Mustafa Jarrar: Lecture Notes on Uninformed Search Algorithms.
Birzeit University, Palestine. 2020

Version 5

Uninformed Search Algorithms

(Chapter 3)

Mustafa Jarrar

Birzeit University
mjarrar@birzeit.edu
www.jarrar.info

1

# Watch this lecture
and download the slides

Download: http://www.jarrar.info/courses/AI/Jarrar.LectureNotes.Ch3.UninformedSearch.pdf
More Courses: http://www.jarrar.info/courses/

Lecture Outline

❑ Achieve intelligence by (searching) a solution!

❑ Problem Formulation

❑ Search Strategies

  ❑ breadth-first

  ❑ uniform-cost search

  ❑ depth-first

  ❑ depth-limited search

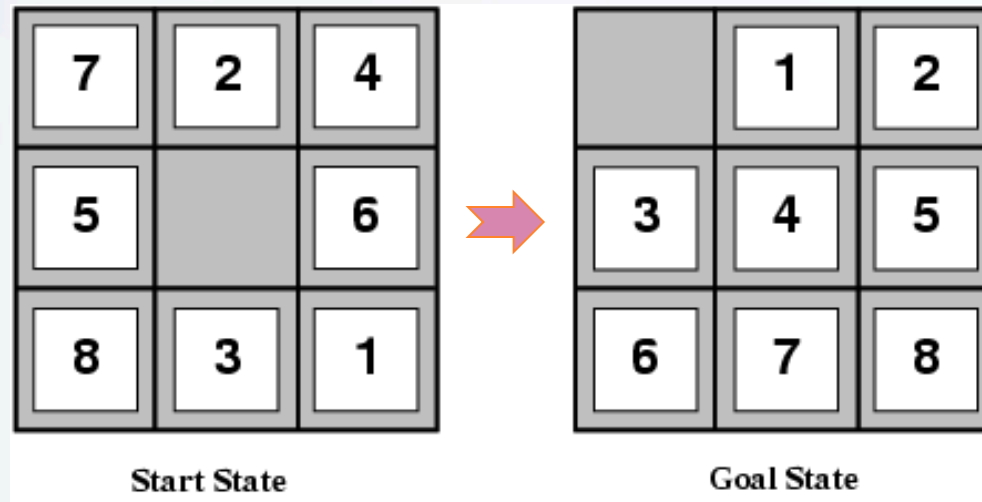 ❑ iterative deepening

  ❑ bi-directional search

Example: Romania

You are in Arad and want to go to Bucharest
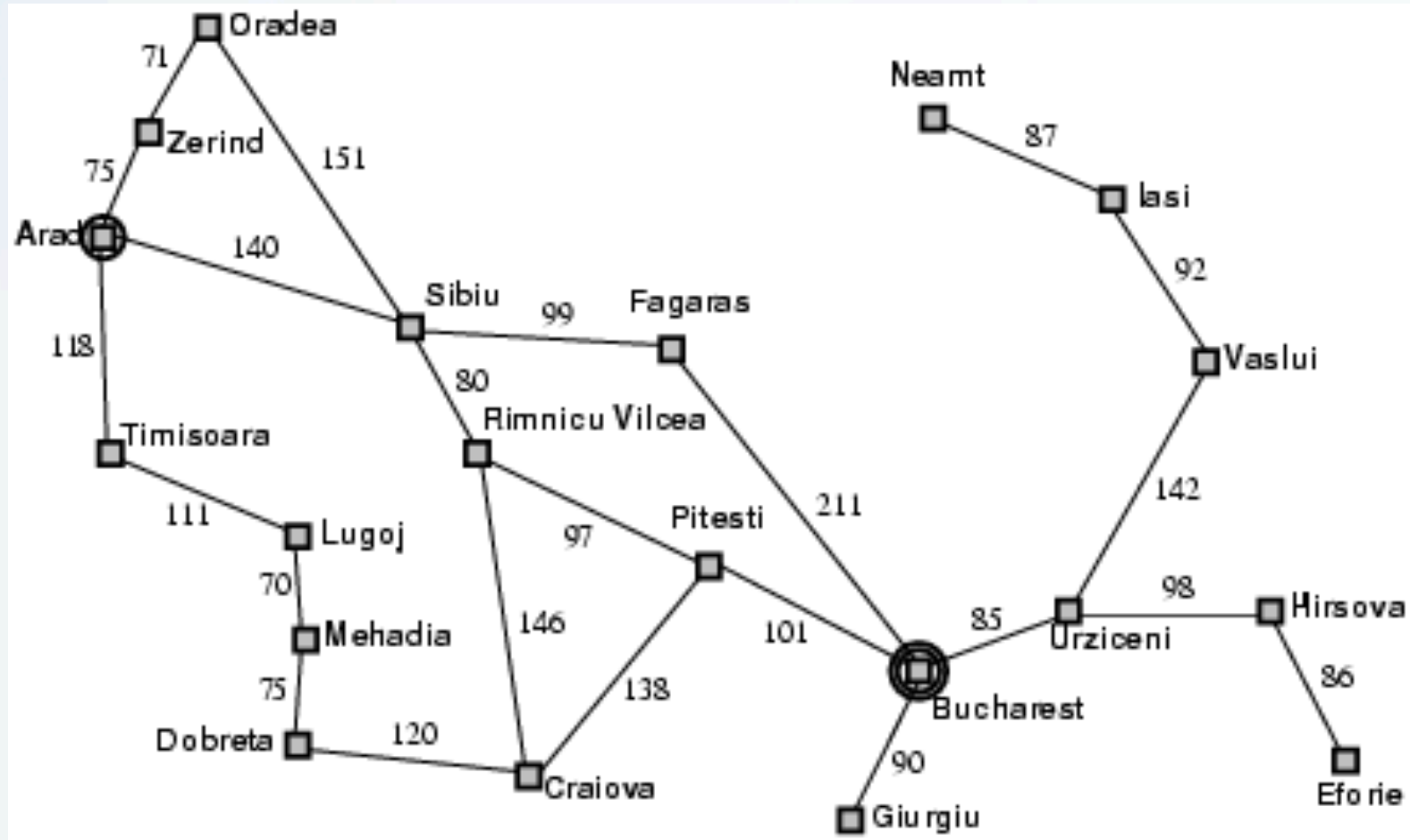
➔ How to design an intelligent agent to find the way between 2 cities?

Example: The 8-puzzle

Start State   Goal State

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

Motivation

- Solve a problem by searching for a solution. Search strategies are important methods for many approaches to problem-solving [2].

- Problem formulation. The use of search requires an abstract formulation of the problem and the available steps to construct solutions.

- Optimizing Search. Search algorithms are the basis for many optimization and planning methods.

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)

State: We regard a problem as state space

here a state is a City

Initial State: the state to start from

In(Arad)

Successor Function: description of the possible actions, give state x, S(X) returns a set of <action, successor> ordered pairs.

S(x)= { <Go(Sibiu), In(Sibiu)>, <Go(Timisoara), In(Timisoara)>,
         <Go(Zerind),In(Zerind)> }

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

In(Sibiu) →No. In(Zerind) →No..... In(Bucharest)→Yes!

Path Cost: a function that assigns a numeric cost to each path.
– e.g., sum of distances, number of actions executed, etc.
– *c(x,a,y)* is the step cost, assumed to be ≥ 0

Solution: a sequence of actions leading from the initial state to a goal state

{Arad →Sibiu → Rimnicu Vilcea → Pitesti → Bucharest}
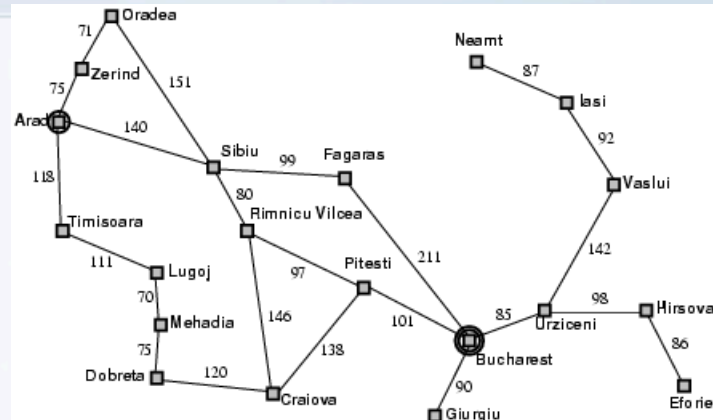
Problem Formulation (The 8- Puzzle Example)

Start State   **Goal State**

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)}

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
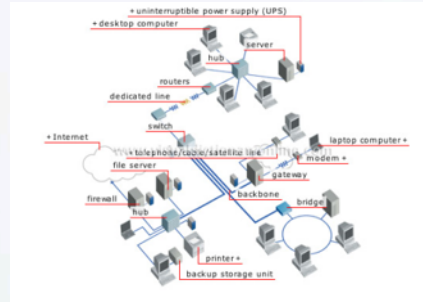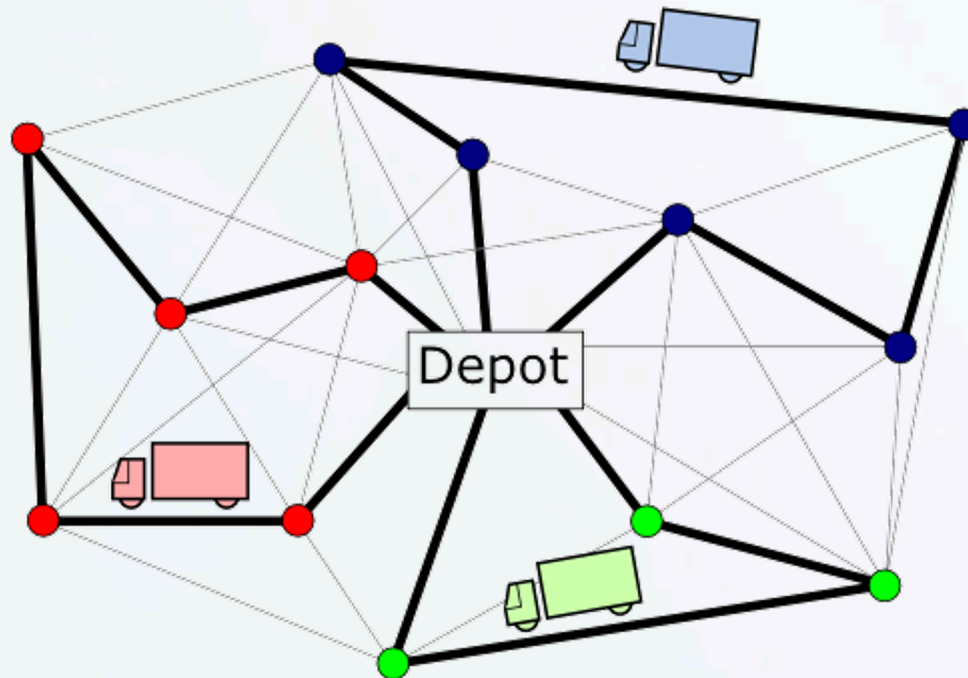
Routing in Computer networks
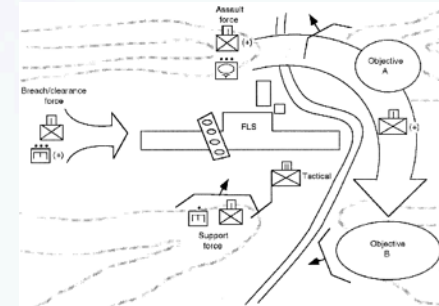
Military operation planning

- **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)

Routing Problem

➔ **What is the state space for each of them?**

A set of places with links between them, which have been visited

# Problem Formulation (Real-life Applications)

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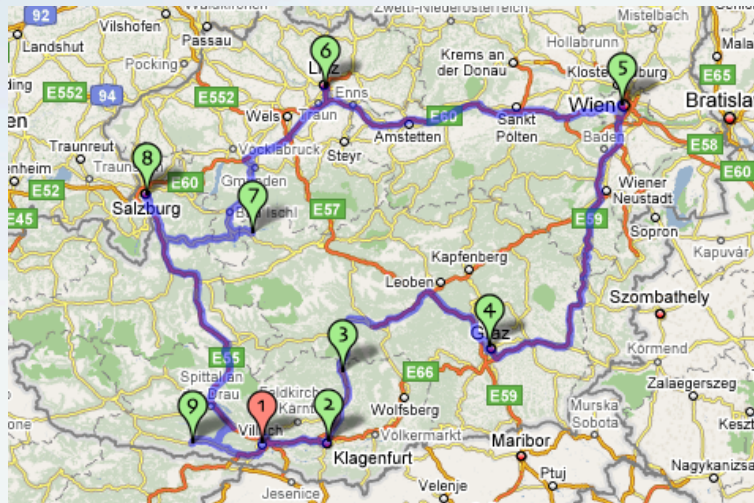
- **States**
  - locations / cities
  - illegal states
    - each city may be visited only once
    - visited cities must be kept as state information
- **Initial state**
  - starting point
  - no cities visited
- **Successor function (operators)**
  - move from one location to another one
- **Goal test**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

Problem Formulation (Real-life Applications)

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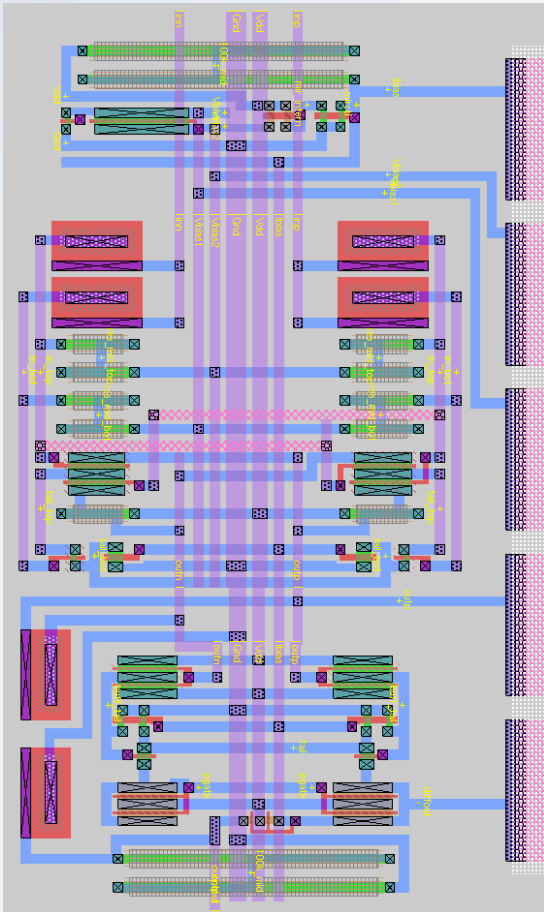
- **States**
  - positions of components, wires on a chip
- **Initial state**
  - incremental: no components placed
  - complete-state: all components placed (e.g. randomly, manually)
- **Successor function (operators)**
  - incremental: place components, route wire
  - complete-state: move component,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

Jarrar © 2020

13

Problem Formulation (Real-life Applications)

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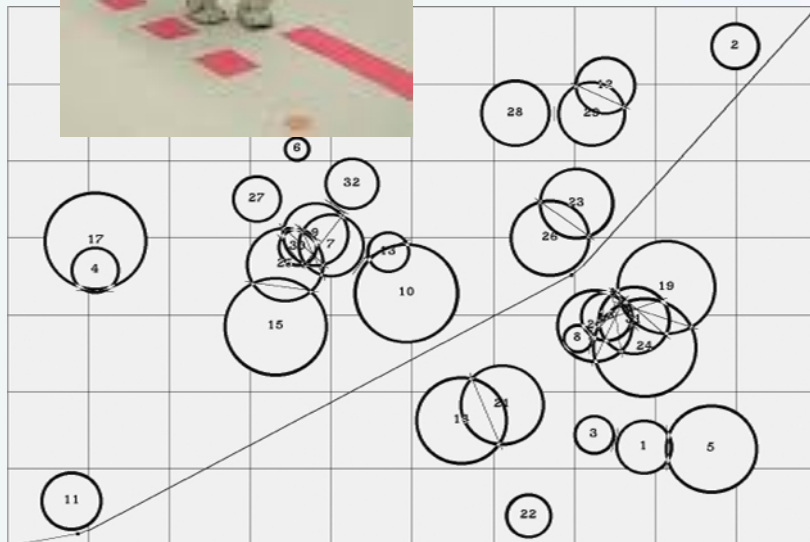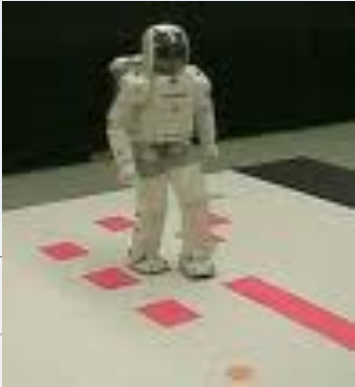
- **States**
  - locations
  - position of actuators
- **Initial state**
  - start position (dependent on the task)
- **Successor function (operators)**
  - movement, actions of actuators
- **Goal test**
  - task-dependent
- **Path cost**
  - maybe very complex
    - distance, energy consumption

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)

Example: Graph Search
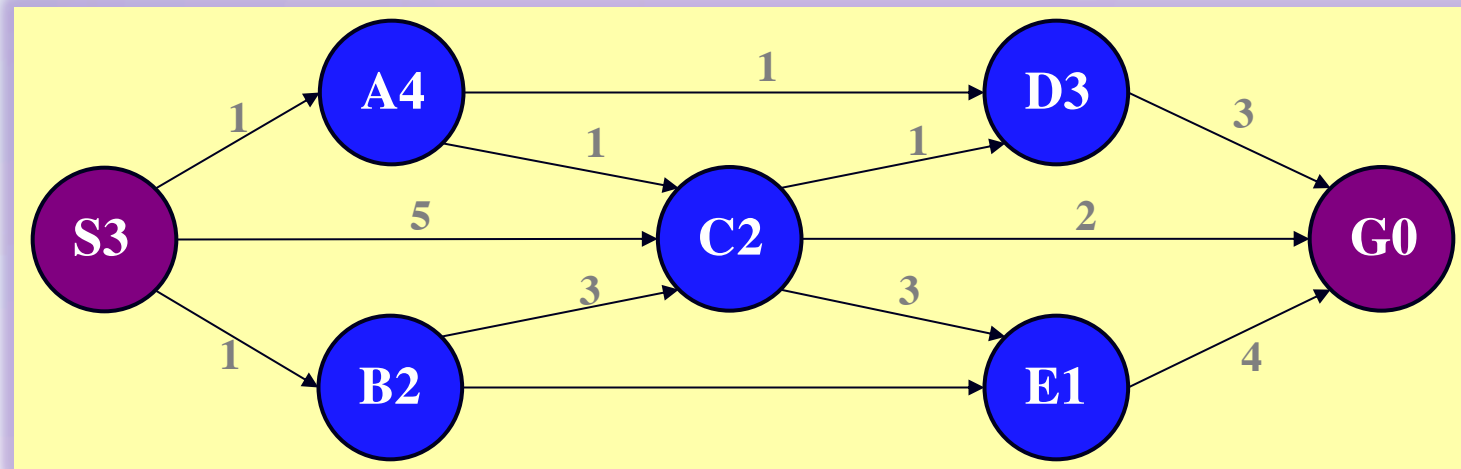
- ## The graph describes the search (state) space
  - Each node in the graph represents one state in the search space
    - e.g. a city to be visited in a routing or touring problem

- ## 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
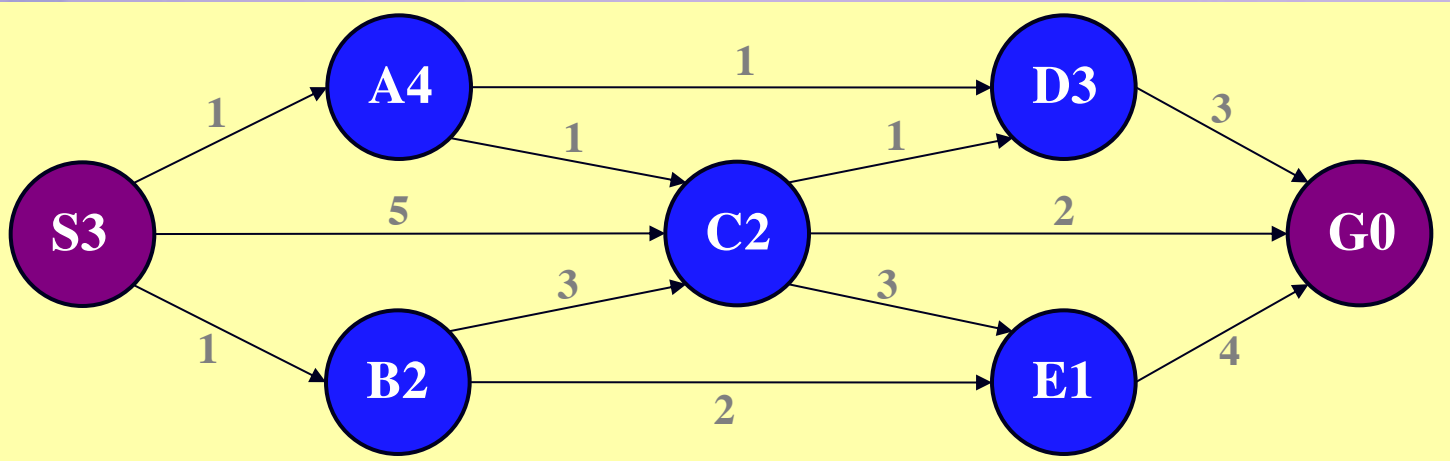
- A tree is generated by traversing the graph.

- 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.
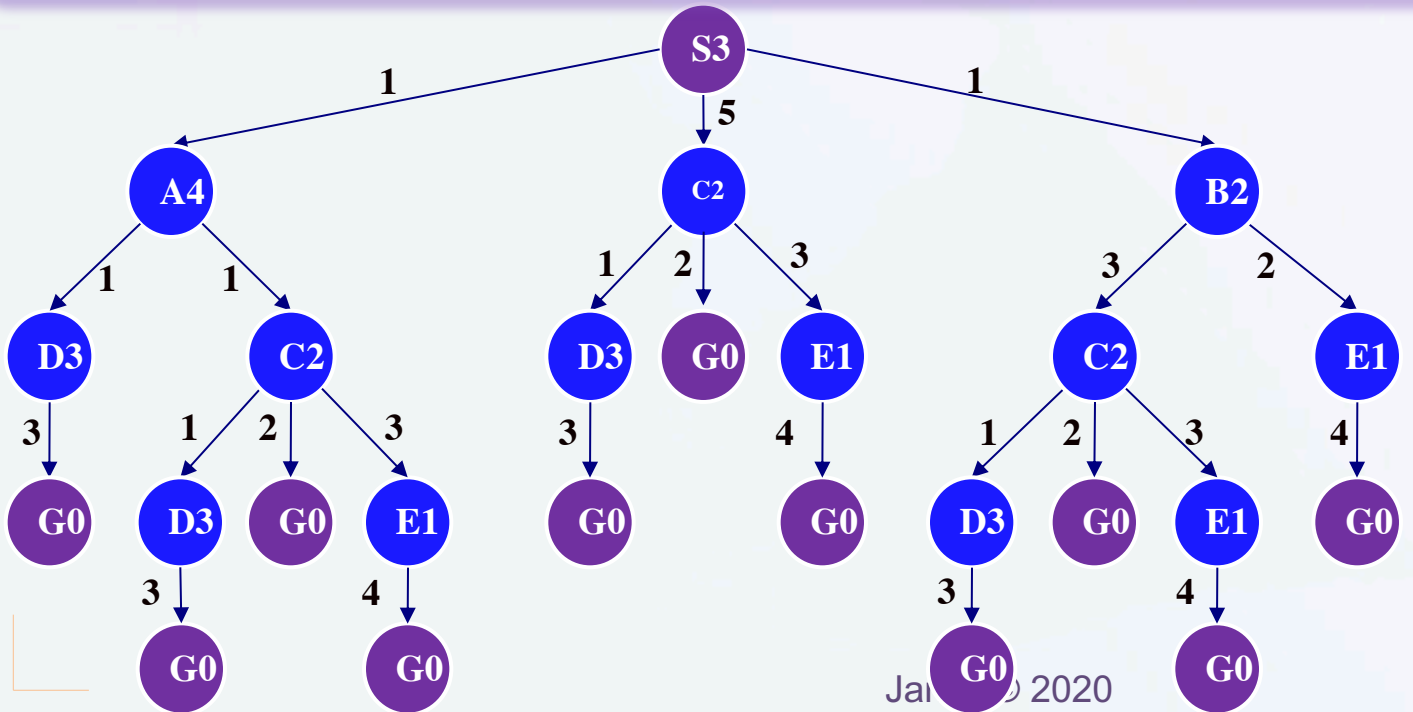
Searching Strategies

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

Jarrar © 2020

20

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

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.
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 Snapshot 1
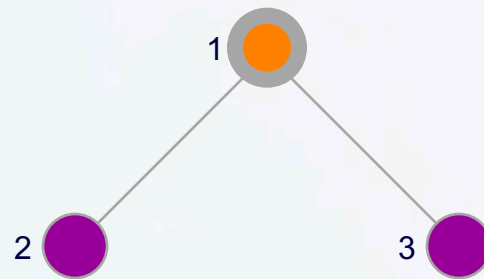
Initial

Visited

Fringe

Current

Visible

Goal

Fringe: [] + [2,3]

Jarrar © 2020

Breadth-First Snapshot 2

Initial

Visited

Fringe

Current

Visible

Goal

Fringe: [3] + [4,5]

Jarrar © 2020
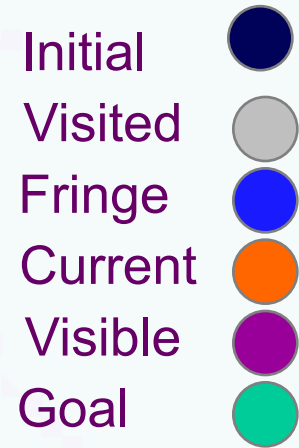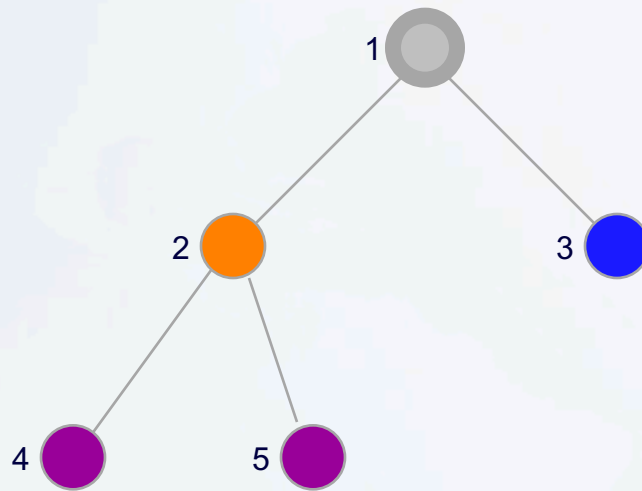
Breadth-First Snapshot 3
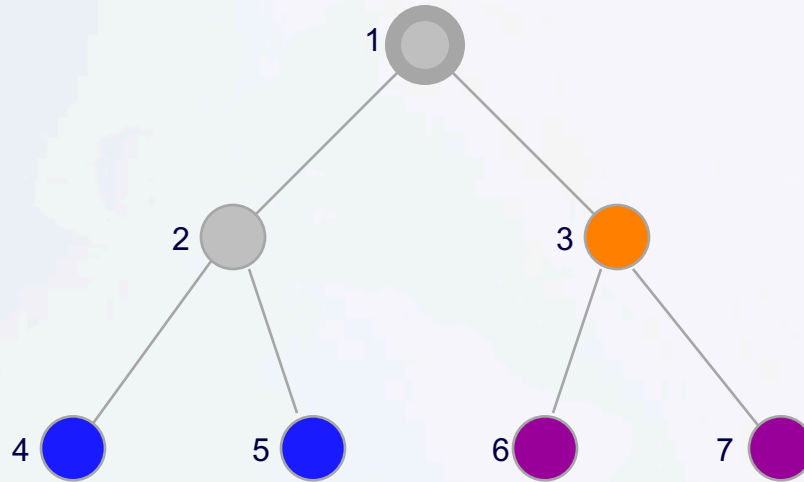
Initial
Visited
Fringe
Current
Visible
Goal

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

Breadth-First Snapshot 4

Initial

Visited

Fringe

Current
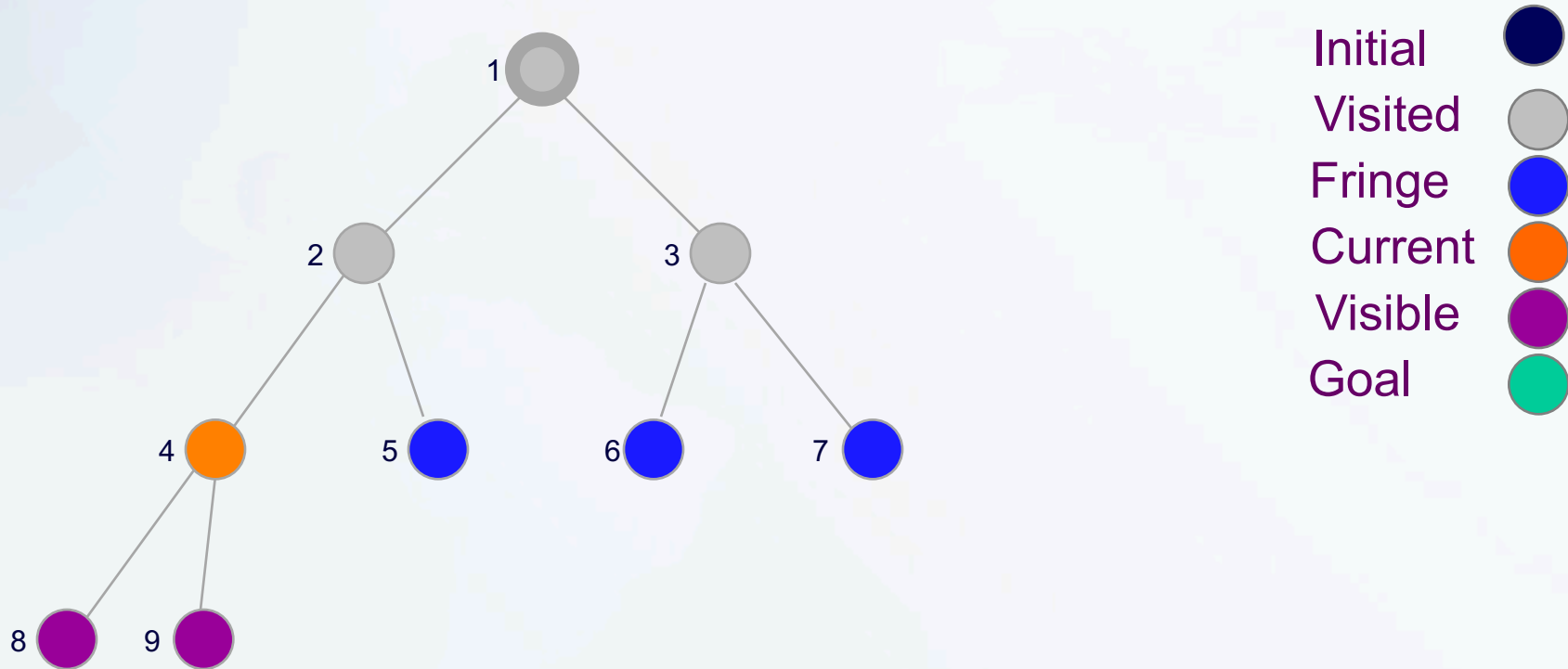
Visible

Goal

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

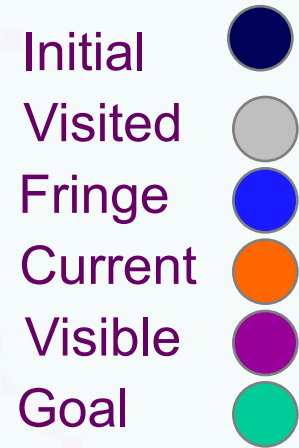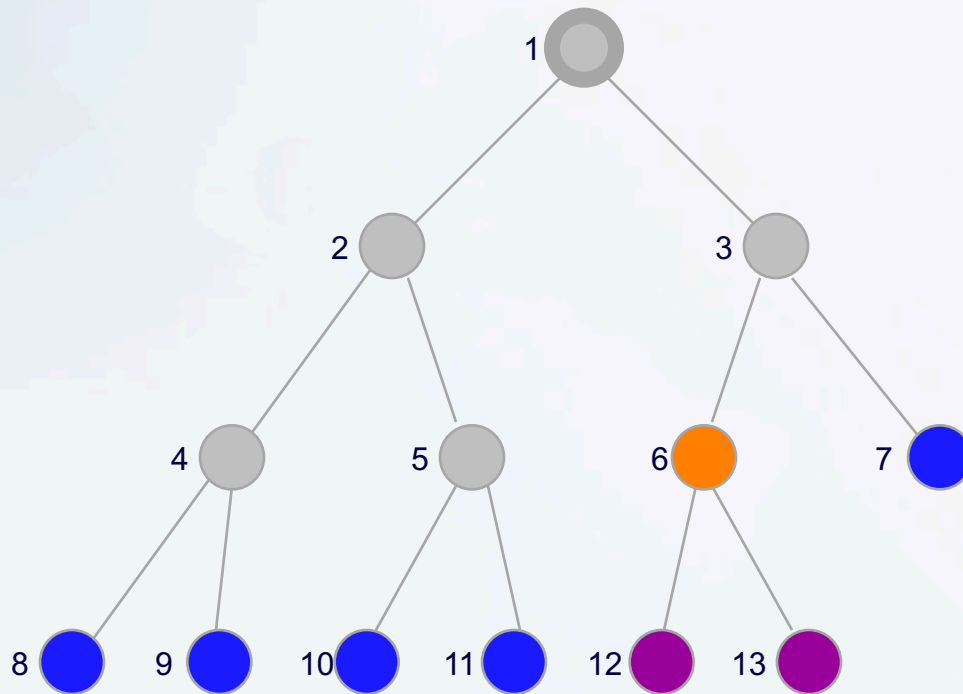# Breadth-First Snapshot 5
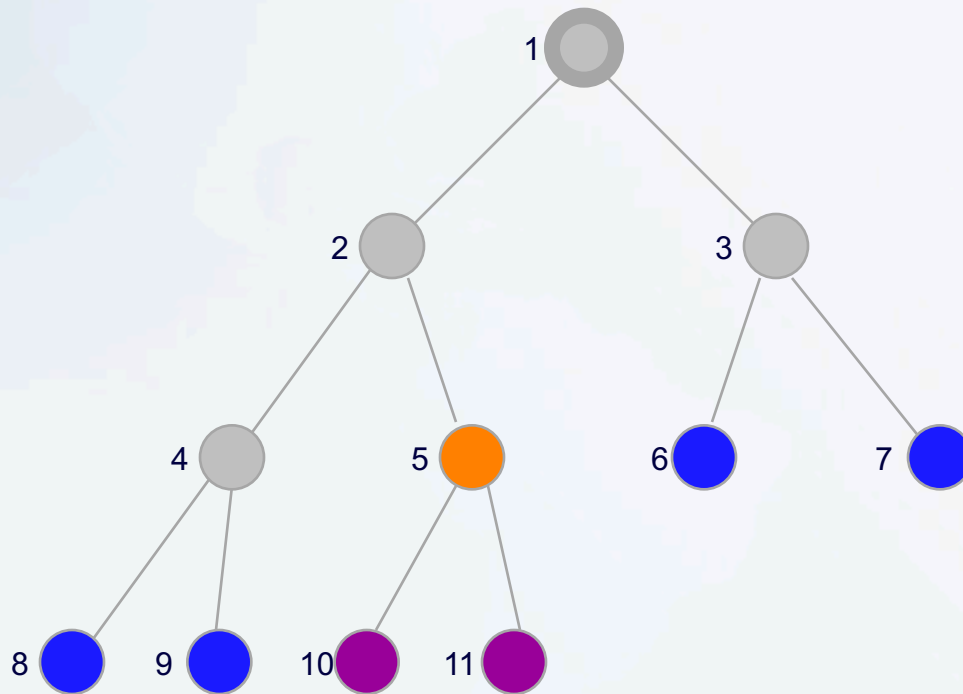
Initial
Visited
Fringe
Current
Visible
Goal

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

Breadth-First Snapshot 6

Initial

Visited
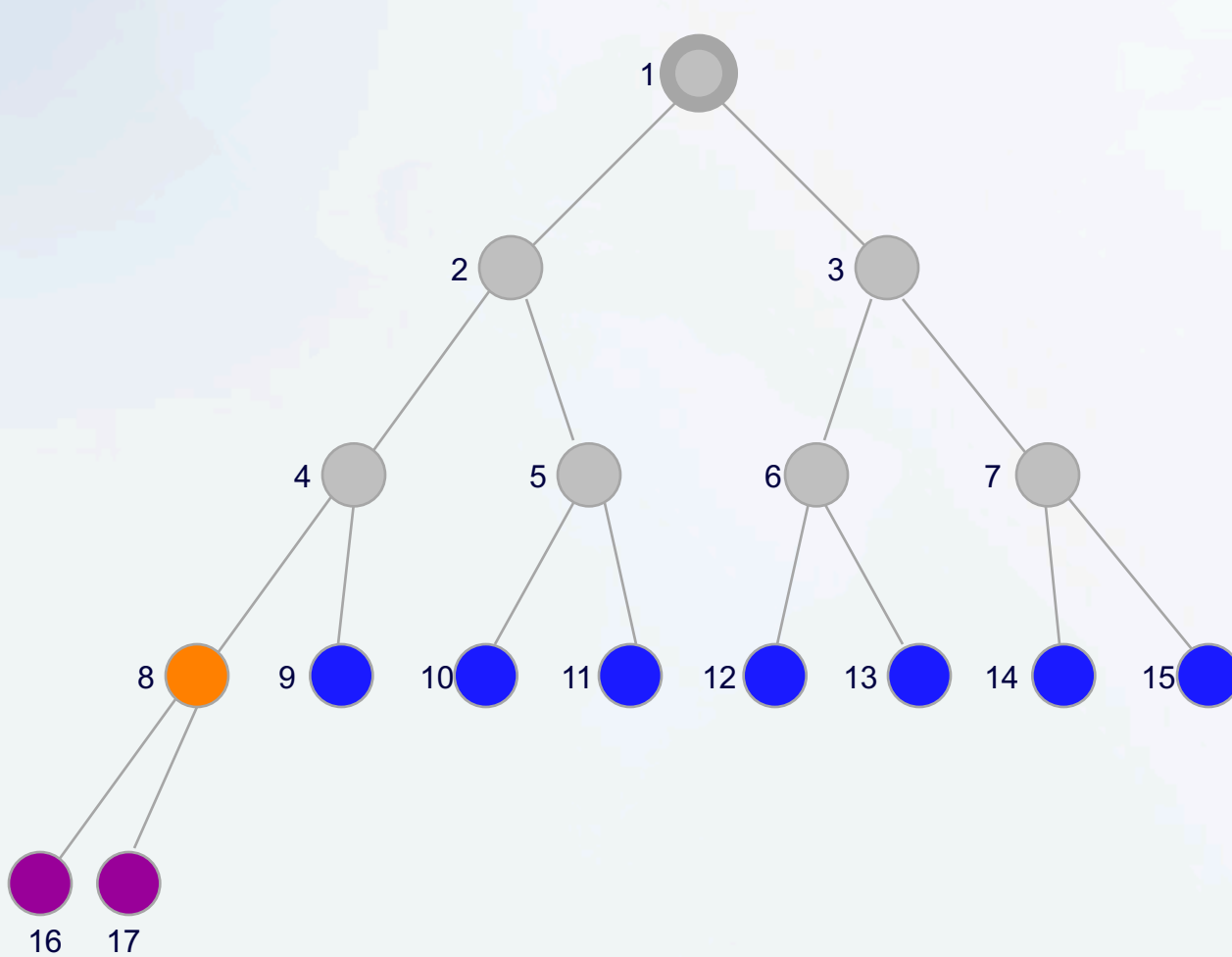
Fringe

Current

Visible

Goal

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

Breadth-First Snapshot 7

Initial
Visited
Fringe
Current
Visible
Goal
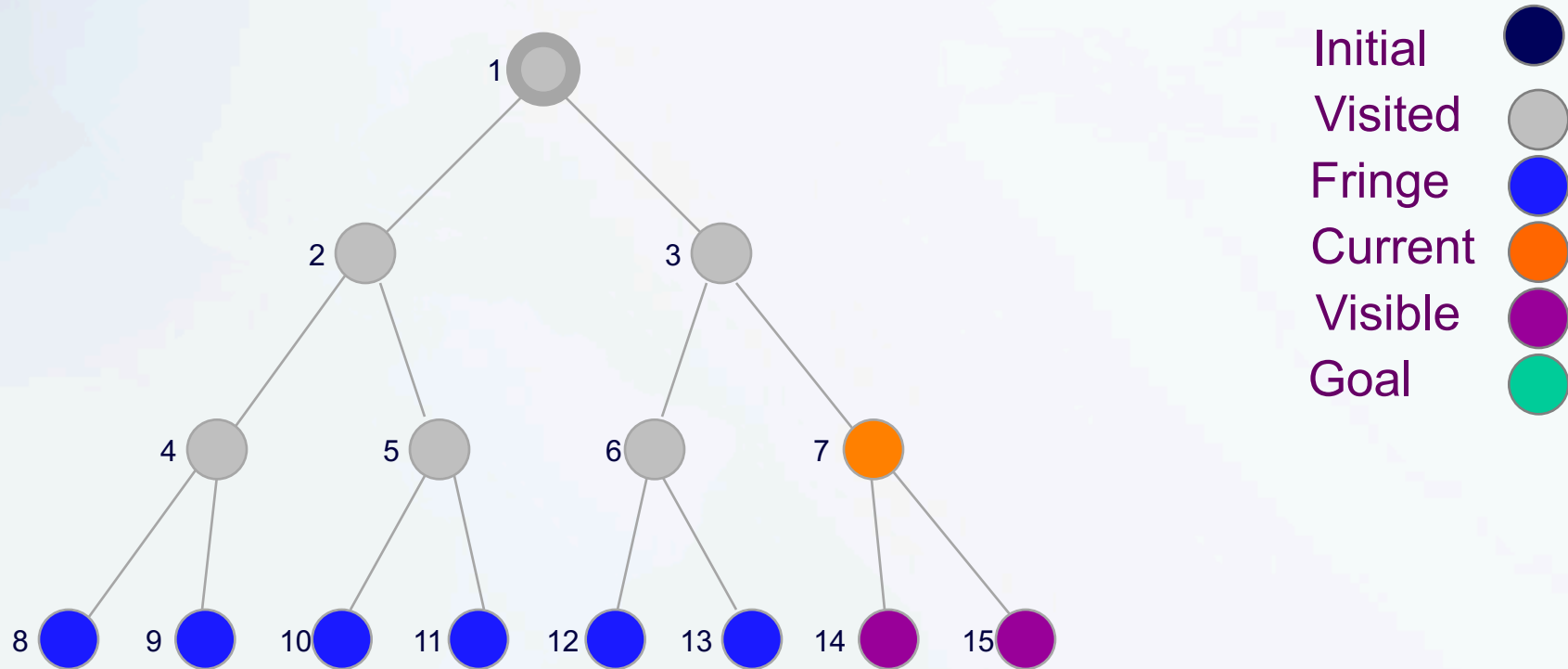
Fringe: [8,9.10,11,12,13] + [14,15]

Breadth-First Snapshot 8

Initial

Visited

Fringe

Current

Visible

Goal

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

Breadth-First Snapshot 9
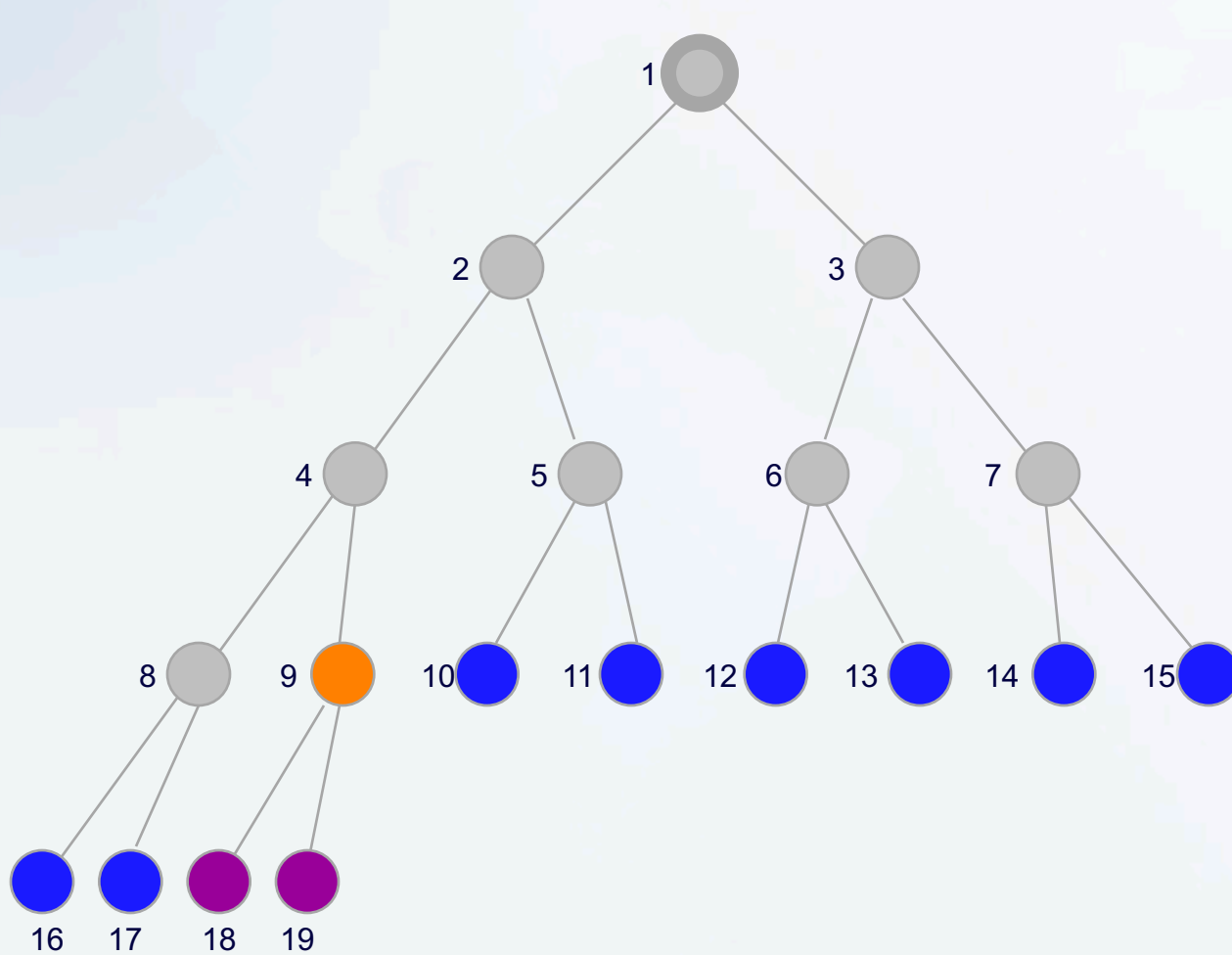
Initial
Visited
Fringe
Current
Visible
Goal

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

Breadth-First Snapshot 10
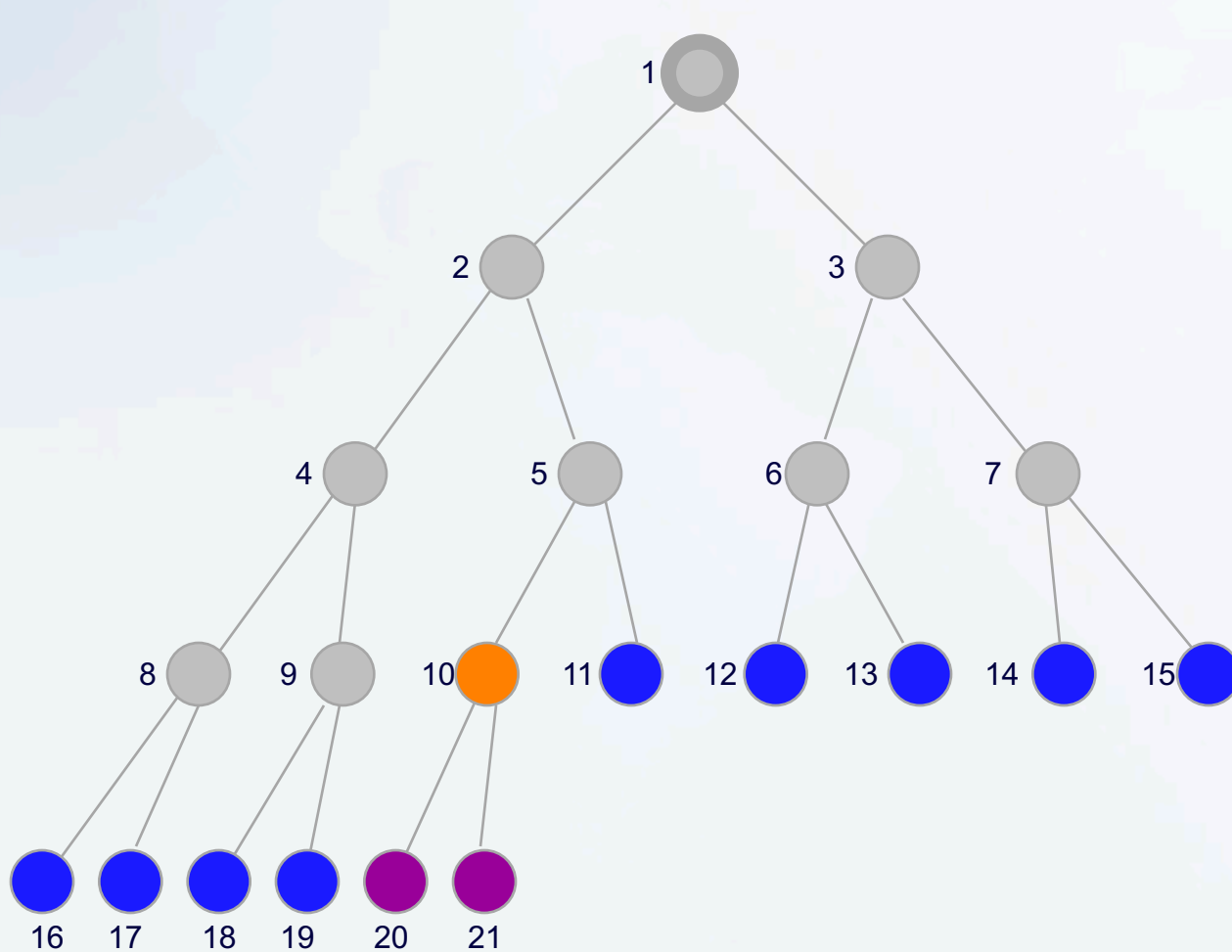
Initial
Visited
Fringe
Current
Visible
Goal

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

33

Breadth-First Snapshot 11
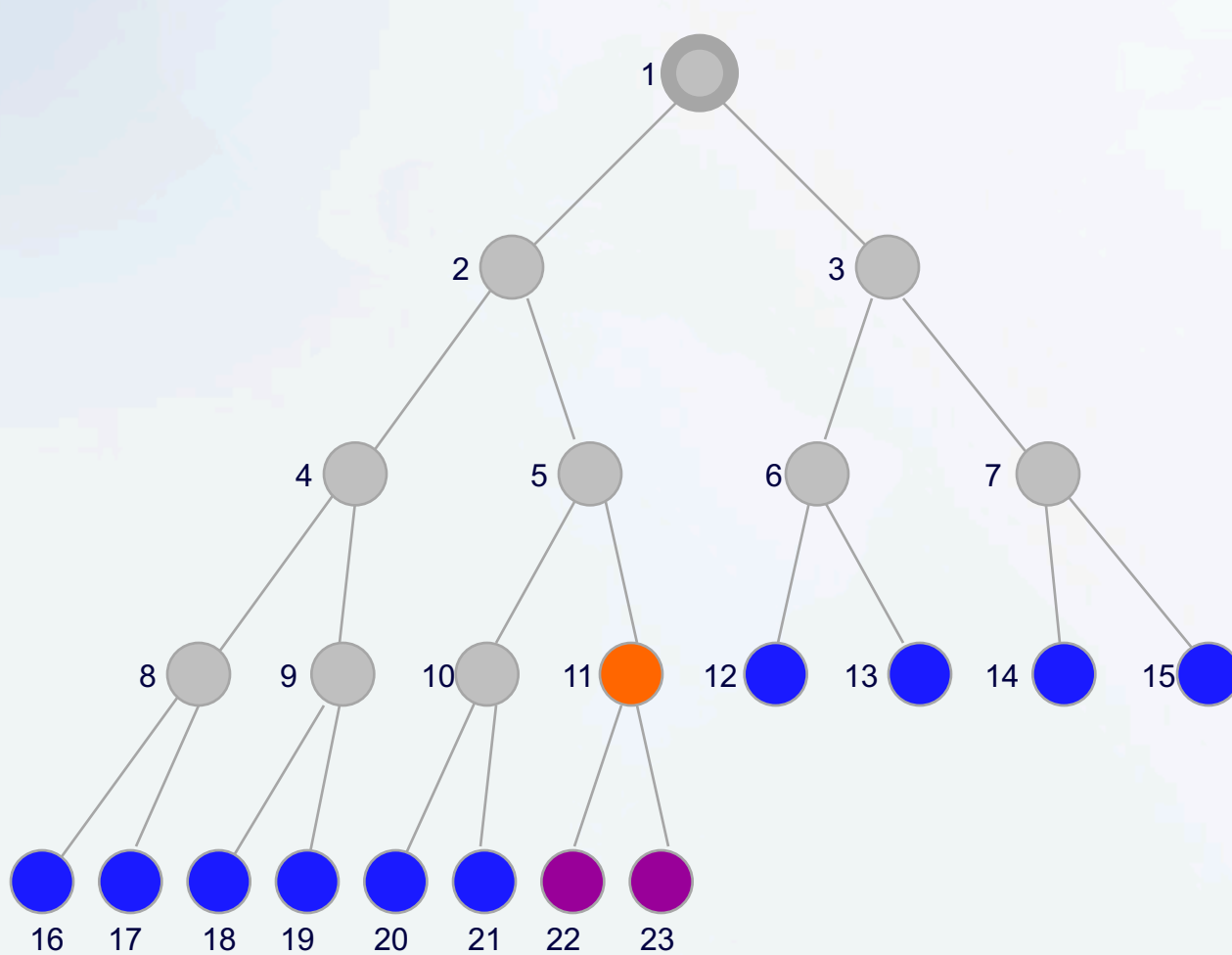
Initial

Visited

Fringe

Current

Visible

Goal

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

Breadth-First Snapshot 12

Initial
Visited
Fringe
Current
Visible
Goal

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]

Jarrar © 2020

Breadth-First Snapshot 13
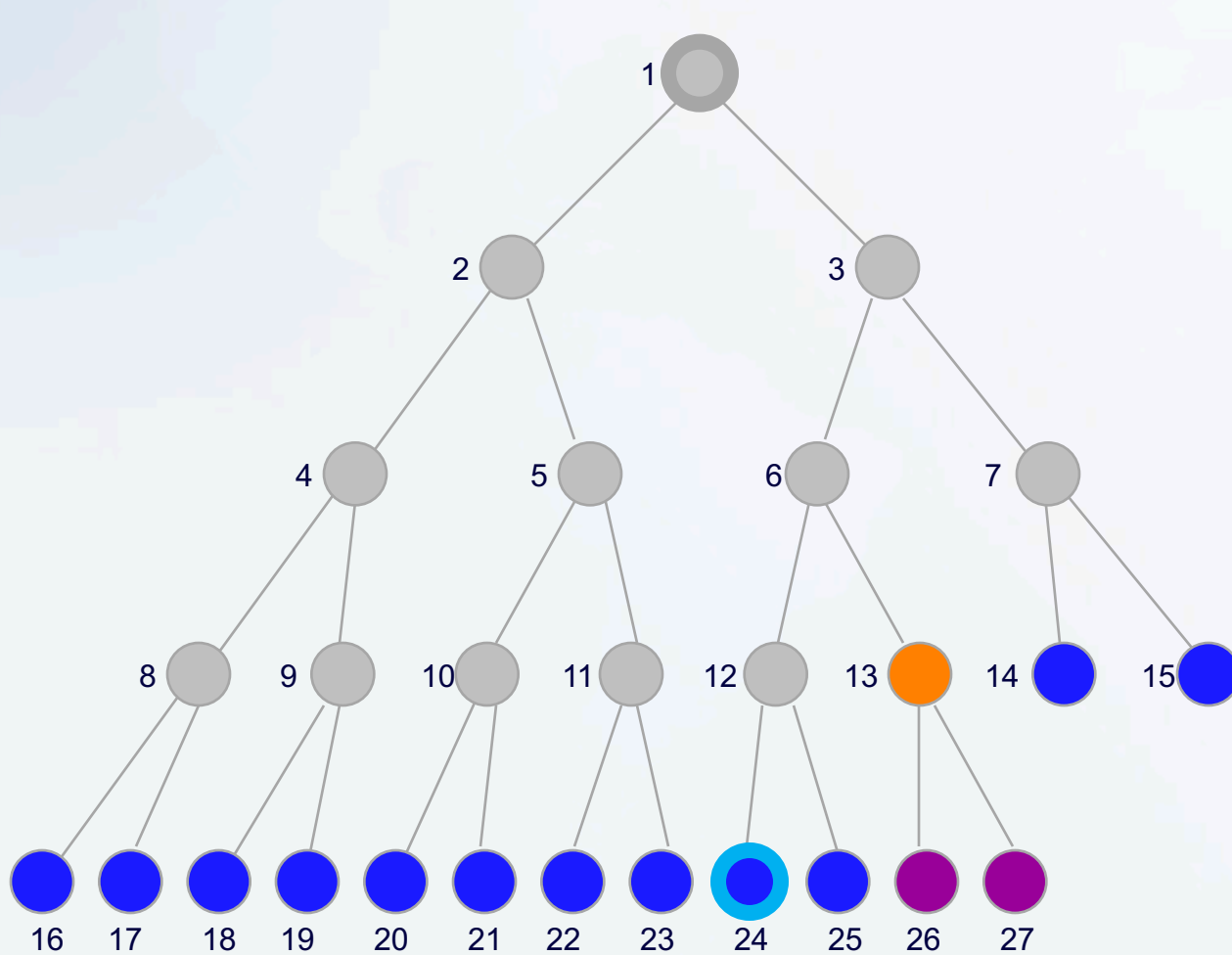
Fringe: [14,15,16,17,18,19,20,21,22,23,24,25] + [26,27]

Jarrar © 2020

Breadth-First Snapshot 14
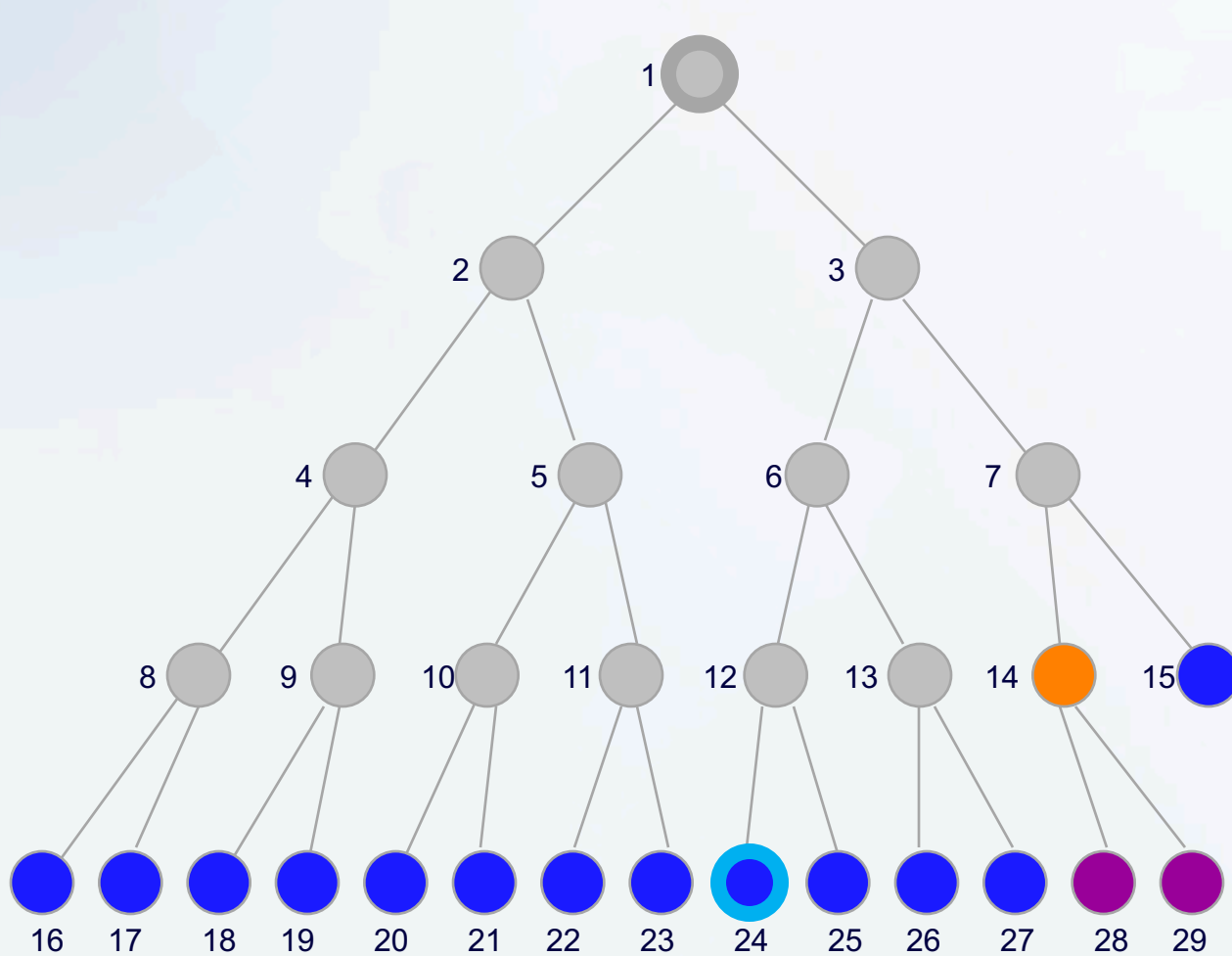
Initial
Visited
Fringe
Current
Visible
Goal

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

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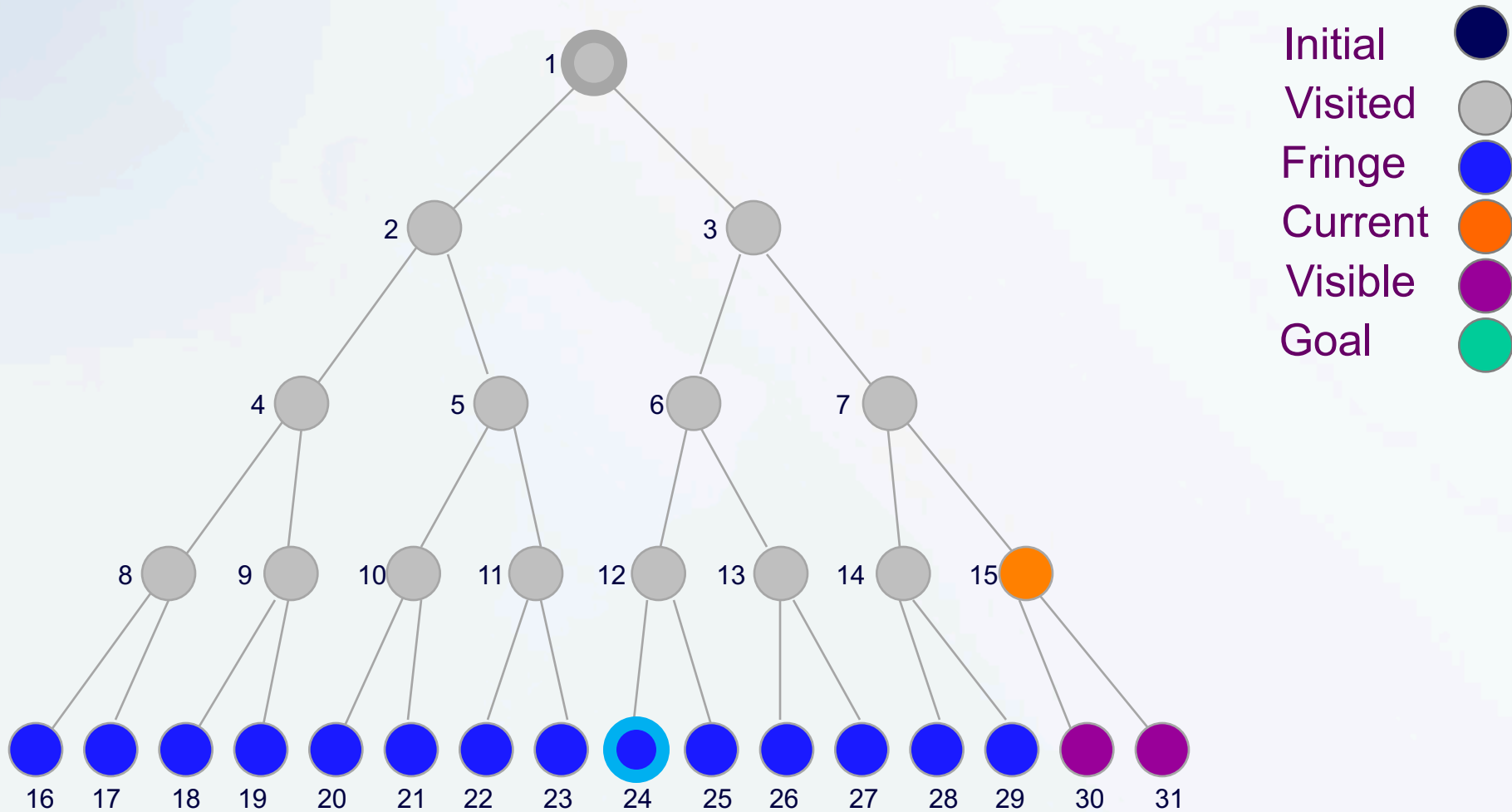
Fringe: [15,16,17,18,19,20,21,22,23,24,25,26,27,28,29] + [30,31]

Breadth-First Snapshot 16

Initial

Visited

Fringe

Current

Visible

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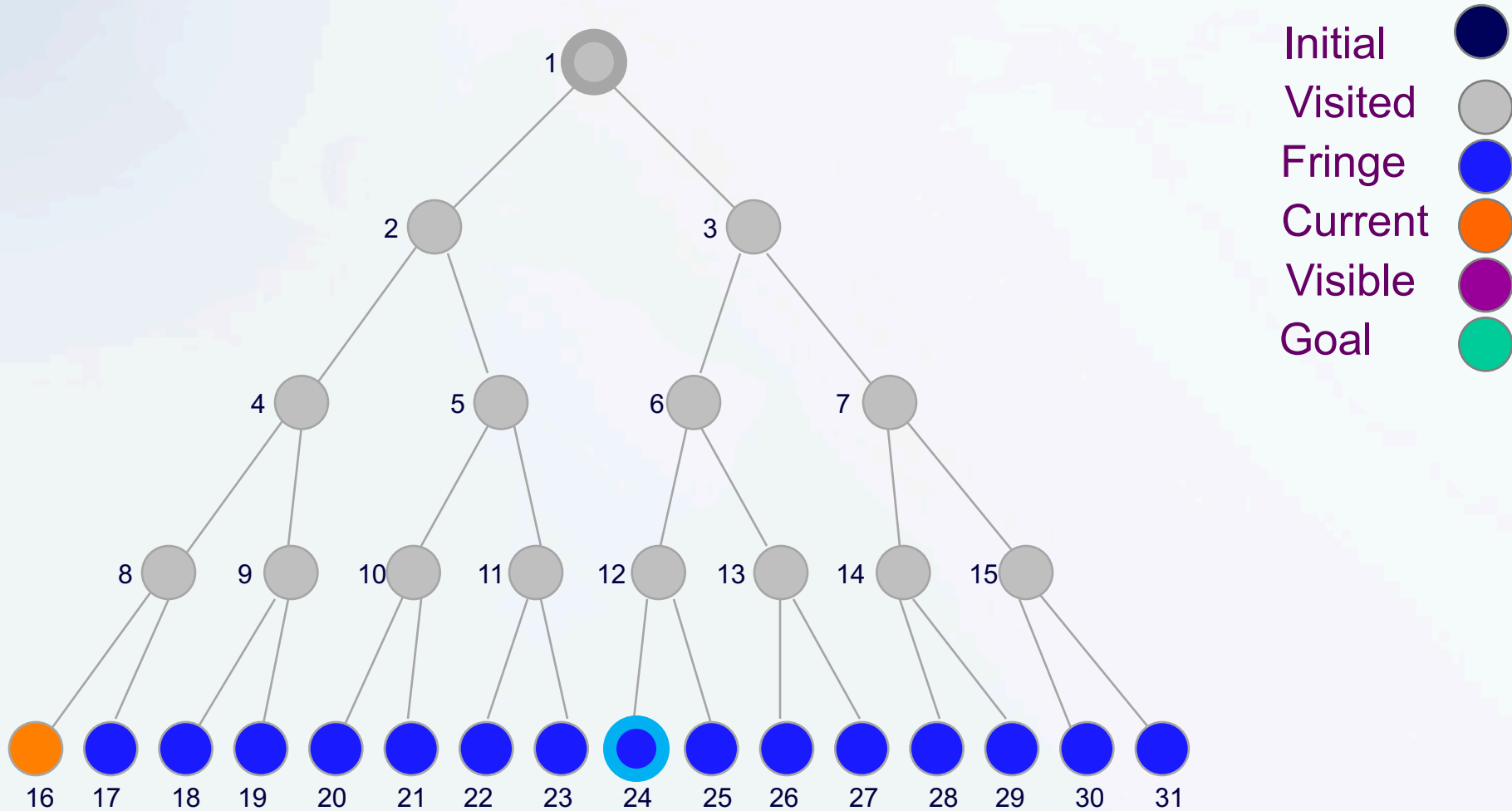
Fringe: [17,18,19,20,21,22,23,24,25,26,27,28,29,30,31]

Breadth-First Snapshot 17

Initial

Visited

Fringe

Current

Visible

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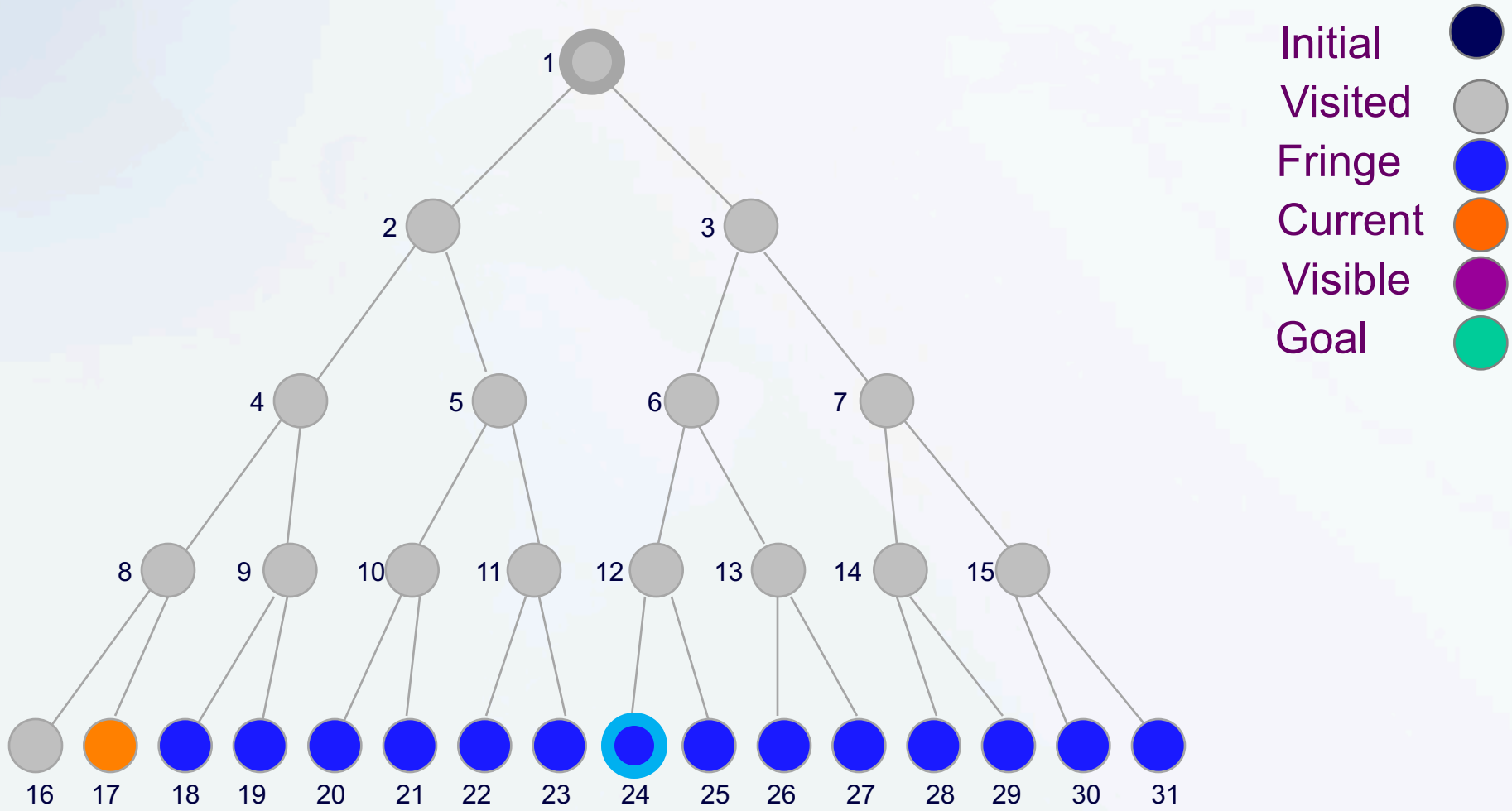
Fringe: [18,19,20,21,22,23,24,25,26,27,28,29,30,31]

Jarrar © 2020

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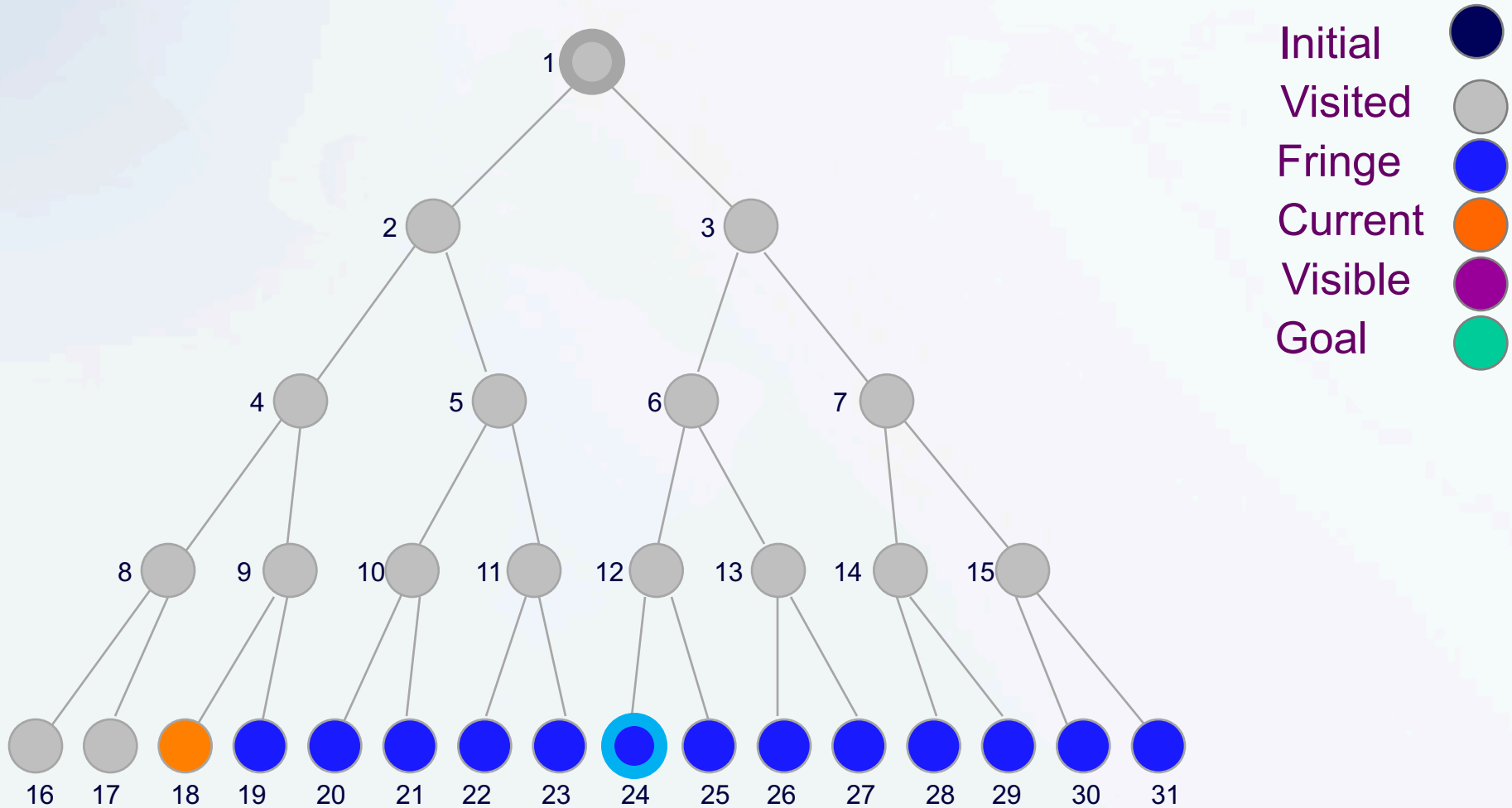
Fringe: [19,20,21,22,23,24,25,26,27,28,29,30,31]

Breadth-First Snapshot 19

Initial
Visited
Fringe
Current
Visible
Goal
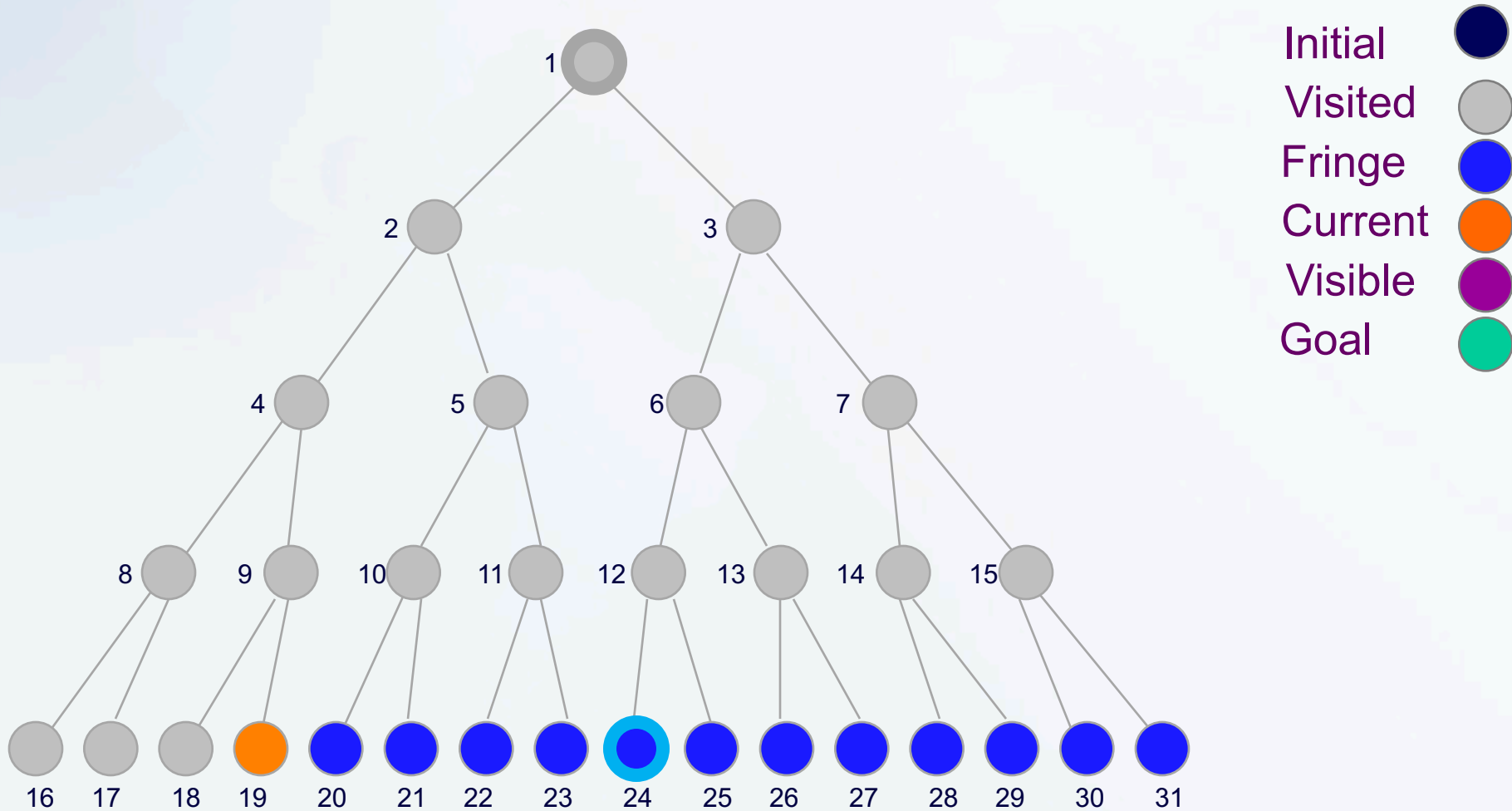
Fringe: [20,21,22,23,24,25,26,27,28,29,30,31]

Jarrar © 2020

Breadth-First Snapshot 20

Initial

Visited

Fringe

Current

Visible

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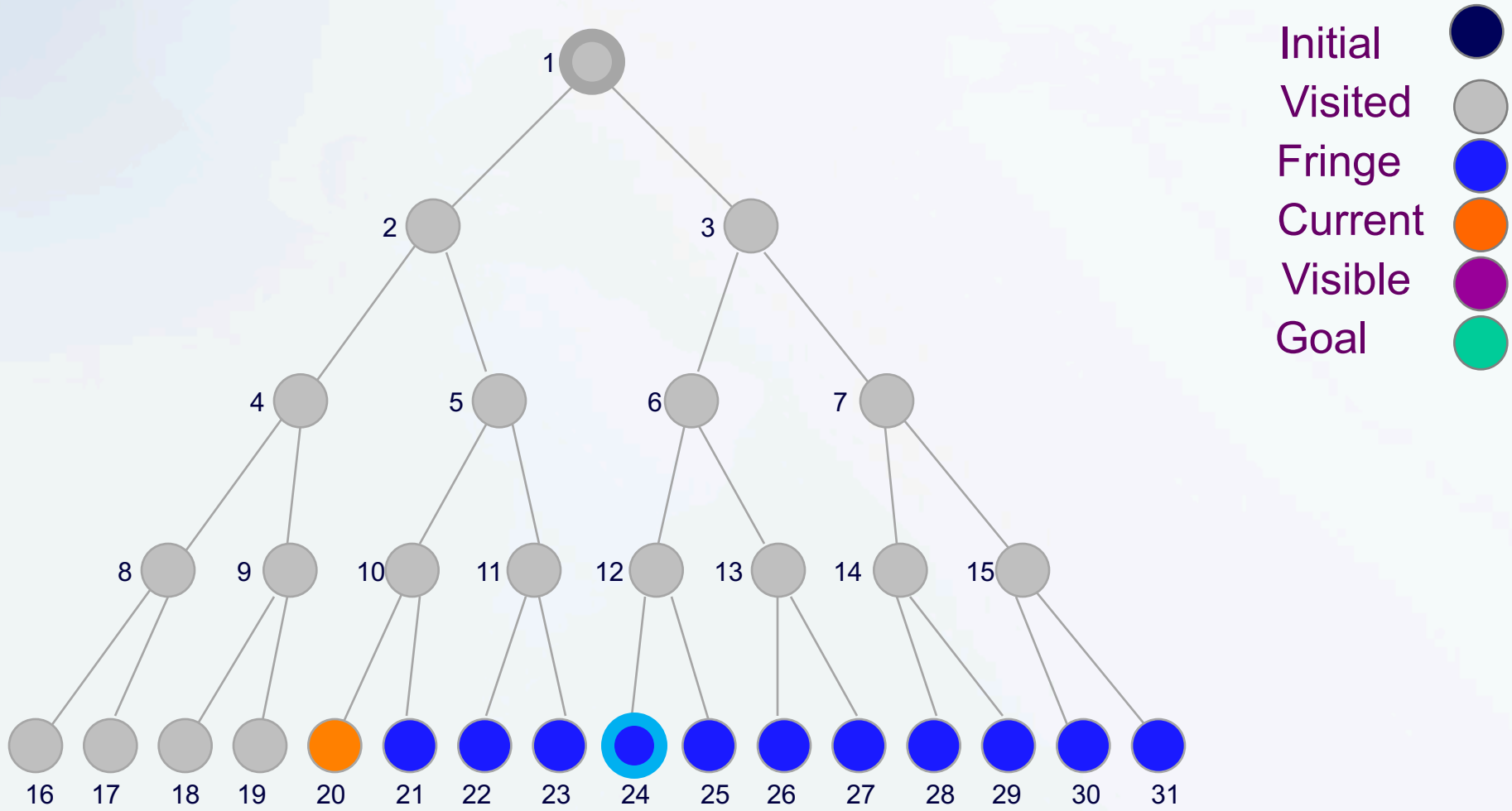
Fringe: [21,22,23,24,25,26,27,28,29,30,31]

Jarrar © 2020

43

Breadth-First Snapshot 21

Initial
Visited
Fringe
Current
Visible
Goal
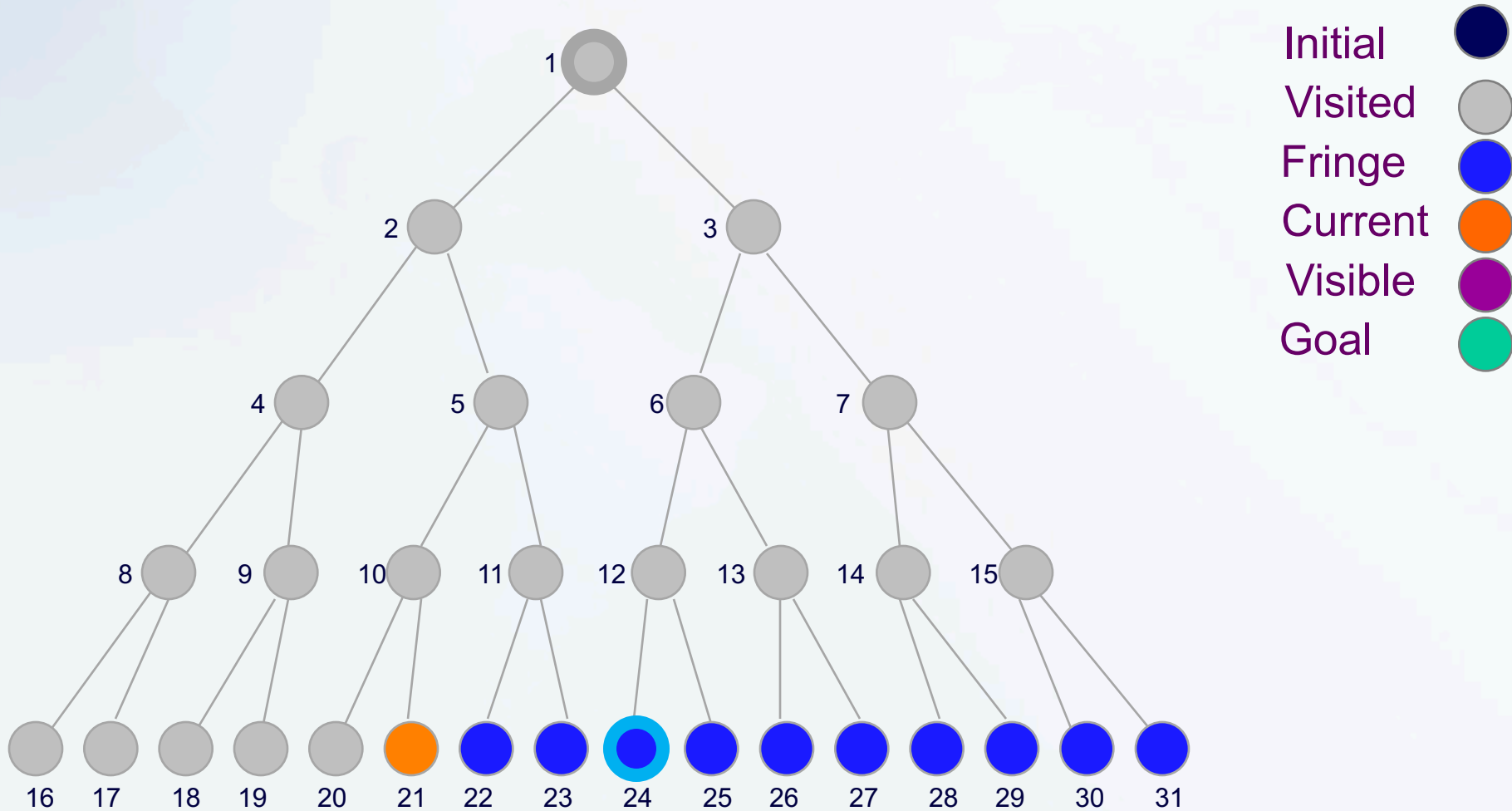
Fringe: [22,23,24,25,26,27,28,29,30,31]

Breadth-First Snapshot 22

Initial
Visited
Fringe
Current
Visible
Goal
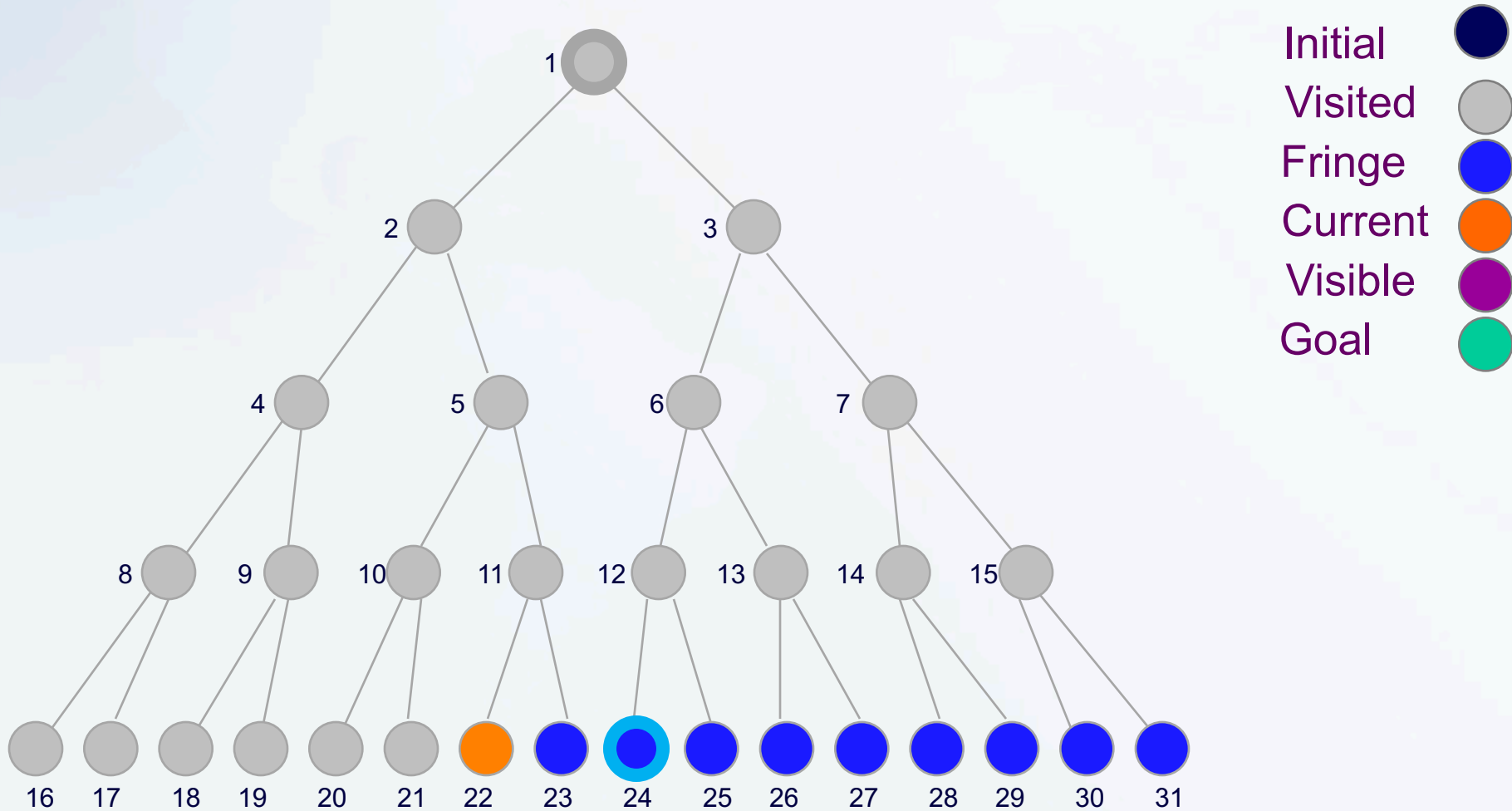
Fringe: [23,24,25,26,27,28,29,30,31]

45

Breadth-First Snapshot 23

Initial
Visited
Fringe
Current
Visible
Goal

1

2   3

4   5   6   7

8   9   10   11   12   13   14   15

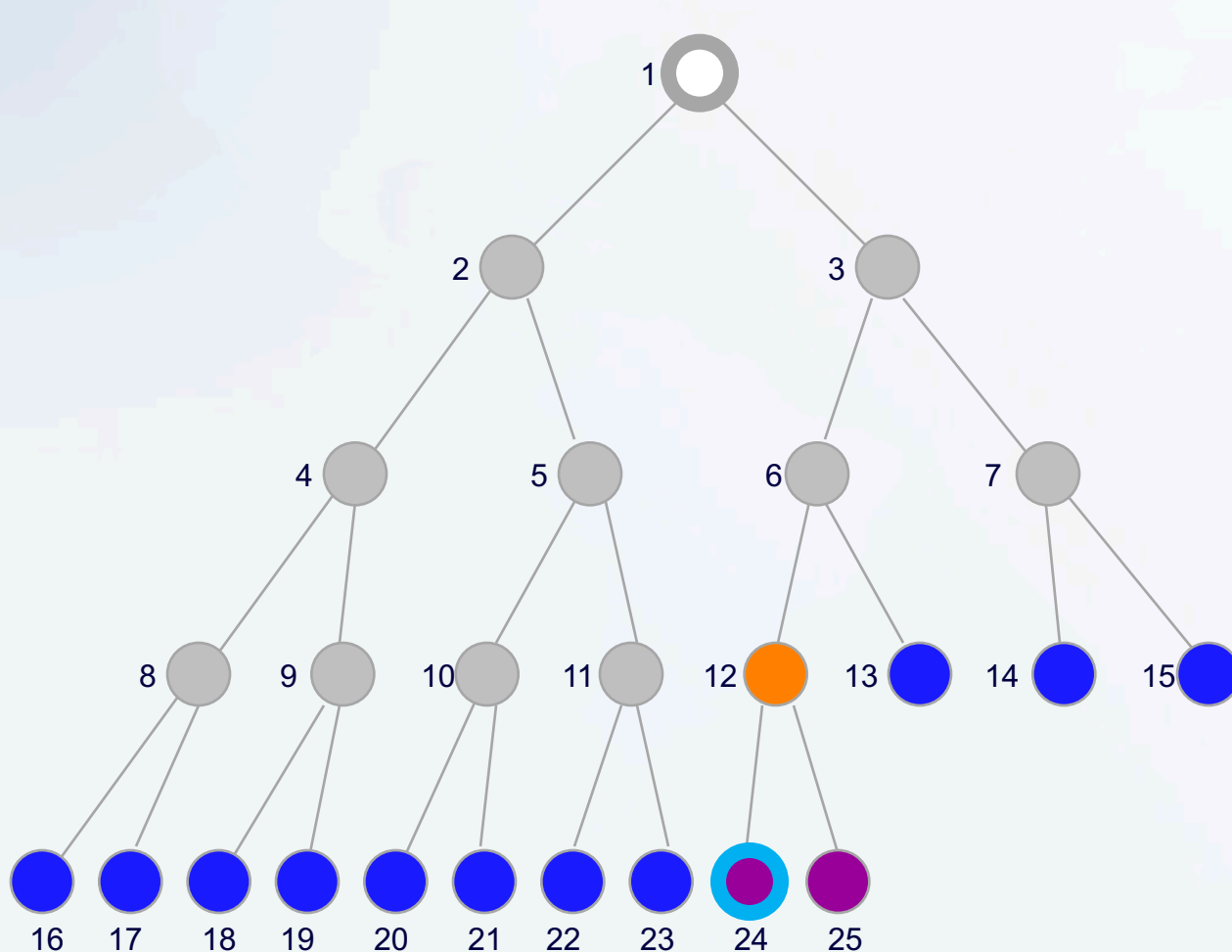16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
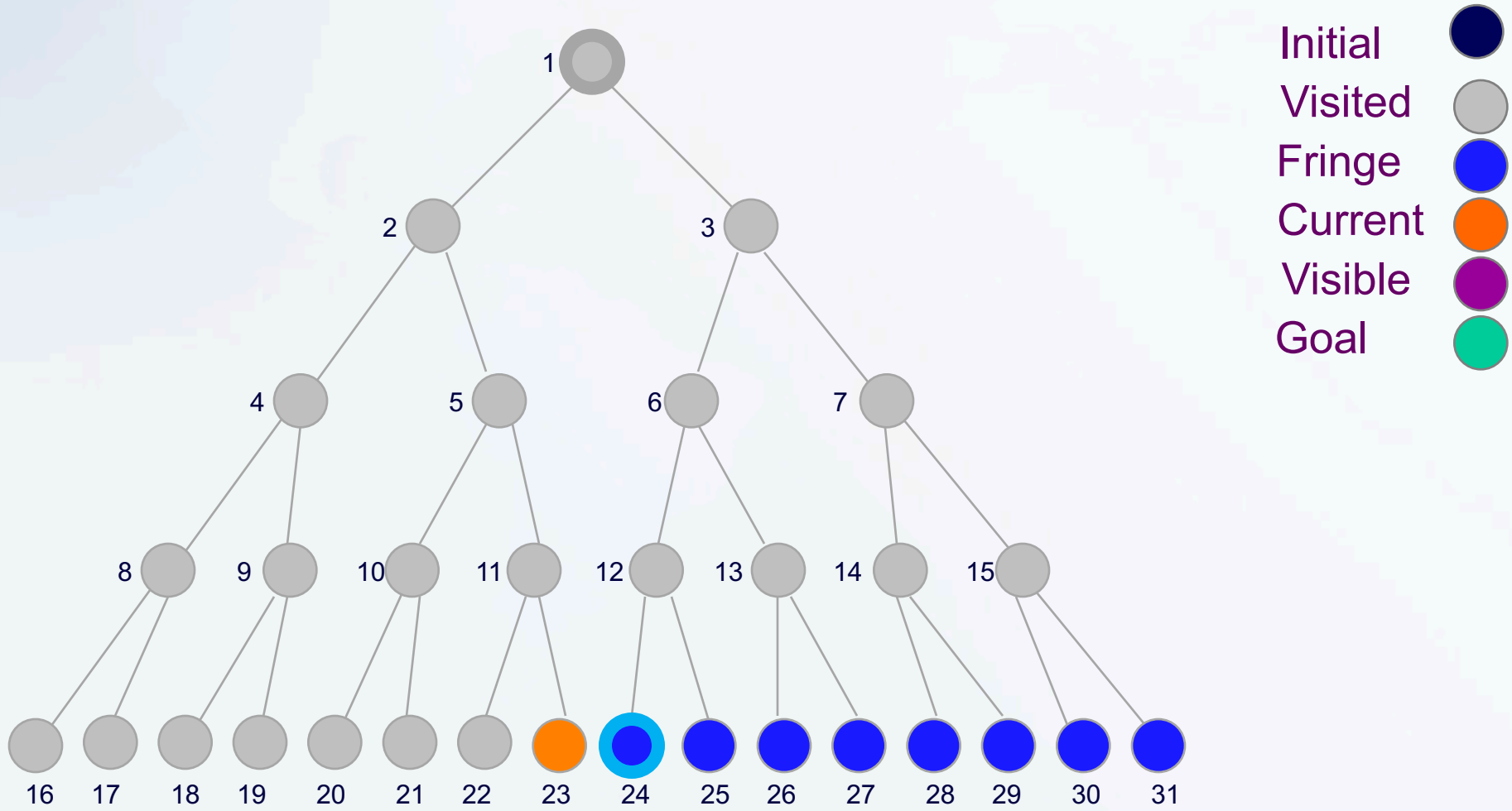
Fringe: [24,25,26,27,28,29,30,31]

Jarrar © 2020

46

Breadth-First Snapshot 24
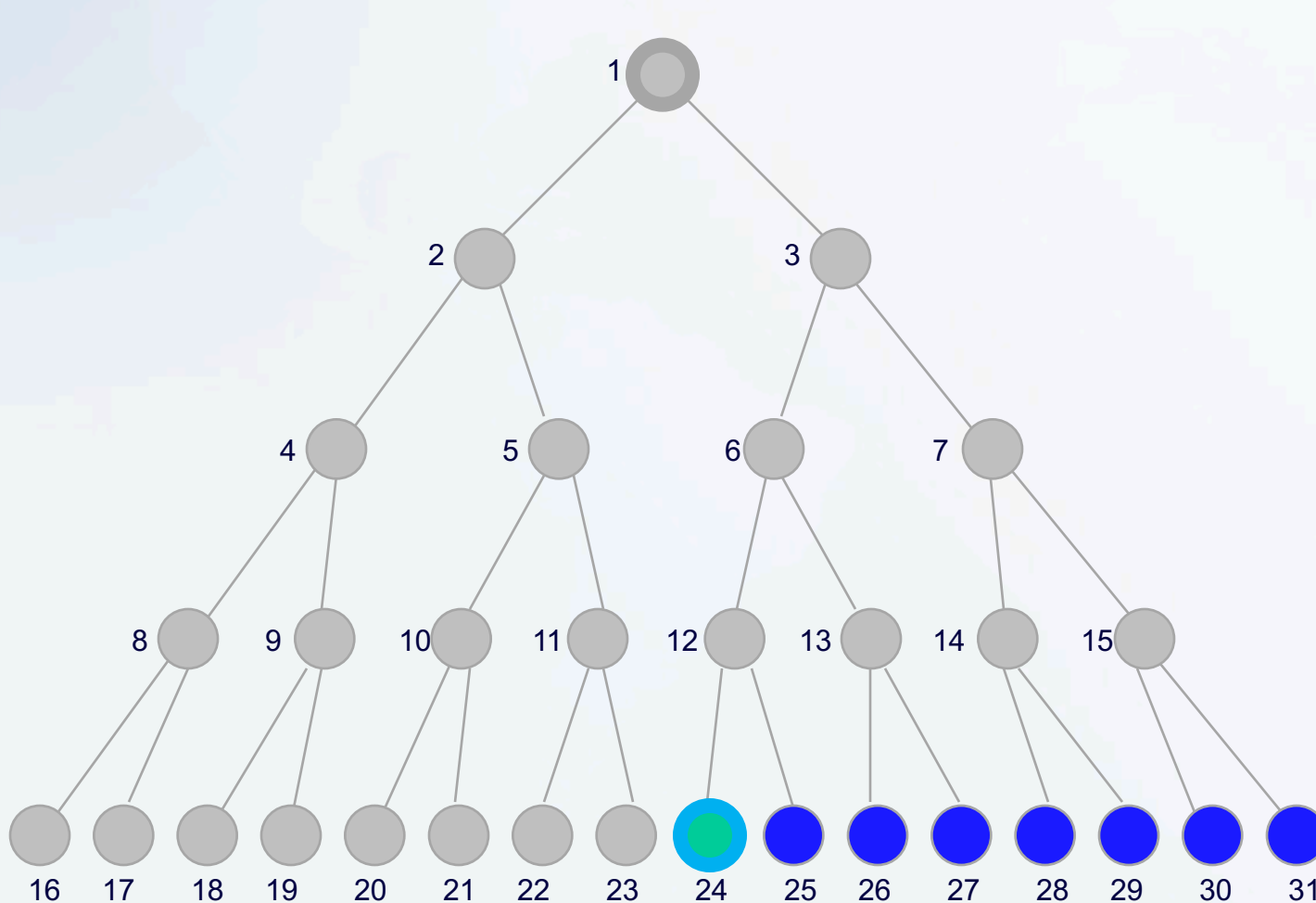
Initial
Visited
Fringe
Current
Visible
Goal

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]

Properties of Breadth-First Search (BFS)

<u>Completeness:</u> Yes (if *b* is finite), a solution will be found if exists.

<u>Time Complexity:</u> $1+b+b^2+b^3+\ldots+b^d + (b^{d+1}- b) = \mathbf{b^{d+1}}$ (nodes until the solution)

<u>Space Complexity:</u> $\mathbf{b^{d+1}}$ (keeps every generated node in memory)

<u>Optimality:</u> Yes (if cost = 1 per step)

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 Search (UCS)


```
generalSearch(problem, priorityQueue)
```
# of nodes tested: 0, expanded: 0

expnd. node	nodes list
	{S}
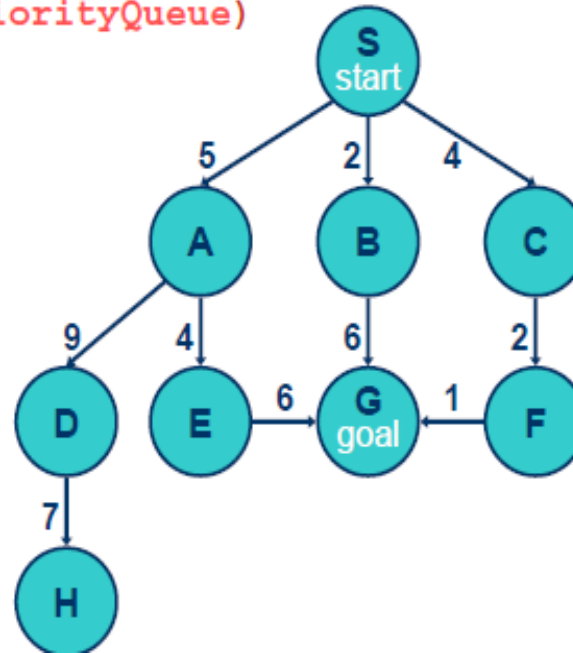
Uniform-Cost Search (UCS)


```
generalSearch(problem, priorityQueue)
```
# of nodes tested: 1, expanded: 1

expnd. node	nodes list
	{S:0}
S not goal	{B:2,C:4,A:5}
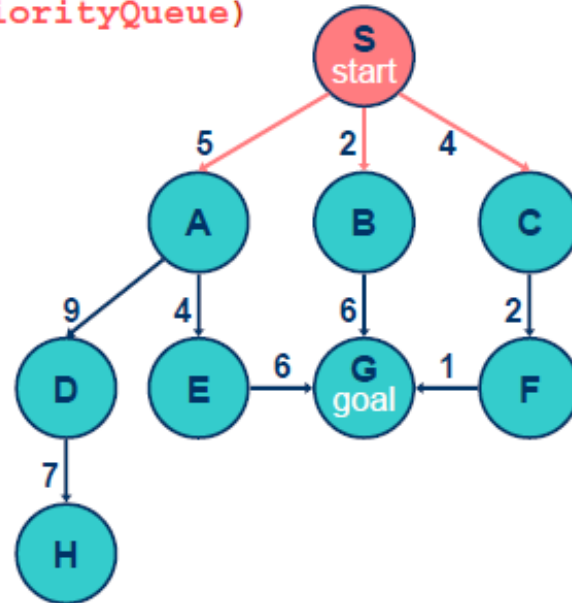
Uniform-Cost Search (UCS)

generalSearch(problem, priorityQueue)

# of nodes tested: 2, expanded: 2

expnd. node	nodes list
	{S}
S	{B:2,C:4,A:5}
B not goal	{C:4,A:5,G:2+6}
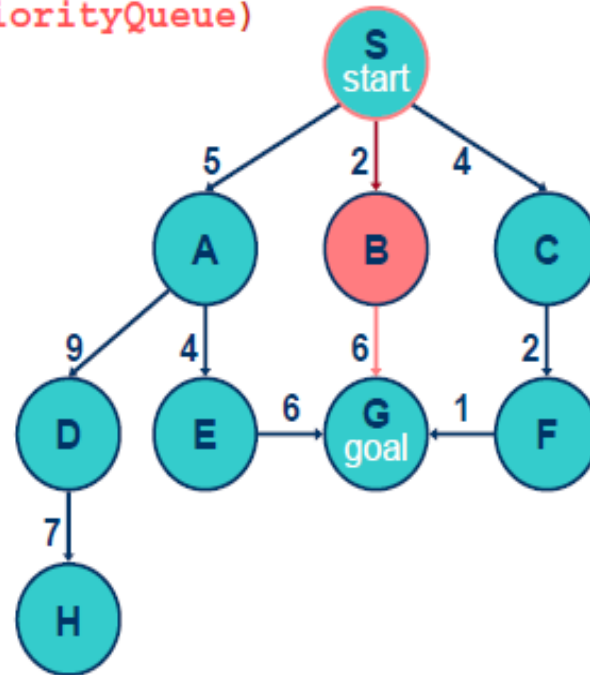
Uniform-Cost Search (UCS)

generalSearch(problem, priorityQueue)

\# of nodes tested: 3, expanded: 3

expnd. node	nodes list
	{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}
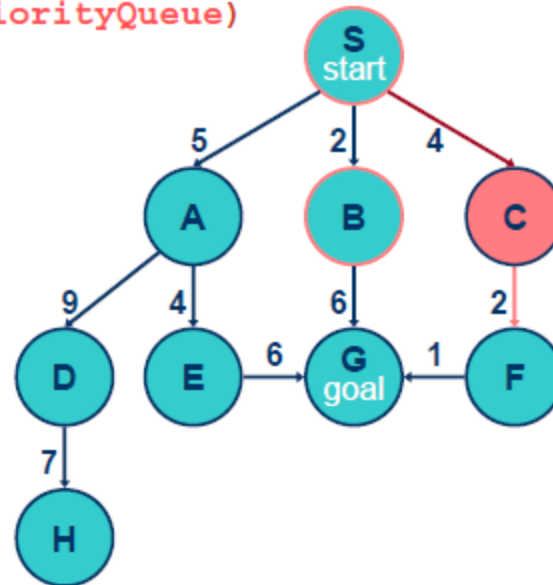
Uniform-Cost Search (UCS)

generalSearch(problem, priorityQueue)

\# of nodes tested: 4, expanded: 4

expnd. node	nodes list
	{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}
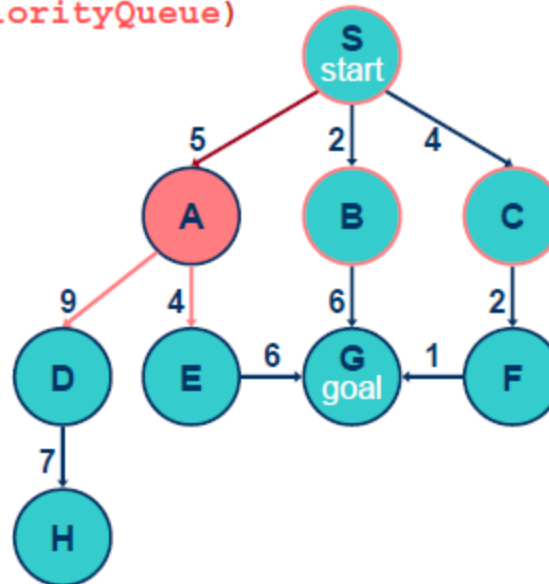
Uniform-Cost Search (UCS)

generalSearch(problem, priorityQueue)
# of nodes tested: 5, expanded: 5

expnd. node	nodes list
	{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}
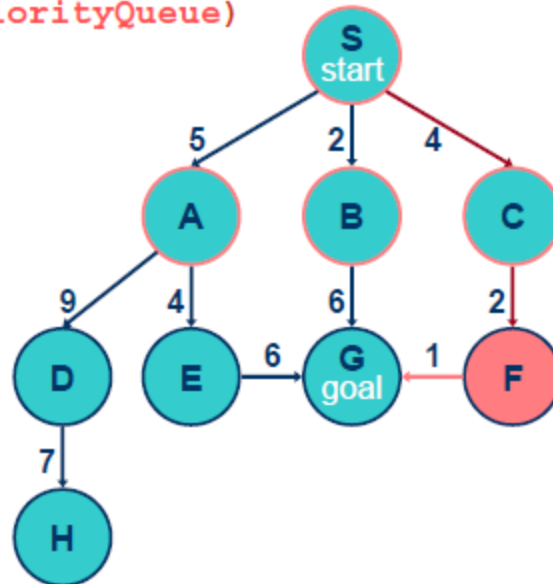
Uniform-Cost Search (UCS)

generalSearch(problem, priorityQueue)
# of nodes tested: 6, expanded: 5

expnd. node	nodes list
	{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
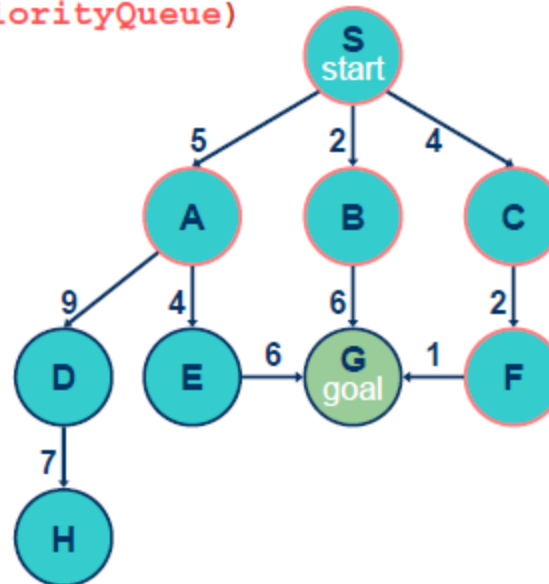
Uniform-Cost Search (UCS)

generalSearch(problem, priorityQueue)
# of nodes tested: 6, expanded: 5

expnd. node	nodes list
	{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}

path: **S,C,F,G**
cost: 7
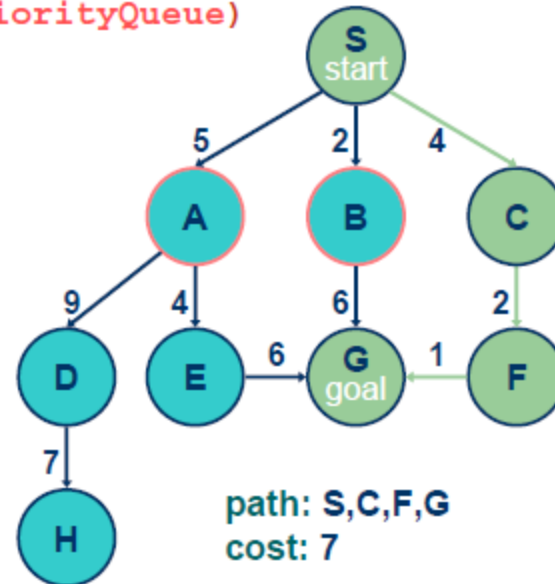
Properties of Uniform-cost Search (UCS)

<u>Completeness</u> Yes (if *b* is finite, and step cost is positive)

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

<u>Space Complexity:</u> as above

<u>Optimality:</u> 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
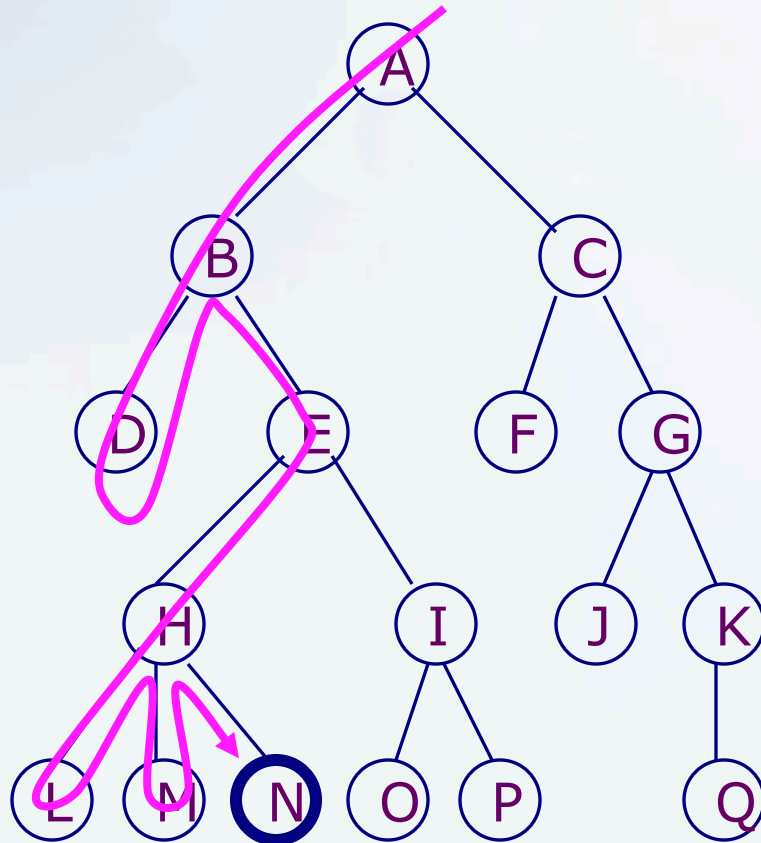
A depth-first search (DFS) explores a path all the way to a leaf before backtracking and exploring another path.

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

Put the root node on a stack;
while (stack is not empty) {
    remove a node from the stack;
    if (node is a goal node) return success;
    put all children of node onto the stack;
}
return failure;

At each step, the stack contains some nodes from each of a number of levels

– The size of stack that is required depends on the branching factor b

– While searching level n, the stack contains approximately $(b-1)*n$ nodes

When this method succeeds, it doesn't give the path

Recursive Depth-First Search

Search(node):
    if node is a goal, { print node and return success; }
    for each child c of node {
        if search(c) is successful, { print c and return success; }
    }
    return failure;

The (implicit) stack contains only the nodes on a path from the root to a goal

- The stack only needs to be large enough to hold the deepest search path
- When a solution is found, the path is on the (implicit) stack, and can be extracted as the recursion "unwinds"

Properties of Depth-First Search

Complete: No: fails in infinite-depth spaces, spaces with loops

– Modify to avoid repeated states along path

– $\rightarrow$ 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: $O(bm)$, i.e., linear space!

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

- <u>Complete?</u> no (if goal beyond l (l<d), or infinite branch length)
- <u>Time?</u> b^l
- <u>Space?</u> $B*l$
- <u>Optimal?</u> No (if l < d)

b	branching factor
l	depth limit

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 failure

    inputs: *problem*, a problem

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

Limit = 0

Iterative deepening search I =1
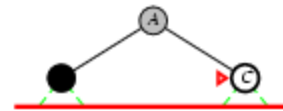
Limit = 1

Iterative deepening search I =2
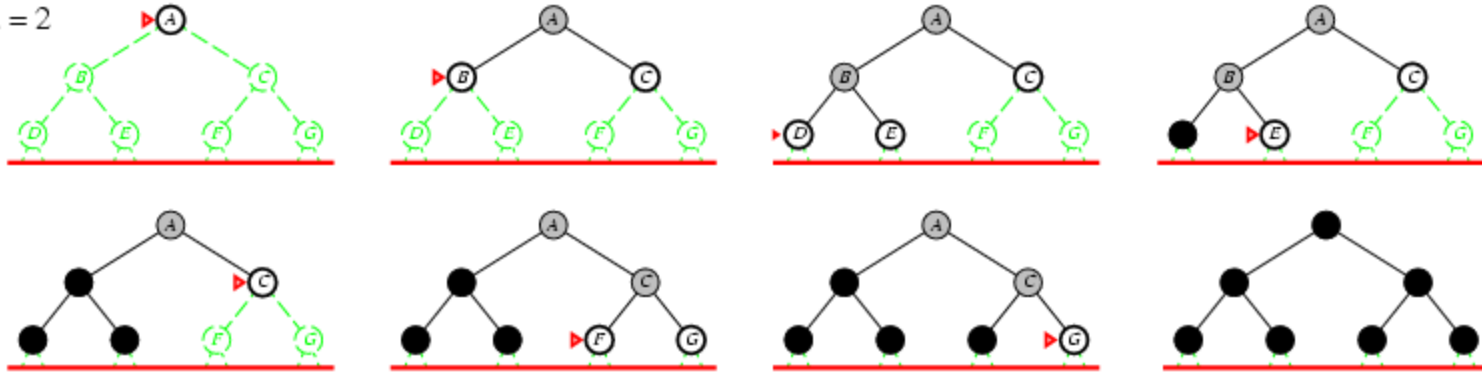
Iterative deepening search I =3
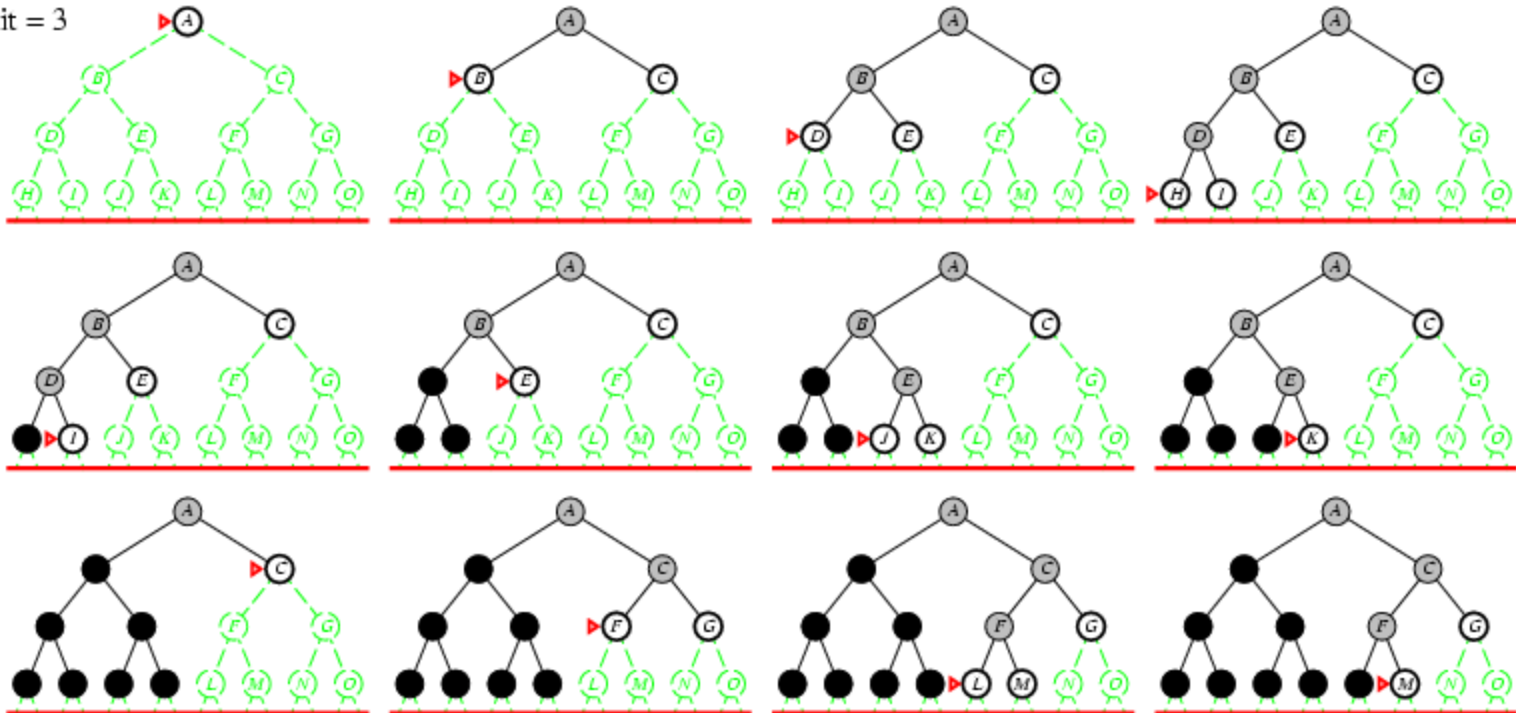
Iterative Deepening Depth-First Search

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: $(d+1)b^0 + d\,b^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$

Space: $O(bd)$

b	branching factor
d	Tree/goal depth

Optimal: Yes, if step cost = 1

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:

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

Time complexity = b^d

Compared with BFS:

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

- 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

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

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 much less

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

- 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

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:

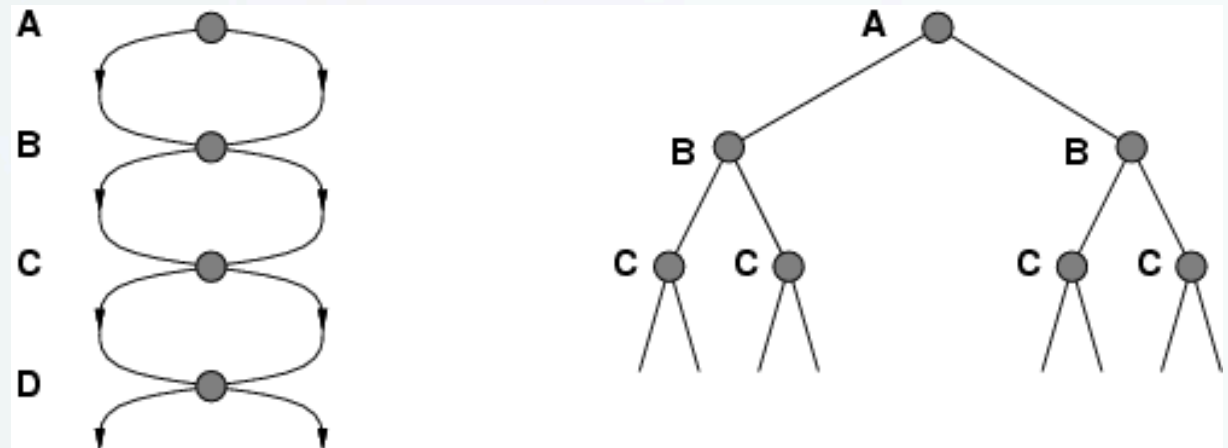– Space is limited and the shortest solution path is required

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

Project (Car Navigator)

Develop a simulation program (in JAVA) that takes as input (Palestine map, current town, goal town), and returns the path to the goal (shown on the screen), and the algorithm's time and space complexities.

- The simulator should implement 2 search algorithms (1 unfirmed (e.g. BFS) and 1 informed (e.g., A*)).
- The size of the map should be include: >60 nodes, branching >2, and depth >6.
- Each student should work alone, and should submit the program via Ritaj, which should include:
    1. Source code (in JAVA)
    2. towns.csv the list of towns, and the coordination (x, y) of each town on the screen.
    3. roads.csv the list of roads/links between towns, and the distance between them.

- The program must be fully running, partial implementations will not be considered, and it is assumed that each student writes every single line in the code him/herself.

- **Evaluation**: the discussion of the project will be online (with the instructor and the TA), and will include: **explaining the code, and answer practical and theoretical questions about all search algorithms taught in the course**.

- Deadline: 7/10/2020
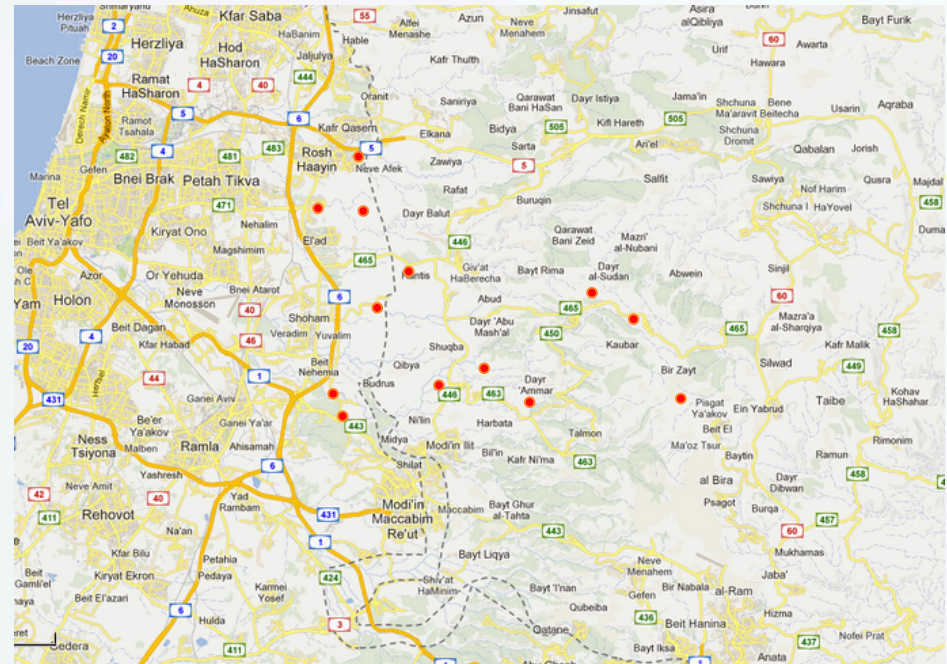
References

[1]   S. Russell and P. Norvig: Artificial Intelligence: A Modern Approach Prentice Hall, 2003, Second Edition

[2]   Franz Kurfess: Notes on Artificial Intelligence
        http://users.csc.calpoly.edu/~fkurfess/Courses/480/F03/Slides/3-Search.pdf

[3]   "Breadth-first Search." *Wikipedia*. Wikimedia Foundation. Web. 3 Feb. 2015. <http://en.wikipedia.org/wiki/Breadth-first_search>.

[4]   David Lee Matuszek: Lecture Notes on Tree Searches
        http://www.cs.nyu.edu/courses/fall06/V22.0102-001/lectures/treeSearching.ppt