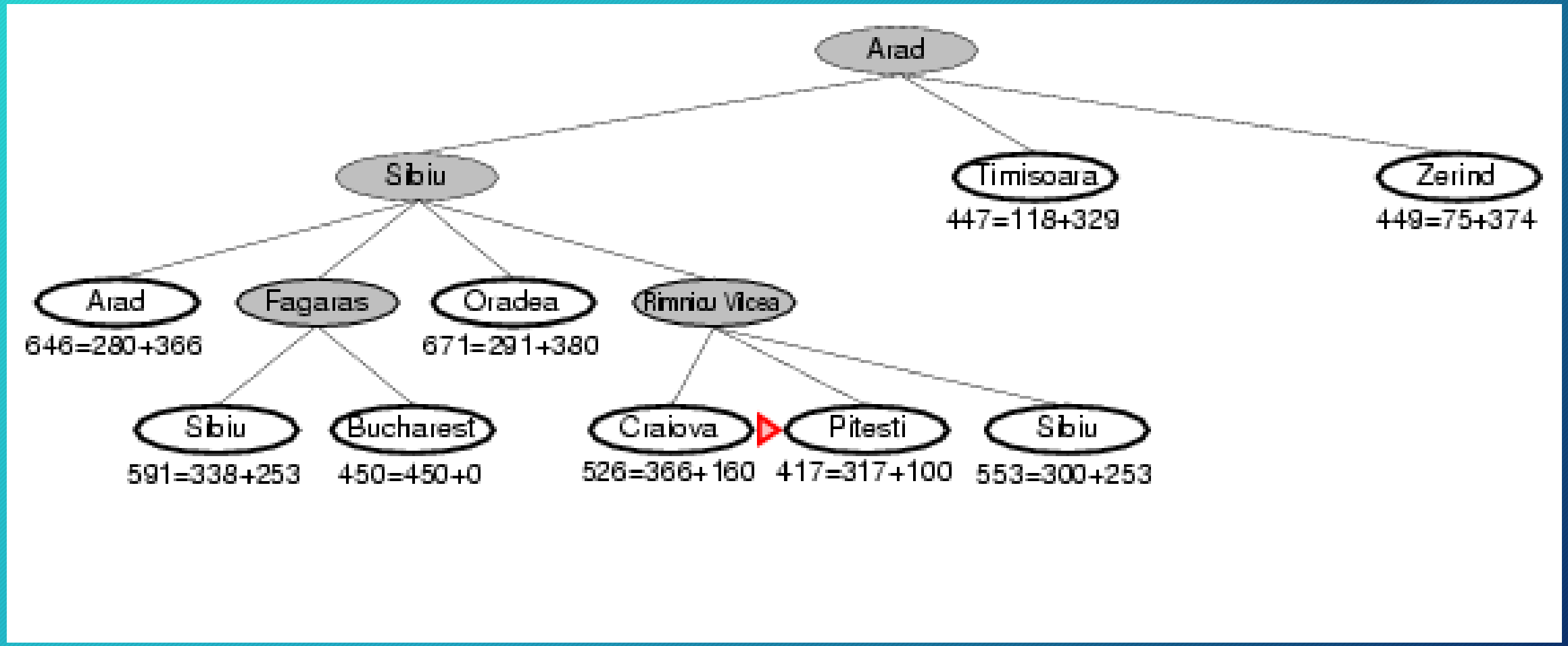
# A* Search Example



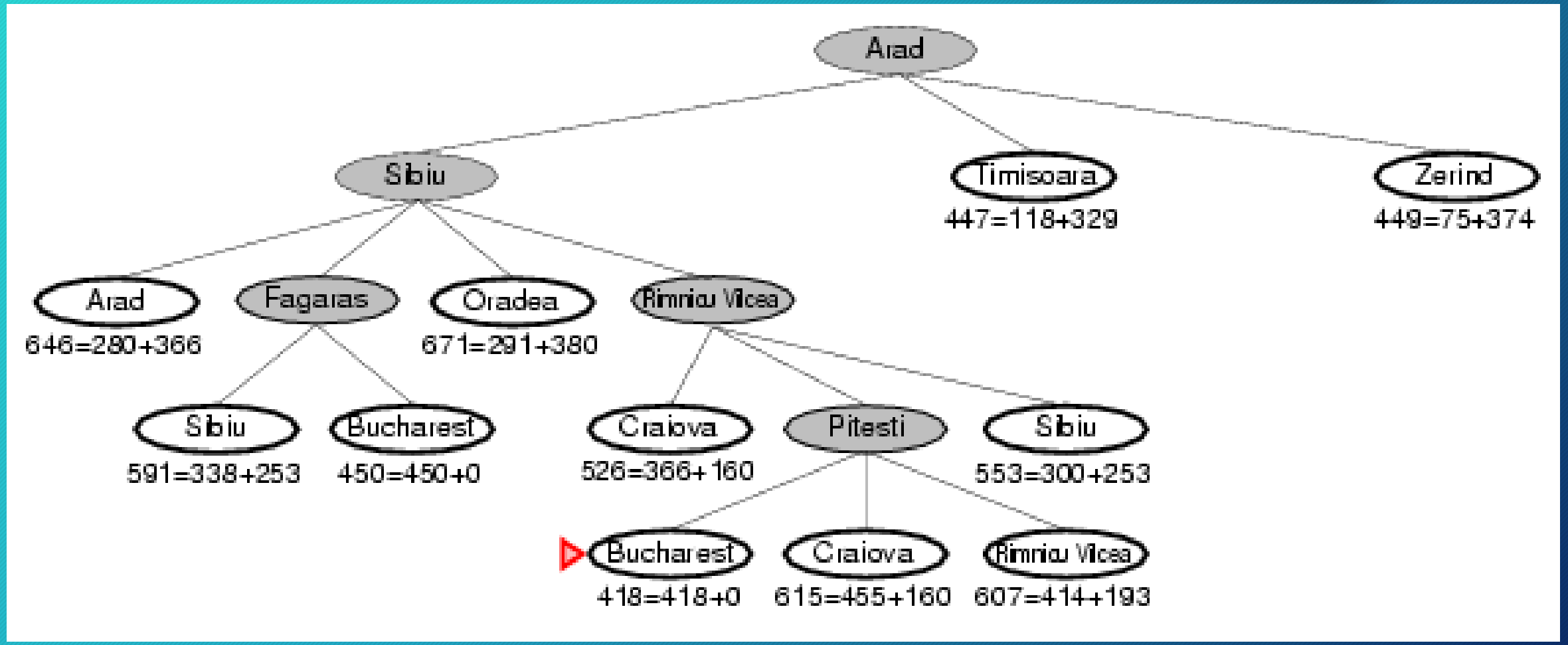
# A* Search Example



# A* Search Example



# A* Search Example



# Properties of A* Search

- Complete? Yes (unless there are infinitely many nodes with f ≤ $f(G)$)
- Time? Exponential
- Space? Keeps all nodes in memory
- Optimal? Yes if h(n) is admissible.

# Results on A* Search

- A heuristics called as admissible if it always under-estimates, i.e., we always have $h(n) \leq f^*(n)$, where $f^*(n)$ denotes the minimum distance to a goal state from state n
- For finite state space, A* always terminates
- At any time before A* terminates, there exists in OPEN a state n that is an optimal path from start state to goal state with $f(n) \leq f^*(s)$

# Results on A* Search (contd…)

- Algorithm A* is admissible, that is, if there is a path from start state to goal state, A* terminates by finding an optimal path.

[Reference Book for proof: *Principles of artificial intelligence* by N. J. Nilsson]

- If $A_1$ and $A_2$ are two versions of A* such that $A_2$ is more informed than $A_1$, then $A_1$ expand at least as many states as does $A_2$

# Admissible Heuristics

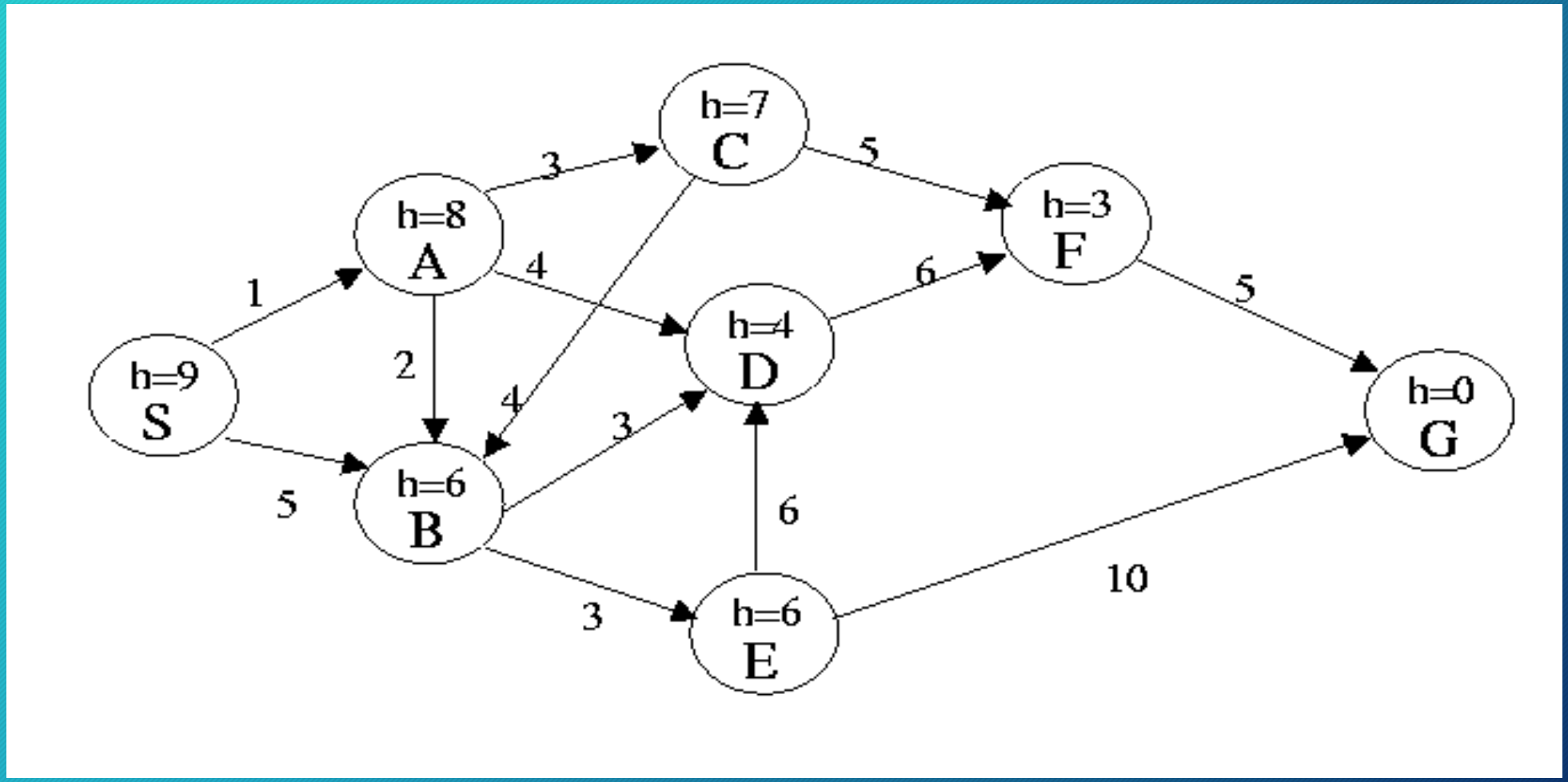
- A heuristic $h(n)$ is *admissible* if for every node $n$,

  $h(n) \leq h^*(n)$, where $h^*(n)$ is the true cost to reach the goal state from $n$.

- An admissible heuristic never overestimates the cost to reach the goal, i.e., it is optimistic

- Example: $h_{SLD}(n)$ (never overestimates the actual road distance)
- Theorem: If $h(n)$ is admissible, A* using `TREE-SEARCH` is optimal

# Example: Search Graphs



# Example: Search Graphs

- In this problem the start state is **S**, and the goal state is **G**. The transition costs are next to the edges, and the heuristic estimate, h, of the distance from the state to the goal is in the state's node. Assume ties are always broken by choosing the state which comes first alphabetically.

1. What is the order of states expanded using Depth First Search? Assume DFS terminates as soon as it reaches the goal state.

- Answer: S, A, B, D, F ,G

2. What is the order of states expanded using Breadth First Search?

- Answer: S, A, B, C, D, E, F, G

3. What is the order of states expanded using Best First Search? Assume BFS terminates as soon as it reaches the goal state.

- Answer: S, B, D, F, G

4. What is the order of states expanded using A* search?

- Answer: S, A, B, D, C, E, F, G

5. What is a least cost path from S to G?

- Answer: S, A, C, F, G

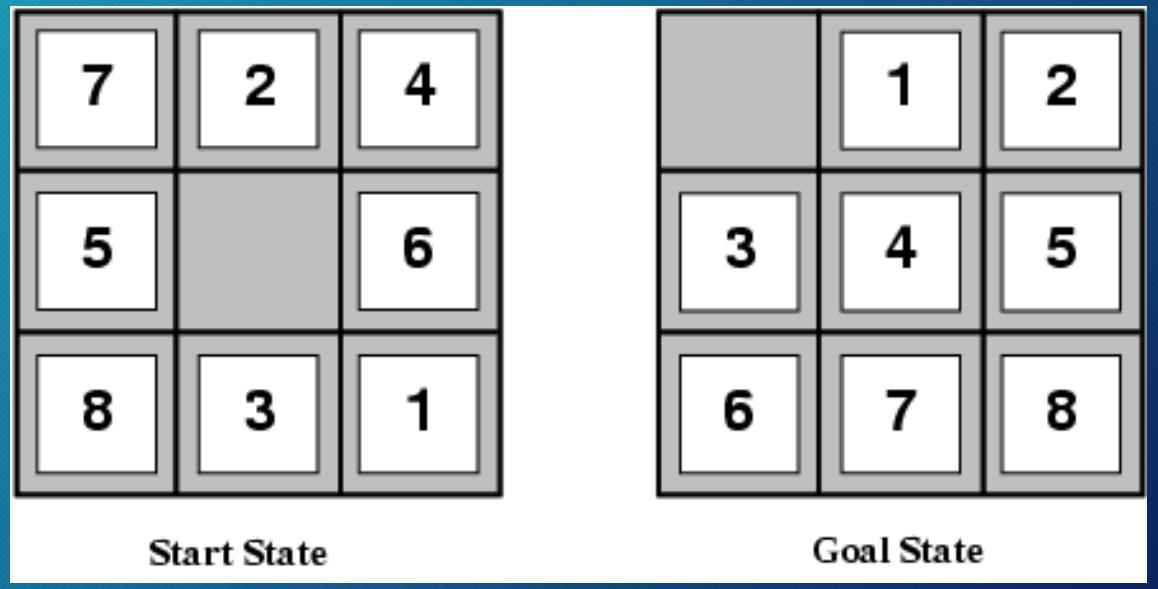
# Example: Search Graphs

Consider following 8-puzzle Problem

- $h_1(n)$ = number of misplaced tiles
- $h_2(n)$ = total Manhattan distance

(i.e., no. of squares from desired location of each tile)

- $h_1(S)$ = ?
- $h_2(S)$ = ?

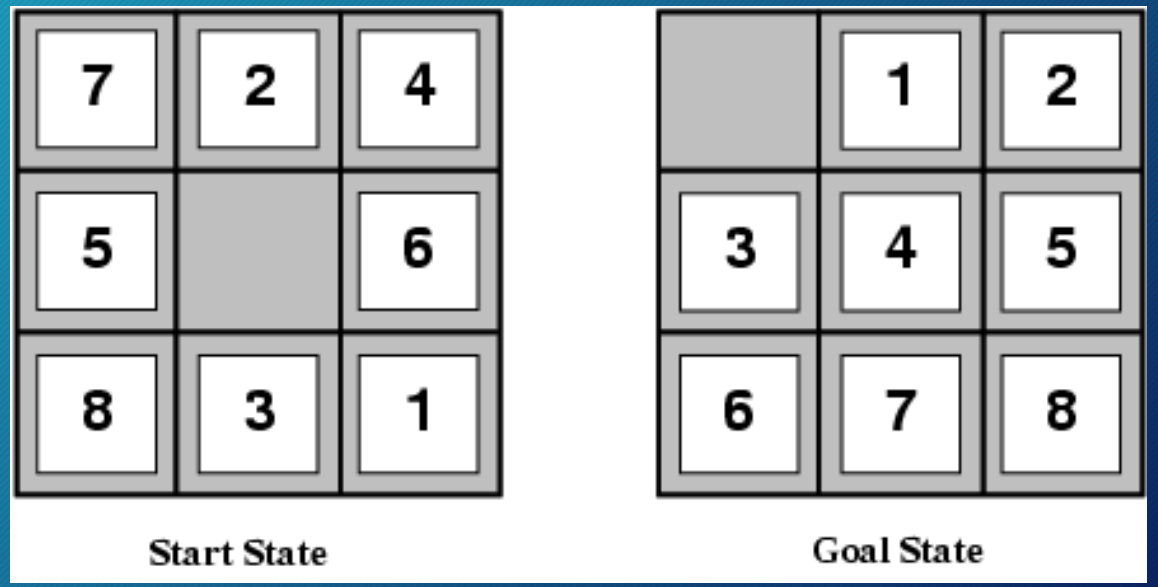


# Example Solution: Search Graphs

Consider following 8-puzzle Problem

- $h_1(n)$ = number of misplaced tiles
- $h_2(n)$ = total Manhattan distance

(i.e., no. of squares from desired location of each tile)




- $h_1(S) = 8$
- $h_2(S) = 3+1+2+2+2+3+3+2 = 18$

# Solution Explanation: Search Graphs

The 8-puzzle was one of the earliest heuristic search problems. As mentioned in Section 3.2, the object of the puzzle is to slide the tiles horizontally or vertically into the empty space until the configuration matches the goal configuration (Figure 3.28).

The average solution cost for a randomly generated 8-puzzle instance is about 22 steps. The branching factor is about 3. (When the empty tile is in the middle, four moves are possible; when it is in a corner, two; and when it is along an edge, three.) This means that an exhaustive tree search to depth 22 would look at about $3^{22} \approx 3.1 \times 10^{10}$ states. A graph search would cut this down by a factor of about 170,000 because only $9!/2 = 181,440$ distinct states are reachable. (See Exercise 3.4.) This is a manageable number, but the corresponding number for the 15-puzzle is roughly $10^{13}$, so the next order of business is to find a good heuristic function. If we want to find the shortest solutions by using A*, we need a heuristic function that never overestimates the number of steps to the goal. There is a long history of such heuristics for the 15-puzzle; here are two commonly used candidates:

# Solution Explanation: Search Graphs

- $h_1$ = the number of misplaced tiles. For Figure 3.28, all of the eight tiles are out of position, so the start state would have $h_1 = 8$. $h_1$ is an admissible heuristic because it is clear that any tile that is out of place must be moved at least once.
- $h_2$ = the sum of the distances of the tiles from their goal positions. Because tiles cannot move along diagonals, the distance we will count is the sum of the horizontal and vertical distances. This is sometimes called the **city block distance** or **Manhattan distance**. $h_2$ is also admissible because all any move can do is move one tile one step closer to the goal. Tiles 1 to 8 in the start state give a Manhattan distance of

$$h_2 = 3 + 1 + 2 + 2 + 2 + 3 + 3 + 2 = 18 .$$

As expected, neither of these overestimates the true solution cost, which is 26.

# THANK YOU

Reference: Artificial Intelligence by Stuart_J_Russell_and_Peter_Norvig
Online Contents