

بِسْمِ اللَّهِ الرَّحْمَنِ الرَّحِيمِ

BISMILLAH ARRAHMAN ARRAHEEM

Artificial Intelligence (CS-401)

Lecture 3: Problem Solving Search

Dr. Fahad Sherwani (Assistant Professor – FAST NUCES)

PhD in Artificial Intelligence

Universiti Tun Hussein Onn Malaysia

fahad.sherwani@nu.edu.pk

Lecture Outline

- ❑ Achieve intelligence by (searching) a solution!
- ❑ Problem Formulation
- ❑ Uninformed (Blind) Search Strategies
 - ❑ breadth-first
 - ❑ uniform-cost search
 - ❑ depth-first
 - ❑ depth-limited search
 - ❑ iterative deepening
 - ❑ bi-directional search

Searching and AI

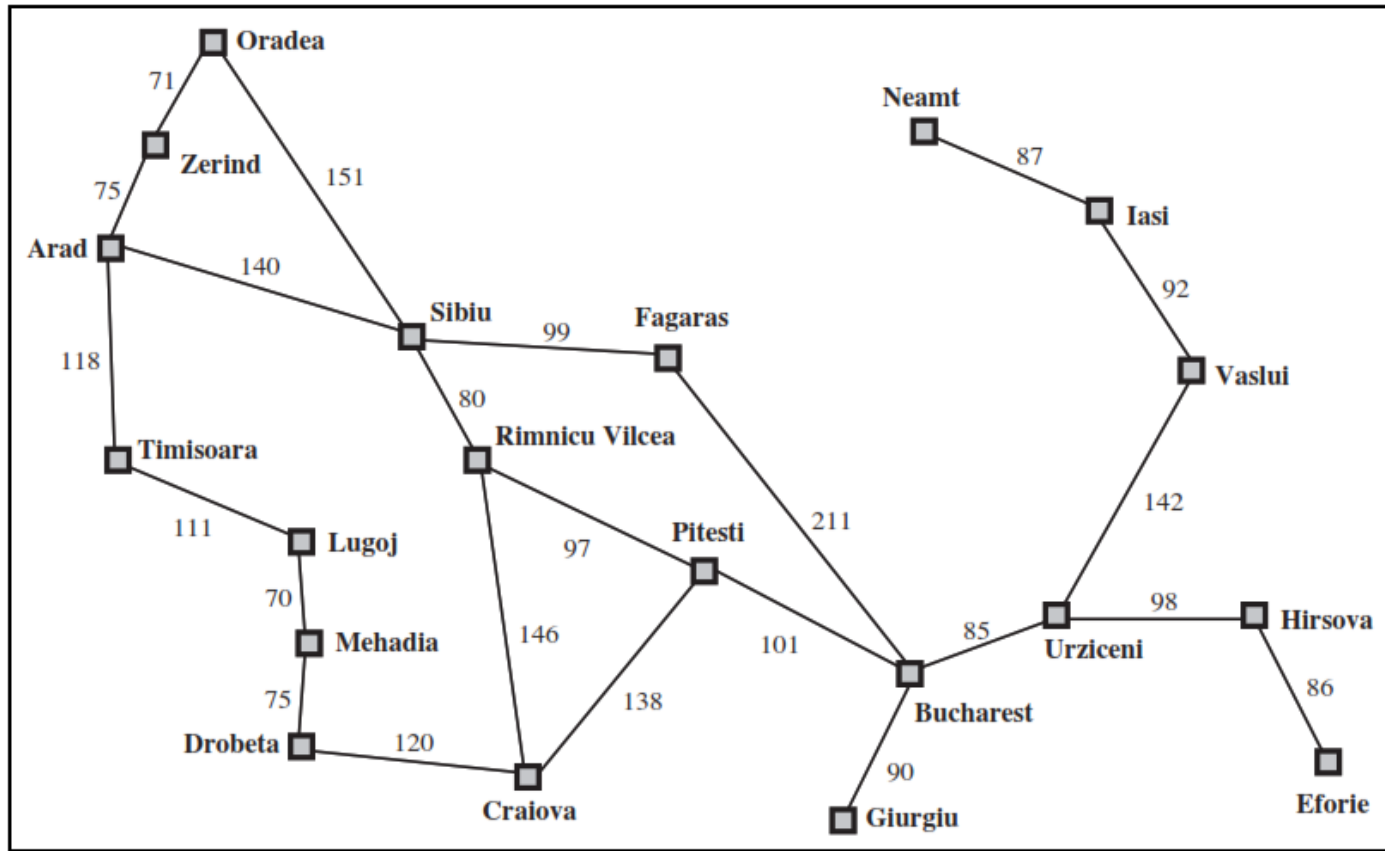
- **Searching** falls under Artificial Intelligence (AI).
- The major **goal of AI** is to give computers the ability to think, or in other words, mimic human behavior in **learning and problem solving**.
- The issue is, unfortunately, computers don't function in the same way our minds do.
- They require a series of **well-reasoned out** steps before finding a solution.
- Your aim, then, is to take a **complicated task** and convert it into **simpler steps** that your computer can handle.

Uninformed (Blind Search)

- The Uninformed Search does not contain any domain knowledge such as closeness or location of goal.
- It operates in a brute force way, as it only includes information about how to traverse the tree and how to identify leaf and goal nodes.
- Uninformed Search applies a way in which search tree is searched without any information, so it is called blind search.
- It examines each node until it achieves the goal node.

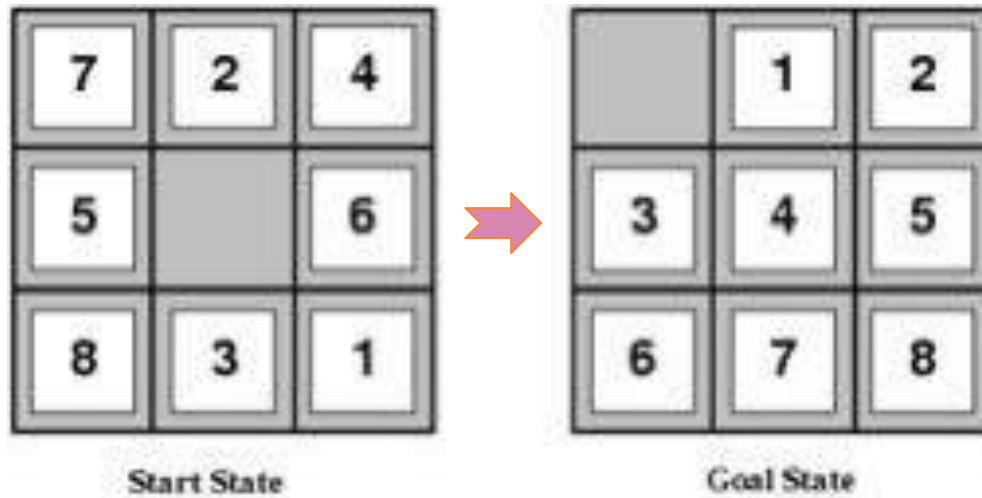
Example: Romania

You are in Arad and want to go to Bucharest



➔ How to design an intelligent agent to find the way between 2 cities?

Example: The 8-puzzle



➔ How to design an intelligent agent to solve the 8-puzzle?

Searching in AI

- **Optimization.** Search algorithms are the basis for many optimization and planning methods.
- **Solve a problem by searching for a solution.** Search strategies are important methods for many approaches to problem-solving.
- **Problem formulation.** The use of search requires an abstract formulation of the problem and the available steps to construct solutions.

Problem Formulation

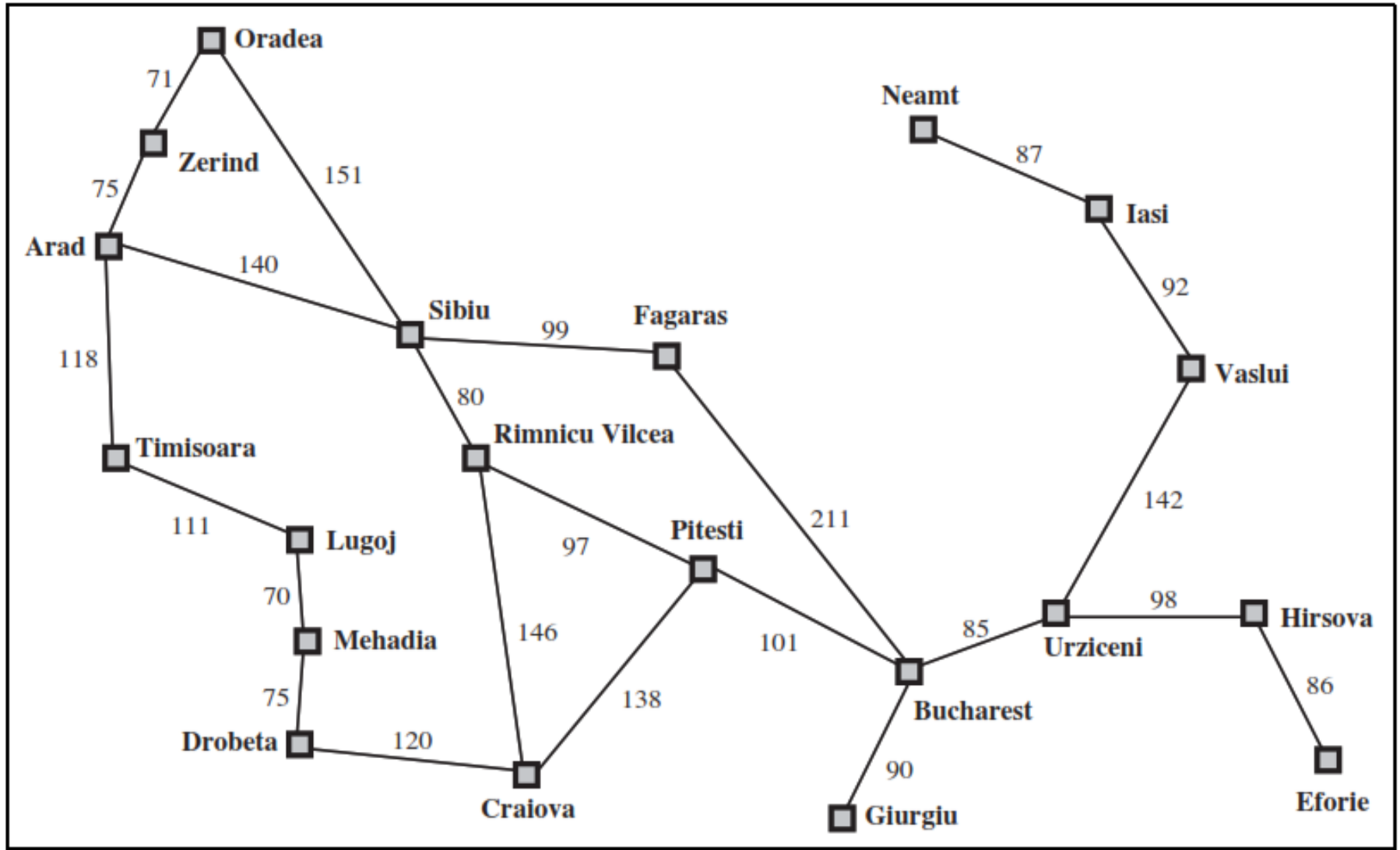
To solve a problem by search, we need to first formulate the problem.

HOW?

Our textbook suggest the following schema to help us formulate problems:

1. State
2. Initial state
3. Actions or Successor Function
4. Goal Test
5. Path Cost
6. Solution

Problem Formulation (The Romania Example)



Solution: a sequence of actions leading from the initial state to a goal state
{Arad → Sibiu → Rimnicu Vilcea → Pitesti → Bucharest}

Problem Formulation (The 8- Puzzle Example)

State: The location of the eight tiles, and the blank

Initial State: $\{(7,0), (2,1), (4,2), (5,3), (_,4), (6,5), (8,6), (3,7), (1,8)\}$

7	2	4
5		6
8	3	1

Start State

	1	2
3	4	5
6	7	8

Goal State

Successor Function: one of the four actions (blank moves Left, Right, Up, Down).

Goal Test: determine a given state is a goal state.

Path Cost: each step costs 1

Solution: $\{(_,0), (1,1), (2,2), (3,3), (4,4), (5,5), (6,6), (7,7), (8,8)\}$

Problem Formulation (Real-life Applications)

Route Finding Problem



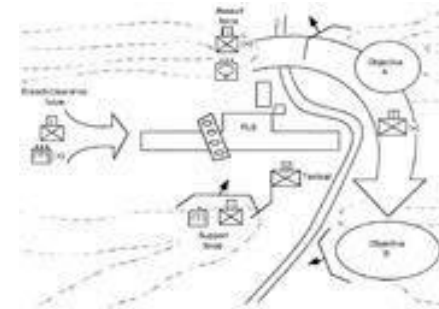
Car
Navigation



Airline travel
planning



Routing in Computer
networks



Military operation
planning

- **States**
 - locations
- **Initial state**
 - starting point
- **Successor function (operators)**
 - move from one location to another
- **Goal test**
 - arrive at a certain location
- **Path cost**
 - may be quite complex
 - money, time, travel comfort, scenery,

Problem Formulation (Real-life Applications)

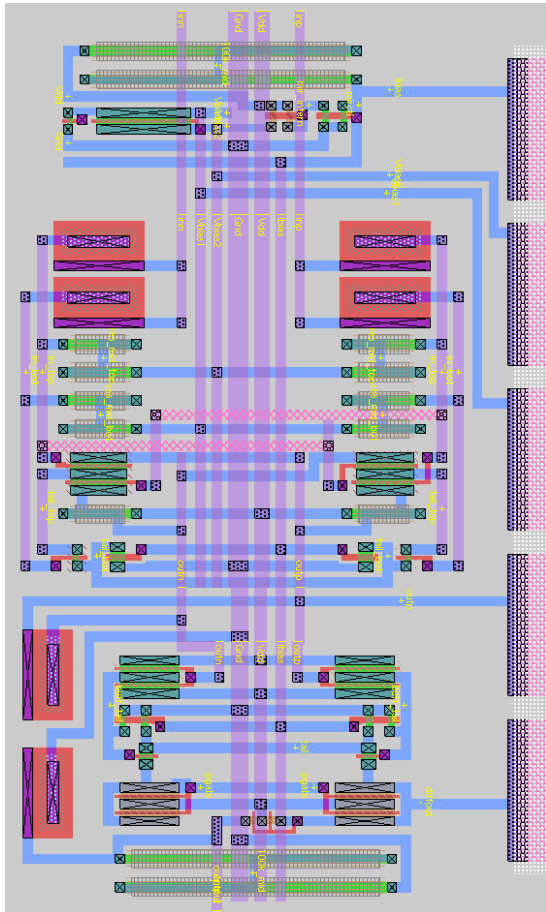
Travel Salesperson Problem



- **States**
 - locations / cities
 - illegal states
 - each city may be visited only once
 - visited cities must be kept as state information
- **Initial state**
 - starting point
 - no cities visited
- **Successor function (operators)**
 - move from one location to another one
- **Goal test**
 - all locations visited
 - agent at the initial location
- **Path cost**
 - distance between locations

Problem Formulation (Real-life Applications)

VLSI layout Problem



- **States**
 - positions of components, wires on a chip
- **Initial state**
 - incremental: no components placed
 - complete-state: all components placed (e.g. randomly, manually)
- **Successor function (operators)**
 - incremental: place components, route wire
 - complete-state: move component, move wire
- **Goal test**
 - all components placed
 - components connected as specified
- **Path cost**
 - maybe complex
 - distance, capacity, number of connections per component

Problem Formulation (Real-life Applications)

Robot Navigation



- **States**
 - locations
 - position of actuators
- **Initial state**
 - start position (dependent on the task)
- **Successor function (operators)**
 - movement, actions of actuators
- **Goal test**
 - task-dependent
- **Path cost**
 - maybe very complex
 - distance, energy consumption

Problem Formulation (Real-life Applications)

Automatic Assembly Sequencing



- **States**
 - location of components
- **Initial state**
 - no components assembled
- **Successor function (operators)**
 - place component
- **Goal test**
 - system fully assembled
- **Path cost**
 - number of moves

Searching for Solutions

Traversal of the search space

- From the initial state to a goal state.
- Legal sequence of actions as defined by successor function.

General procedure

- Check for goal state
- Expand the current state
 - Determine the set of reachable states
 - Return “failure” if the set is empty
- Select one from the set of reachable states
- Move to the selected state

A search tree is generated

- Nodes are added as more states are visited

Search Terminology

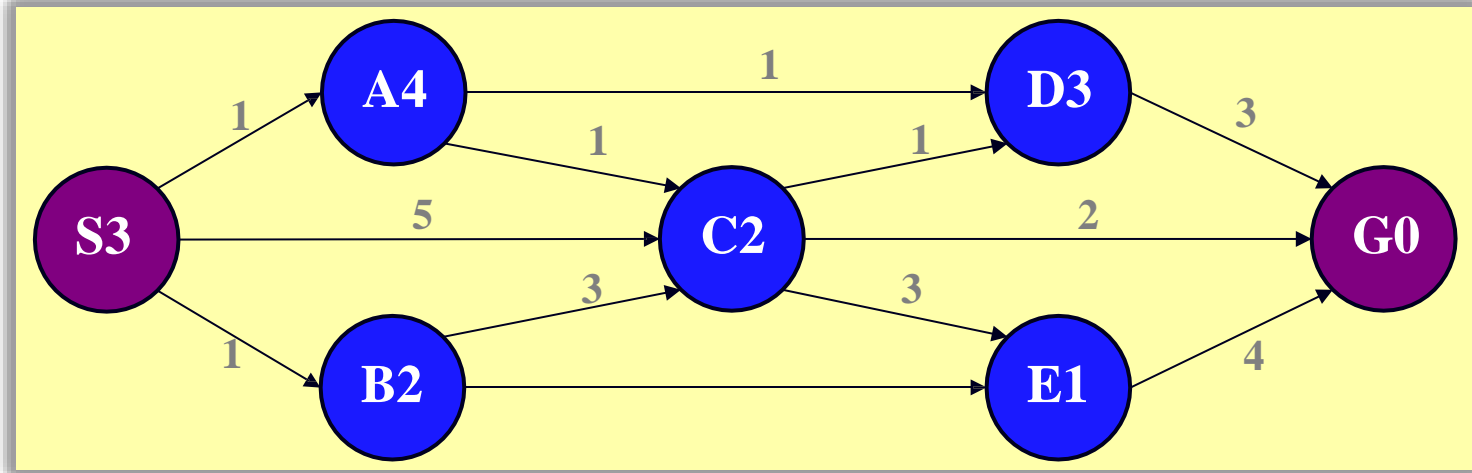
Search Tree

- Generated as the search space is traversed
 - The search space itself is not necessarily a tree, frequently it is a graph
 - The tree specifies possible paths through the search space
- Expansion of nodes
 - As states are explored, the corresponding nodes are expanded by applying the successor function
 - this generates a new set of (child) nodes
 - The *fringe* (frontier/queue) is the set of nodes not yet visited
 - newly generated nodes are added to the fringe
- Search strategy
 - Determines the selection of the next node to be expanded
 - Can be achieved by ordering the nodes in the fringe
 - e.g. queue (FIFO), stack (LIFO), “best” node w.r.t. some measure (cost)

Search Tree Vs Graph Tree

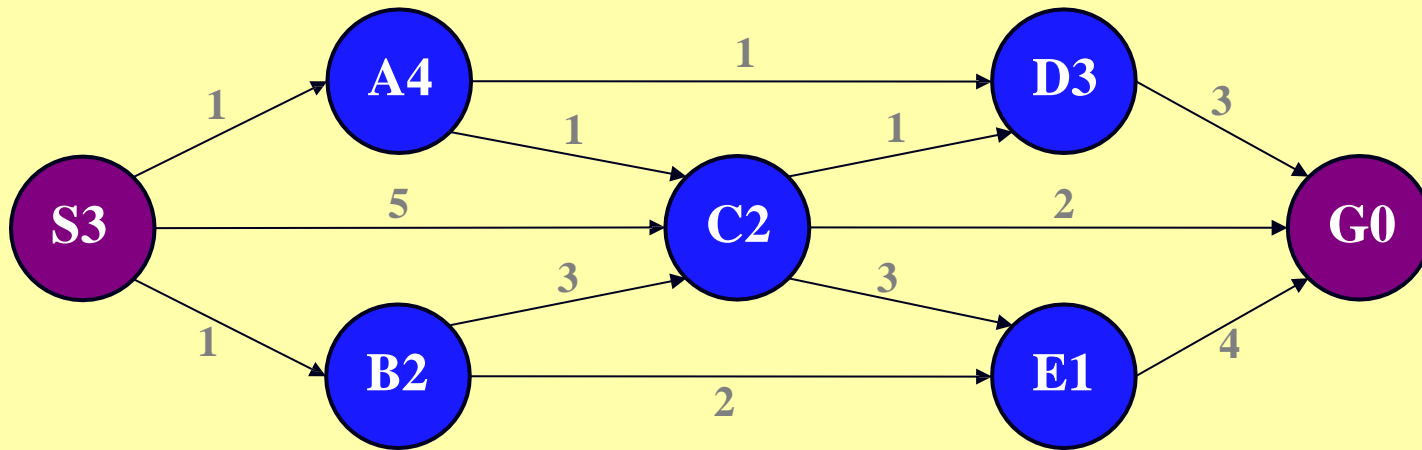
BASIS FOR COMPARISON	TREE	GRAPH
Path	Only one between two vertices.	More than one path is allowed.
Root node	It has exactly one root node.	Graph doesn't have a root node.
Loops	No loops are permitted.	Graph can have loops.
Complexity	Less complex	More complex comparatively
Traversal techniques	Pre-order, In-order and Post-order.	Breadth-first search and depth-first search.
Number of edges	$n-1$ (where n is the number of nodes)	Not defined
Model type	Hierarchical	Network

Example: Graph Search

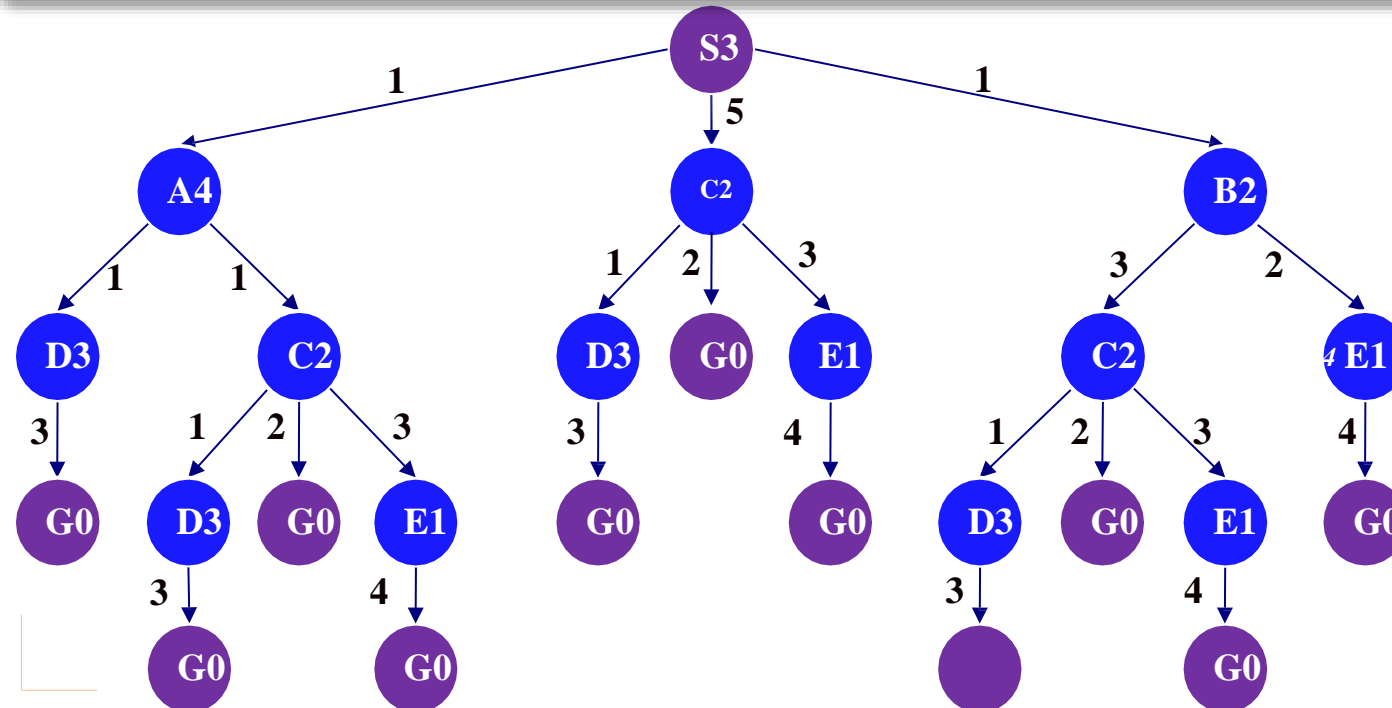


- The graph describes the search (state) space
 - Each node in the graph represents one state in the search space
 - e.g. a city to be visited in a routing or touring problem
- This graph has additional information
 - Names and properties for the states (e.g. S3)
 - Links between nodes, specified by the successor function
 - properties for links (distance, cost, name, ...)

Traversing a Graph as Tree



- A tree is generated by traversing the graph.
- The same node in the graph may appear repeatedly in the tree.
- The arrangement of the tree depends on the traversal strategy (search method)
- The initial state becomes the root node of the tree
- In the fully expanded tree, the goal states are the leaf nodes.
- Cycles in graphs may result in infinite branches



Kruskal's Algorithm

Kruskal's Algorithm includes 4 Steps:

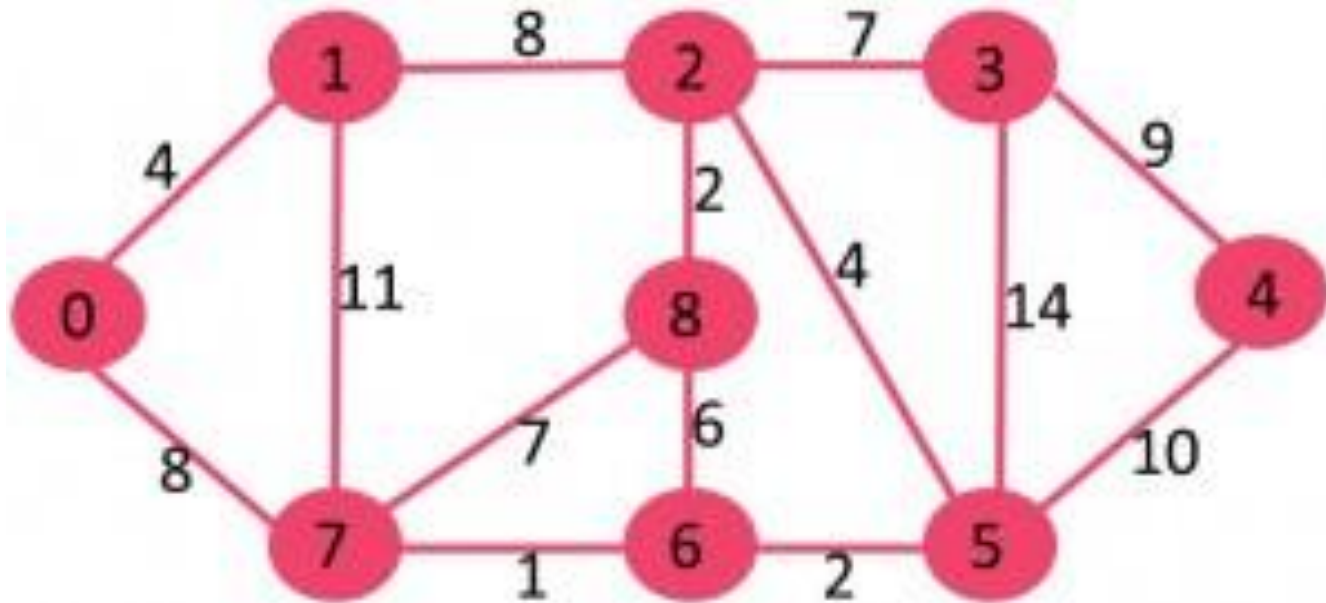
Step 1: List all the edges of the graph in order of increasing weights.

Step 2: Select the smallest edge of the graph.

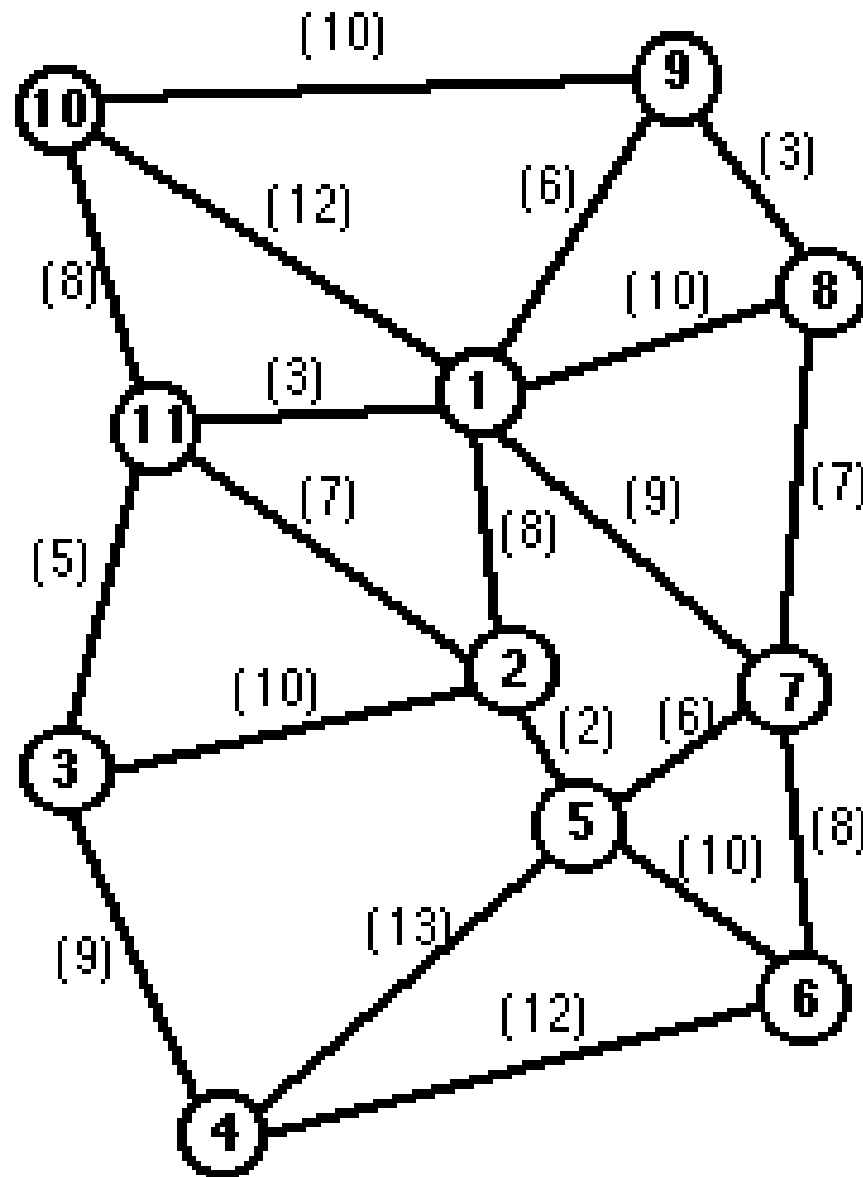
Step 3: Select the next smallest edge that do not makes any circuit.

Step 4: Continue this process until all the Vertices are explored and $(V_n - 1)$ edges have been selected

Minimum Spanning Tree (MST) Kruskal's Algorithm



Find MST using Kruskal's Algorithm



Prim's Algorithm

Prim's Algorithm includes 4 Steps:

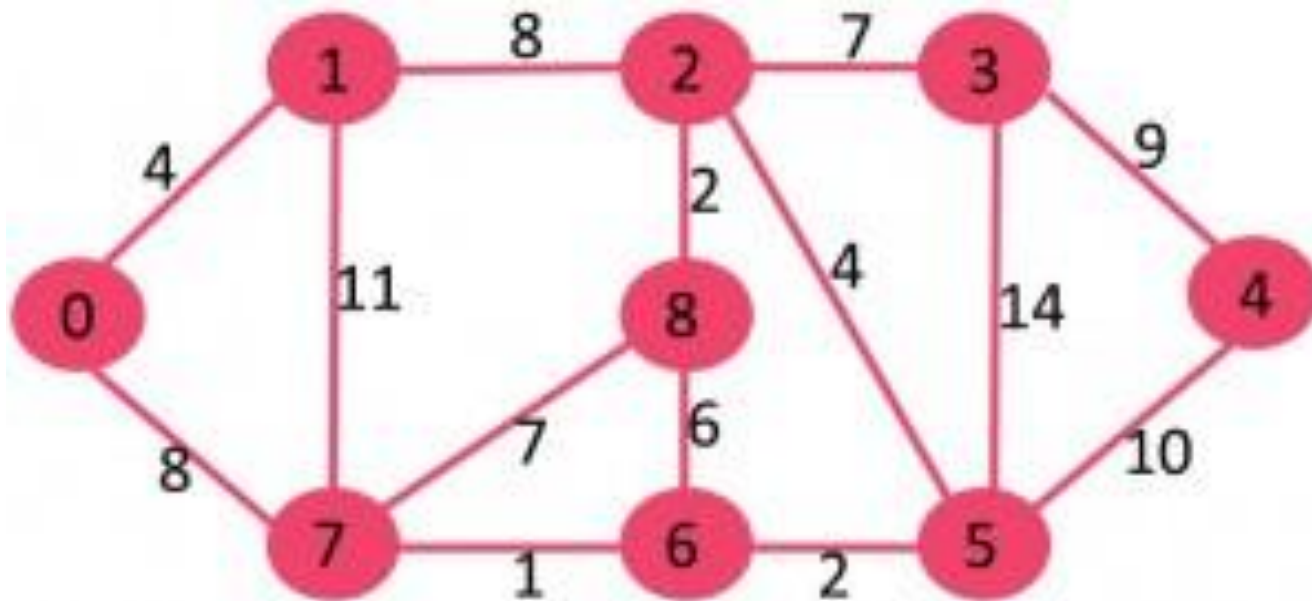
Step 1: Draw an n by n ($n \times n$) vertices' matrix and label them as V_1, V_2, \dots, V_n along with the given weights of the edges.

Step 2: Starting from vertex V_1 and connect it to its nearest neighbor by searching in row 1.

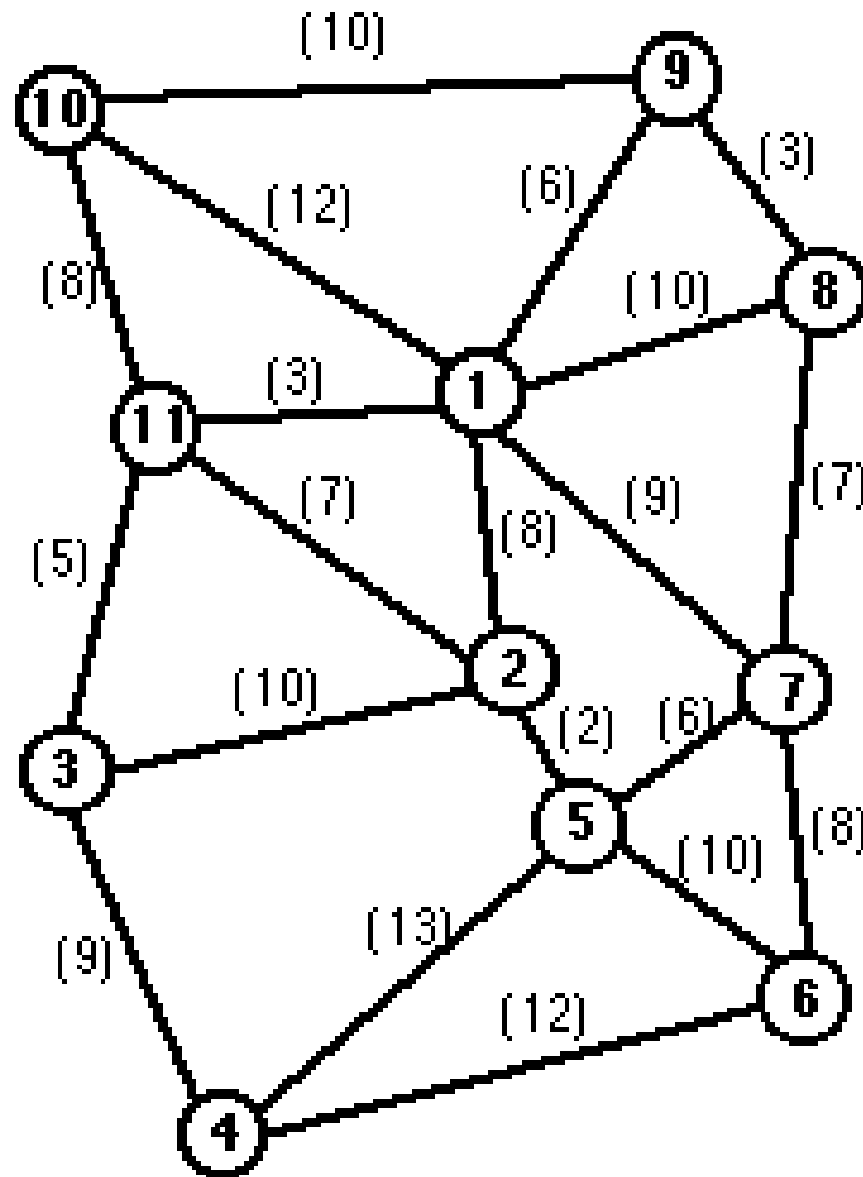
Step 3: Consider V_1 and V_i as one subgraph and connect as step 2 while do not forming any circuit.

Step 4: Continue this process until we get the MST having n vertices and $(n-1)$ edges.

Prim's Algorithm



Find MST using Prim's Algorithm



Prim's Vs Kruskal's

Prim's Algorithm	Kruskal's Algorithm
It starts to build the Minimum Spanning Tree from any vertex in the graph.	It starts to build the Minimum Spanning Tree from the vertex carrying minimum weight in the graph.
It traverses one node more than one time to get the minimum distance.	It traverses one node only once.
Prim's algorithm has a time complexity of $O(V^2)$, V being the number of vertices and can be improved up to $O(E + \log V)$ using Fibonacci heaps.	Kruskal's algorithm's time complexity is $O(E \log V)$, V being the number of vertices.
Prim's algorithm gives connected component as well as it works only on connected graph.	Kruskal's algorithm can generate forest(disconnected components) at any instant as well as it can work on disconnected components
Prim's algorithm runs faster in dense graphs.	Kruskal's algorithm runs faster in sparse graphs.

Searching Strategies

Uninformed Search

- breadth-first
 - uniform-cost search
 - depth-first
 - depth-limited search
 - iterative deepening
 - bi-directional search

Informed Search

- best-first search
- search with heuristics
- memory-bounded search
- iterative improvement search

Most of the effort is often spent on the selection of an appropriate search strategy for a given problem:

- Uninformed Search (blind search)
 - number of steps, path cost unknown
 - agent knows when it reaches a goal
- Informed Search (heuristic search)
 - agent has background information about the problem
 - map, costs of actions

Evaluation of Search Strategies

A search strategy is defined by picking the **order of node expansion**

Strategies are evaluated along the following dimensions:

- **Completeness**: if there is a solution, will it be found
- **Time complexity**: How long does it takes to find the solution
- **Space complexity**: memory required for the search
- **Optimality**: will the best solution be found

Time and space complexity are measured in terms of

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

1. Breadth-First Search (BFS) Algorithm

Breadth-First Search

- It is the **most common** search strategy for **traversing a tree or graph**.
- This algorithm searches **breadthwise** in a tree or graph, so it is called **breadth-first search**.
- BFS algorithm **starts searching** from the **root node** of the tree and expands all the successor nodes at the **current level** before moving to node of **next level**.
- BFS is implemented using **FIFO Queue data structure**.

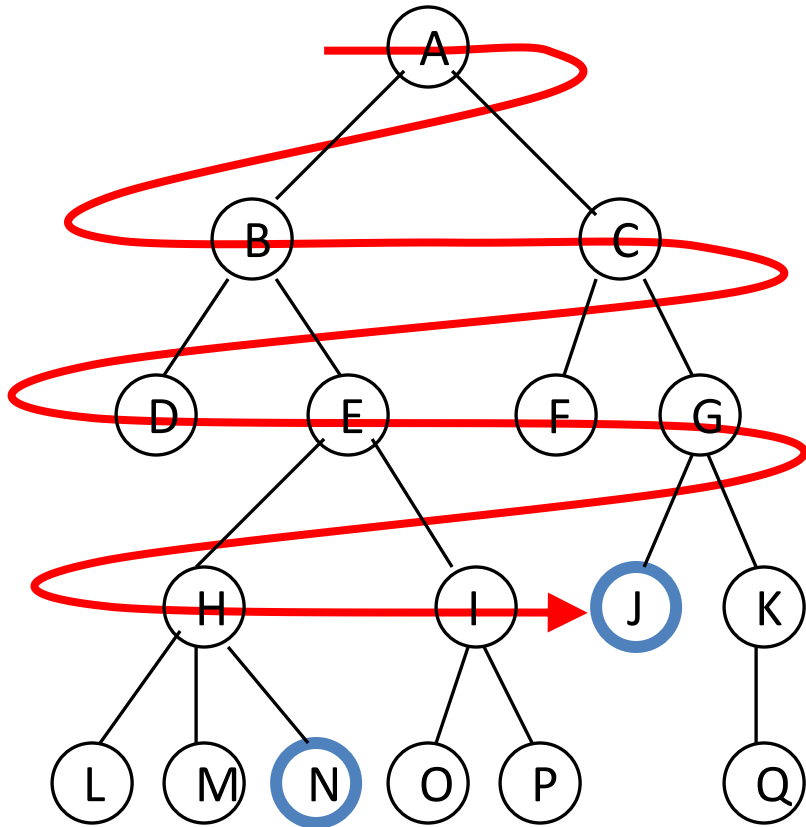
Breadth-First Search

All the nodes reachable from the current node are explored first (shallow nodes are expanded before deep nodes).

Algorithm (Informal)

1. Enqueue the root/initial node (**Queue Structure**).
2. Dequeue a node and examine it.
 1. If the element sought is found in this node, quit the search and return a result.
 2. Otherwise enqueue any successors (the direct child nodes) that have not yet been discovered.
3. If the queue is empty, every node on the graph has been examined – quit the search and return "not found".
4. Repeat from Step 2.

Breadth-First Search



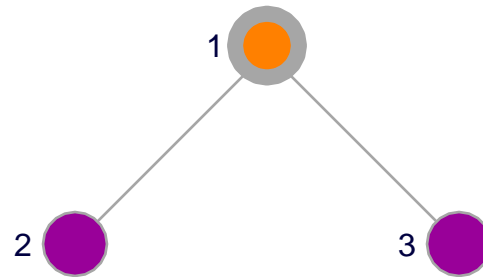
A **breadth-first search** (BFS) explores nodes nearest the root before exploring nodes further away on each level







For example, after searching **A**, then **B**, then **C**, the search proceeds with **D**, **E**, **F**, **G**

Node are explored in the Level order **A B C D E F G H I J K L M N O P Q**

J will be found before **N**

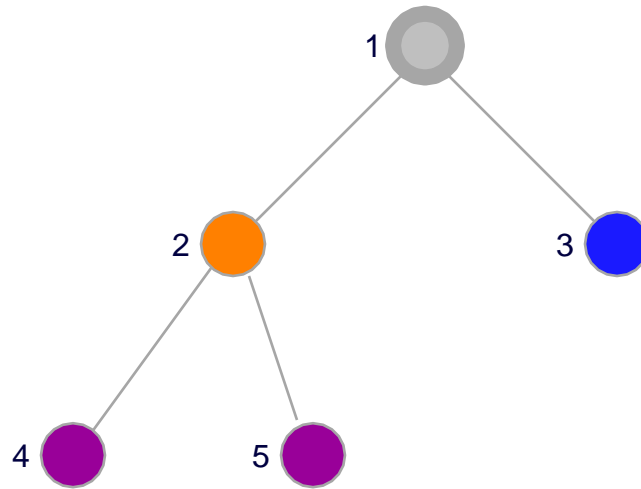
Breadth-First Snapshot 1









Initial	
Visited	
Fringe	
Current	
Visible	
Goal	

Fringe: [] + [2,3]

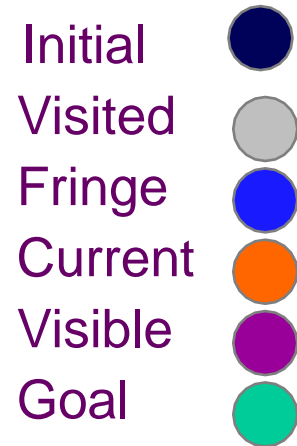
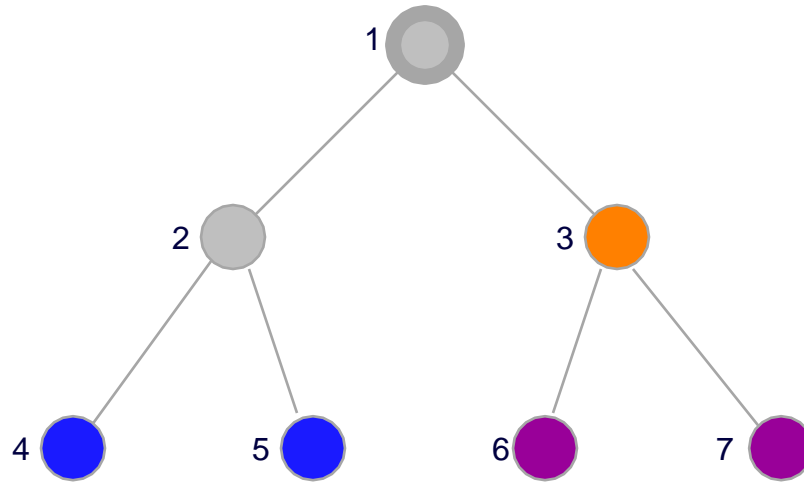
Breadth-First Snapshot 2



Initial	
Visited	
Fringe	
Current	
Visible	
Goal	

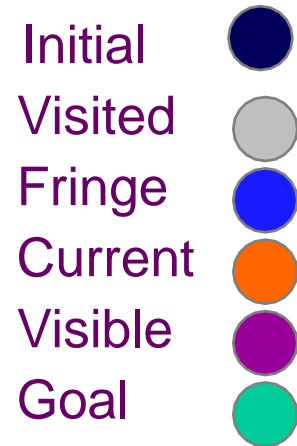
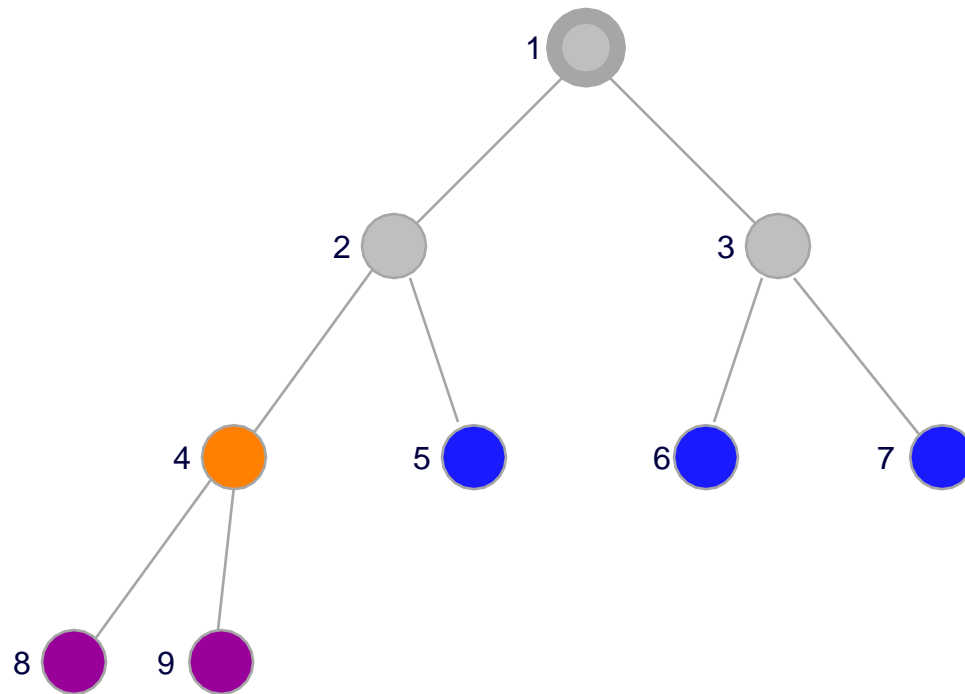
Fringe: [3] + [4,5]

Breadth-First Snapshot 3



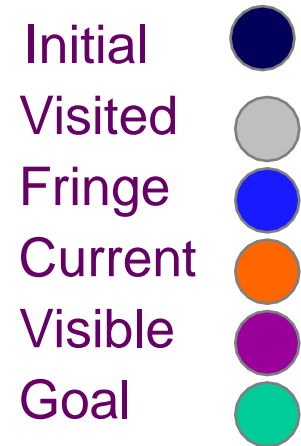
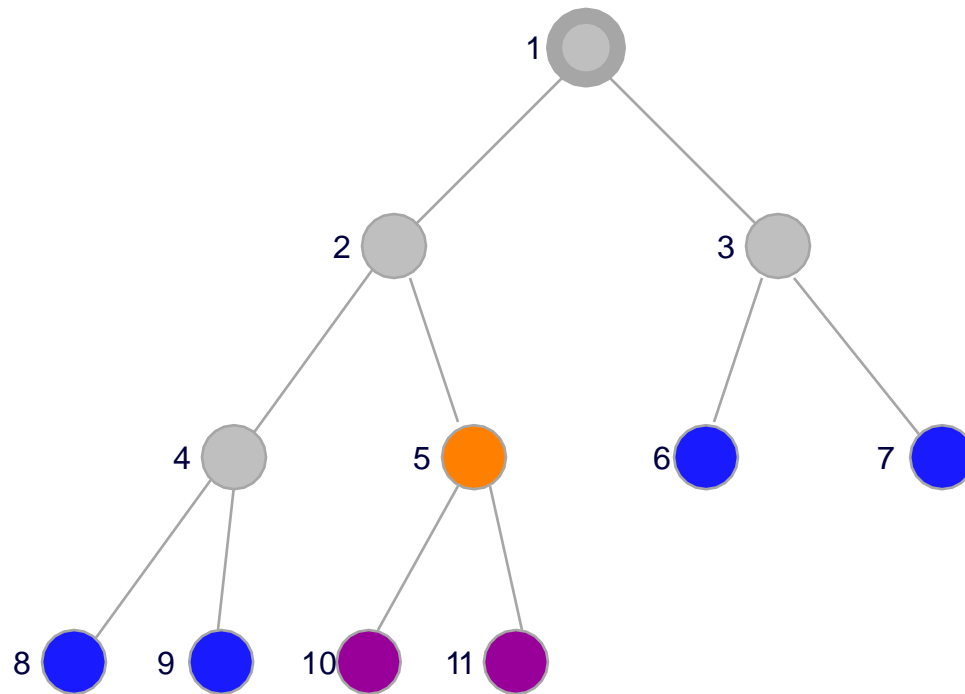
Fringe: [4,5] + [6,7]

Breadth-First Snapshot 4



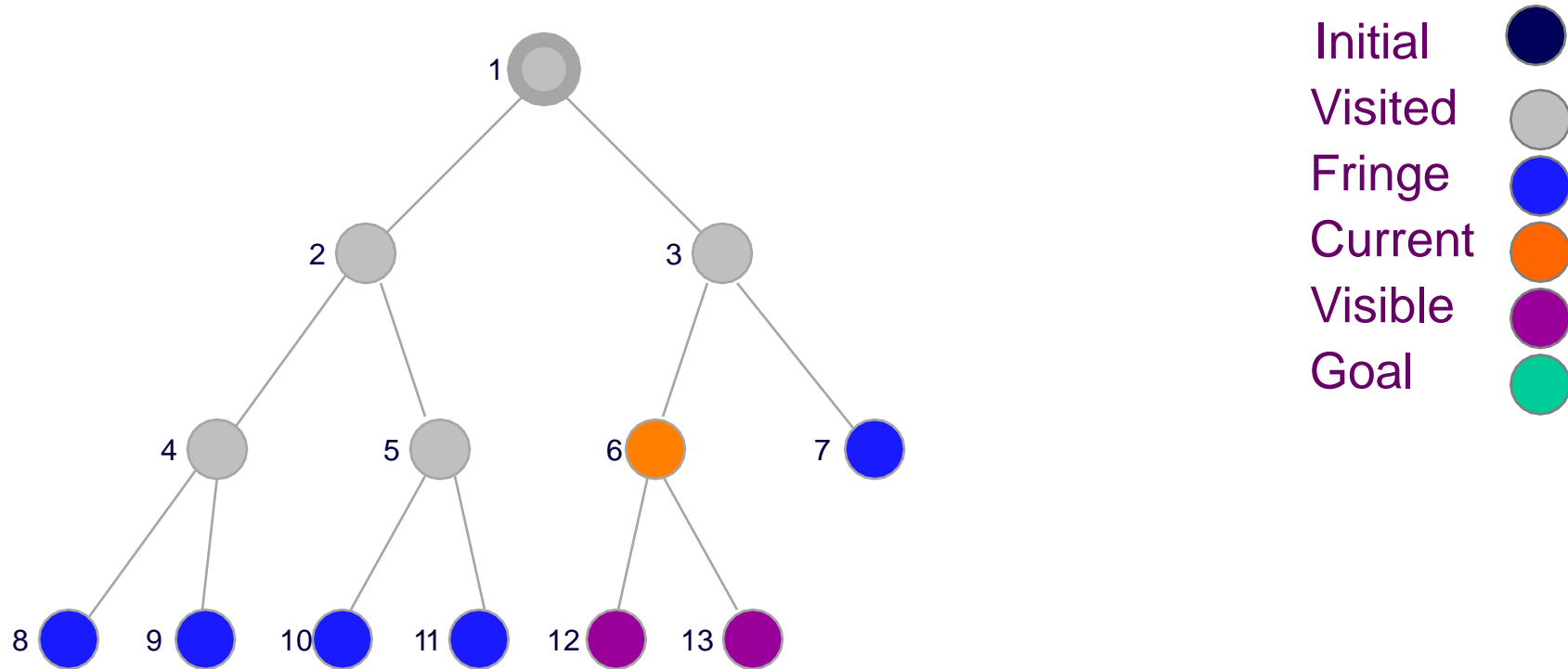
Fringe: [5,6,7] + [8,9]

Breadth-First Snapshot 5



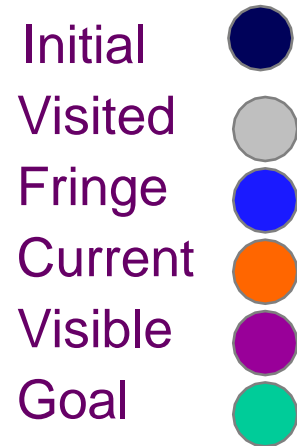
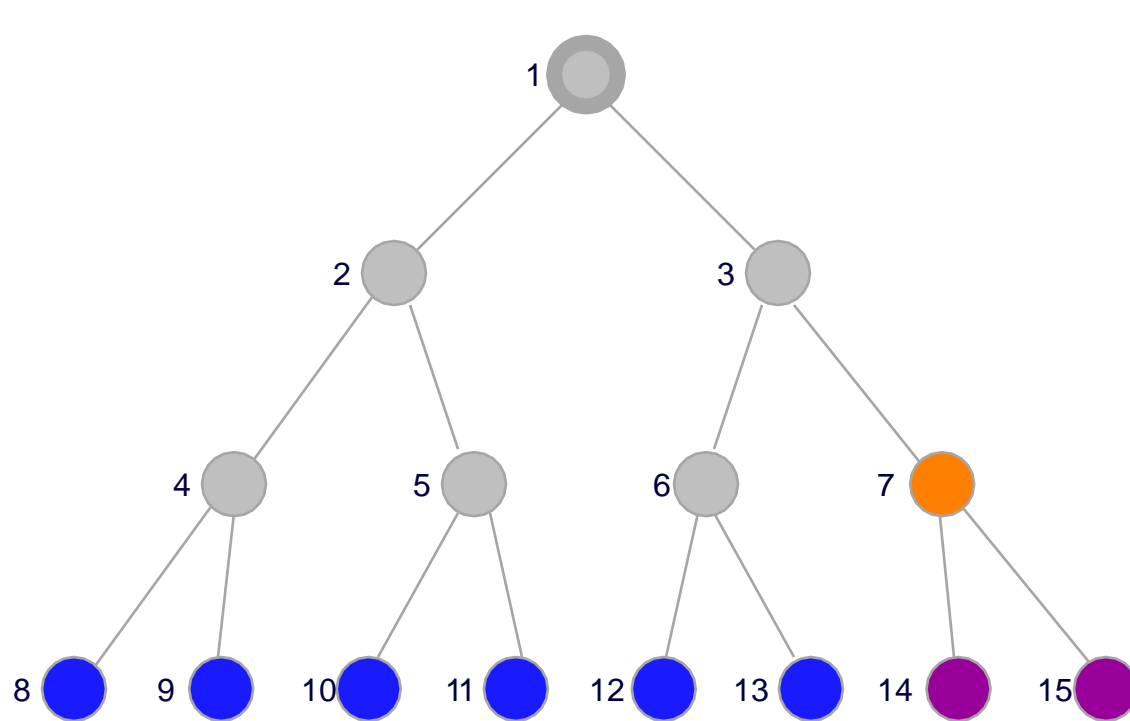
Fringe: [6,7,8,9] + [10,11]

Breadth-First Snapshot 6



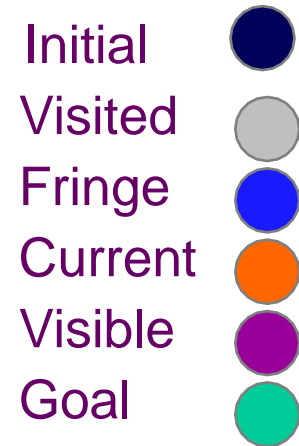
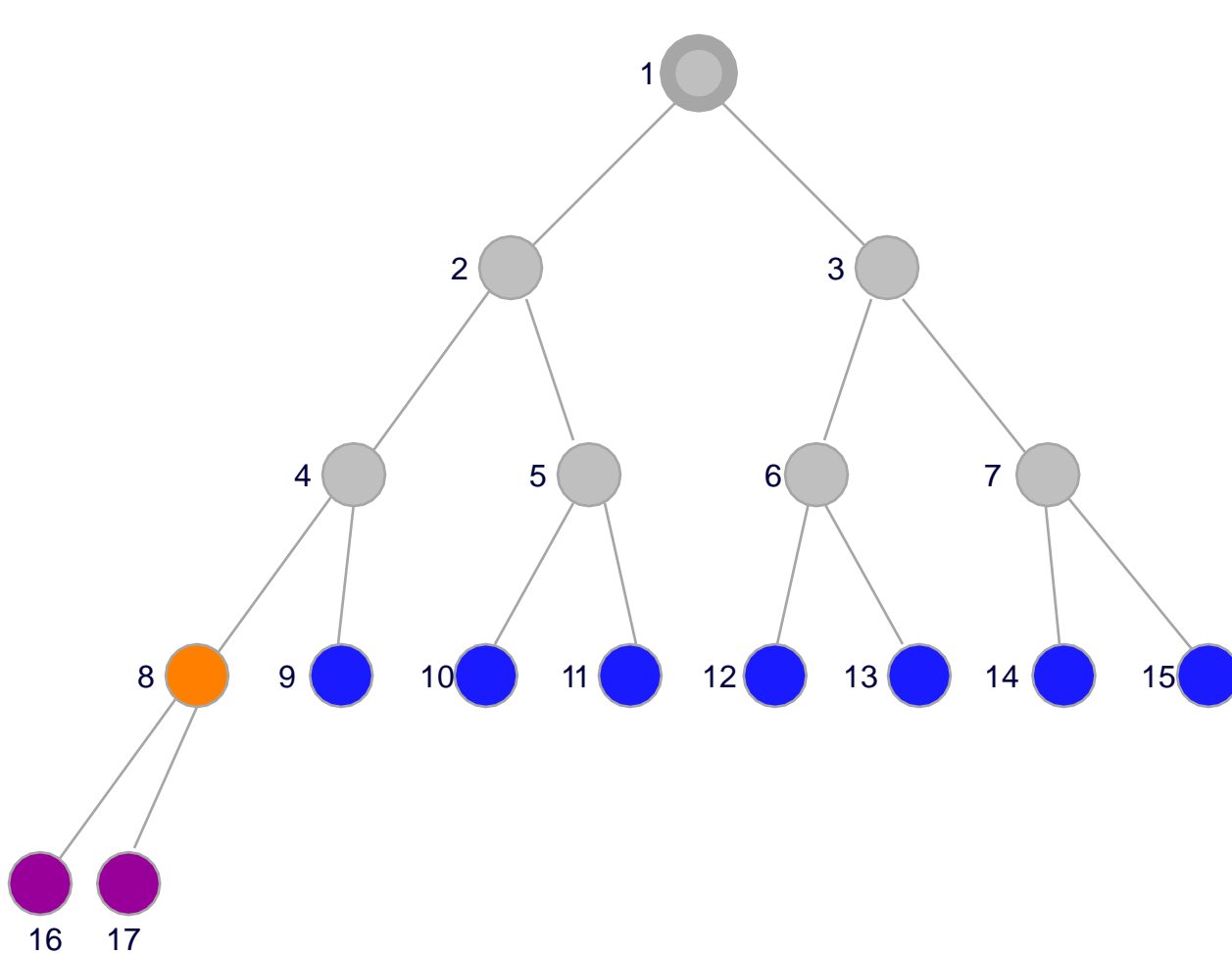
Fringe: [7,8,9,10,11] + [12,13]

Breadth-First Snapshot 7



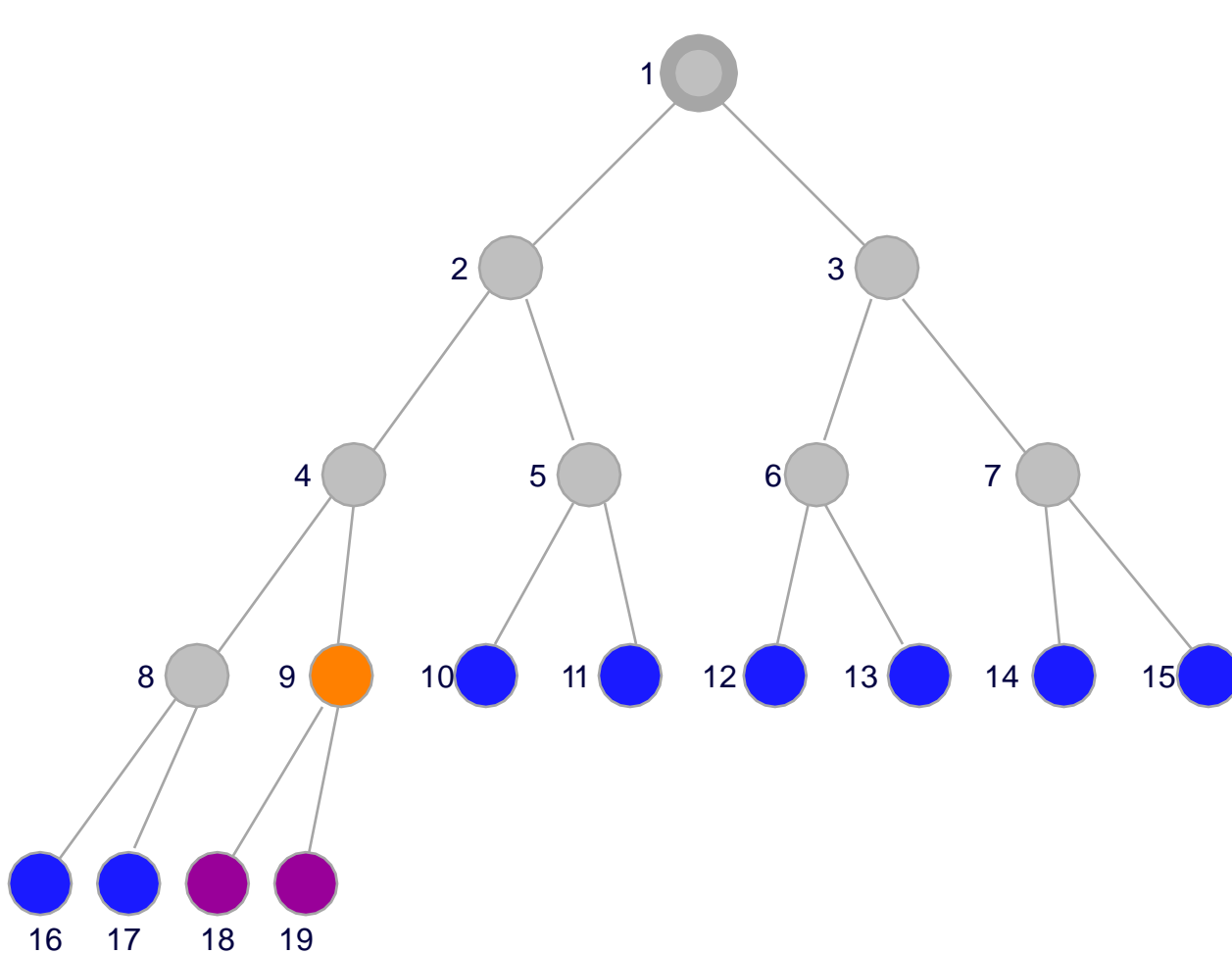
Fringe: [8,9,10,11,12,13] + [14,15]

Breadth-First Snapshot 8



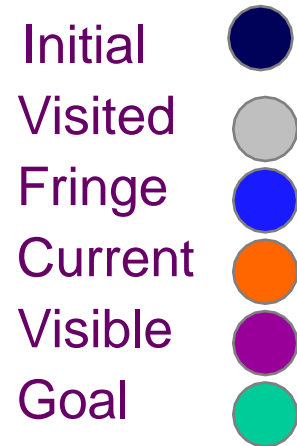
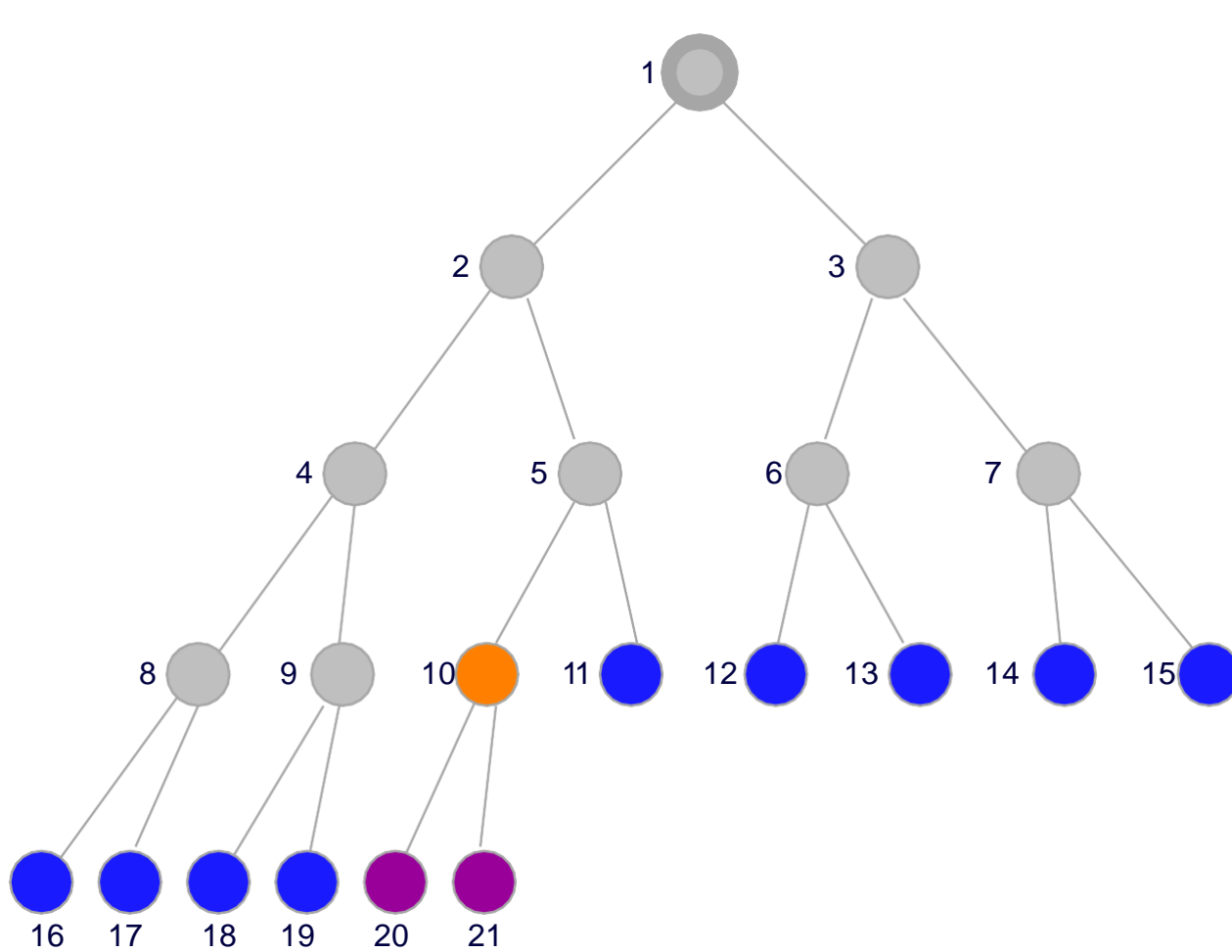
Fringe: [9,10,11,12,13,14,15] + [16,17]

Breadth-First Snapshot 9



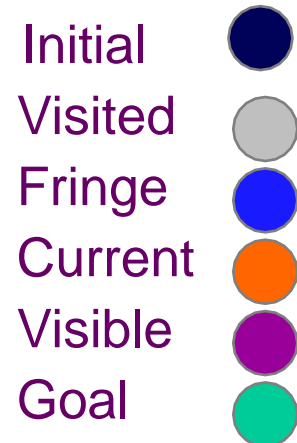
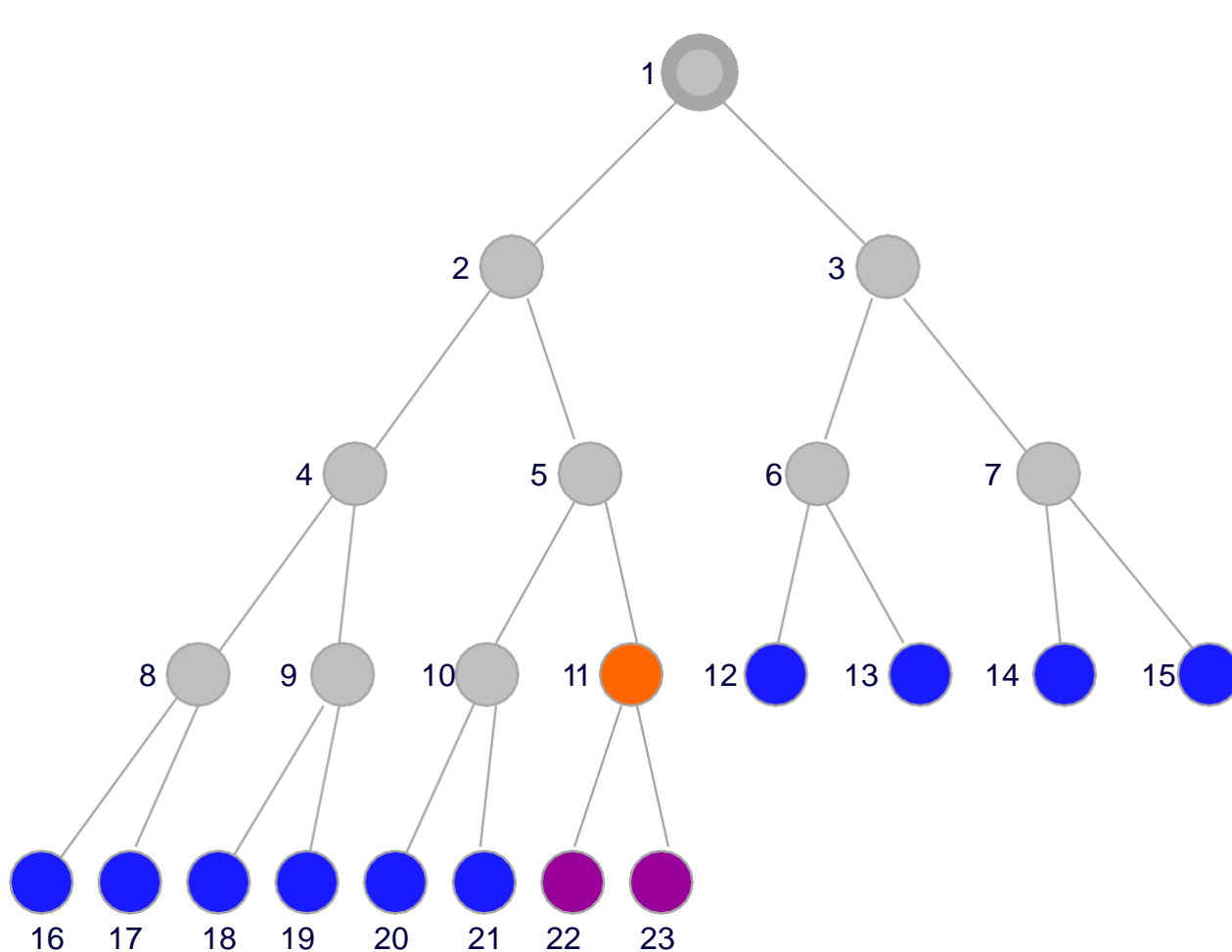
Fringe: [10,11,12,13,14,15,16,17] + [18,19]

Breadth-First Snapshot 10



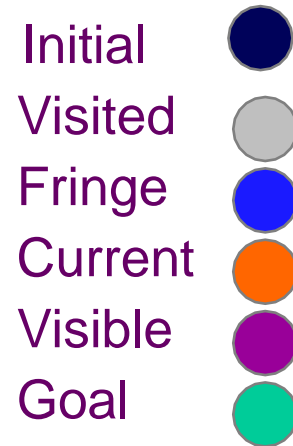
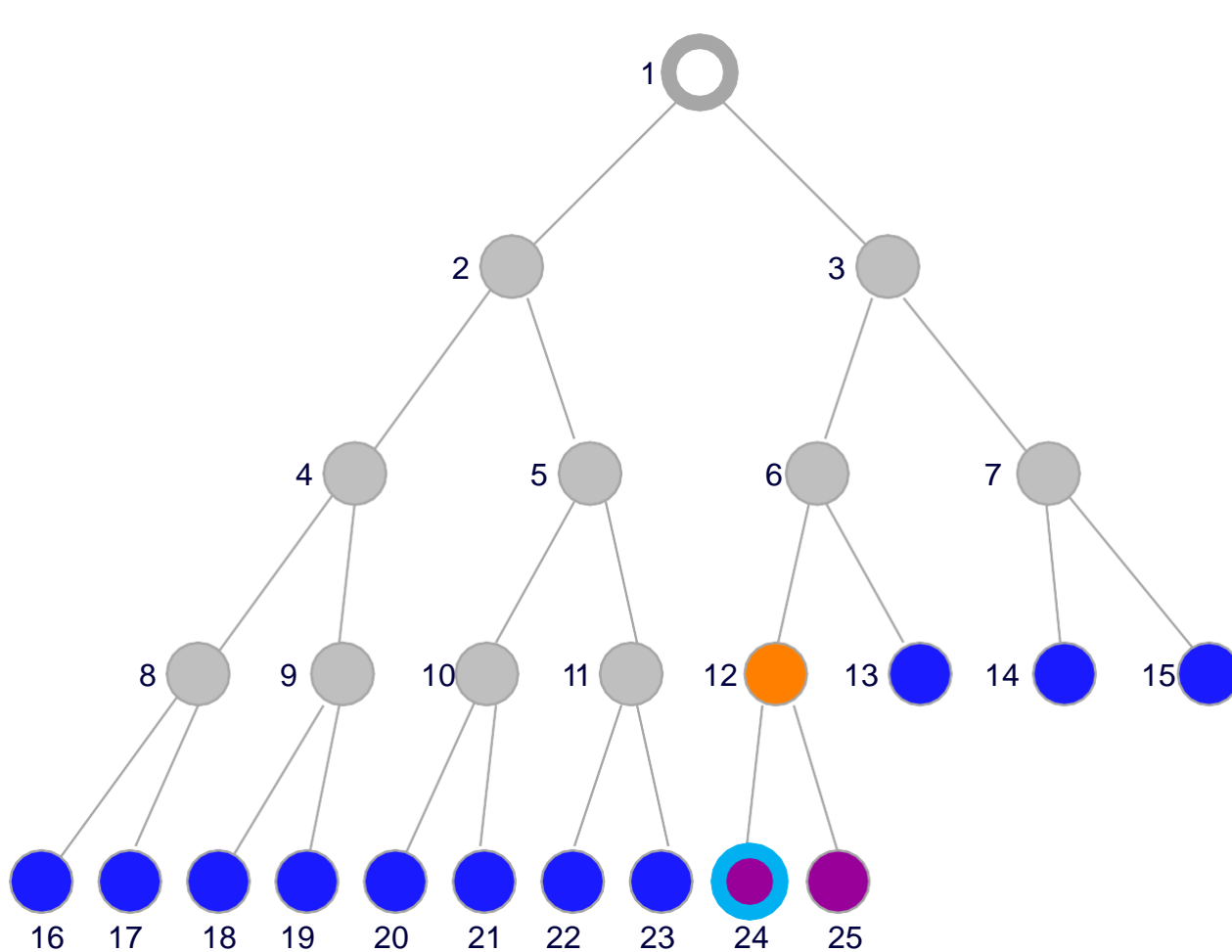
Fringe: [11,12,13,14,15,16,17,18,19] + [20,21]

Breadth-First Snapshot 11



Fringe: [12, 13, 14, 15, 16, 17, 18, 19, 20, 21] + [22,23]

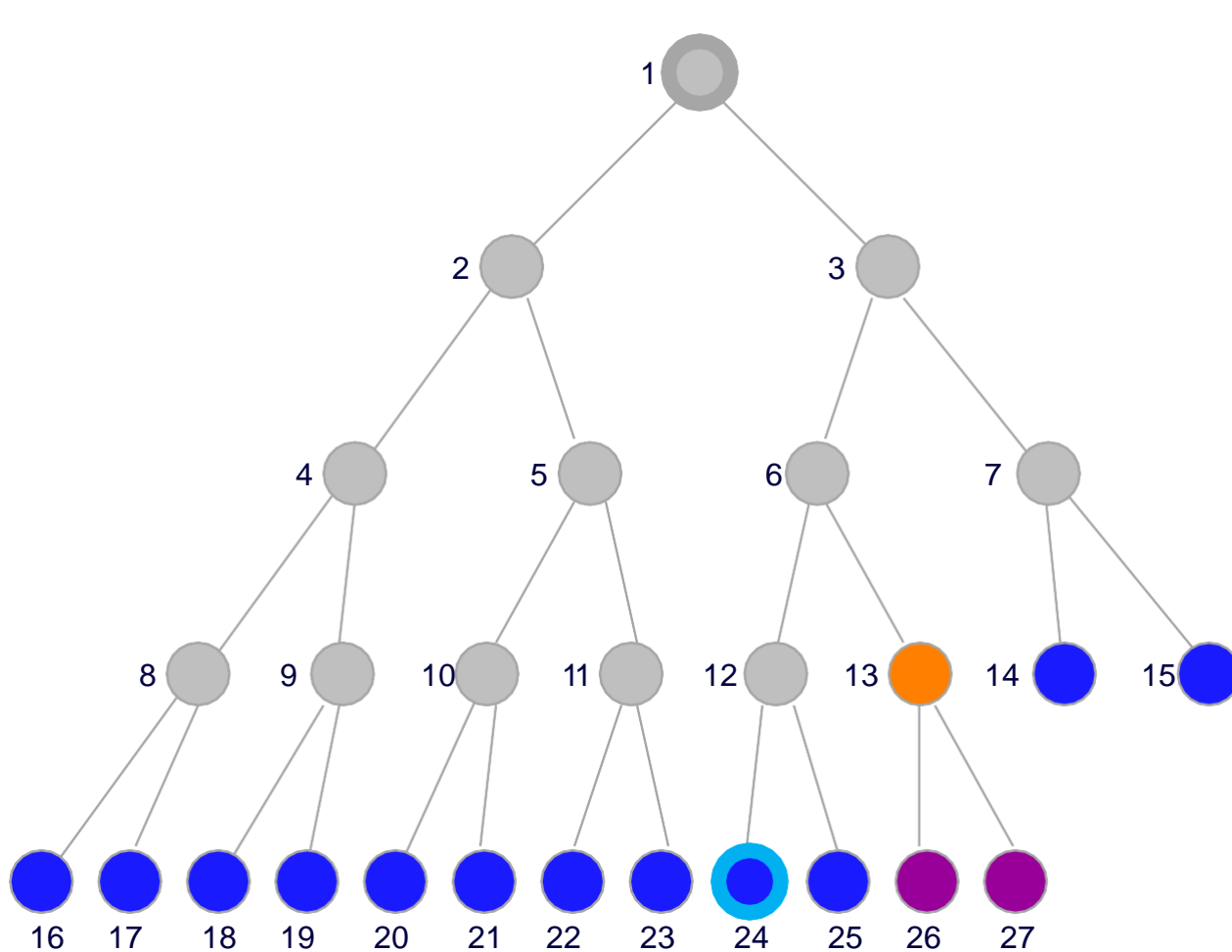
Breadth-First Snapshot 12



Note:
The goal node is "visible" here, but we can not perform the goal test yet.

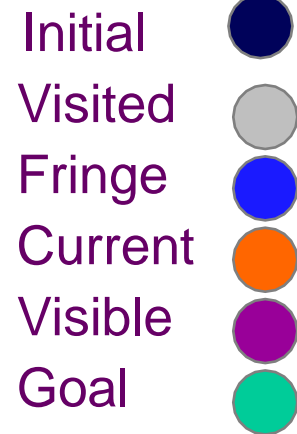
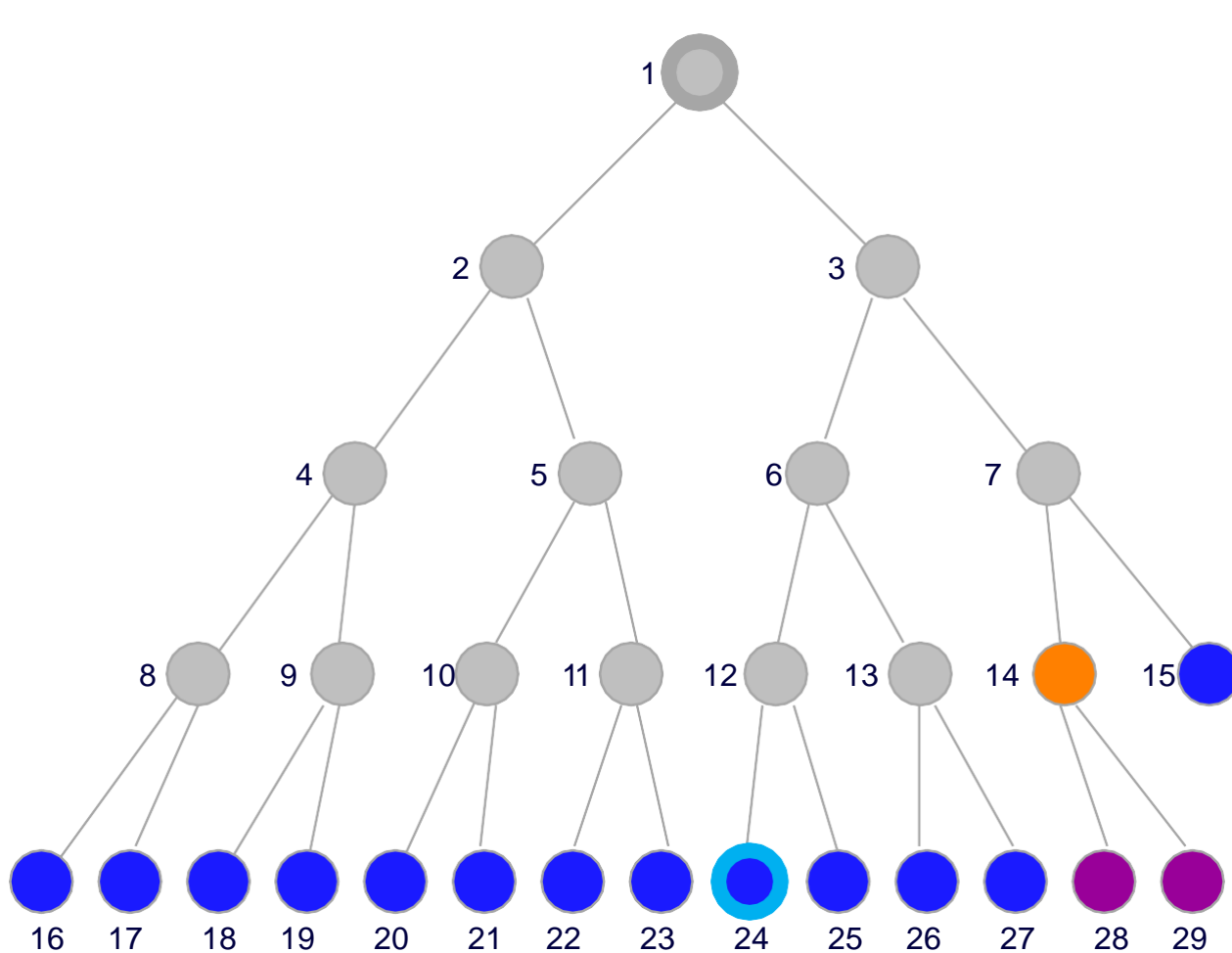
Fringe: [13,14,15,16,17,18,19,20,21] + [22,23]

Breadth-First Snapshot 13



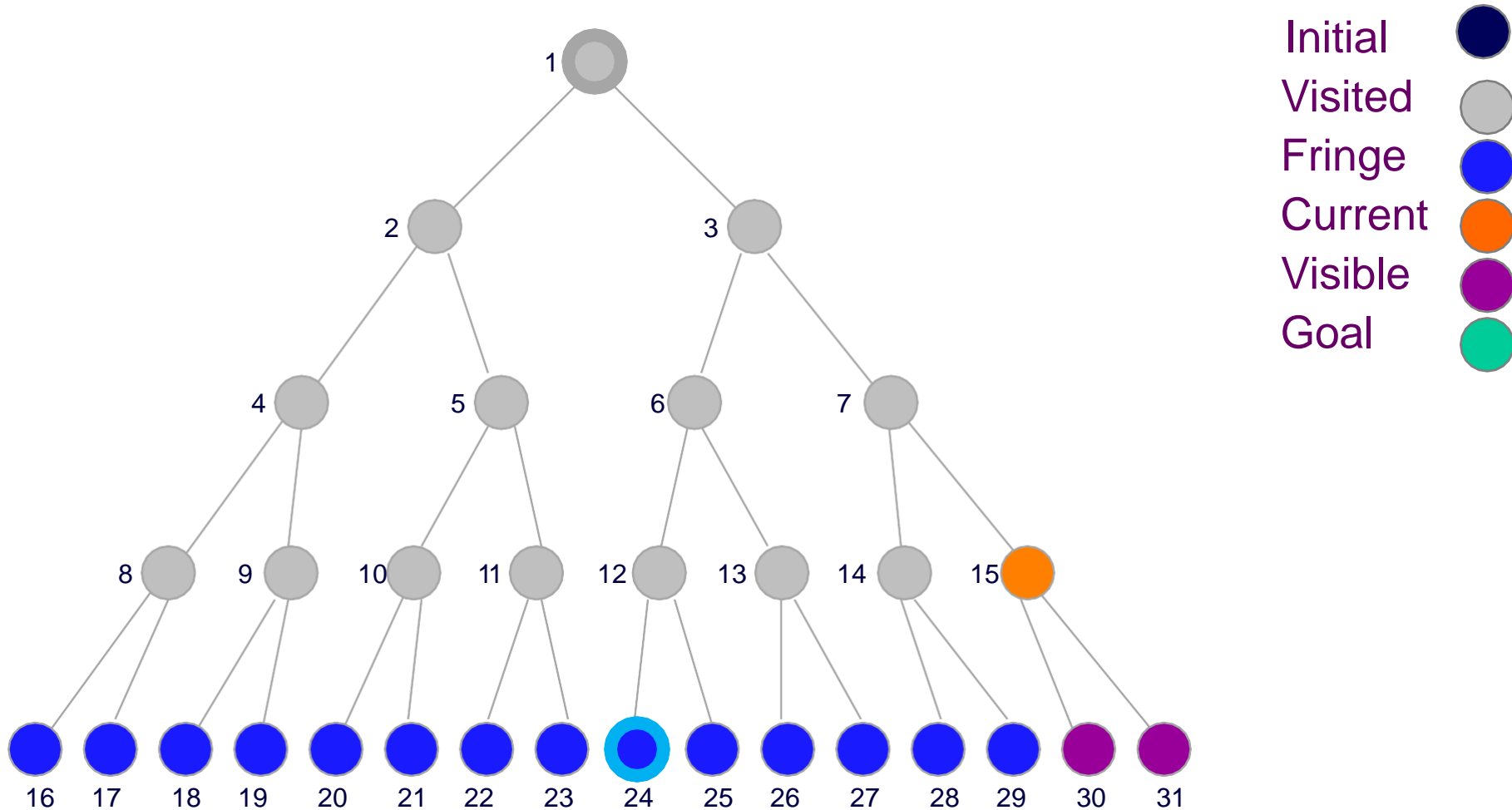
Fringe: [14,15,16,17,18,19,20,21,22,23,24,25] + [26,27]

Breadth-First Snapshot 14



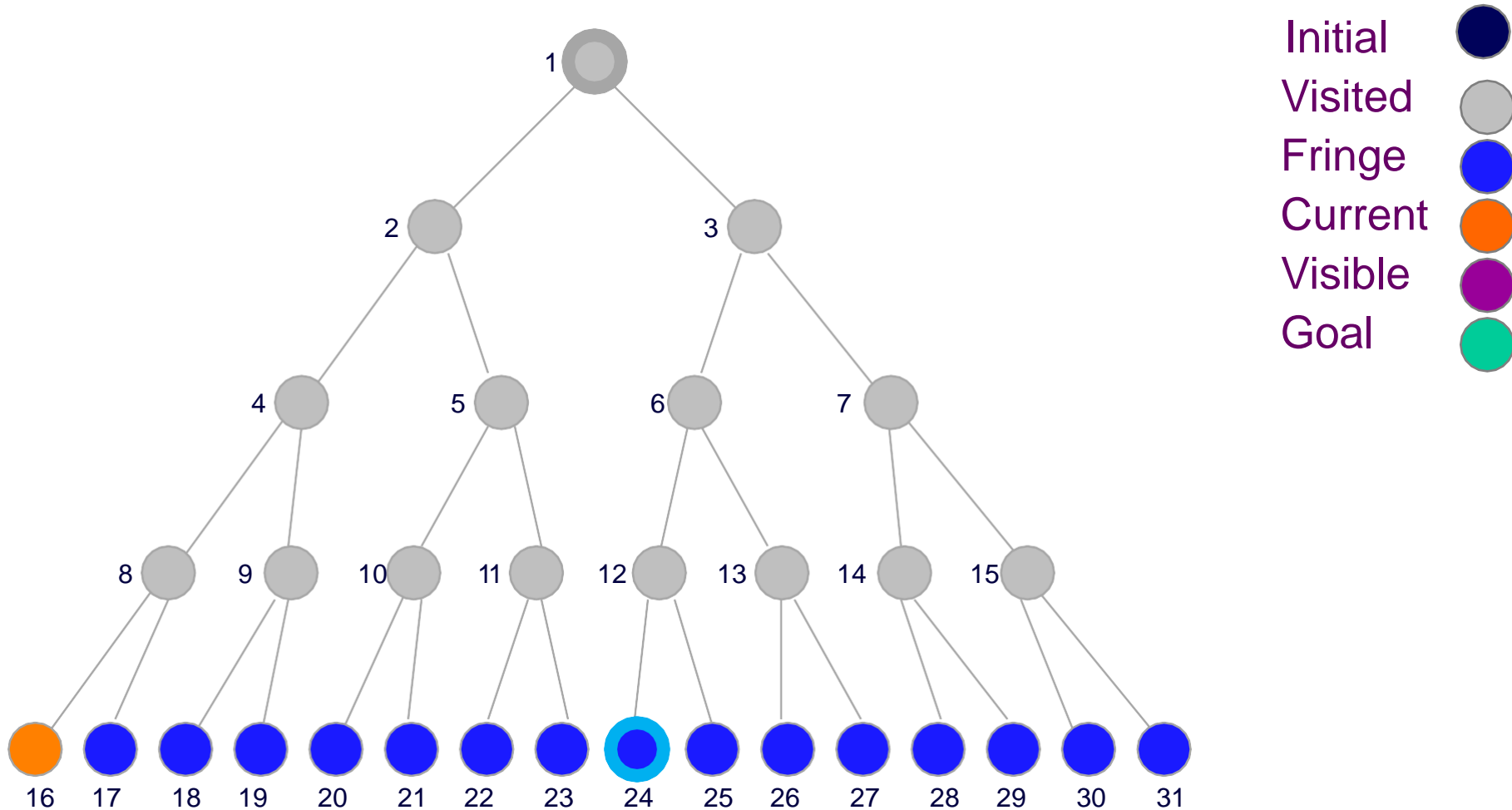
Fringe: [15,16,17,18,19,20,21,22,23,24,25,26,27] + [28,29]

Breadth-First Snapshot 15



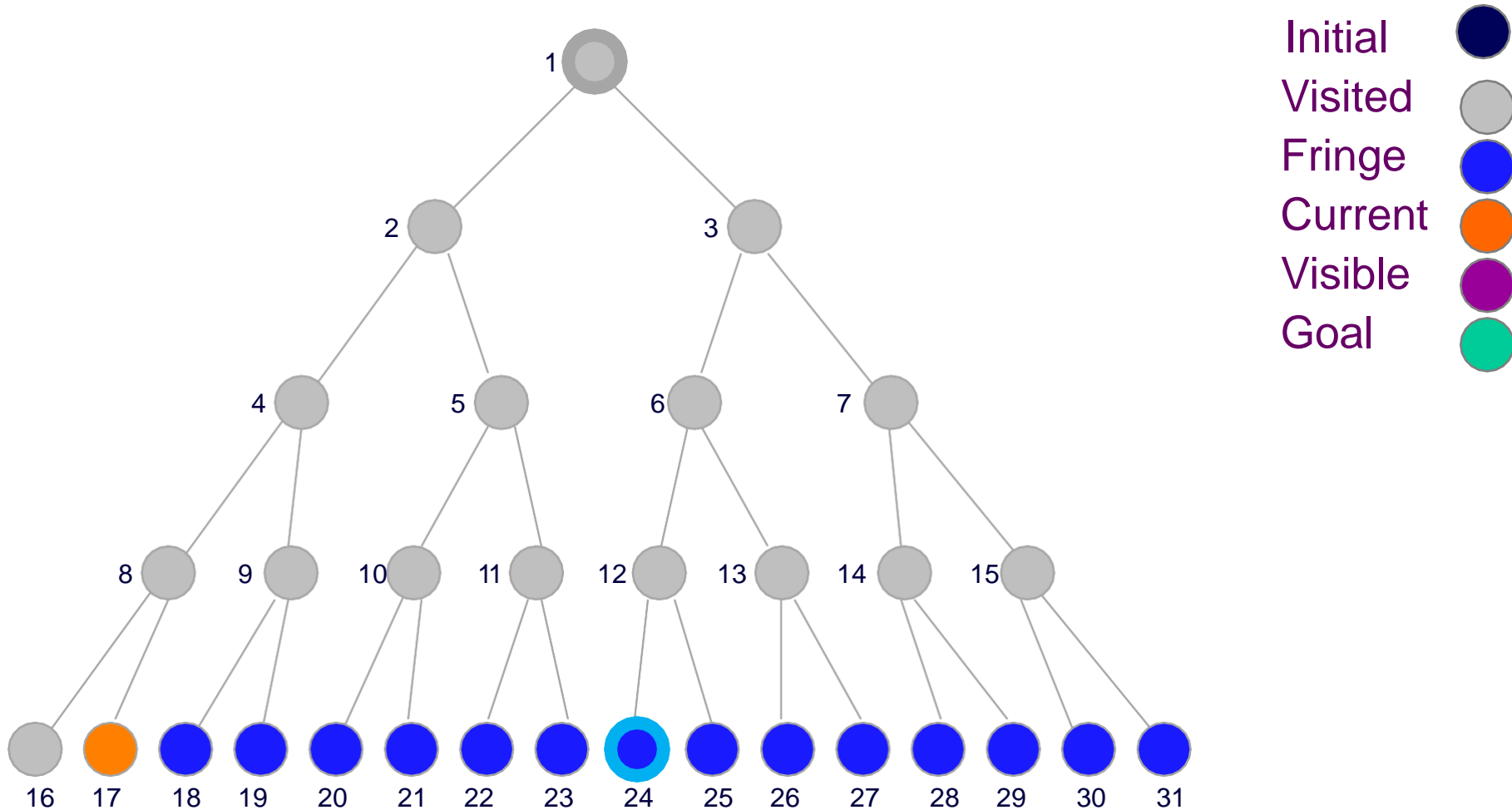
Fringe: [15,16,17,18,19,20,21,22,23,24,25,26,27,28,29] + [30,31]

Breadth-First Snapshot 16



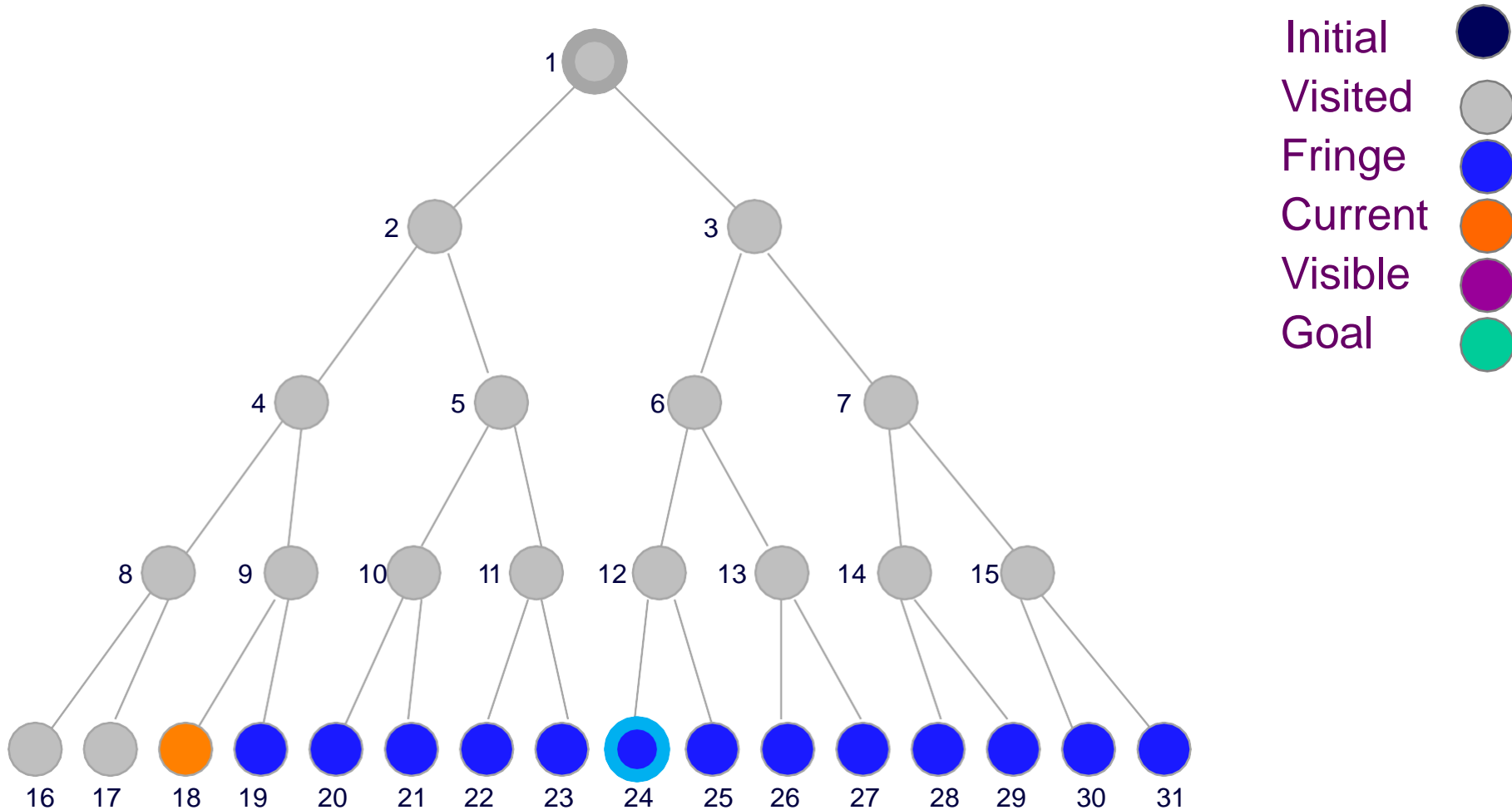
Fringe: [17,18,19,20,21,22,23,24,25,26,27,28,29,30,31]

Breadth-First Snapshot 17



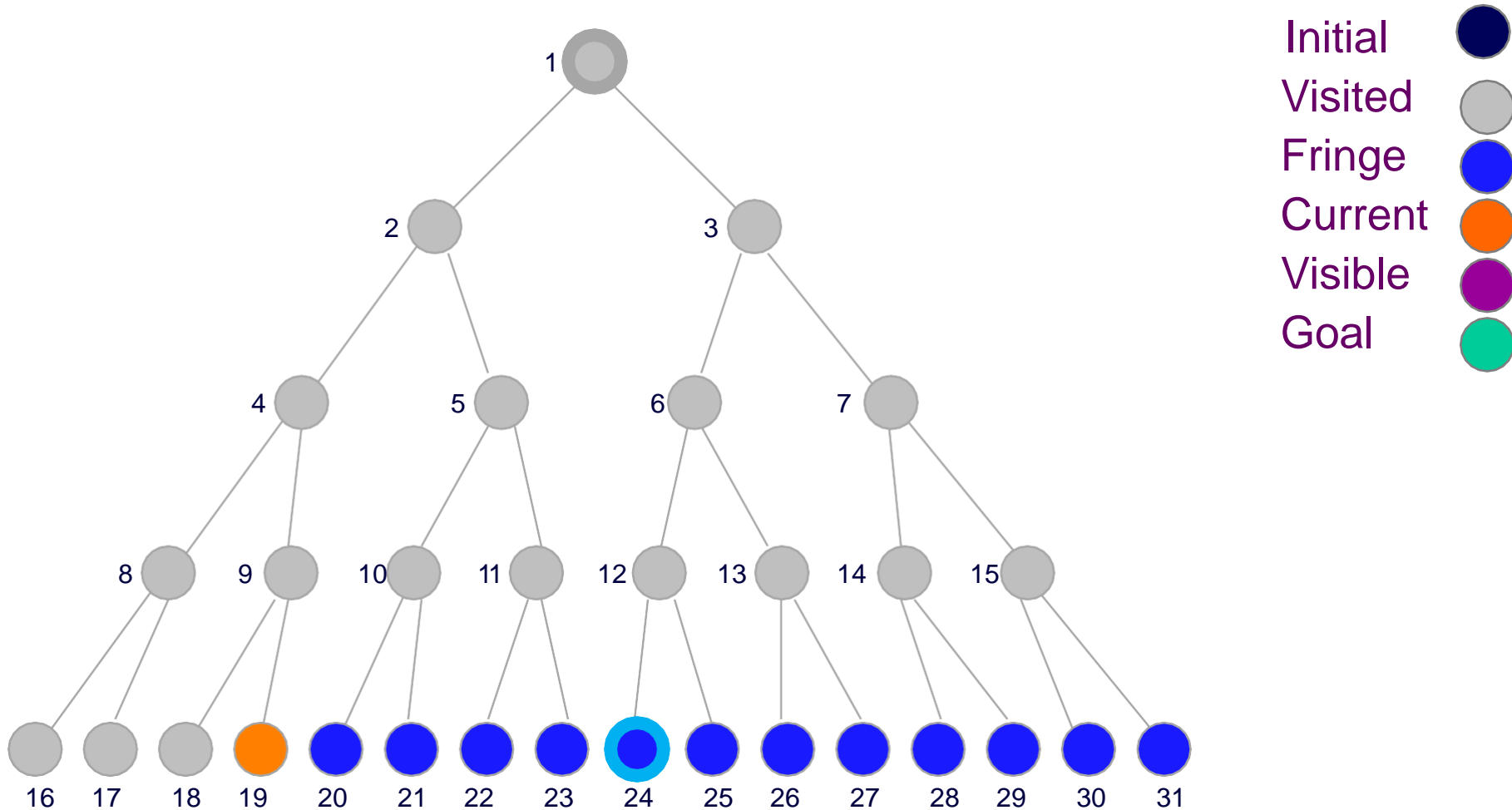
Fringe: [18,19,20,21,22,23,24,25,26,27,28,29,30,31]

Breadth-First Snapshot 18



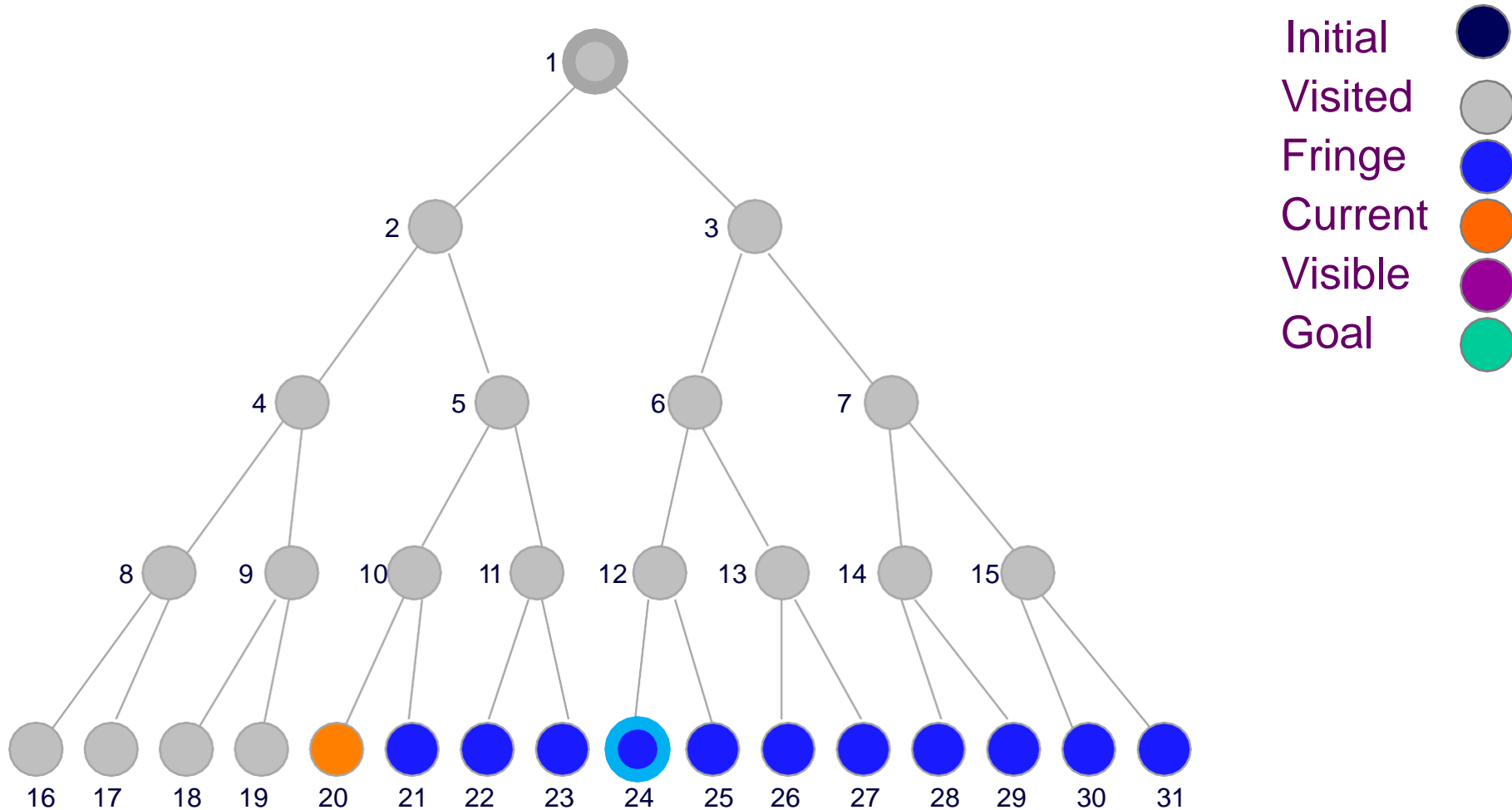
Fringe: [19,20,21,22,23,24,25,26,27,28,29,30,31]

Breadth-First Snapshot 19



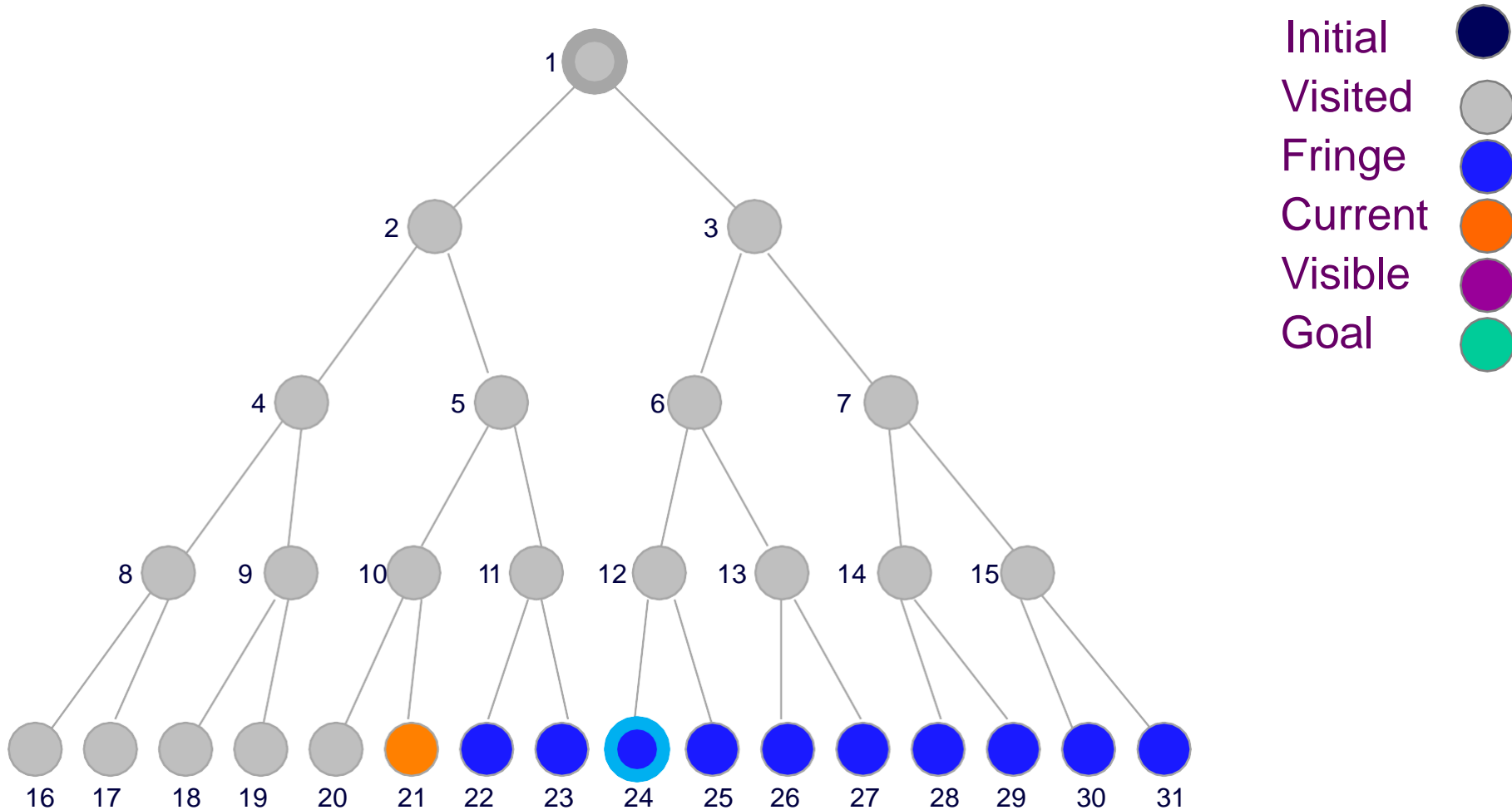
Fringe: [20,21,22,23,24,25,26,27,28,29,30,31]

Breadth-First Snapshot 20



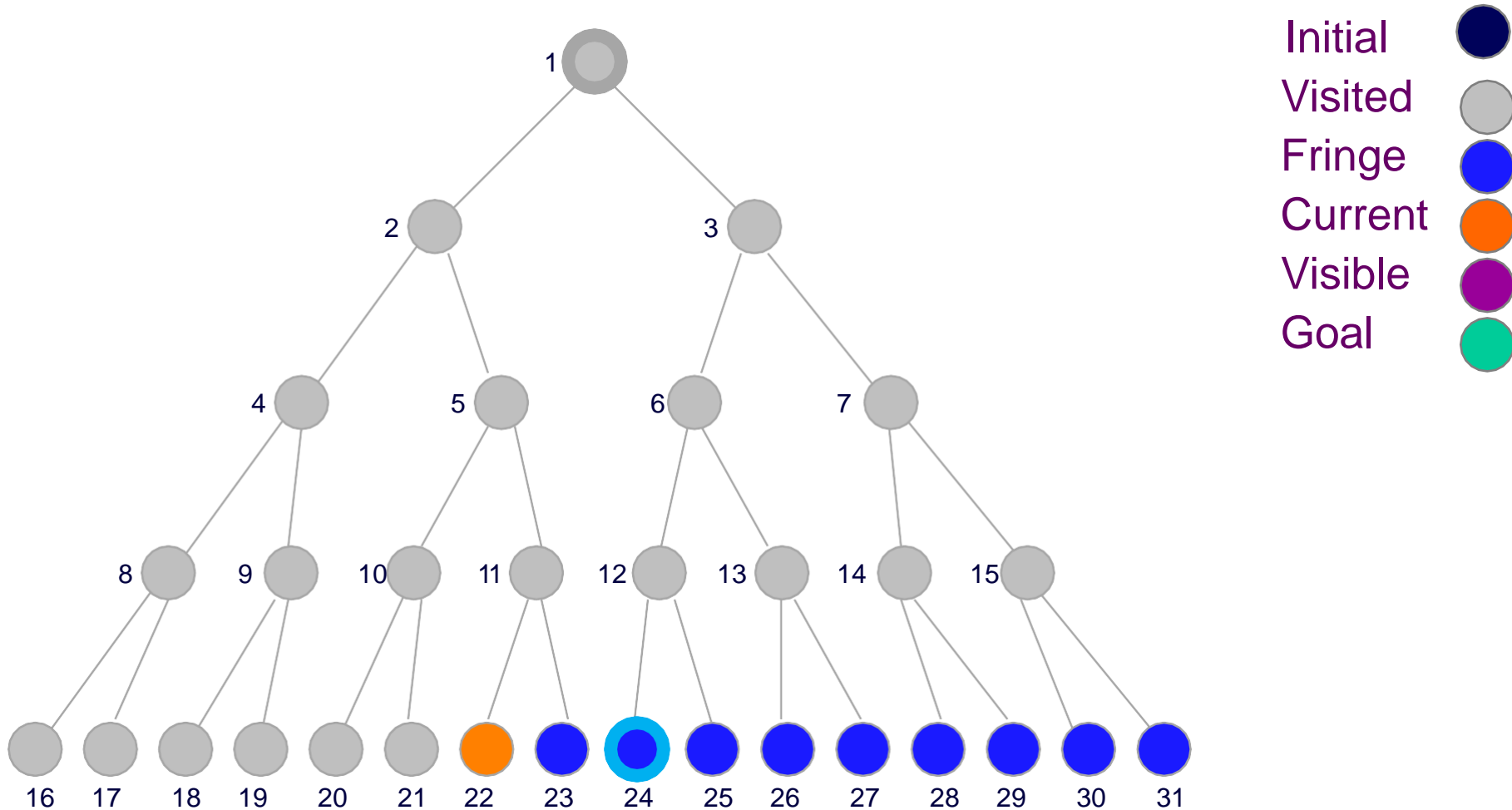
Fringe: [21,22,23,24,25,26,27,28,29,30,31]

Breadth-First Snapshot 21



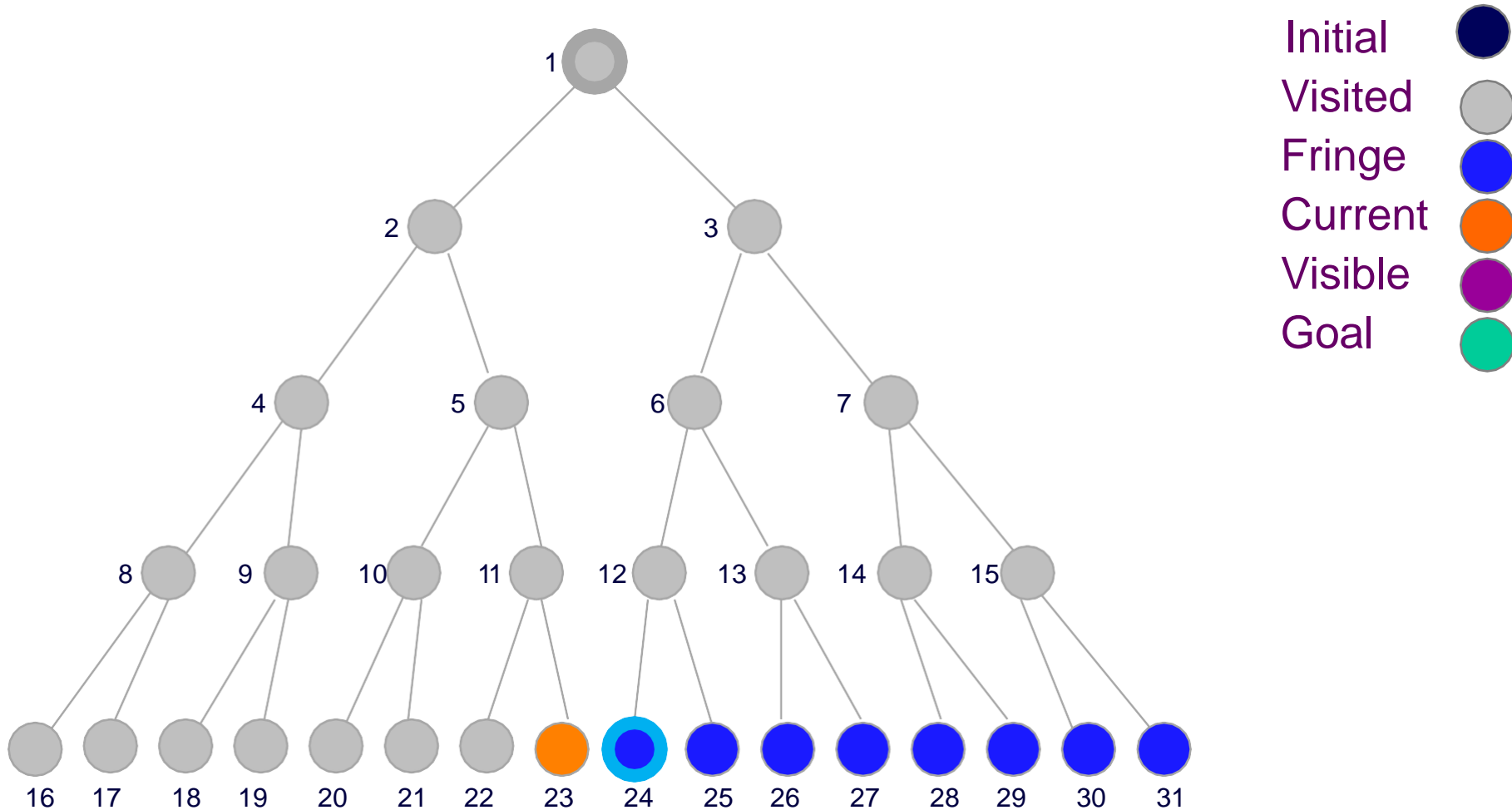
Fringe: [22,23,24,25,26,27,28,29,30,31]

Breadth-First Snapshot 22



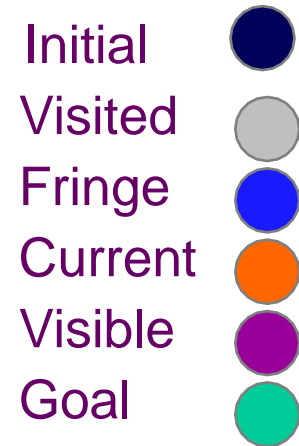
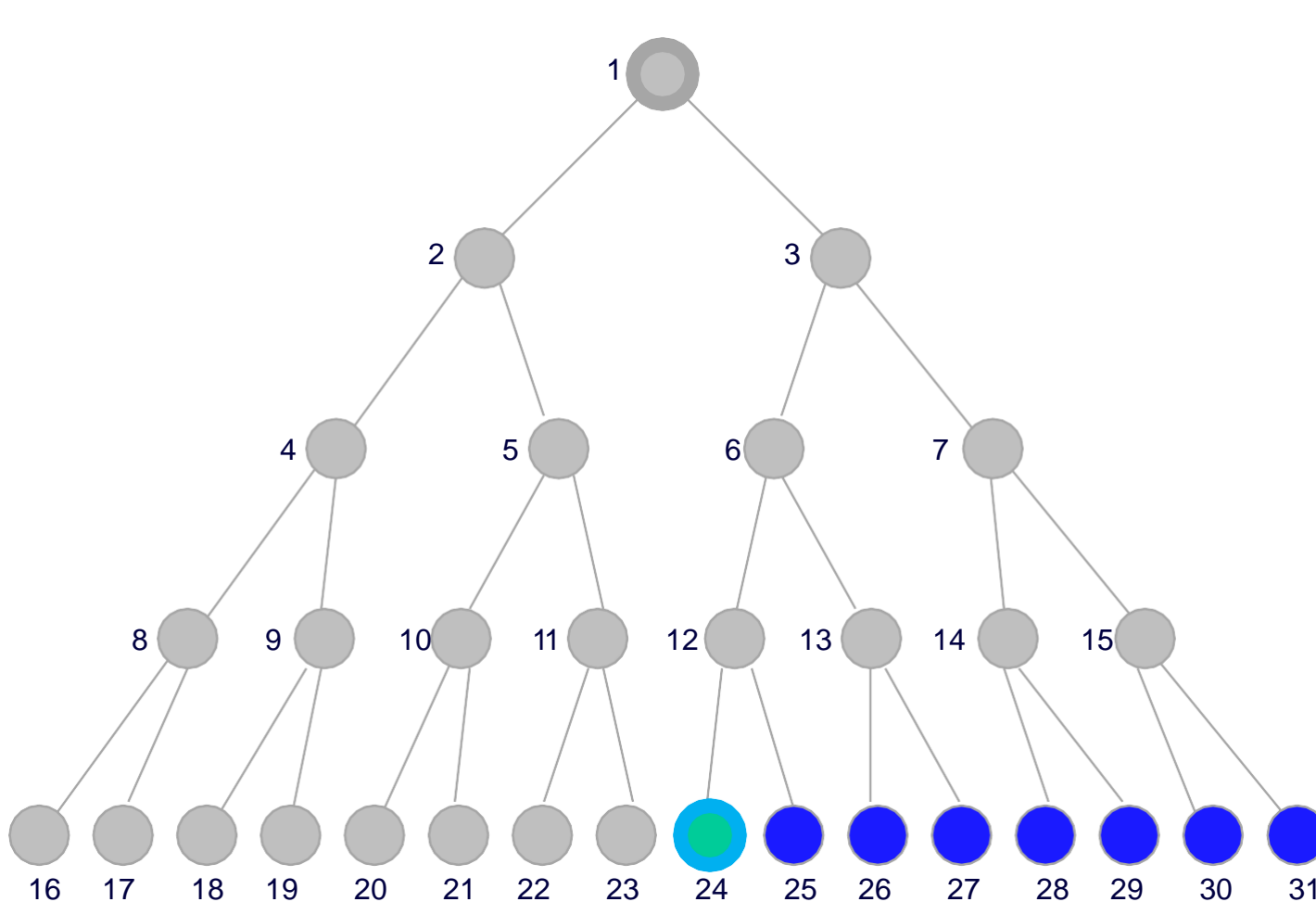
Fringe: [23,24,25,26,27,28,29,30,31]

Breadth-First Snapshot 23



Fringe: [24,25,26,27,28,29,30,31]

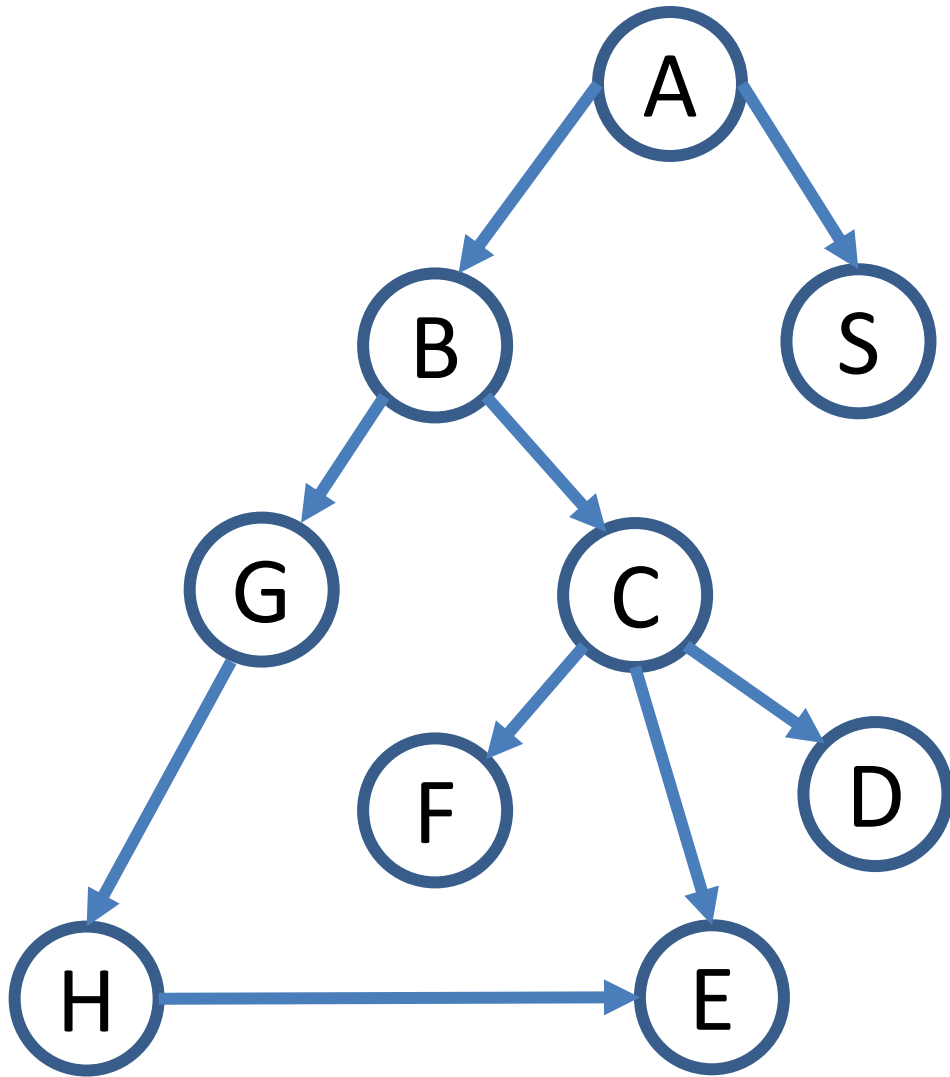
Breadth-First Snapshot 24



Note:
The goal test
is positive for
this node, and
a solution is
found in 24
steps.

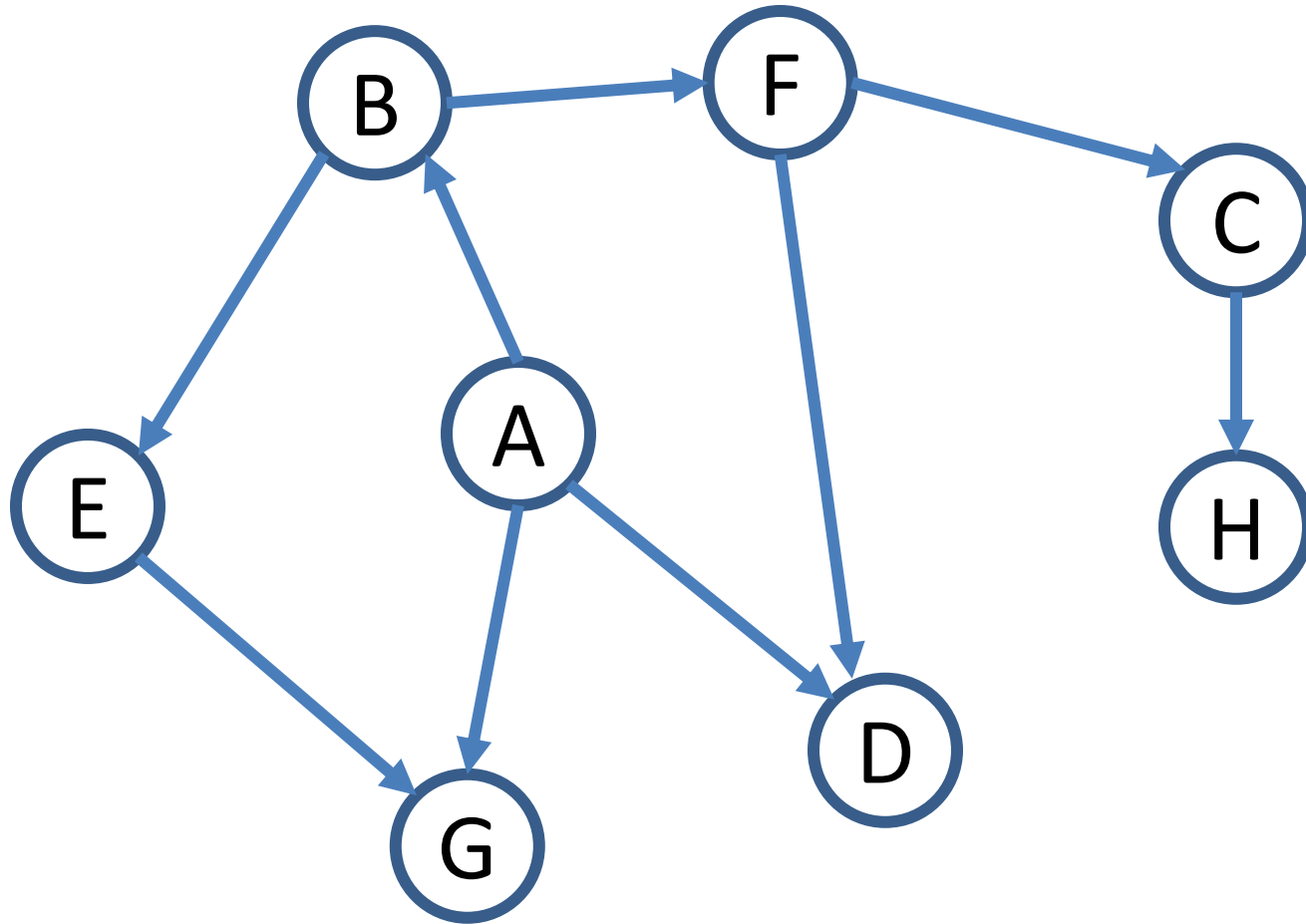
Fringe: [25,26,27,28,29,30,31]

Example BFS



Solve using BFS

ABDGEFCH



Properties of Breadth-First Search (BFS)

Completeness: Yes (if b is finite), a solution will be found if exists.

Time Complexity: (nodes until the solution)

Optimality: Yes

Criterion	Breadth-First
Complete?	Yes
Time	$O(b^{d+1})$
Space	$O(b^{d+1})$
Optimal?	Yes

b Branching Factor

d The depth of the goal

Suppose the branching factor $b=10$, and the goal is at depth $d=12$:

- Then we need $O10^{12}$ time to finish. If O is 0.001 second, then we need 1 billion seconds (31 year). And if each O costs 10 bytes to store, then we also need 1 terabytes.

➔ Not suitable for searching large graphs

2- Uniform-Cost -First

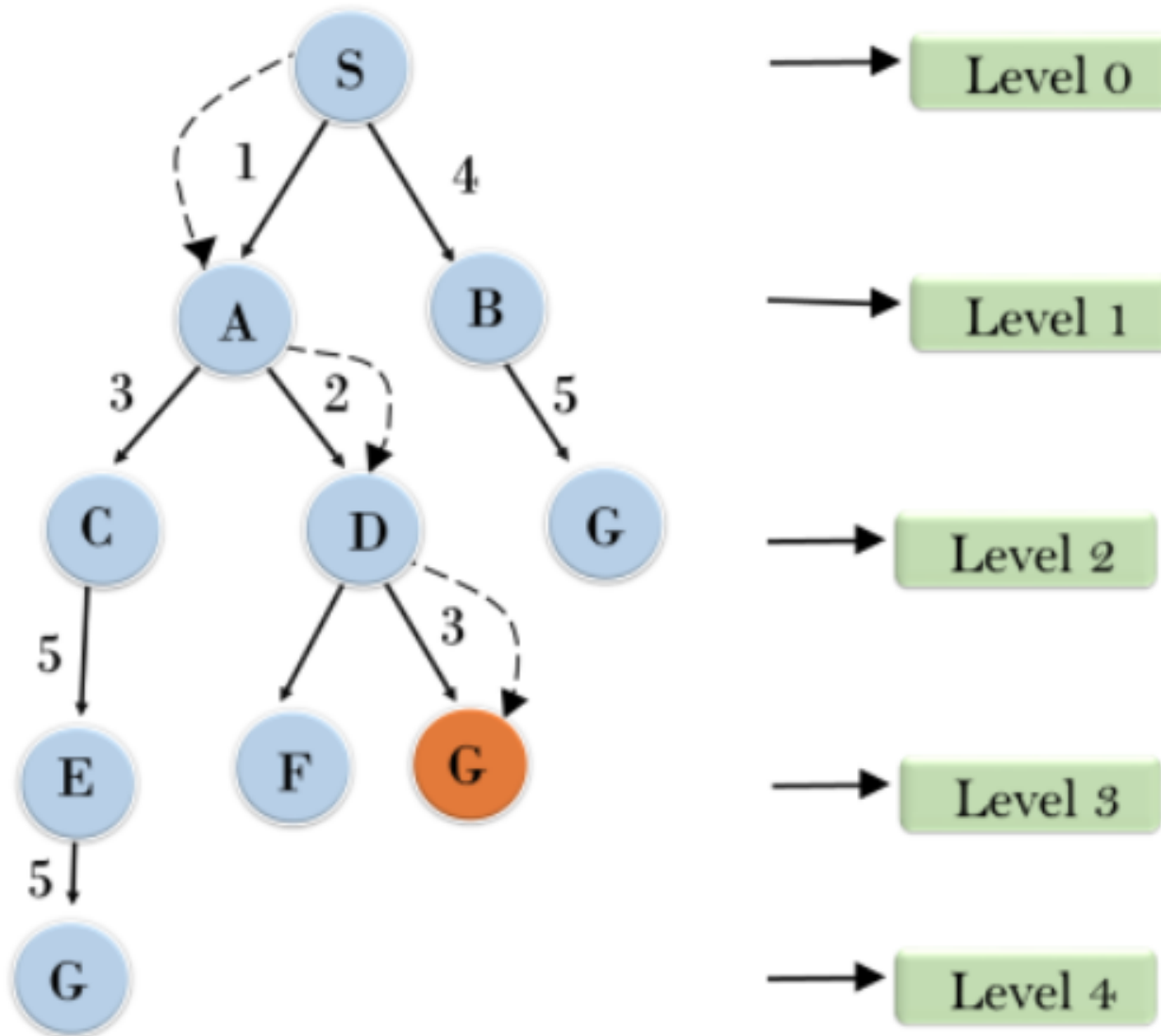
Uniform-Cost -First

Visits the next node which has the least total cost from the root, until a goal state is reached.

- Similar to BREADTH-FIRST, but with an evaluation of the cost for each reachable node.
- $g(n)$ = path cost(n) = sum of individual edge costs to reach the current node.

Uniform-Cost -First

Uniform Cost Search

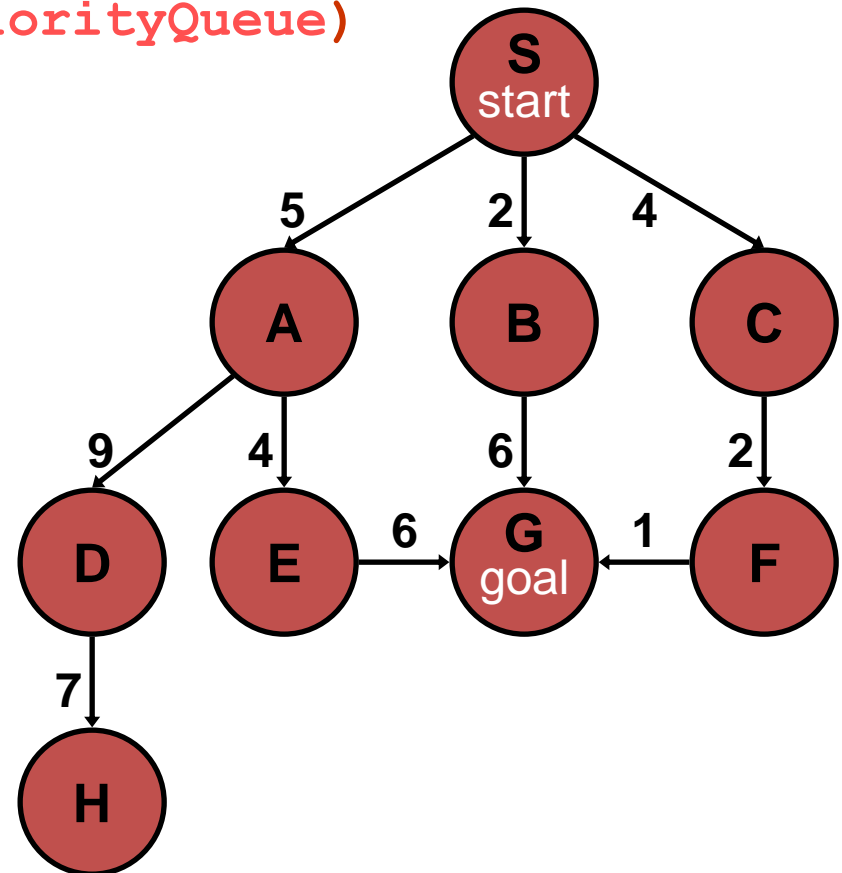


Uniform-Cost Search (UCS)

`generalSearch(problem, priorityQueue)`

of nodes tested: 0, expanded: 0

Closed Node	Open Node
	{S}

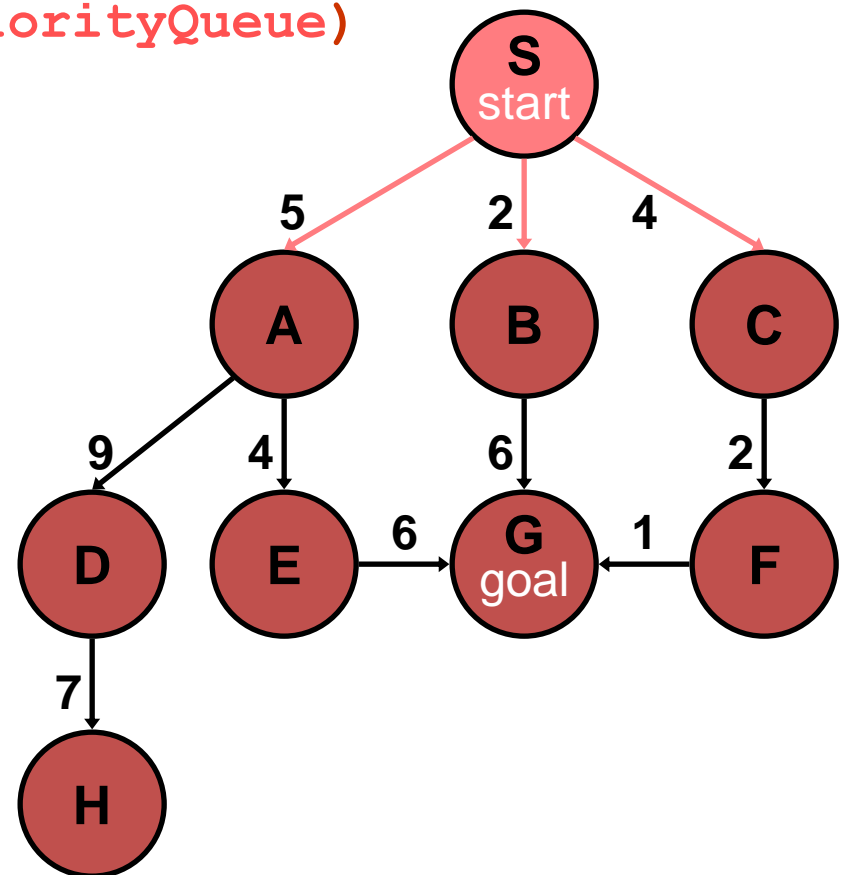


Uniform-Cost Search (UCS)

generalSearch(problem, priorityQueue)

of nodes tested: 1, expanded: 1

Closed Node	Open Node
	{S:0}
S not goal	{B:2,C:4,A:5}

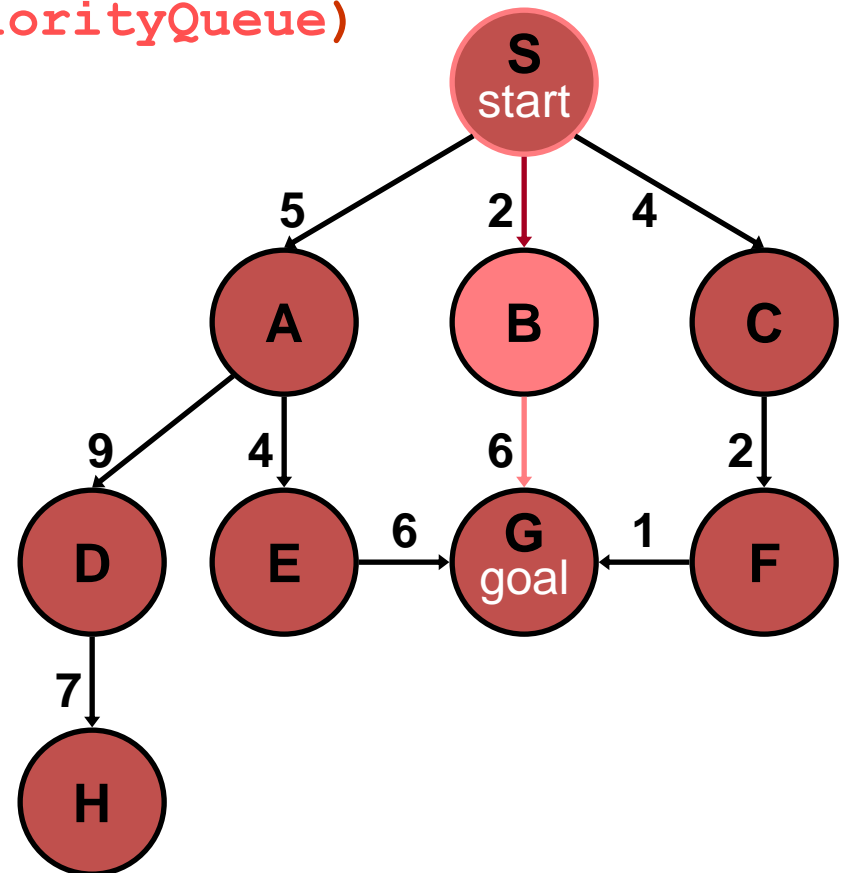


Uniform-Cost Search (UCS)

```
generalSearch(problem, priorityQueue)
```

of nodes tested: 2, expanded: 2

Closed Node	Open Node
	{S}
S	{B:2,C:4,A:5}
B not goal	{C:4,A:5,G:2+6}

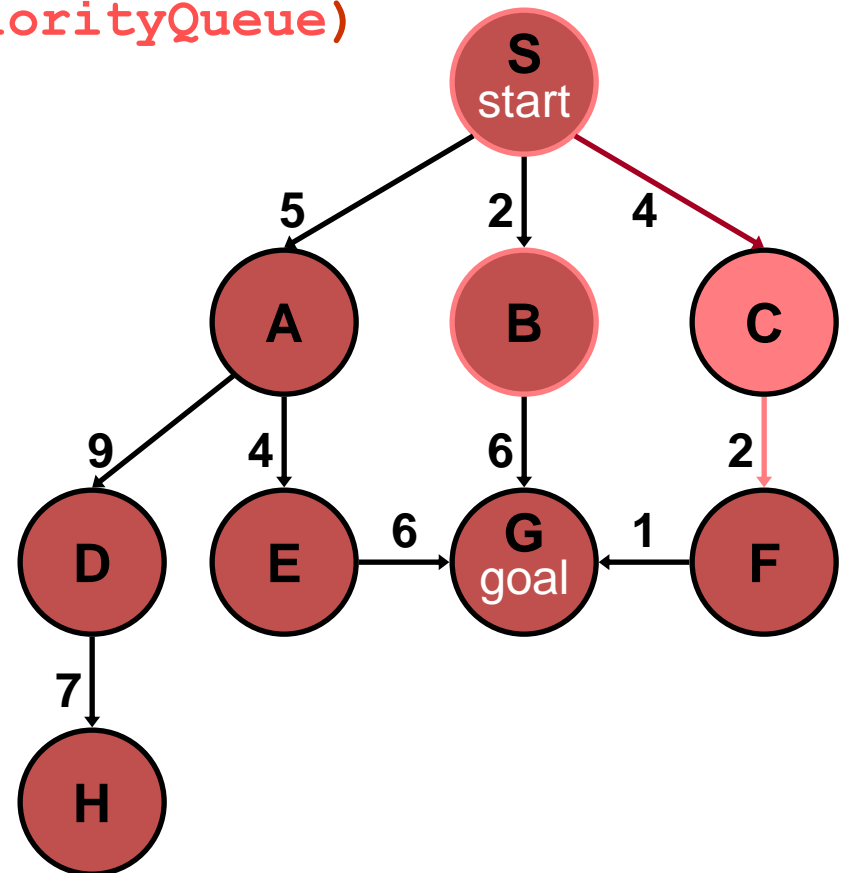


Uniform-Cost Search (UCS)

generalSearch(problem, priorityQueue)

of nodes tested: 3, expanded: 3

Closed Node	Open Node
	{S}
S	{B:2,C:4,A:5}
B	{C:4,A:5,G:8}
C not goal	{A:5,F:4+2,G:8}

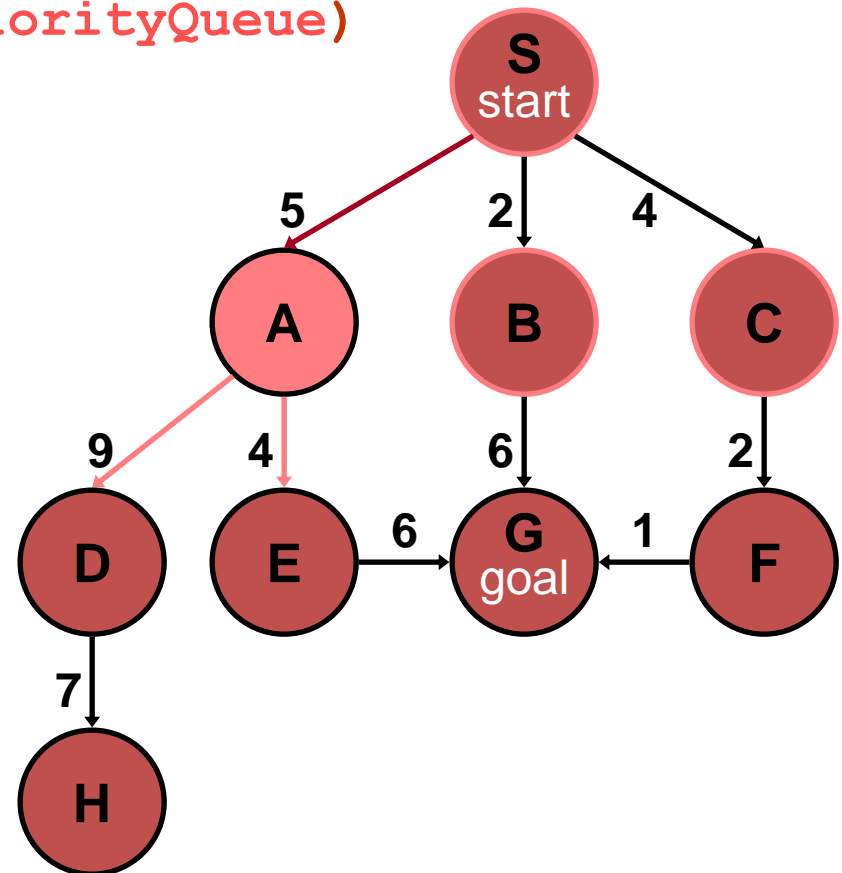


Uniform-Cost Search (UCS)

generalSearch(problem, priorityQueue)

of nodes tested: 4, expanded: 4

Closed Node	Open Node
	{S}
S	{B:2,C:4,A:5}
B	{C:4,A:5,G:8}
C	{A:5,F:6,G:8}
A not goal	{F:6,G:8,E:5+4, D:5+9}

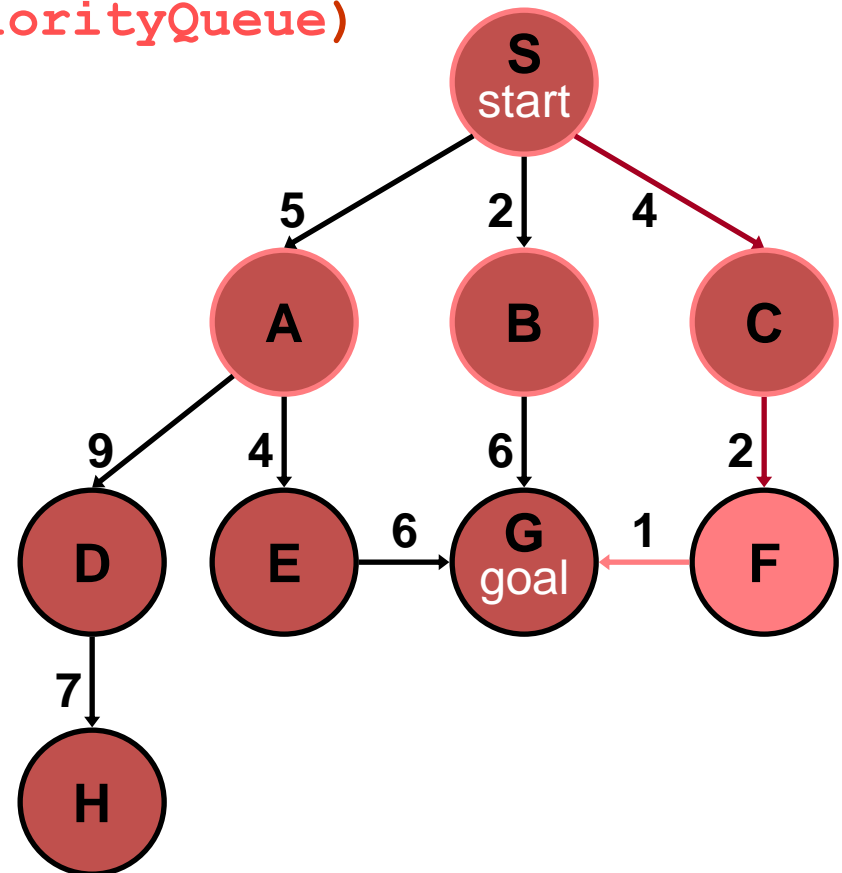


Uniform-Cost Search (UCS)

generalSearch(problem, priorityQueue)

of nodes tested: 5, expanded: 5

Closed Node	Open Node
	{S}
S	{B:2,C:4,A:5}
B	{C:4,A:5,G:8}
C	{A:5,F:6,G:8}
A	{F:6,G:8,E:9,D:14}
F not goal	{G:4+2+1,G:8,E:9,D:14}

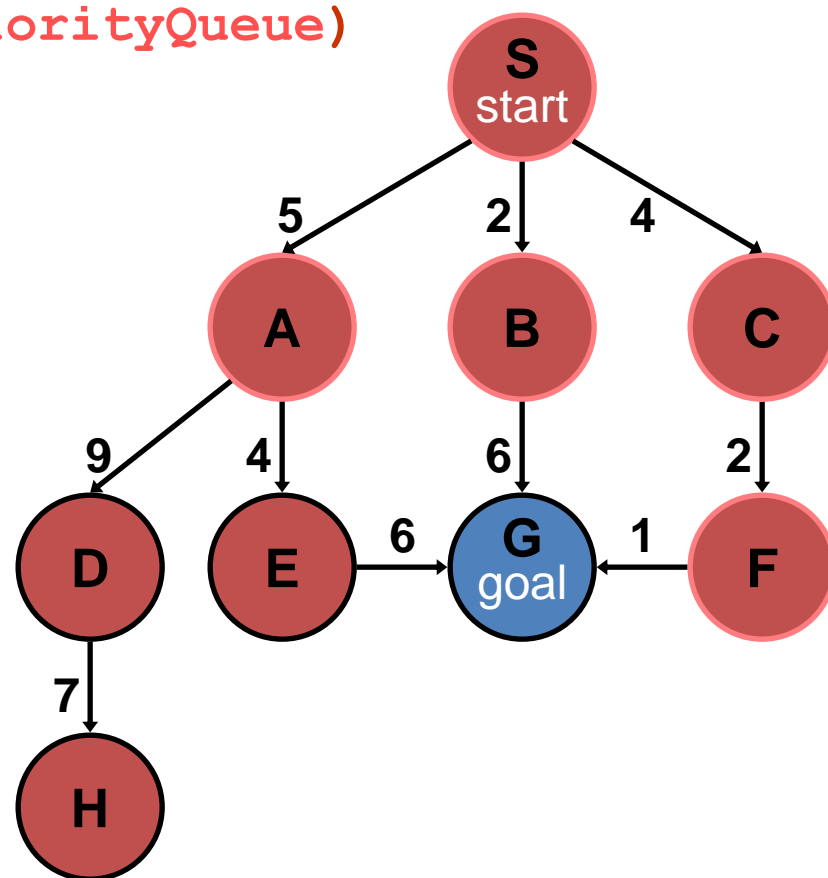


Uniform-Cost Search (UCS)

generalSearch(problem, priorityQueue)

of nodes tested: 6, expanded: 5

Closed Node	Open Node
	{S}
S	{B:2,C:4,A:5}
B	{C:4,A:5,G:8}
C	{A:5,F:6,G:8}
A	{F:6,G:8,E:9,D:14}
F	{G:7,G:8,E:9,D:14}
G goal	{G:8,E:9,D:14} no expand

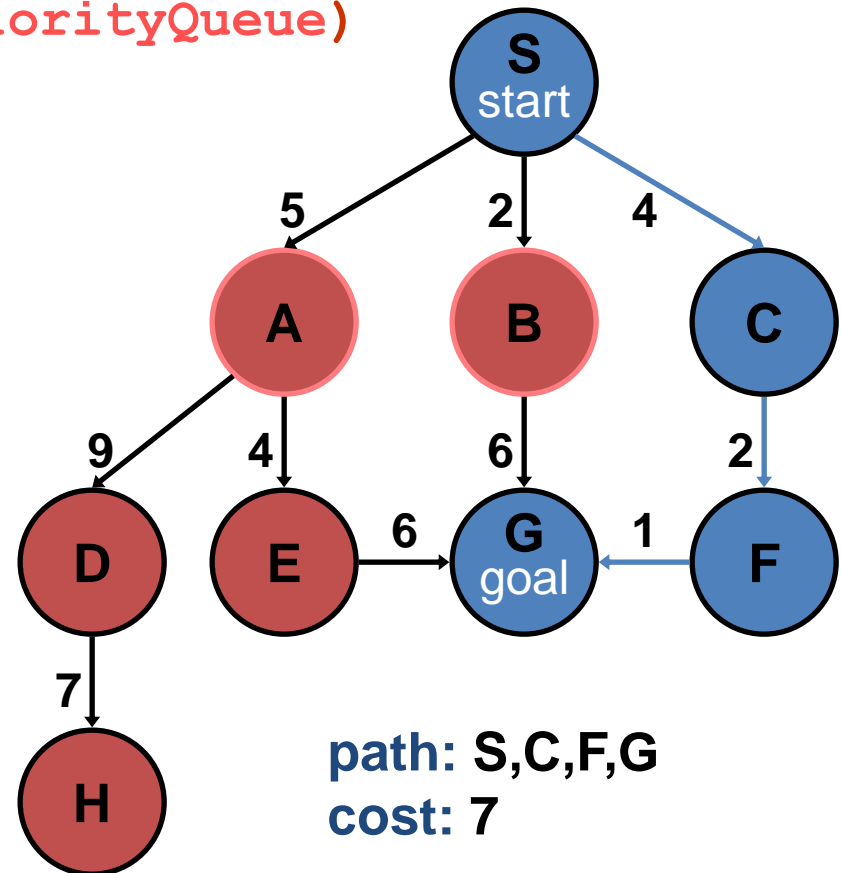


Uniform-Cost Search (UCS)

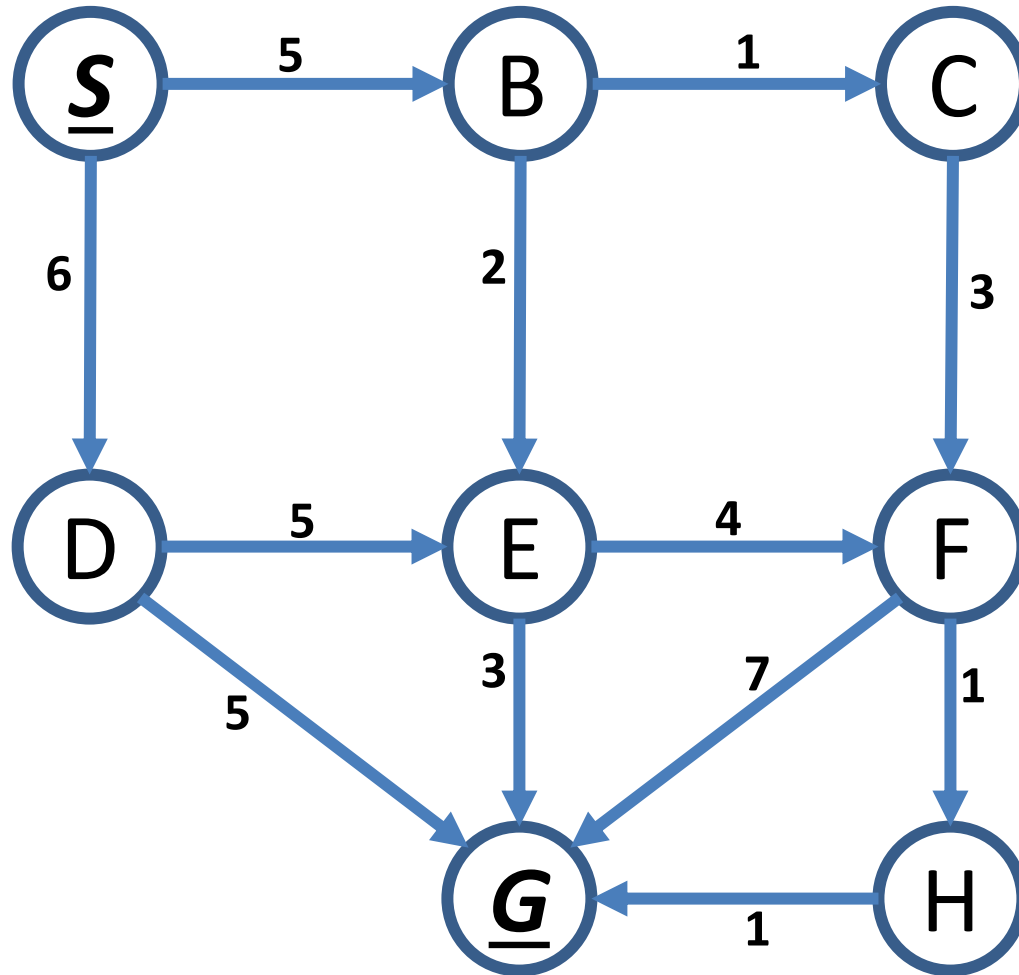
generalSearch(problem, priorityQueue)

of nodes tested: 6, expanded: 5

Closed Node	Open Node
	{S}
S	{B:2,C:4,A:5}
B	{C:4,A:5,G:8}
C	{A:5,F:6,G:8}
A	{F:6,G:8,E:9,D:14}
F	{G:7,G:8,E:9,D:14}
G	{G:8,E:9,D:14}



Uniform-Cost Search (UCS)



Properties of Uniform-cost Search (UCS)

Completeness Yes (if b is finite, and step cost is positive)

Time Complexity much larger than b^d , and just b^d if all steps have the same cost.

Space Complexity: as above

Optimality: Yes

Criterion	Uniform-Cost
Complete?	Yes
Time	$O(b^{\lceil C^*/\epsilon \rceil})$
Space	$O(b^{\lceil C^*/\epsilon \rceil})$
Optimal?	Yes

b Branching Factor

d Depth of the goal/tree

Requires that the goal test being applied when a node is removed from the nodes list rather than when the node is first generated while its parent node is expanded.

Breadth-First vs. Uniform-Cost

Breadth-first search (BFS) is a special case of uniform-cost search when all edge costs are positive and identical.

Breadth-first always expands the shallowest node

- Only optimal if all step-costs are equal

Uniform-cost considers the overall path cost

- Optimal for any (reasonable) cost function
 - non-zero, positive
- Gets stuck down in trees with many fruitless, short branches
 - low path cost, but no goal node

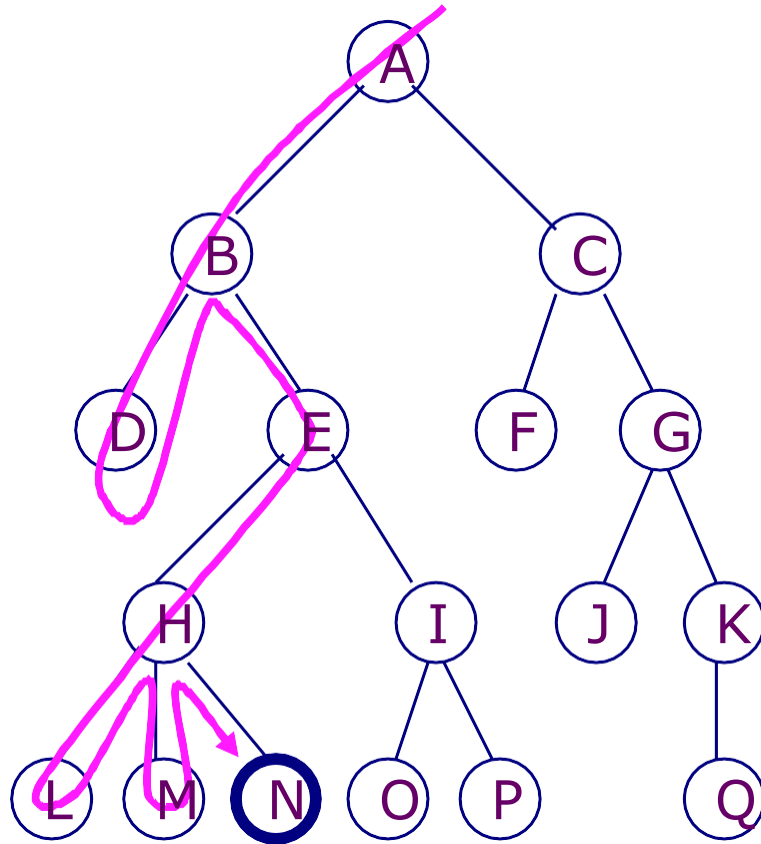
Both are complete for non-extreme problems

- Finite number of branches
- Strictly positive search function

3- Depth-First Search

Depth-First Search

Based on [4]



A **depth-first search (DFS)** explores a path all the way to a leaf before **backtracking** and exploring another path.

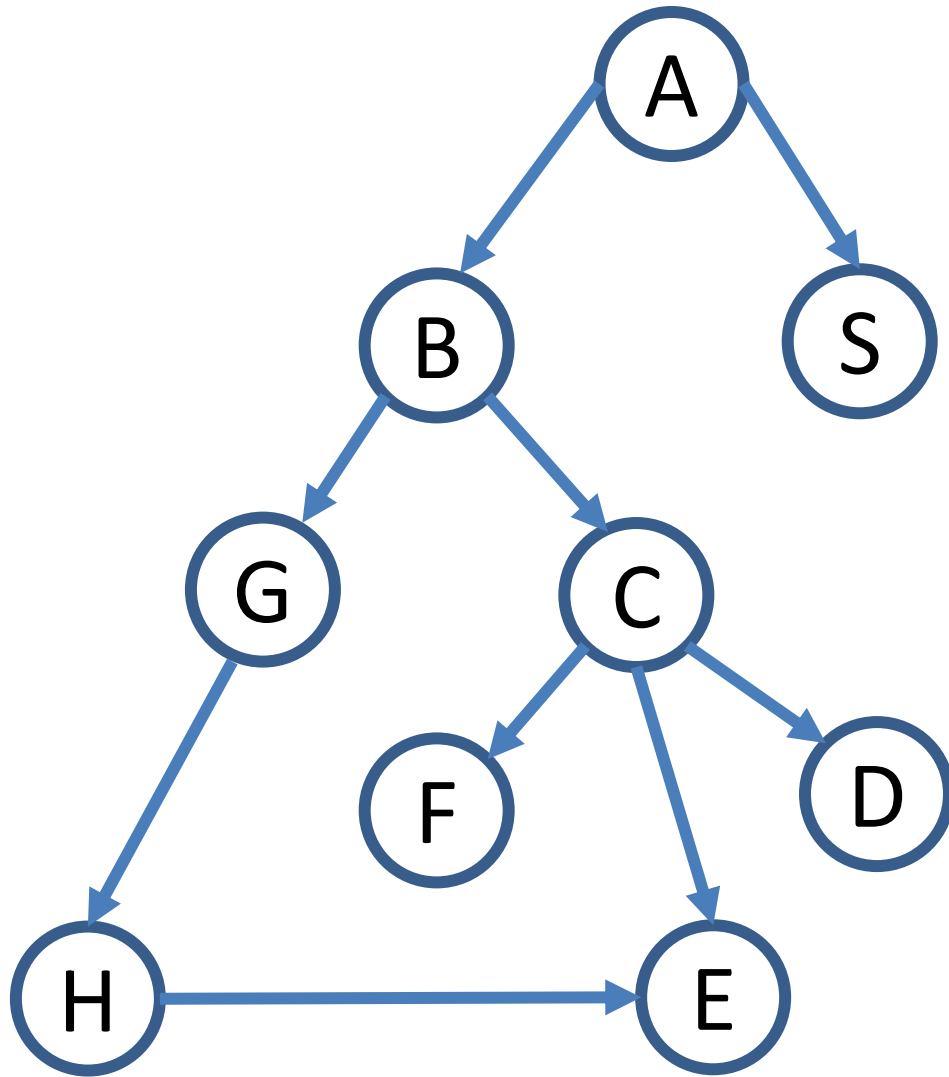
For example, after searching **A**, then **B**, then **D**, the search backtracks and tries another path from **B**.

Node are explored in the order **A B D E H L M N I O P C F G J K Q**

Depth-First Search

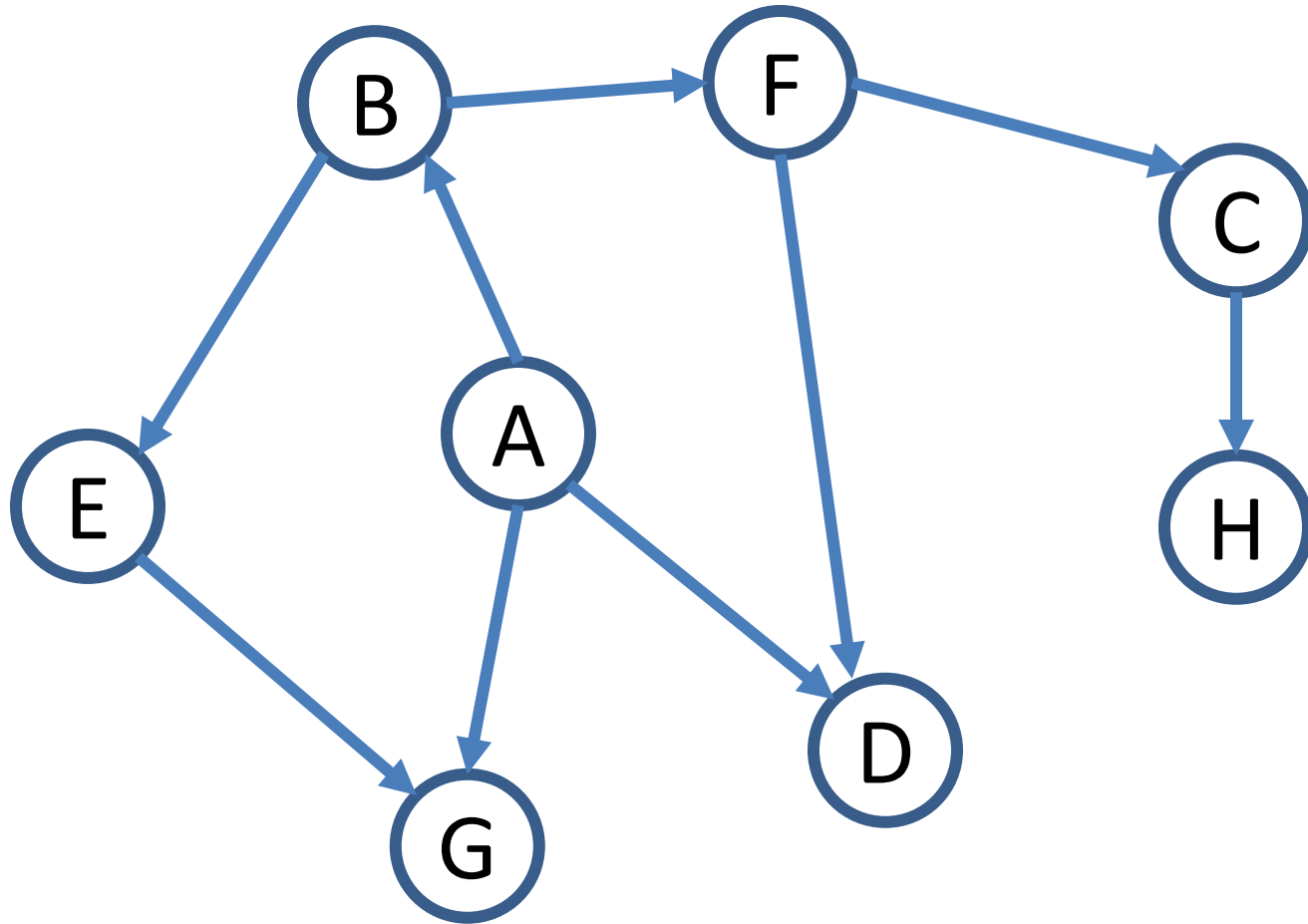
1. Start
2. Push Root Node to Stack
 - Mark Root Node as Visited
 - Print Root Node as Output
3. Check Top of the Stack If Stack is Empty, Go to Step 6
4. Else, Check Adjacent Top of the Stack
 - If Adjacent is not Visited
 - Push Node to Stack
 - Mark Node as Visited
 - Print Node as Output
 - Else Adjacent Visited
5. Go to Step 3
6. Stop

Example DFS



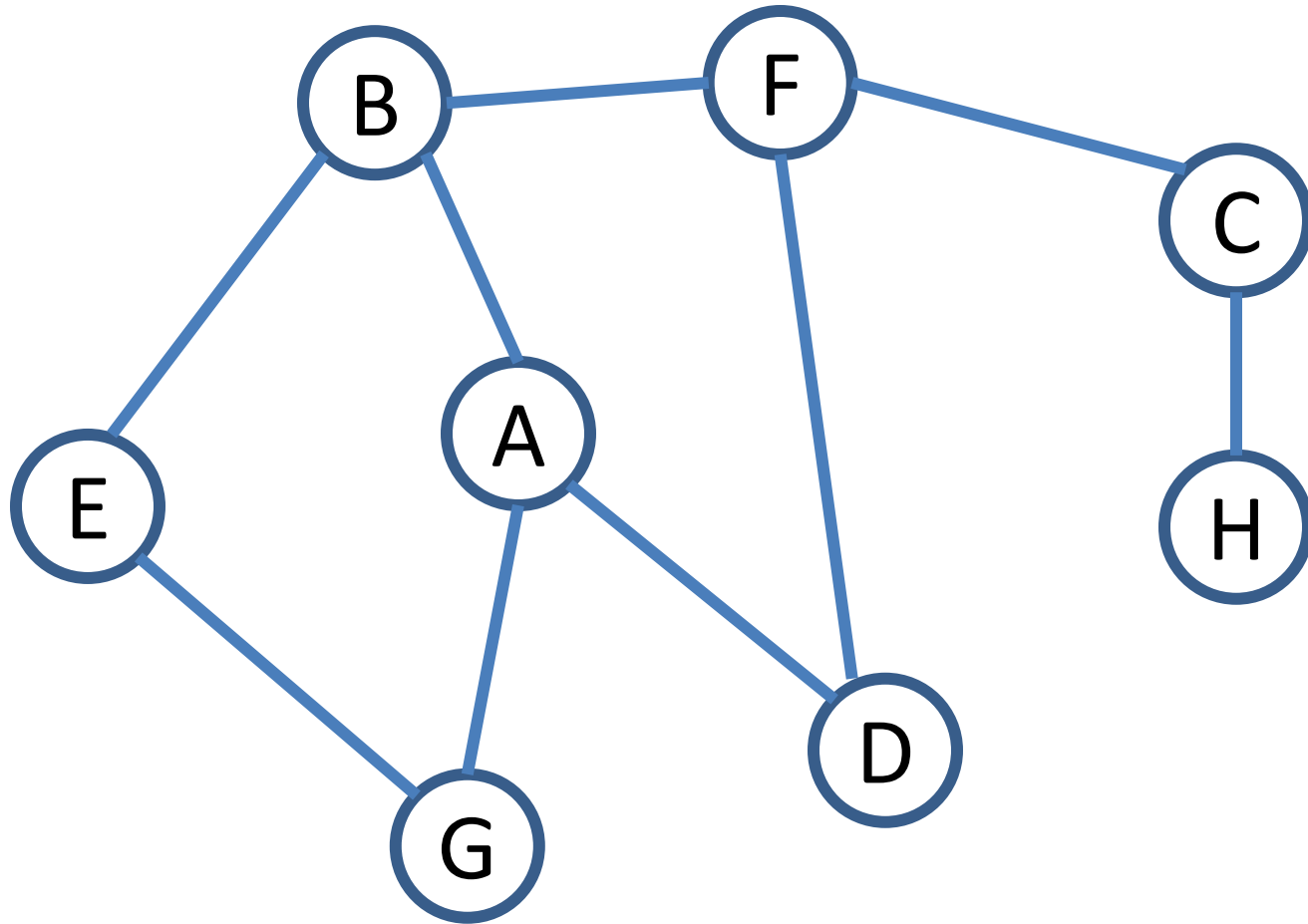
Example DFS

ABEGFCHD



Example DFS

AGEDFCHB



Properties of Depth-First Search

Complete: No: fails in infinite-depth spaces, spaces with loops
– Yes, complete in finite spaces

Time: $O(b^m)$: terrible if m is much larger than d
– but if solutions are dense, may be much faster than breadth-first

Space: $O(bm)$

Optimal: No

Criterion	Depth-First
Complete?	No
Time	$O(b^m)$
Space	$O(bm)$
Optimal?	No

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

Depth-First vs. Breadth-First

Depth-first goes off into one branch until it reaches a leaf node

- Not good if the goal is on another branch
- Neither complete nor optimal
- Uses much less space than breadth-first
 - Much fewer visited nodes to keep track, smaller fringe

Breadth-first is more careful by checking all alternatives

- Complete and optimal (Under most circumstances)
- Very memory-intensive

For a large tree, breadth-first search memory requirements maybe excessive

For a large tree, a depth-first search may take an excessively long time to find even a very nearby goal node.

➔ How can we combine the advantages (and avoid the disadvantages) of these two search techniques?

4- Depth-Limited Search

Similar to depth-first, but with a limit

- i.e., nodes at depth l have no successors
 - Overcomes problems with infinite paths
 - Sometimes a depth limit can be inferred or estimated from the problem description
 - In other cases, a good depth limit is only known when the problem is solved
 - must keep track of the depth
- Complete? no (if goal beyond l ($l < d$), or infinite branch length)
 - Time? $O(b^l)$
 - Space? $O(bl)$
 - Optimal? No (if $l < d$)

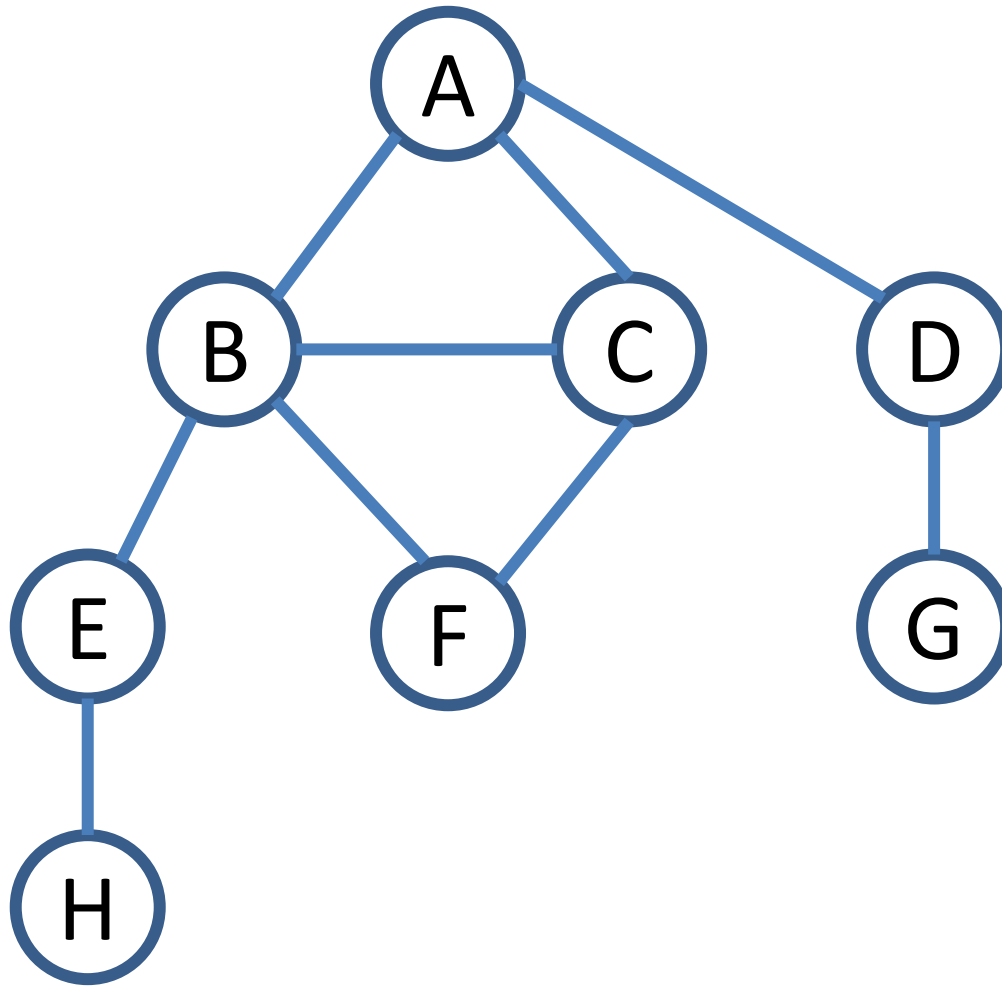
b branching factor
 l depth limit

Criterion	Depth-Limited
Complete?	No
Time	$O(b^l)$
Space	$O(bl)$
Optimal?	No

4- Example DLS

Depth = 2

ABCFEDG



5- Iterative Deepening Depth-First Search

Iterative Deepening Depth-First Search

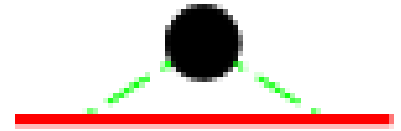
Applies LIMITED-DEPTH with increasing depth limits

- Combines advantages of BREADTH-FIRST and DEPTH-FIRST
- It searches to depth 0 (root only), then if that fails it searches to depth 1, then depth 2, etc.

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution, or fail-  
ure  
  inputs: problem, a problem  
  for depth  $\leftarrow$  0 to  $\infty$  do  
    result  $\leftarrow$  DEPTH-LIMITED-SEARCH(problem, depth)  
    if result  $\neq$  cutoff then return result
```

Iterative deepening search $l=0$

Limit = 0

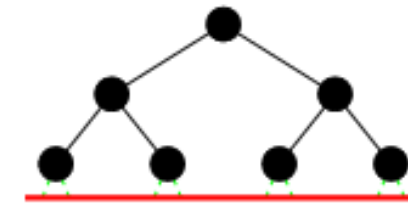
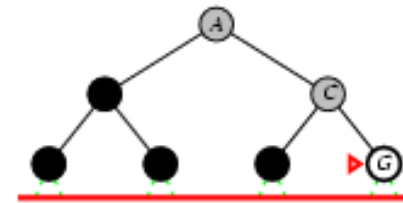
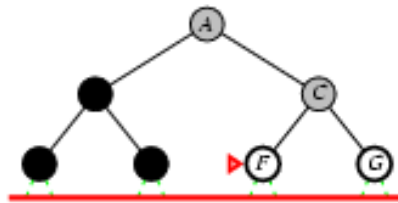
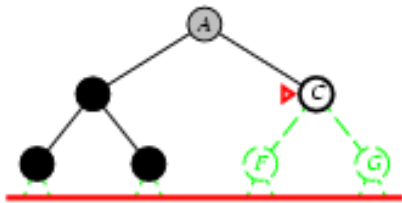
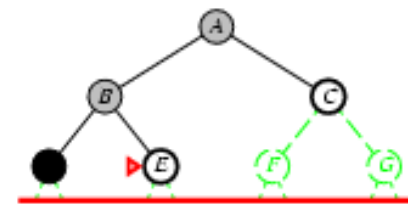
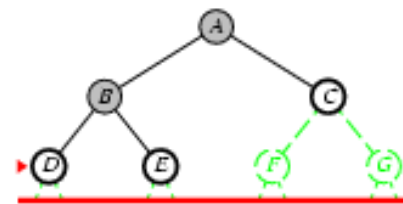
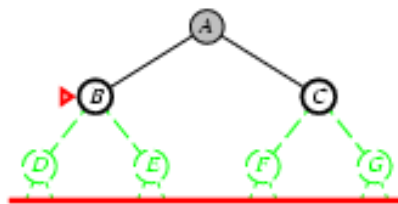


Iterative deepening search $l = 1$



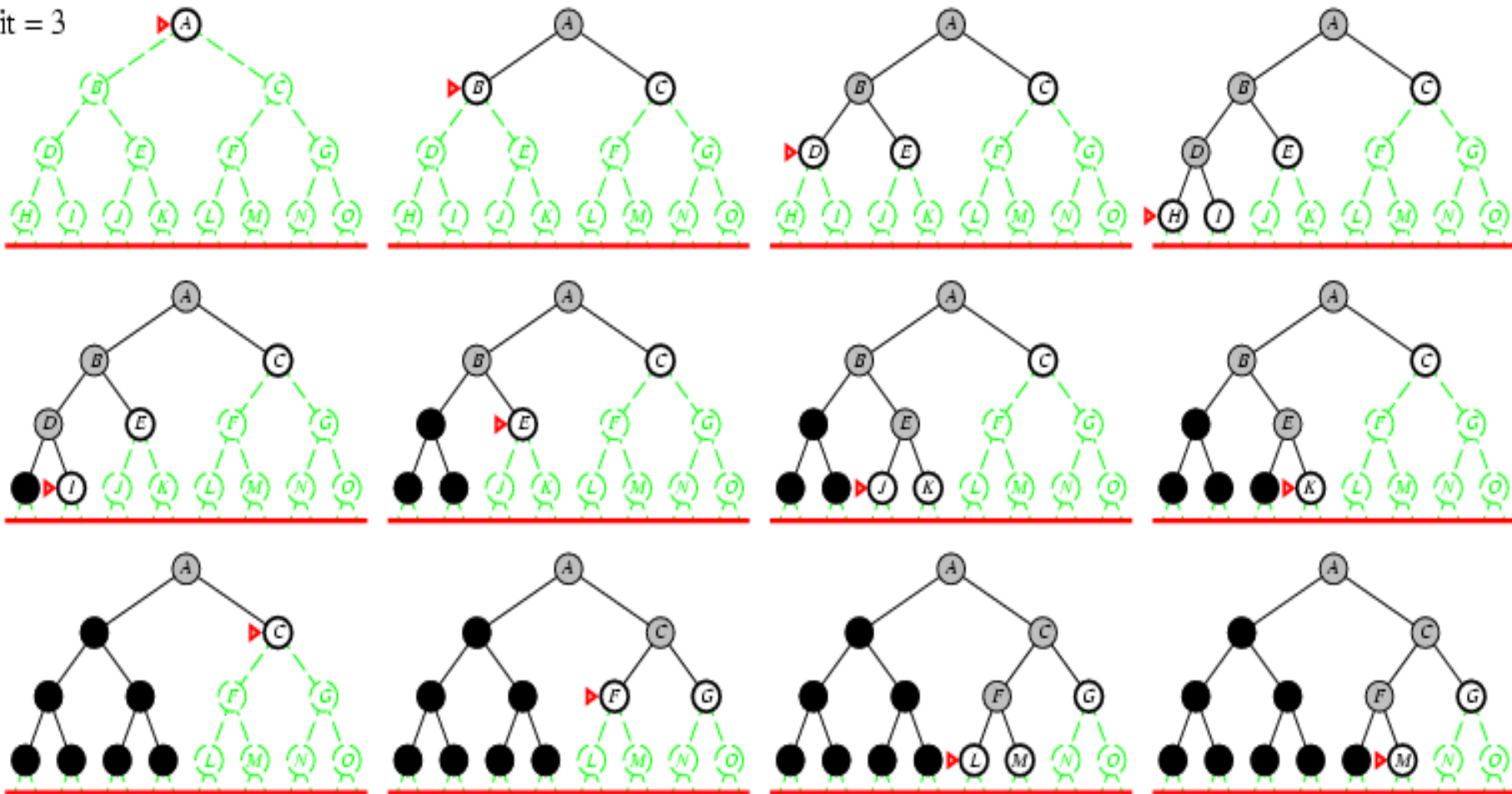
Iterative deepening search $l=2$

Limit = 2

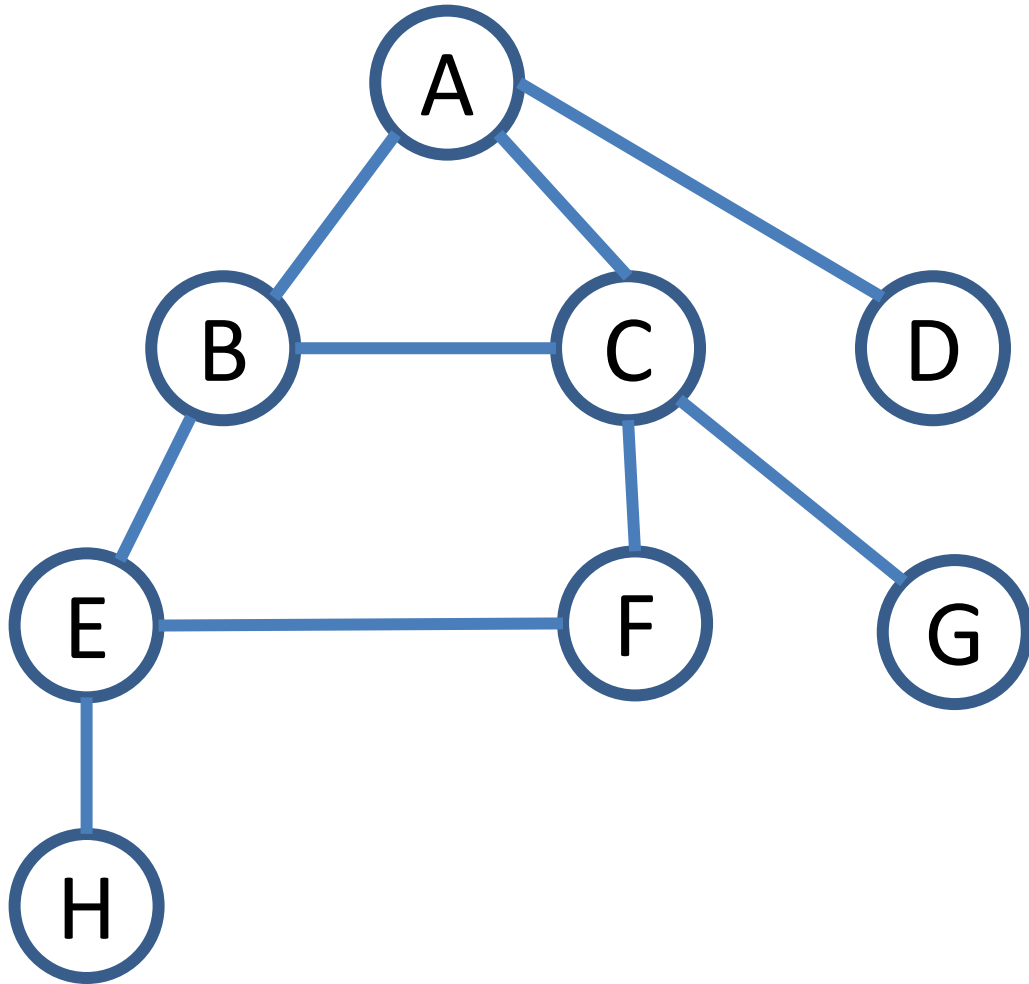


Iterative deepening search $l=3$

Limit = 3



Example IDS



Iterative Deepening Depth-First Search

If a goal node is found, it is a nearest node and the path to it is on the stack.

- Required stack size is limit of search depth (plus 1).
- Many states are expanded multiple times
 - doesn't really matter because the number of those nodes is small
- In practice, one of the best uninformed search methods
 - for large search spaces, unknown depth

Properties of Iterative Deepening Search

Complete: Yes (if the b is finite)

Time: $(d+1)b^0 + d b^1 + (d-1)b^2 + \dots + b^d = O(b^d)$

Space: $O(bd)$

Optimal: Yes, if step cost = 1

Criterion	Iterative Deepening
Complete?	Yes
Time	$O(b^d)$
Space	$O(bd)$
Optimal?	Yes

b	branching factor
d	Tree/goal depth

Iterative Deepening Search

The nodes in the bottom level (level d) are generated once, those on the next bottom level are generated twice, and so on:

$$N_{IDS} = (d)b + (d-1)b^2 + \dots + (1)b^d$$

$$\text{Time complexity} = b^d$$

Compared with BFS:

$$N_{BFS} = b + b^2 + \dots + b^d + (b^{d+1} - b)$$

- Suppose $b = 10$, $d = 5$,

$$N_{IDS} = 50 + 400 + 3,000 + 20,000 + 100,000 = 123,456$$

$$N_{BFS} = 1 + 10 + 100 + 1,000 + 10,000 + 100,000 = 111,111$$

➔ IDS behaves better in case the search space is large and the depth of goal is unknown.

Iterative Deepening Search

Based on [4]

When searching a binary tree to depth 7:

- DFS requires searching 255 nodes
- Iterative deepening requires searching 502 nodes
- Iterative deepening takes only about twice as long

When searching a tree with branching factor of 4 (each node may have four children):

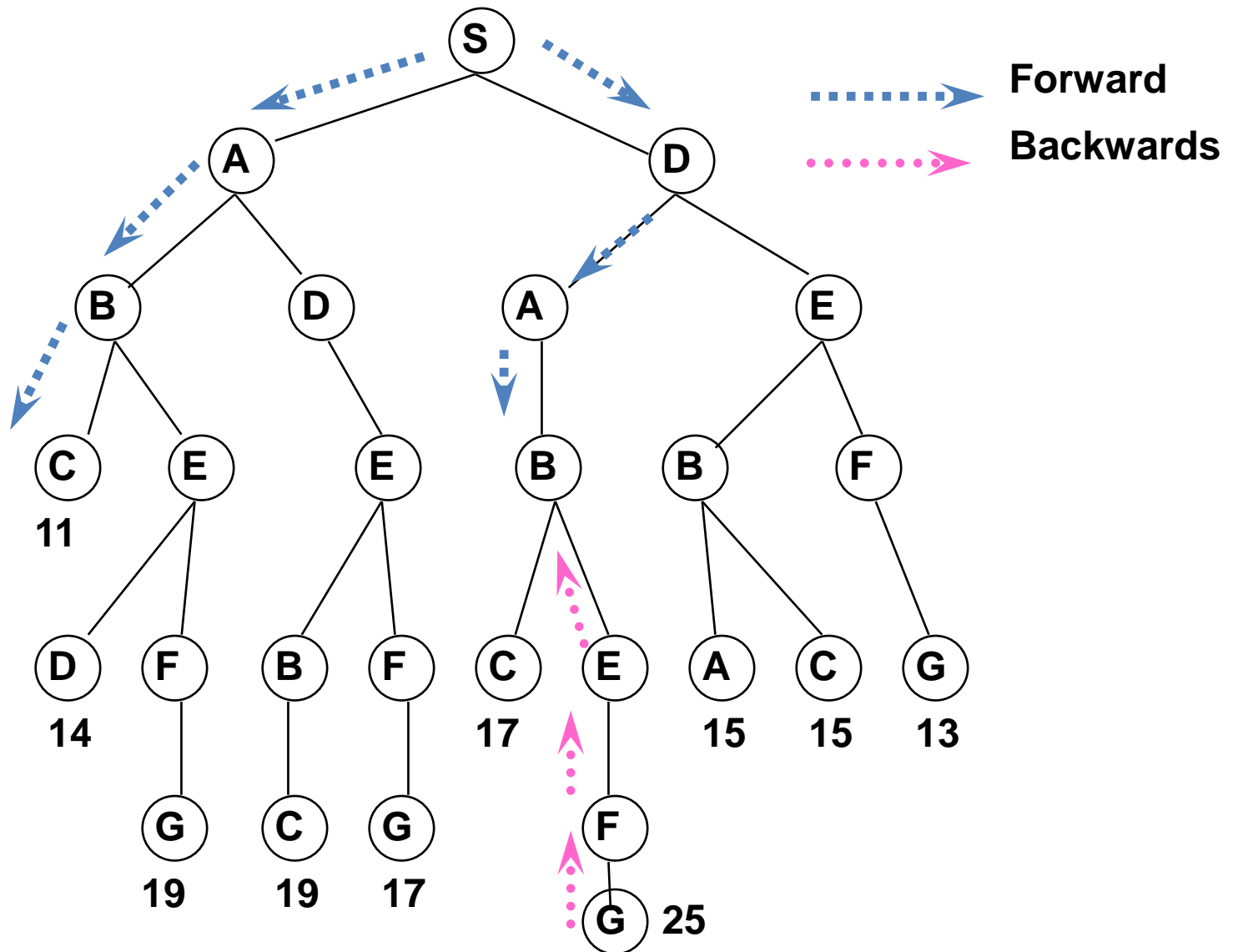
- DFS requires searching 21845 nodes
- Iterative deepening requires searching 29124 nodes
- Iterative deepening takes about $4/3 = 1.33$ times as long

The higher the branching factor, the lower the relative cost of iterative deepening depth first search

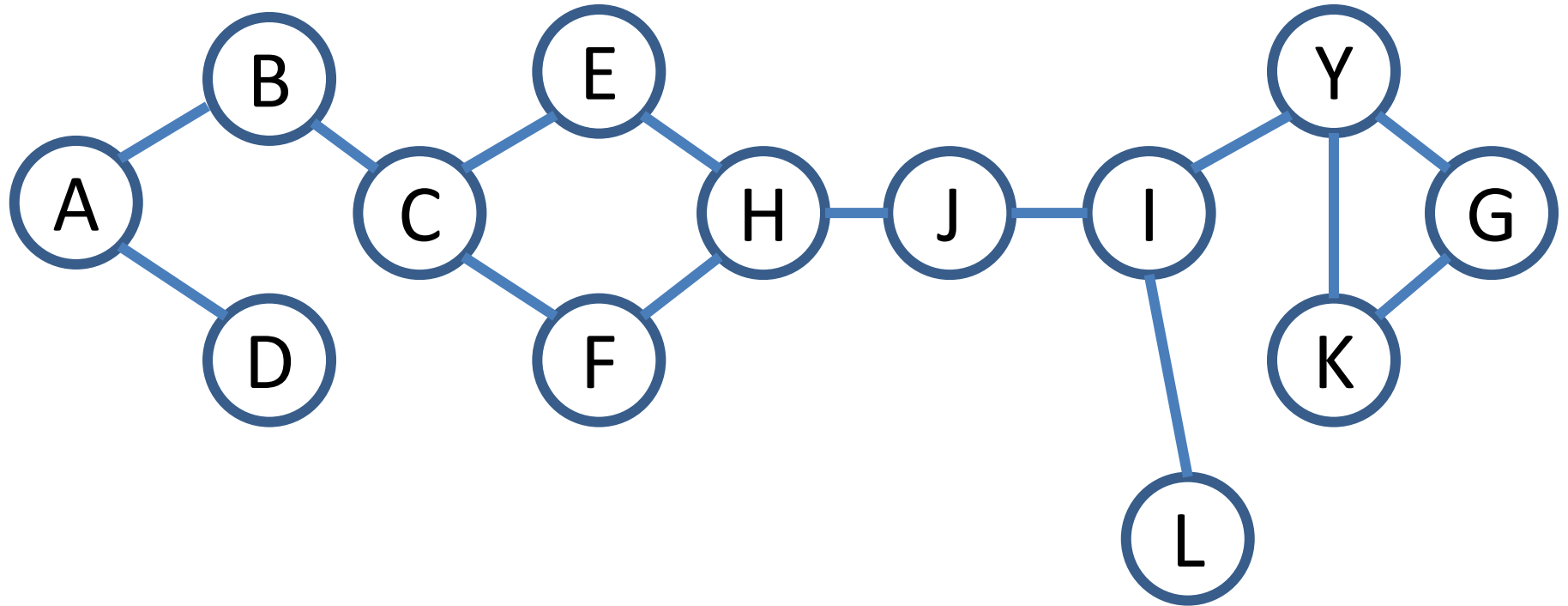
6- Bi-directional Search

- Both search **forward** from **initial state**, and **backwards** from **goal**.
- Stop when the two searches meet in the middle.
- Motivation: $b^{d/2} + b^{d/2}$ is much less than b^d
- Implementation
 - Replace the goal test with a check to see whether the frontiers of the two searches intersect, if yes \rightarrow solution is found

6- Bi-directional Search



6- Bi-directional Search Example Using BFS



6- Bi-directional Search

Search simultaneously from two directions

- Forward from the initial and backward from the goal state, until they meet in the middle (i.e., if a node exists in the fringe of the other).
- The idea is to have $(b^{d/2} + b^{d/2})$ instead of b^d , which is much less

May lead to substantial savings (if it is applicable), but it has several limitations

- Predecessors must be generated, which is not always possible
- Search must be coordinated between the two searches
- One search must keep all nodes in memory

Time Complexity	$b^{d/2}$
Space Complexity	$b^{d/2}$
Completeness	yes (b finite, breadth-first for both directions)
Optimality	yes (all step costs identical, breadth-first for both directions)

b	branching factor
d	tree depth

Summary (When to use what)

Breadth-First Search:

- Some solutions are known to be shallow

Uniform-Cost Search:

- Actions have varying costs
- Least cost solution is the required

This is the only uninformed search that worries about costs.

Depth-First Search:

- Many solutions exist
- Know (or have a good estimate of) the depth of solution

Iterative-Deepening Search:

- Space is limited and the shortest solution path is required

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

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

Iterative deepening search uses only linear space and not much more time than other uninformed algorithms

Summary

Breadth-first search (BFS) and depth-first search (DFS) are the foundation for all other search techniques.

We might have a **weighted tree**, in which the edges connecting a node to its children have differing “weights”

- We might therefore look for a “least cost” goal

The searches we have been doing are **blind searches**, in which we have no prior information to help guide the search

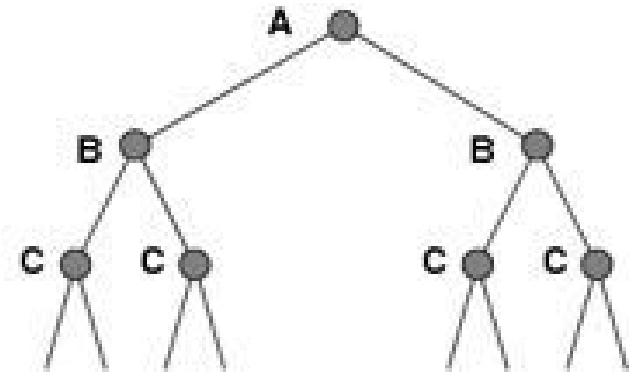
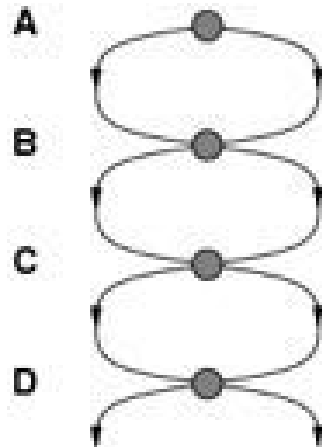
Conclusions

- Interesting problems can be cast as search problems, but you've got to set up state space
- Problem formulation usually requires abstracting away real-world details to define a state space that can be explored using computer algorithms.
- Once problem is formulated in abstract form, complexity analysis helps us picking out best algorithm to solve problem.
- Variety of uninformed search strategies; difference lies in method used to pick node that will be further expanded.

Improving Search Methods

Make algorithms more efficient

- avoiding repeated states



Use additional knowledge about the problem

- properties (“shape”) of the search space
 - more interesting areas are investigated first
- pruning of irrelevant areas
 - areas that are guaranteed not to contain a solution can be discarded