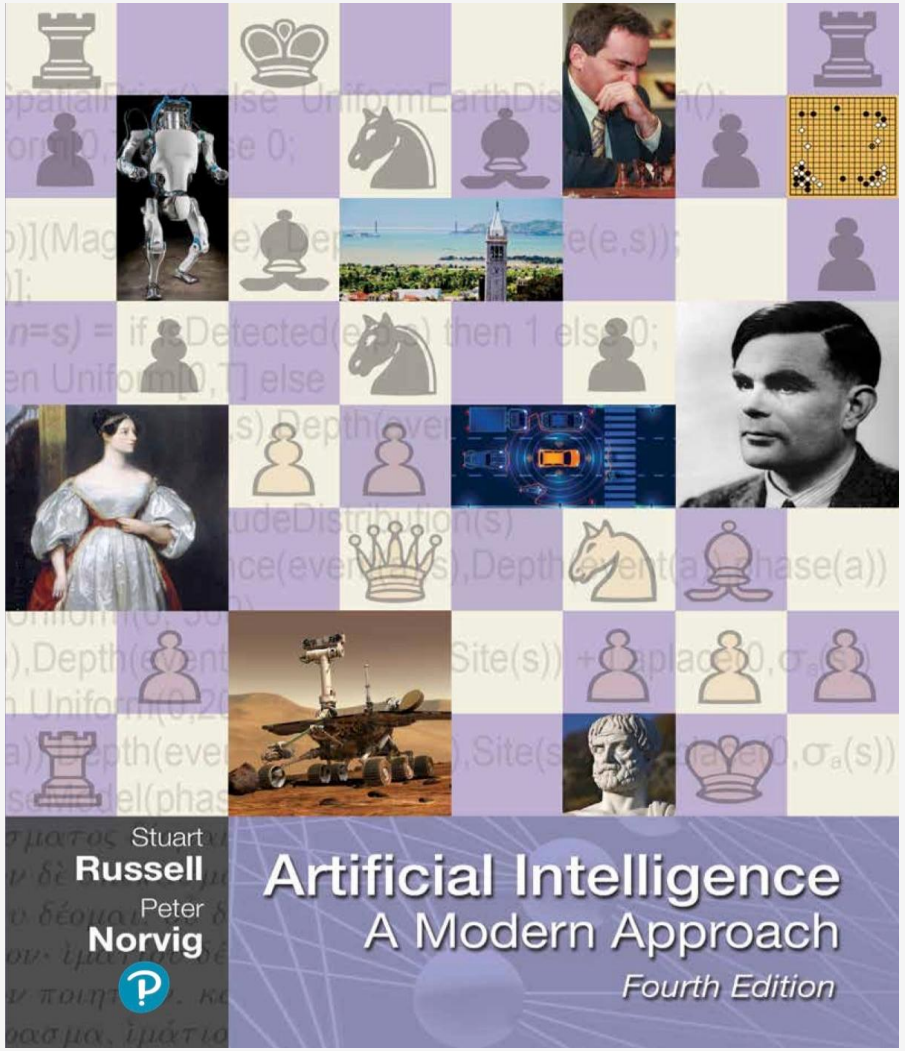
Artificial Intelligence

Prepared by:

Dr. Sara Sweidan

Supplementary Textbook



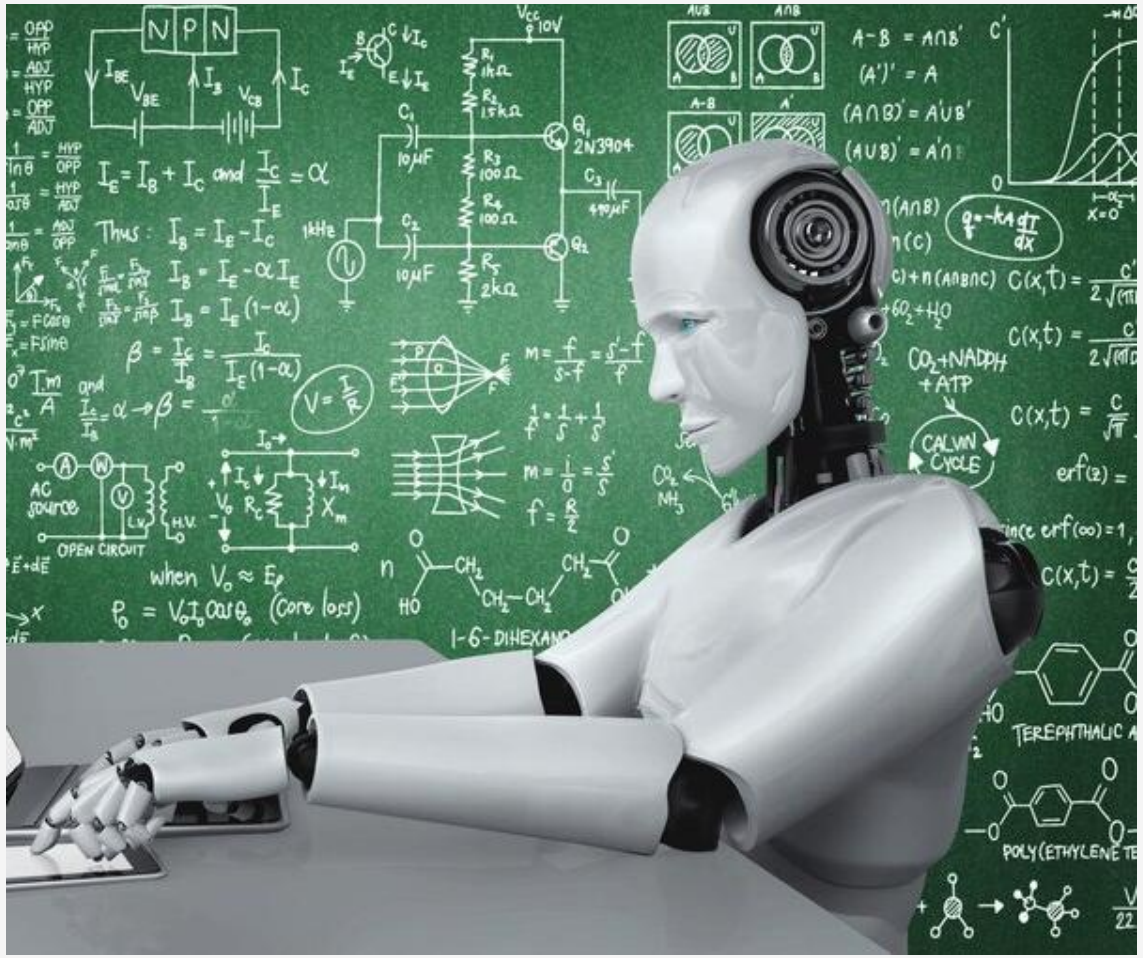
Book Title:
Artificial Intelligence, A modern approach

Authors:
Stuart Russell, Peter Norvig.

Publisher: Pearson.

Edition:
Fourth edition (2021)

Problem Solving as Search & State Space Search

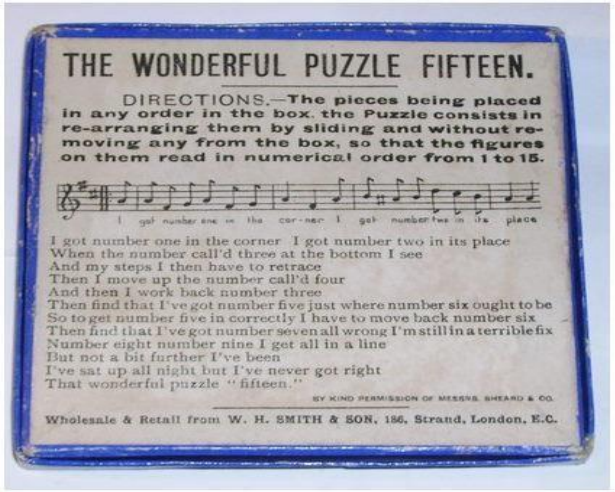


- Solving problems by searching.
- State Space Search Graph & Strategies
- Basic Idea of Search Algorithm

Why this chapter??

- To Understand and Learn:
  - How we can make an AI / Agent to solve the problem.
  - Problem solving using State Space Search
  - State Space and way to formulate a well defined problem
  - Strategies of search in state space/ ways to find solution in the state space.
  - Methods to compare the strategies
- Solving a problem means finding a sequence of actions that will eventually lead to desired goal.
- Finding a sequence of action when correct action to take is not obvious is called plan.

Solving Problems by SEARCHING



THE WONDERFUL PUZZLE FIFTEEN.

DIRECTIONS.—**The pieces being placed in any order in the box, the Puzzle consists in re-arranging them by sliding and without removing any from the box, so that the figures on them read in numerical order from 1 to 15.**

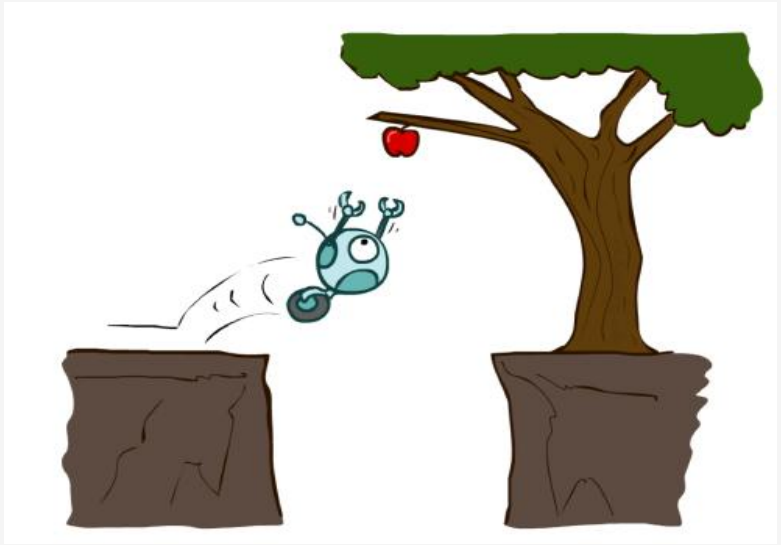
I got number one in the corner I got number two in its place
When the number call'd three at the bottom I see
And my steps I then have to retrace
Then I move up the number call'd four
And then I work back number three
Then find that I've got number five just where number six ought to be
So to get number five in correctly I have to move back number six
Then find that I've got number seven all wrong I'm still in a terrible fix
Number eight number nine I get all in a line
But not a bit further I've been
I've sat up all night but I've never got right
That wonderful puzzle "fifteen."

BY KIND PERMISSION OF MESSRS. SHEARD & CO.

Wholesale & Retail from W. H. SMITH & SON, 186, Strand, London, E.C.

TypesofAgents

a Reflex Agent :

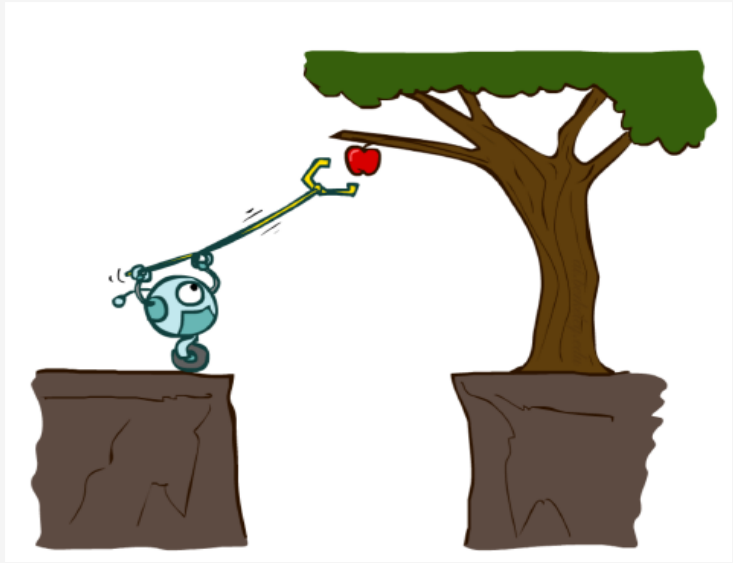


Considers how the world IS

- Choose action based on current percept.
- Do not consider the future consequences of actions.

Can a reflex agent be rational?

a Planning agent :



Considers how the world WOULD BE

- Decisions based on (hypothesized) consequences of actions.
- Must have a model of how the world evolves in response to actions.
- Must formulate a goal.

Source: D. Klein, P. Abbeel

Problem Solving - Definition

- Problem Solving is a process of generating solutions from observed data.
- Finding a sequence of actions that form a path to a goal state.
- Problem is characterized by set of states and set of operations and a set of goals.
- The method of solving problem through AI involves the process of:
  - Defining the search space,
  - Deciding start and goal states
  - Finding the path from start state to goal state through search space.

Problem Solving

- Search space or problem space is an abstract space that has all valid states that can be generated by applying any combination of operators to states.
- Problem space can have one or more solutions
- Solution is a combination of operators and states that achieve the goals
- Search refers to a search for a solution in a problem space.

Problem Solving

Here we assume that the agents always have access to the world's information, such as a map. Thus, the four methods/phases to solve the problem by an agent or AIs:

- Goal Formulation: Decide the goal
- Problem Formulation:
  - Define a problem
  - Define state and actions allowed to reach a goal
  - This is abstract modeling of the problem
- Search:
  - Sequence of action agents take to reach the goal
  - Such a sequence is the solution.
  - Agent might have to simulated multiple sequences
- Execution:
  - Agent can now execute the solution.

State Space

- Definition:
  - State space is defined as a set of all possible states for a given problem
- Problem modeling technique.
- formalizes a problem in terms of initial state, goal state and actions

Problem-solving Agent

SimpleProblemSolvingAgent(percept)

state = UpdateState(state, percept)

if sequence is empty then

goal = FormulateGoal(state)

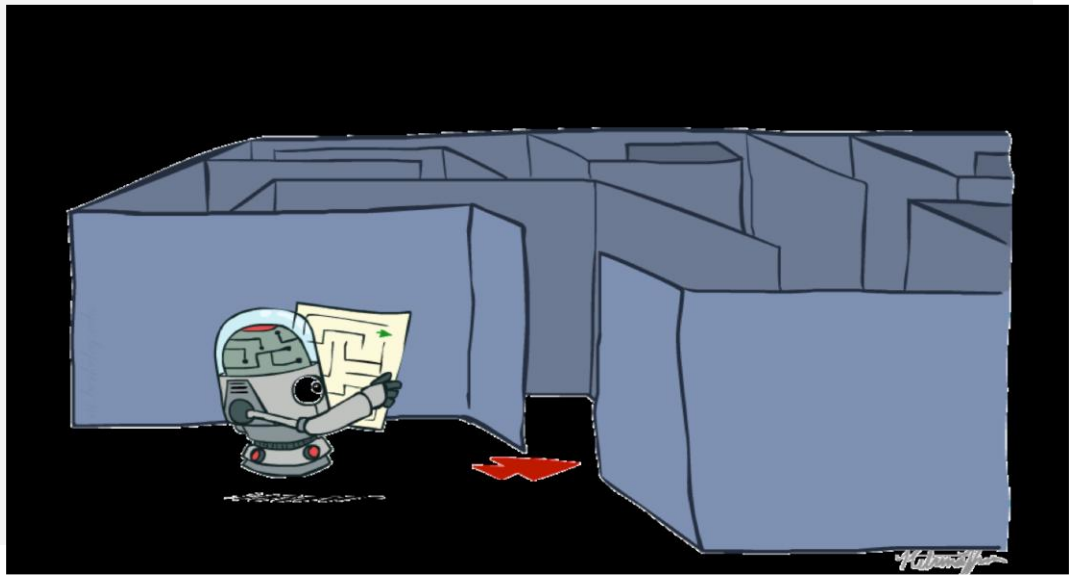
problem = FormulateProblem(state, g)

sequence = Search(problem)

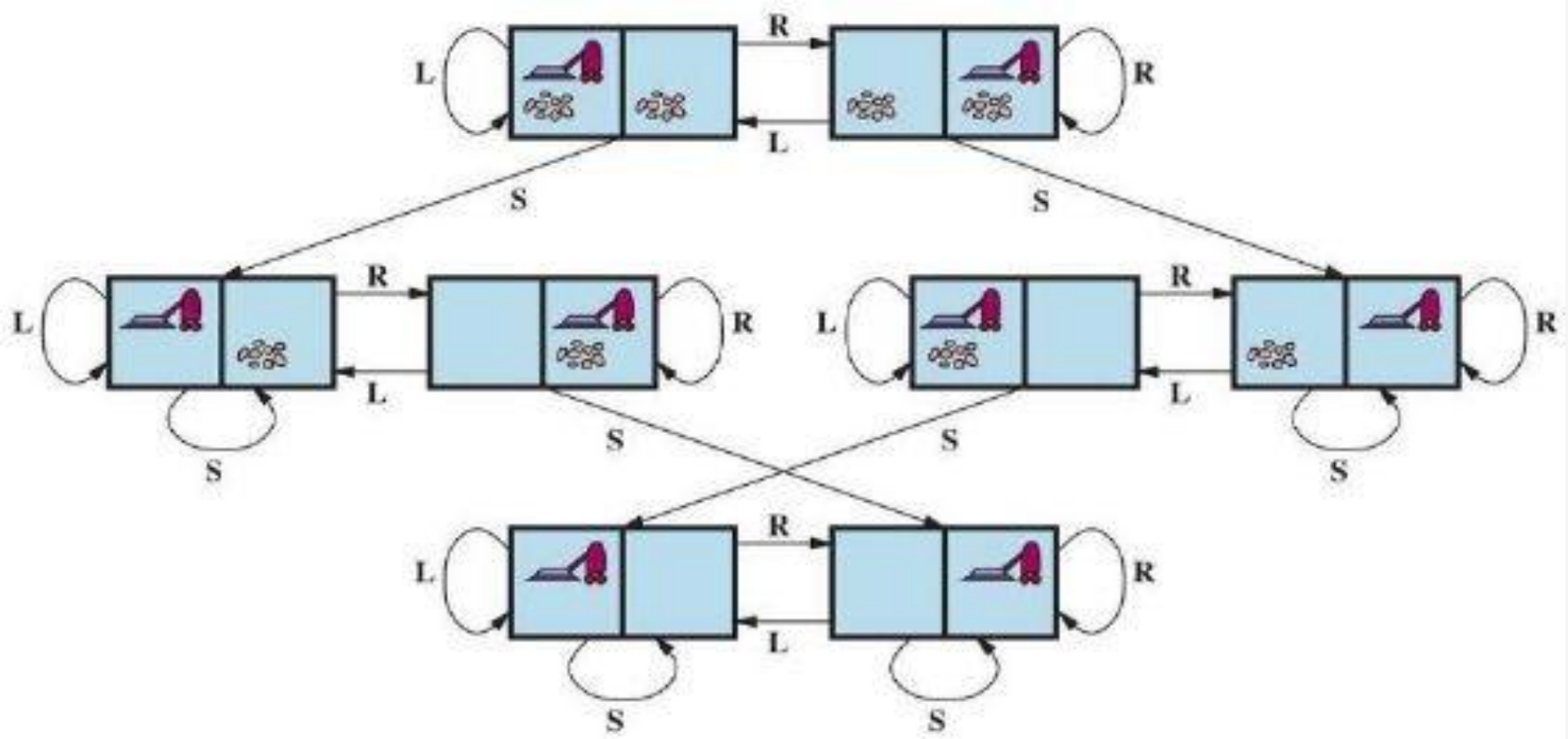
action = First(sequence)

sequence = Rest(sequence)

Return action



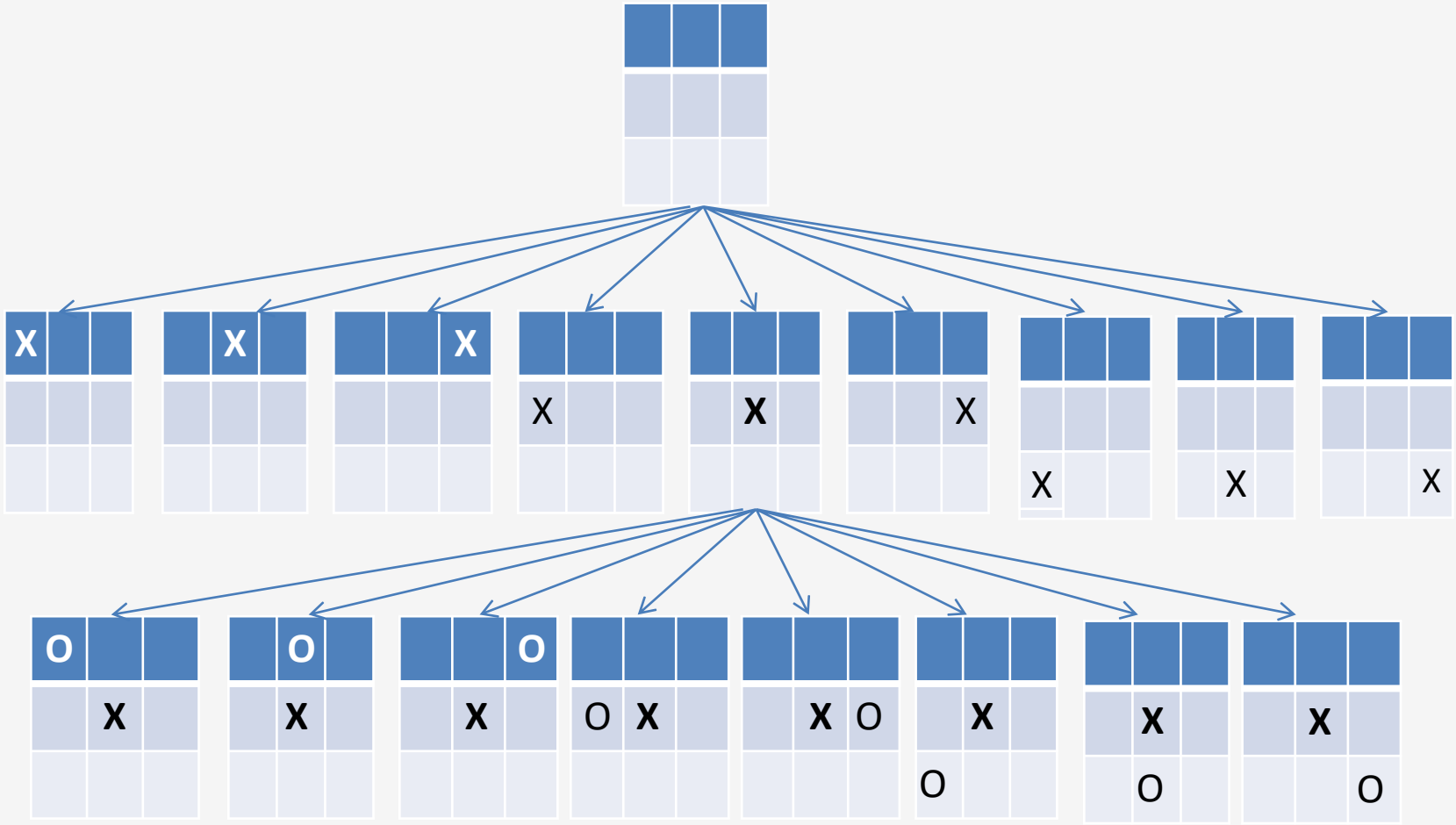
Problem Formulation: Using Cleaning Robot



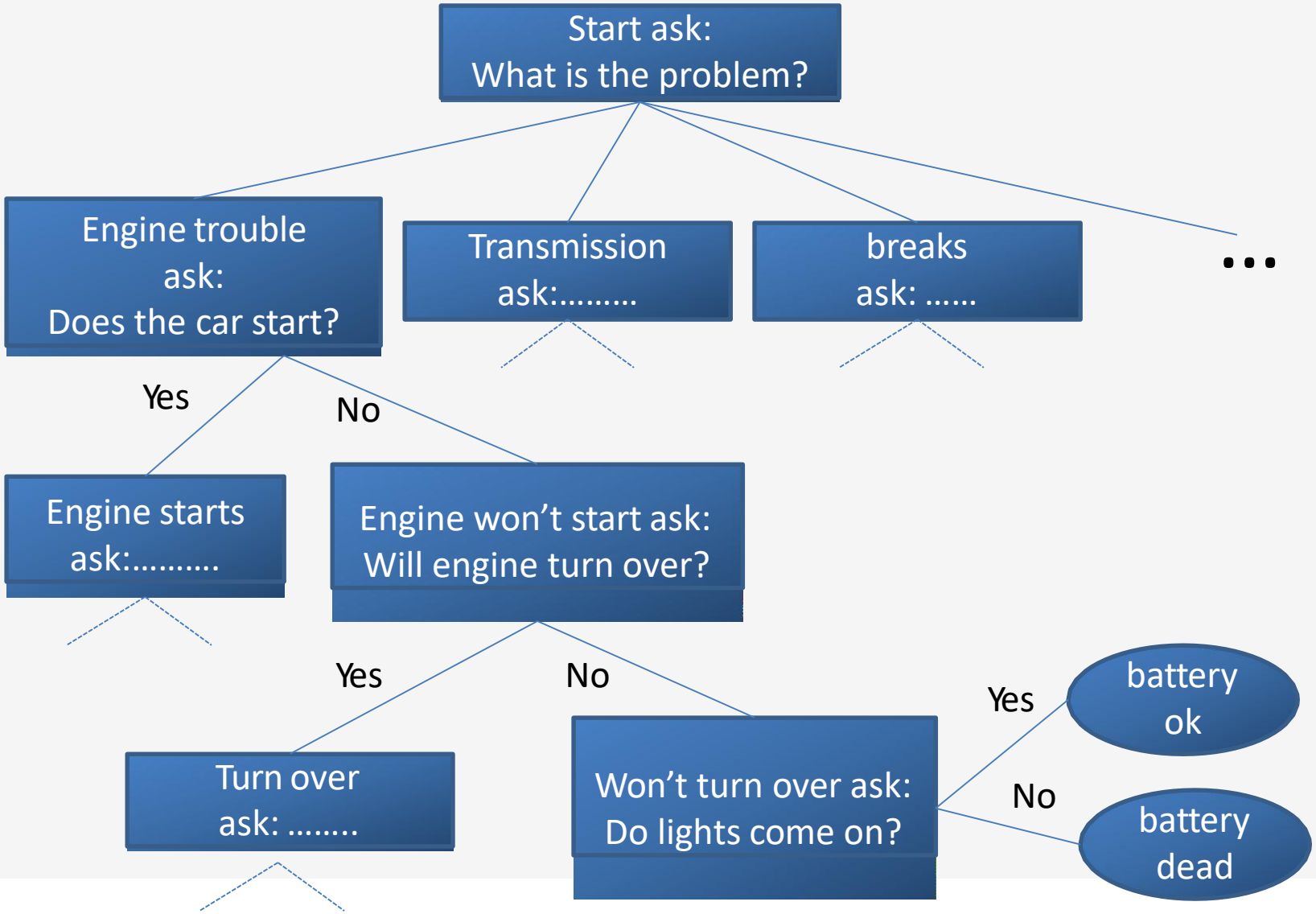
Formulation

- States: All possible state
- Initial State: Any state can be designed as an initial state.
- Actions: {moveLeft(), moveRight(), doClean()}
- Transition Model:
  - doClean() removes any dirt from the cell
  - moveLeft() moves toward the left, left cell exist
  - moveRight() moves toward the right, right cell exist
- Goal States:
  - The states in which every cell is clean
- Action Cost: Each action costs 1.

Example: Tic-Tac-Toe Game



Example: Mechanical Fault Diagnosing



State Space Search Strategies

There are two distinct ways for searching a state space graph:

- **Data-Driven Search:** (Forward chaining)

Start searching from the given data of a problem instance toward a goal.

- **Goal-Driven Search:** (Backward chaining)

Start searching from a goal state to facts or data of the given problem.

State Space Search Strategies..

Selecting Search Strategy

Data-Driven Search is suggested if:

- The data are given in the initial problem statement.
- There are few ways to use the given facts.
- There are large number of potential goals.
- It is difficult to form a goal or hypothesis.

Goal- Driven Search is appropriate if:

- A goal is given in the problem statement or can easily formulated.
- There are large number of rules to produce a new facts.
- Problem data are not given but acquired by the problem solver.

How Human Beings Think..?

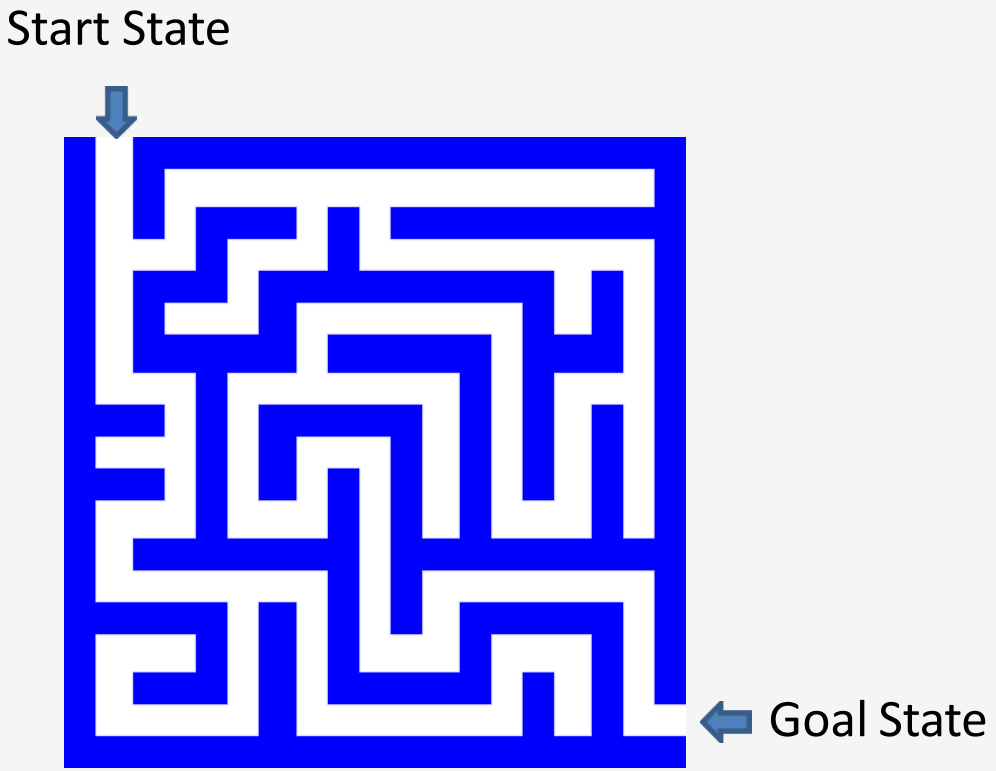
- Human beings do not search the entire state space (*exhaustive search*).
- Only alternatives that experience has shown to be effective are explored.
- Human problem solving is based on judgmental rules that limit the exploration of search space to those portions of state space that seem somehow promising.
- These judgmental rules are known as *“heuristics”.*

Heuristic Search

- A heuristic is a strategy for selectively exploring the search space.
- It guides the search along lines that have a high probability of success.
- It employs knowledge about the nature of a problem to find a solution.
- It does not guarantee an optimal solution to the problem but can come close most of the time.
- Human beings use many heuristics in problem-solving.

Search

We will consider the problem of designing **goal-based agents** in **fully observable**, **deterministic**, **discrete** environments.



Search

We will consider the problem of designing **goal-based agents** in **fully observable**, **deterministic**, **discrete** environments.

The agent must find a *sequence of actions* that reaches the goal. The **performance measure** is defined by:

(a) reaching the goal, *and ..*

(b) how “expensive” the path to the goal is.

Search Problem Components

Initial state

Actions

Transition model

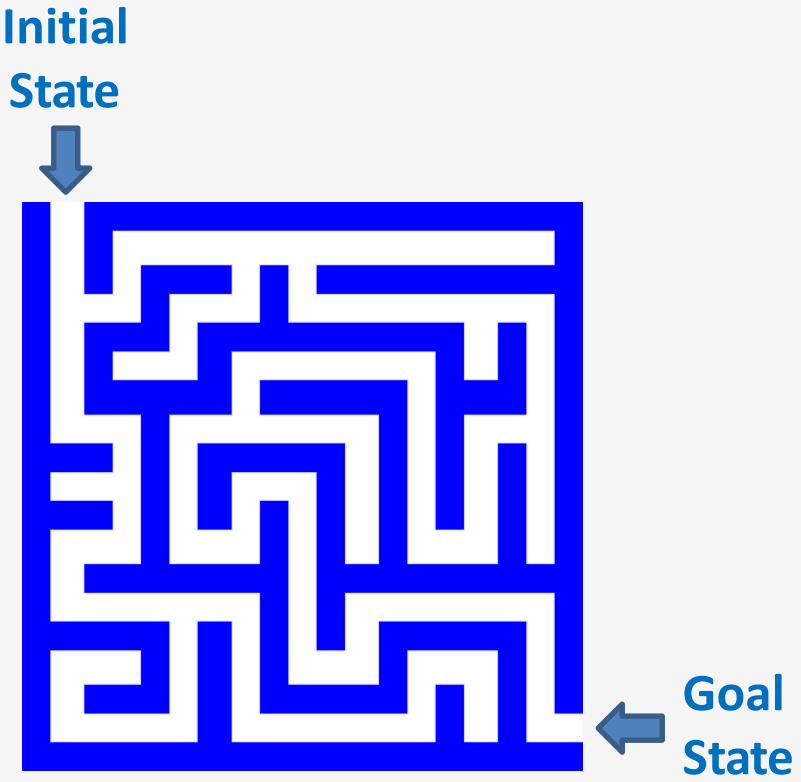
What state results from performing a given action in each state?

Goal state

Solution Path

Path cost

Assume that it is a sum of non-negative *step costs*



The **optimal solution** is the sequence of actions that gives the *lowest* path cost for reaching the goal.

Search Space Definitions

State

- A description of a possible state of the world
- Includes all features of the world that are pertinent to the problem

Initial state

- Description of all pertinent aspects of the state in which the agent starts the search

Goal test

- Conditions the agent is trying to meet (e.g., have $1M)

Goal state

- Any state which meets the goal condition

  ex: Thursday, have $1M, live in NYC

Action

- Function that maps (transitions) from one state to another

Search Space Definitions

Problem formulation

- Describe a general problem as a search problem

Solution

- Sequence of actions that transitions the world from the initial state to a goal state

Solution cost (additive)

- Sum of the cost of operators
- Alternative: sum of distances, number of steps, etc.

Search

- Process of looking for a solution
- Search algorithm takes problem as input and returns solution
- We are searching through a space of possible states

Execution

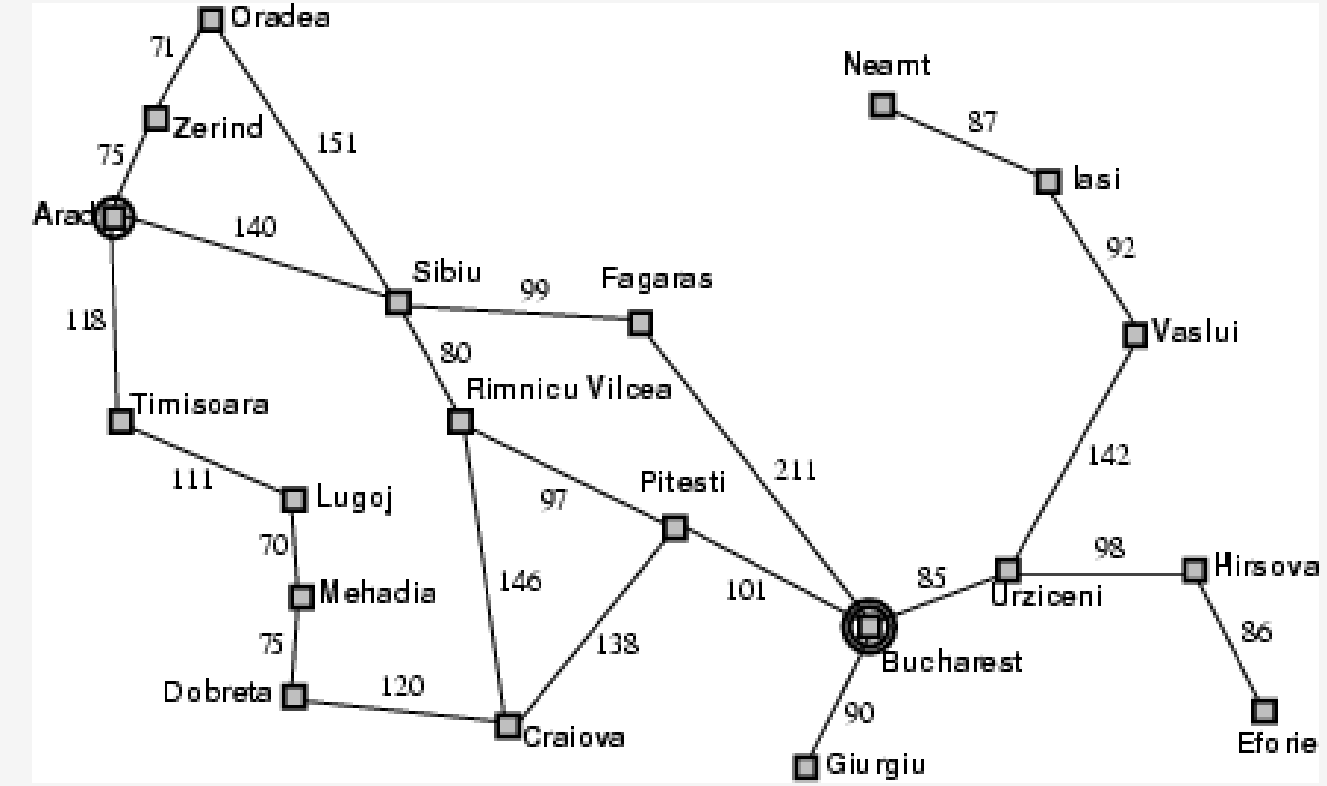
- Process of executing sequence of actions (solution)

Example: Romania

- On vacation in Romania; currently in Arad.
- Flight leaves tomorrow from Bucharest.

A search problem is defined by:

1. Initial state (e.g., Arad)
2. Operators (e.g., Arad -> Zerind, Arad -> Sibiu, etc.)
3. Goal test (e.g., at Bucharest)
4. Solution cost (e.g., path cost)



Example: Romania

- On vacation in Romania; currently in Arad.
- Flight leaves tomorrow from Bucharest.

Initial state

- Arad

Actions

- Go from one city to another

Transition Model

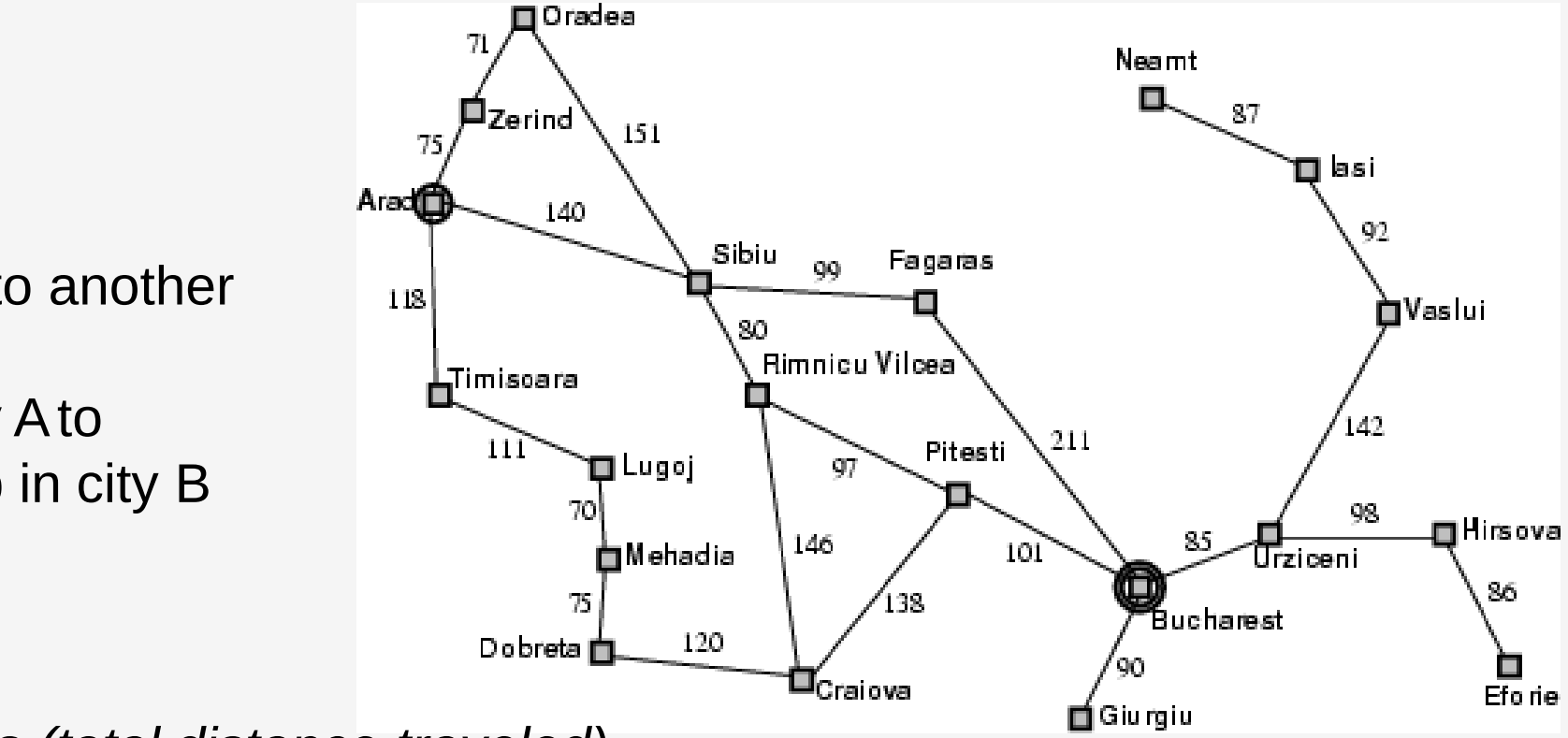
- If you go from city A to city B, you end up in city B

Goal State

- Bucharest

Path Cost

- Sum of edge costs *(total distance traveled)*

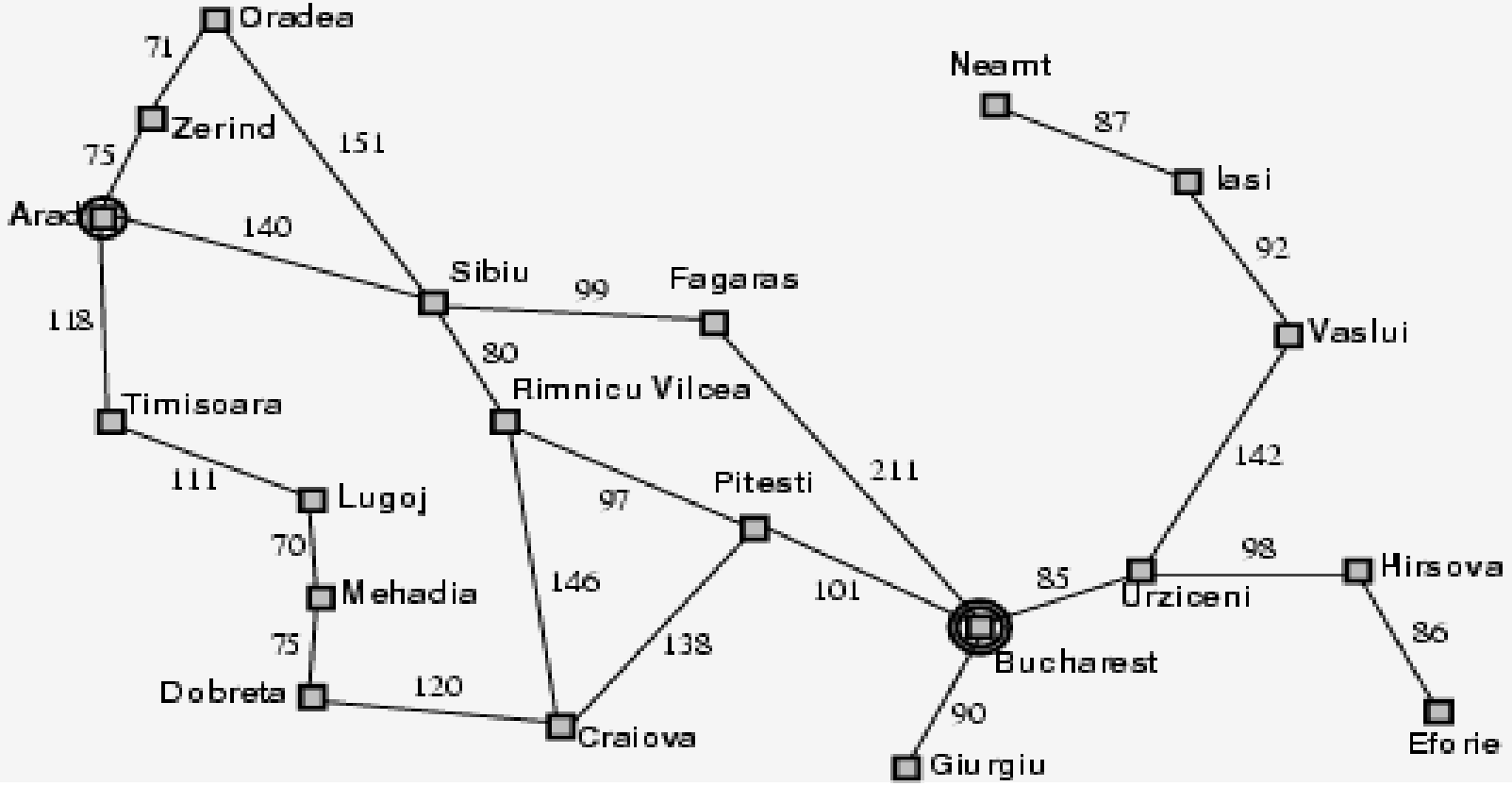


StateSpace

The initial state, actions, and transition model define the **state space** of the problem;

- The set of all states reachable from initial state by any sequence of actions.
- Can be represented as a **directed graph** where the nodes are states and links between nodes are actions.

What is the state space for the Romania problem?



Search problems and solutions

- A search **problem** can be defined as a set of possible **states** that the environment can be in. we call this **state space**.
- The **initial state** that the agent starts in. For example: Arad.
- A set of one or more **goal states**. Sometimes there is one goal state (Bucharest), or a set of alternative goal states.
- The **actions** available to the agent. Given a state *S*, action(s) returns a finite set of actions that can be executed in *S*. we say that each of these actions is applicable in *S.*
- *Actions(Arad)= {To_Sibiu, To_Timisoara, To_Zerind}.*

Search problems and solutions

- A **transition model**, which describes what each action does. Results (s, a) returns the state that results from doing action a in state s. For example, Results(*Arad*, *To_Zerind*) = *Zerind*.
- An **action cost function**, gives numeric cost of applying action *a* in the state *S* to reach s'.
- A sequence of actions forms a **path**, and a **solution** is a path from the initial state to a goal state. We assume that action costs are additive; that is, the total cost of a path is the sum of the individual action cost.
- An **optimal solution** has the lowest path cost among all solutions.
- The state space can be represented as a **graph** in which the vertices are states and the directed edges between them are actions.

Search problems and solutions

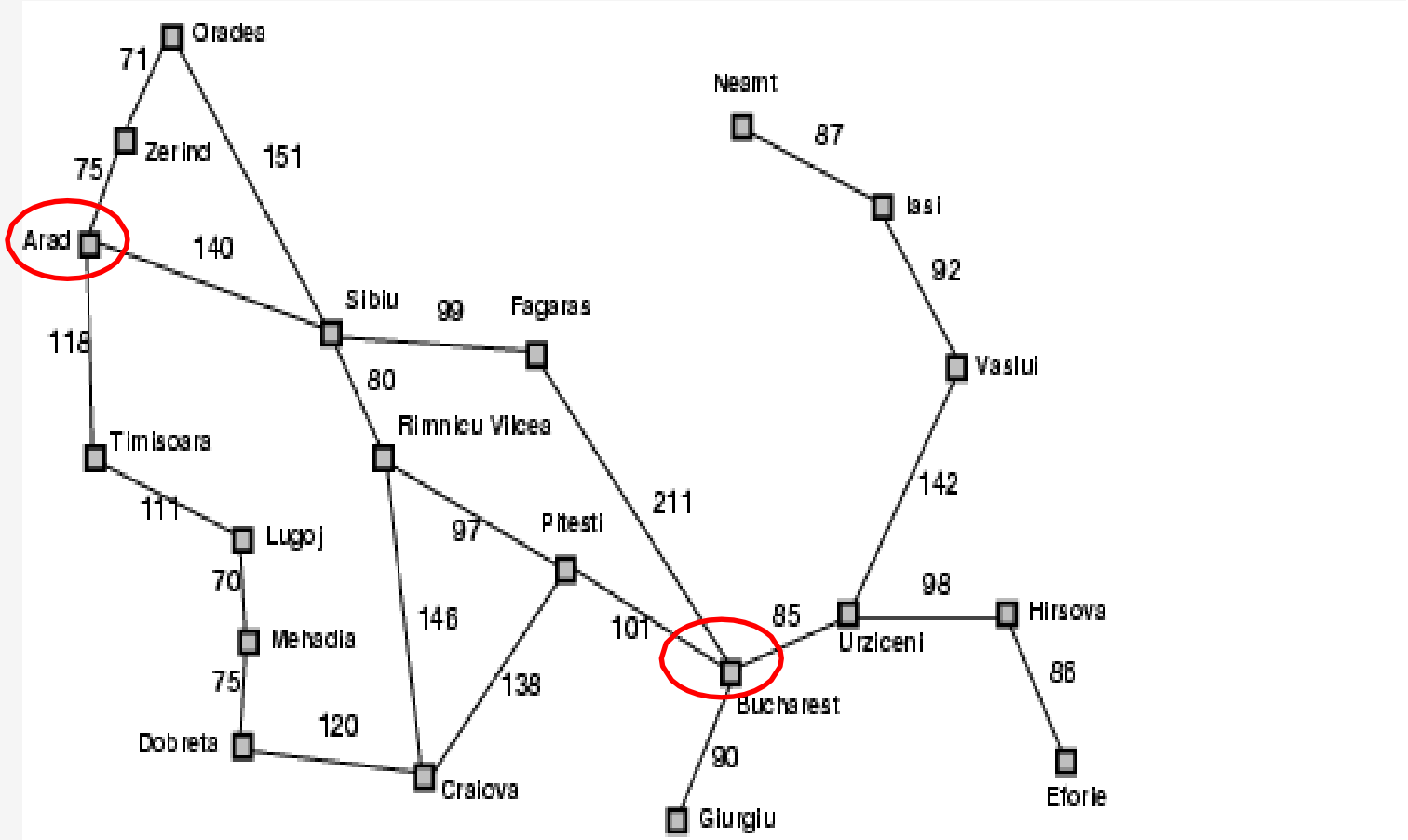
State space: cities

Successor function: Roads: go to adjacent city with cost = distance

Start state: Arad

Goal test: is state == Bucharest?

Solution?

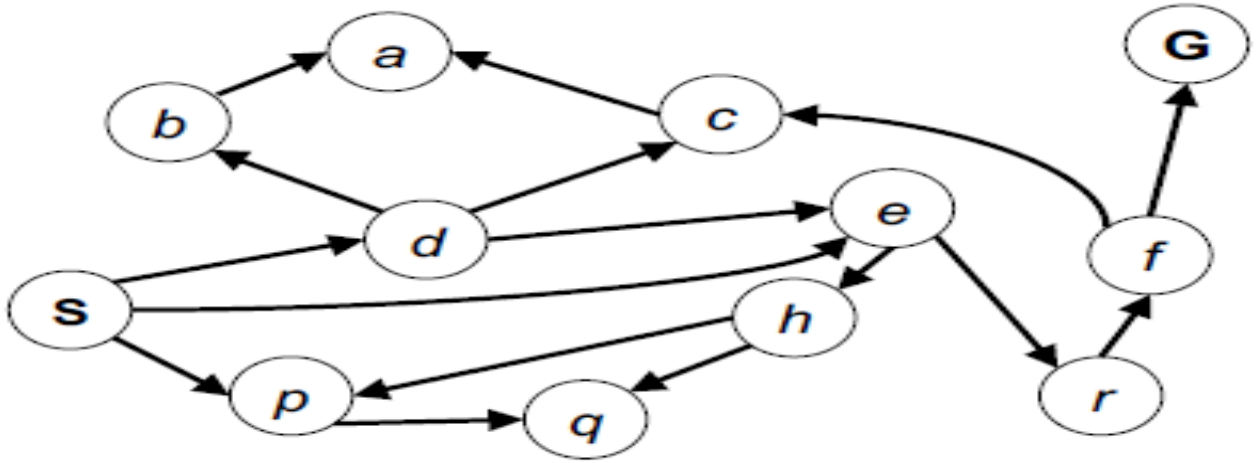


State Space

- A **model** is the formulation of the problem.
- State space graph: a mathematical representation of the search problem.

  Nodes are world configurations. **Arcs** represent successors (action results).

  The **goal test** is a set of goal nodes (maybe only ones).
- **Each state occurs only once.**



State Space

An AI problem can be represented as a *state space graph.*

A **graph** is a set of *nodes* and *links* that connect them.

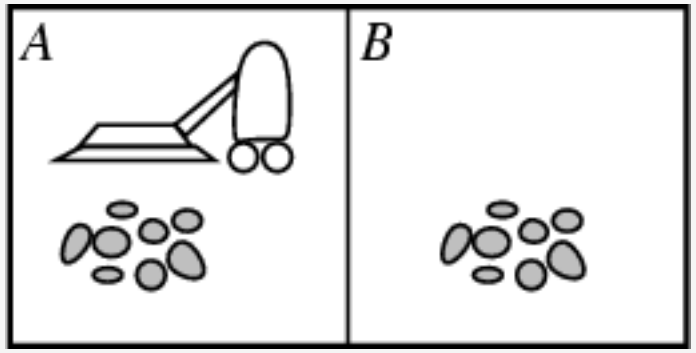
Graph theory:

- Labeled graph.
- Directed graph.
- Path.
- Rooted graph.
- Tree.
- Parent.
- Child.
- Sibling.
- Ancestor.
- Descendant.

Data structure for Search tree (Node)

we will assume a **node is a data structure** with five components:
• the state in the state space to which the node corresponds;
• the node in the search tree that generated this node (this is called the **parent node);**
• the operator that was applied to generate the node;
• the number of nodes on the path from the root to this node (the **depth** of the node);
• the path cost of the path from the initial state to the node.
The node data type is thus:
datatype node
components: STATE, PARENT-NODE, OPERATOR, DEPTH, PATH-COST

Example: Vacuum World

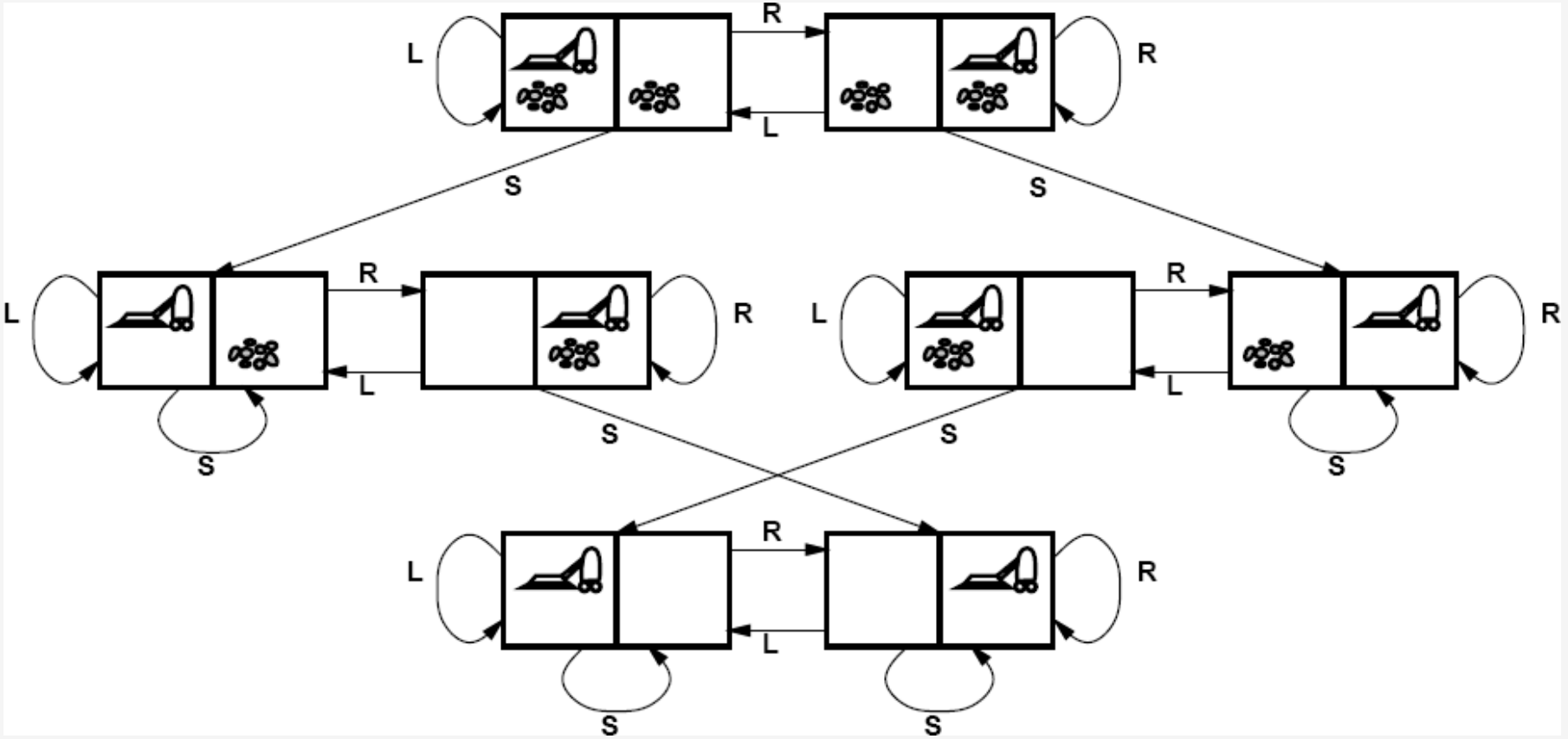


- **States:**
  - Agent location and dirt location
  - How many possible states?
  - What if there are *n* possible locations?
    - The size of the state space grows exponentially with the "size" of the world!
- **Actions:**
  - Left, right, suck.
- **Transition Model .. ?**

Example: Vacuum World

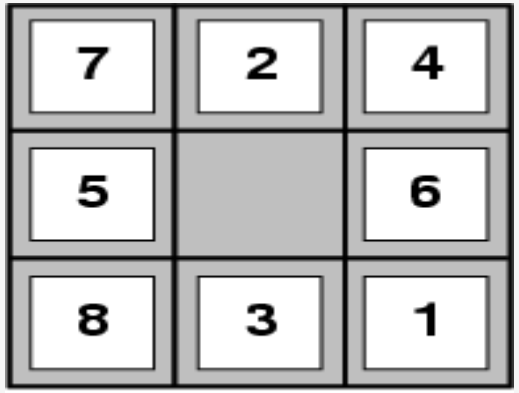
State Space Graph

- **Transition Model:**

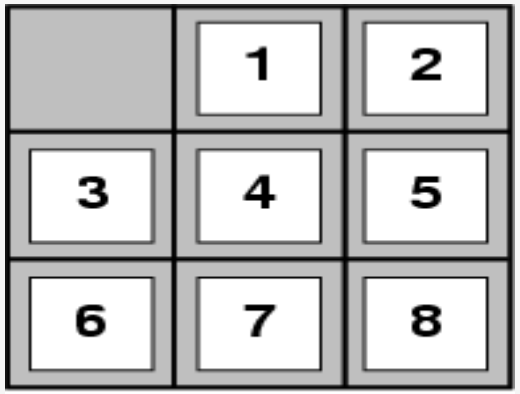


Example: the 8-Puzzle

- **States**
  - Locations of tiles
    - 8-puzzle: 181,440 states (9!/2)
    - 15-puzzle: ~10 trillion states
    - 24-puzzle: ~10^{25} states
- **Actions**
  - Move blank left, right, up, down
- **Path Cost**
  - 1 per move

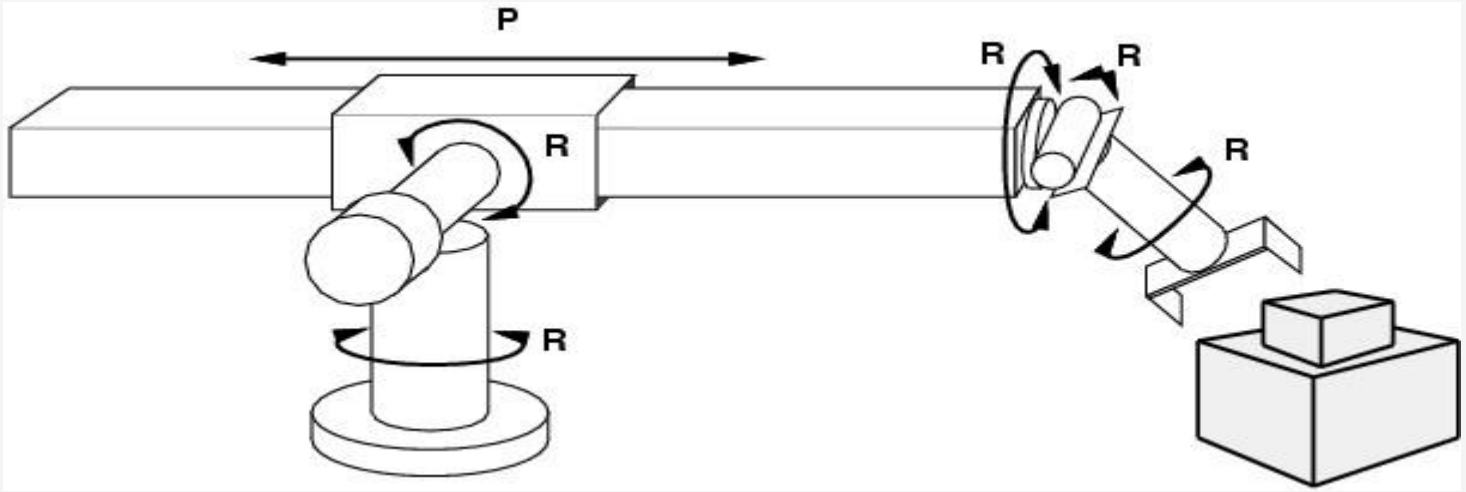


Start State



Goal State

Example: Robot Motion Planning



- **States**
  - Real-valued joint parameters *(angles, displacements)*.
- **Actions**
  - Continuous motions of robot joints.
- **Goal State**
  - Configuration in which object is grasped.
- **Path Cost**
  - Time to execute, etc.

Example: Rubik's Cube

States: list of colors for each cell on each face

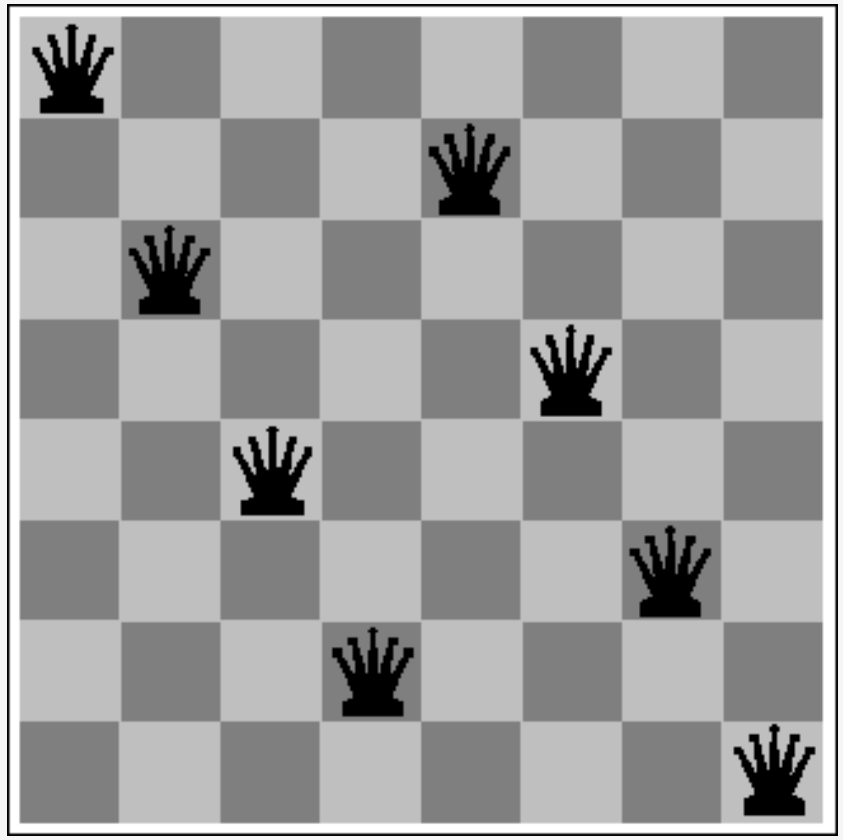
Initial state: one specific cube configuration

Action: rotate row x or column y on face z direction a

Goal: configuration has only one color on each face

Path cost: 1 per move

Example: Eight Queens



States: locations of 8 queens on chess board

Initial state: one specific queen's configuration

Actions: move queen x to row y and column z

Goal: no queen can attack another (cannot be in same row, column, or diagonal)

Example: Tic-Tac-Toe

- **Nodes(N)**: all the different configuration of Xs and Os that the game can have.
- **Arcs (A)**: generated by legal moves by placing X or O in unused location.
- **Start state (S)**: an empty board.
- **Goal states (GD)**: a board with three Xs in a row, column, or diagonal.
- The arcs are directed, then no cycles in the state space, *directed acyclic graph* (DAG).
- **Complexity**: 9! Different paths can be generated.

Example: Traveling Salesperson

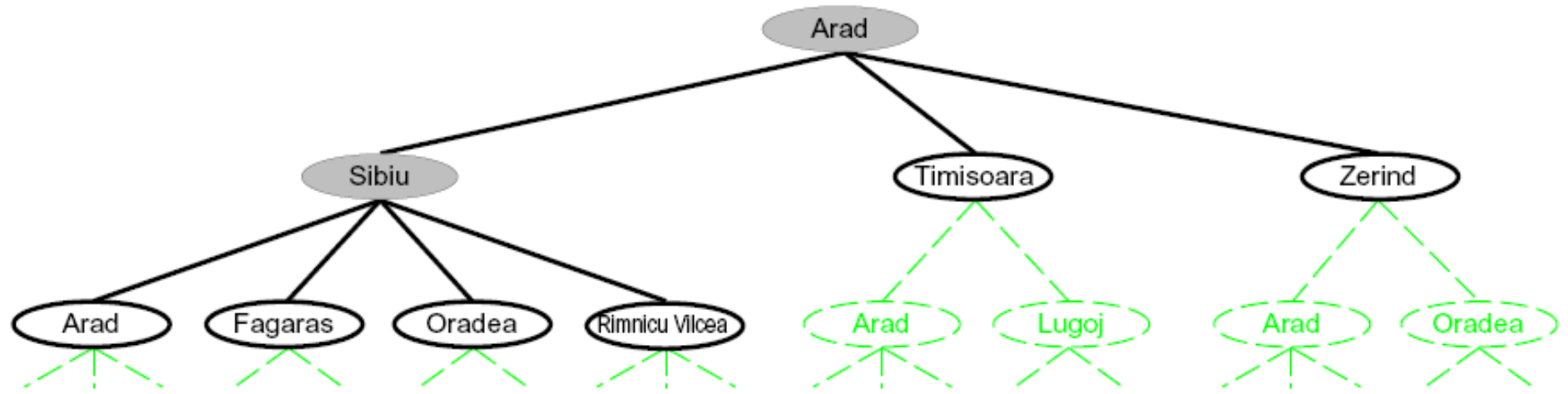
A salesperson has five cites to visit and ten must return home.

- **Nodes(N)**: represent 5 cites.
- **Arcs(A)**: labeled with weight indicating the cost of traveling between connected cites.
- **Start state(S)**: a home city.
- **Goal states(GD)**: an entire path contains a complete circuit with minimum cost.
- **Complexity**: (n-1)! Different cost-weighted paths can be generated.

Performance Evaluation of Search Strategies

- It is criteria used to choose search algorithms
- Evaluating the algorithms performance considering four factors:
  - **COMPLETENESS**
    - Is the algorithm guaranteed to find a solution when there is one, and to correctly report failure when there is not?
  - **COST OPTIMALITY**
    - Does the algorithm find a solution to the problem with the lowest path cost of all solutions?
  - **TIME COMPLEXITY**
    - How long does it take to find the solution? This can be measured in seconds or more abstractly by the number of states and actions considered?
  - **SPACE COMPLEXITY**
    - How much memory is needed to perform the search?

Searching with a general search tree



Search:

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

Important ideas: **fringe, expansion, exploration strategy**

Search..?

Given:

- ✓ **Initial state**
- ✓ **Actions**
- ✓ **Transition model**
- ✓ **Goal state**
- ✓ **Path cost**

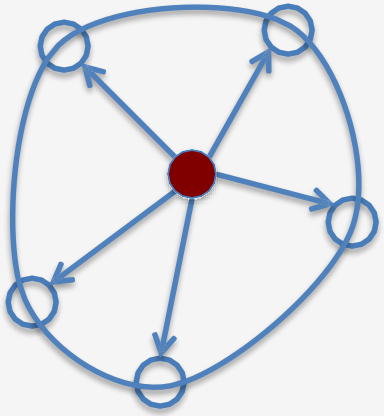
How do we find the optimal solution?

Search: Basic_idea

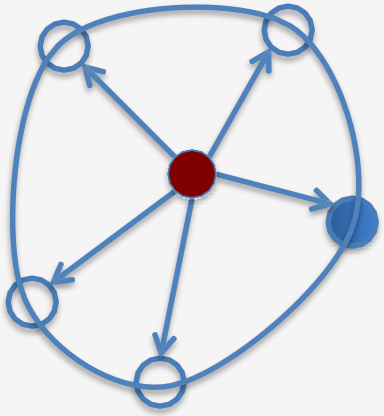
- Let’s begin at the start state and **expand** it by making a list of all possible successor states.
- Maintain a **frontier** or a list of unexpanded states.
- At each step, pick a state from the frontier to expand.
- Keep going until you reach a goal state.
- Try to expand as few states as possible.

Search: Basic_idea

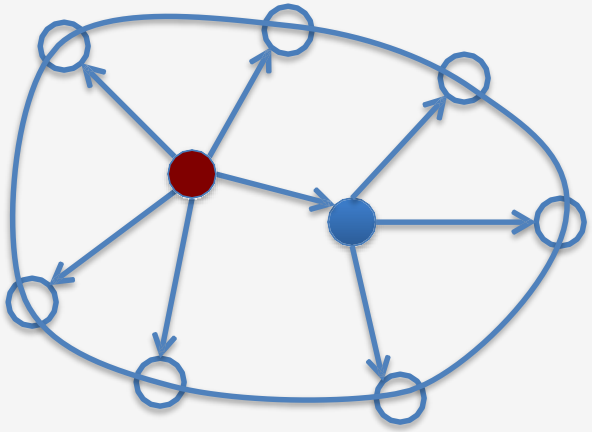
Search: Basic_idea



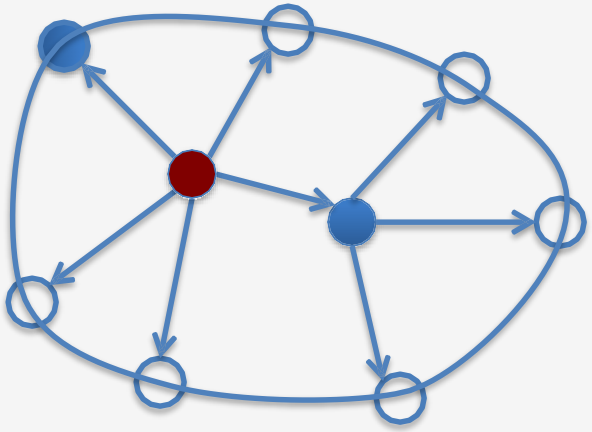
Search: Basic_idea



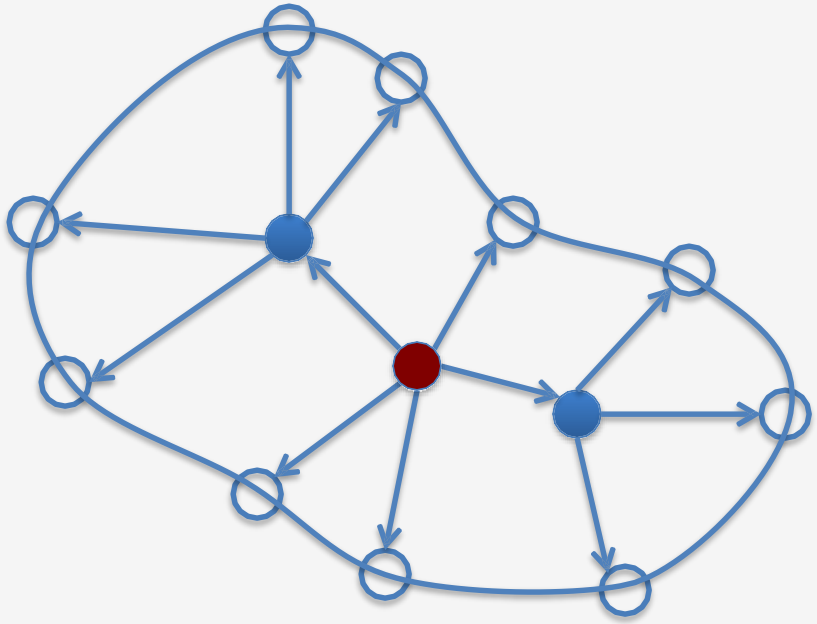
Search: Basic_idea



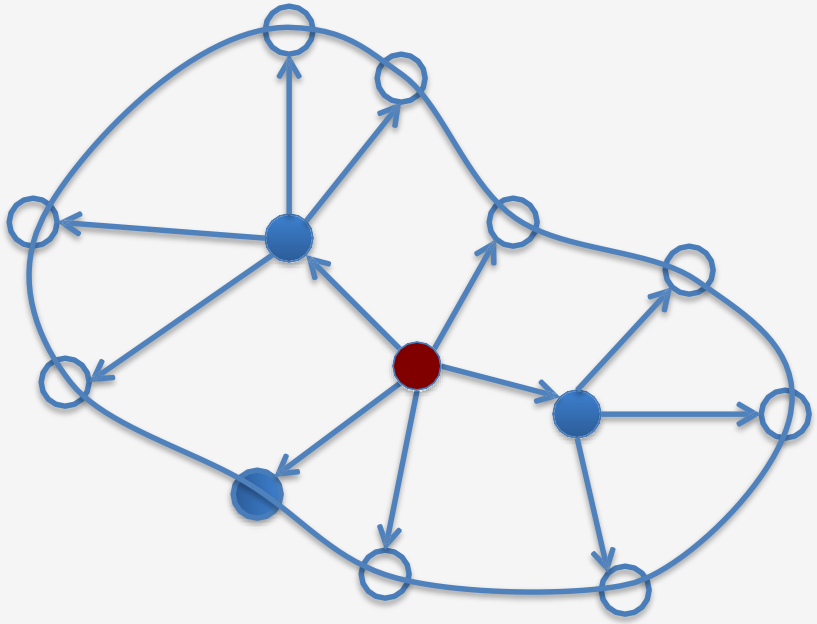
Search: Basic_idea



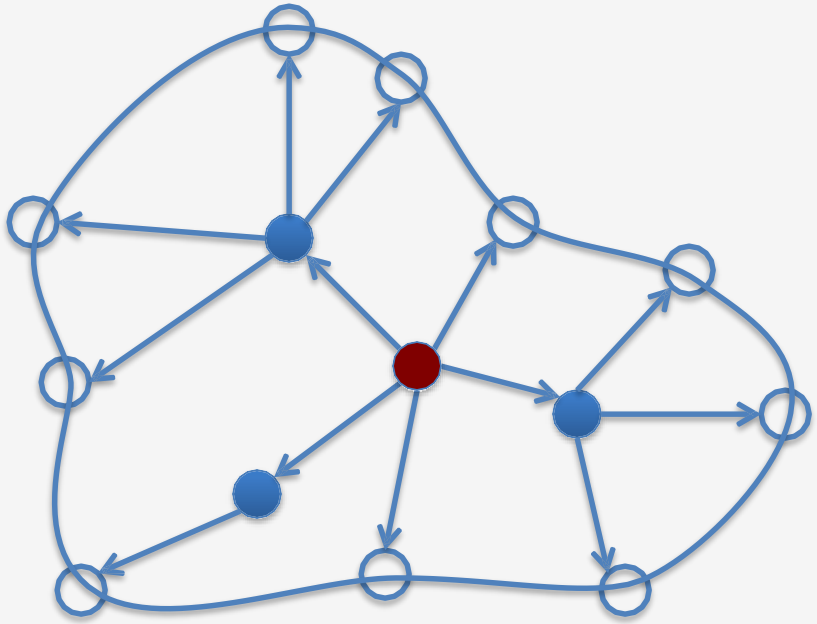
Search: Basic_idea



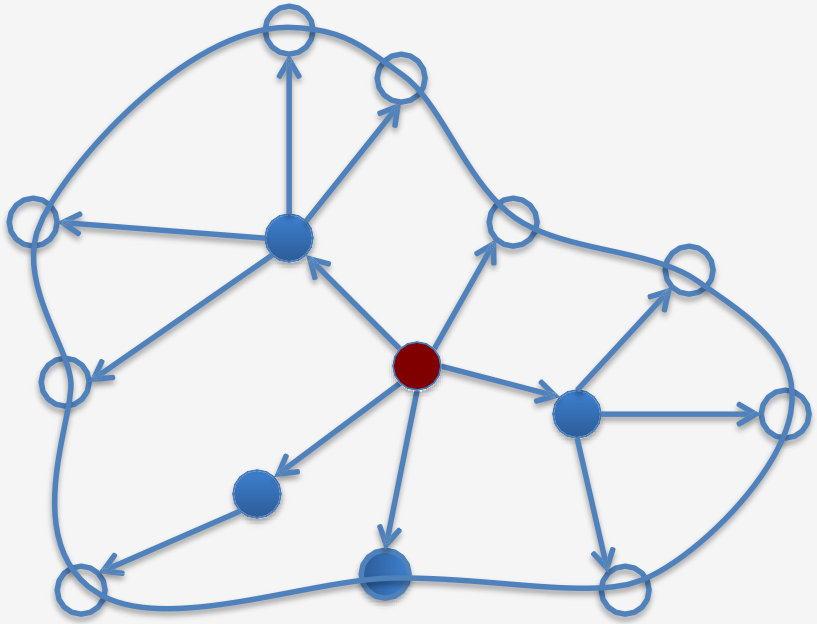
Search: Basic_idea



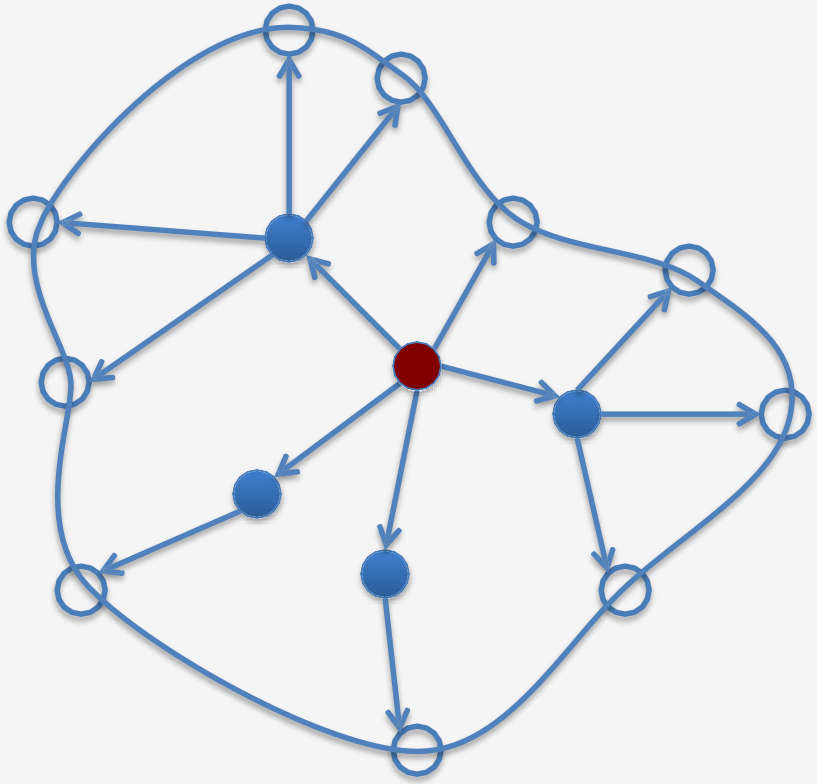
Search: Basic_idea



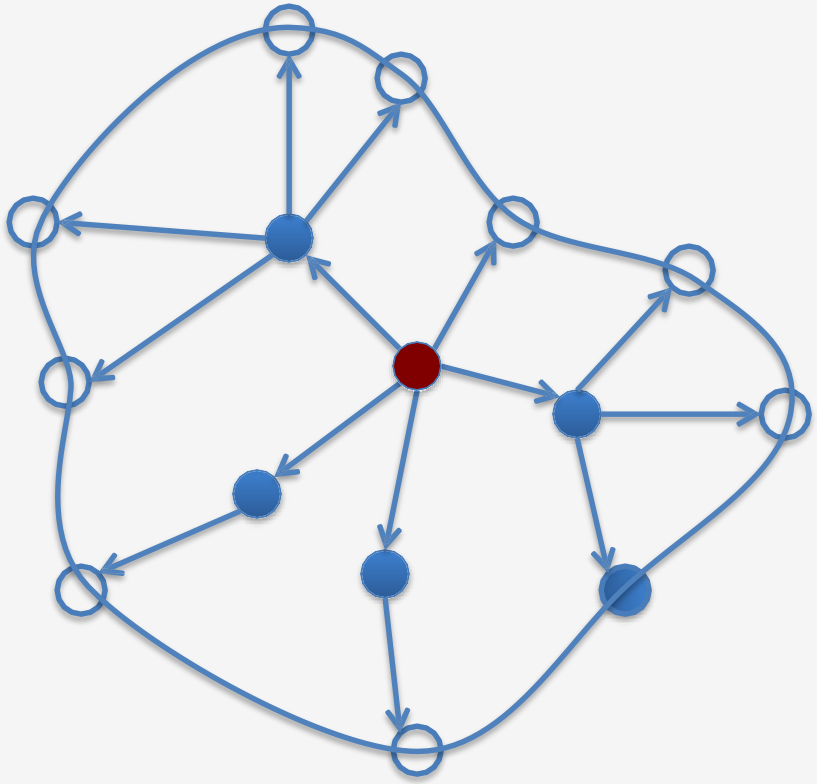
Search: Basic_idea



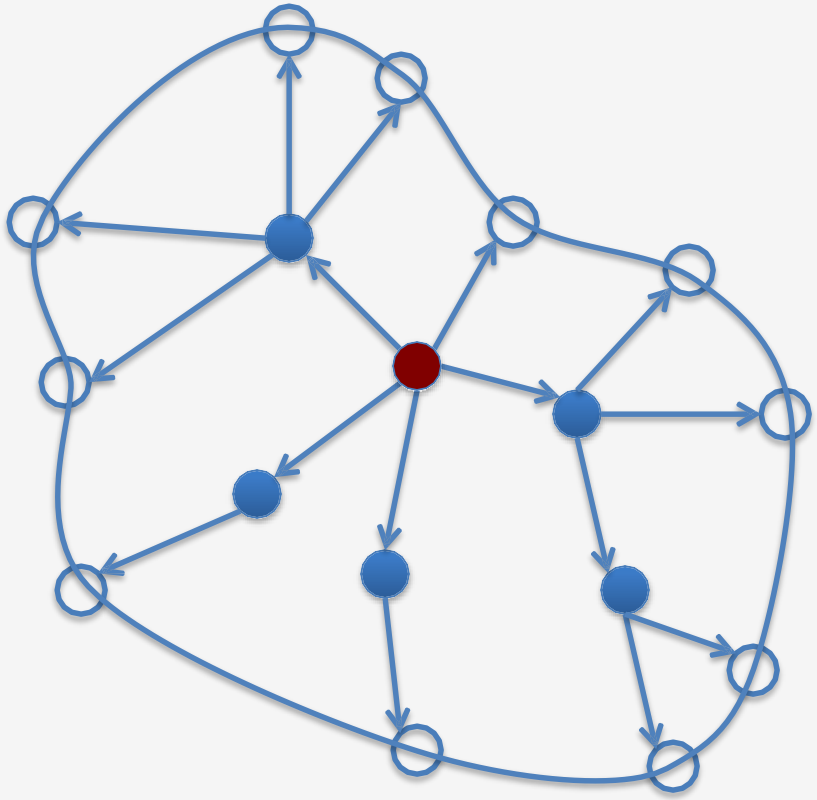
Search: Basic_idea



Search: Basic_idea



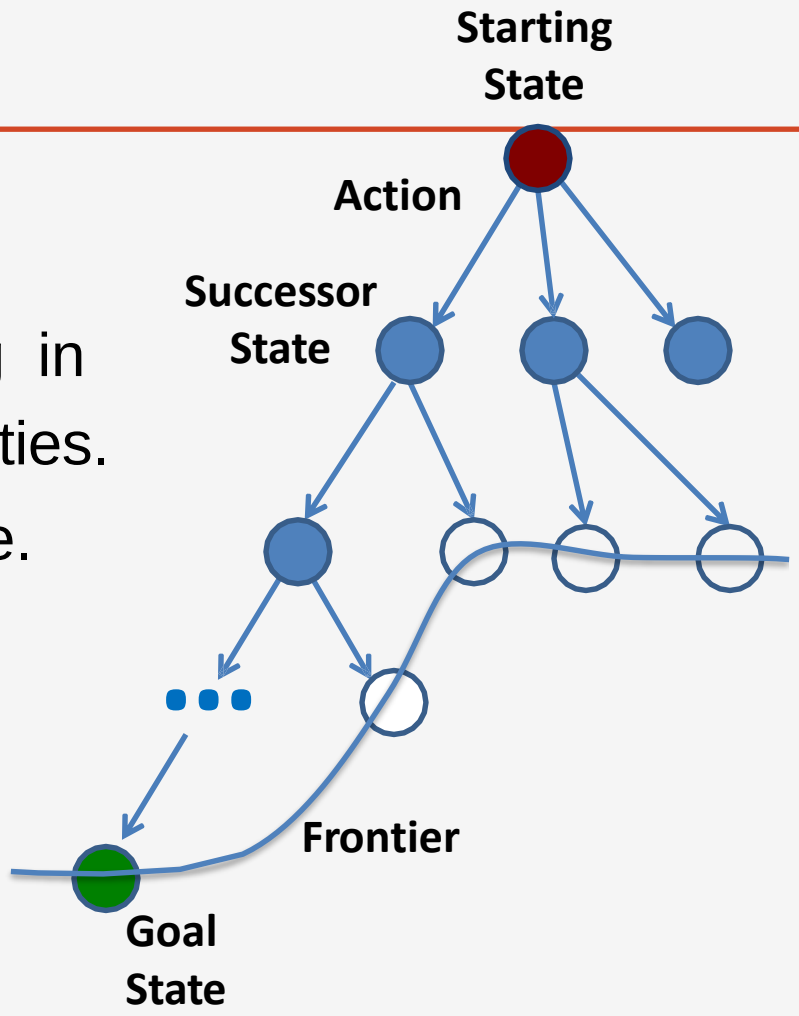
Search: Basic_idea



Search Tree (the What-if tree)

"What if" tree of sequences of actions and outcomes;

- I.e., When we are searching, we are not acting in the world, merely "thinking" about the possibilities.
- The root node corresponds to the starting state.
- The children of a node correspond to the successor states of that node's state.
- A path through the tree corresponds to a sequence of actions.
- A solution is a path ending in the goal state



Nodes vs. **States?** **A state represents the world, while a node is a data structure that is part of the search tree. A node must keep a pointer to its parent, path cost, and possibly other information**.

Tree Search Algorithm Outline

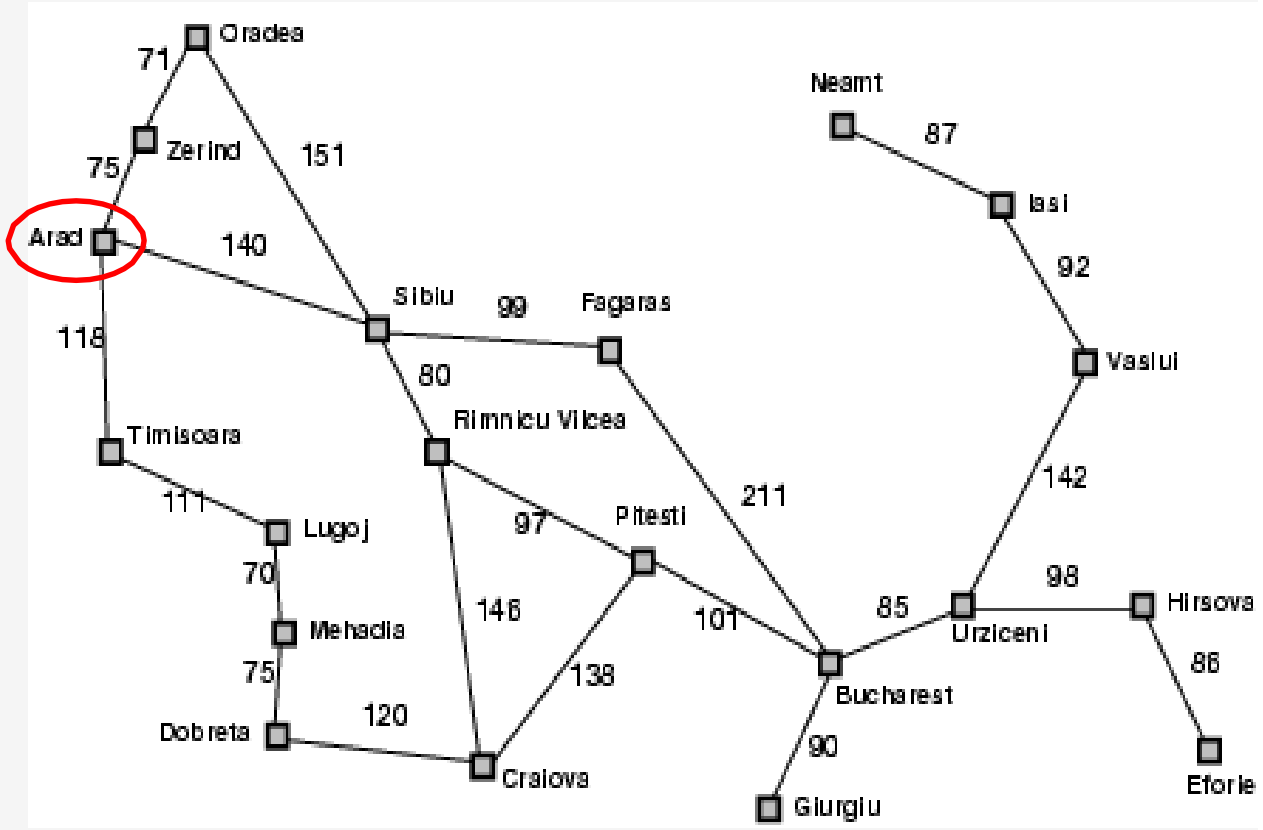
Initialize the **frontier** using the **starting state.**

While the frontier is not empty:

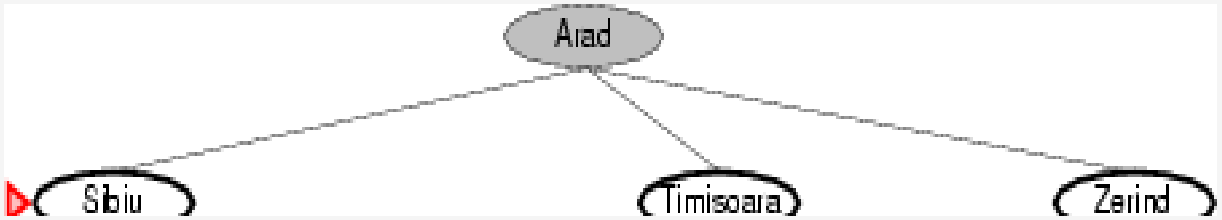
- Choose a frontier node according to **search strategy** and take it off the frontier.
- If the node contains the **goal state**, return solution.
- Else **expand** the node and add its children to the frontier.

Tree Search Example

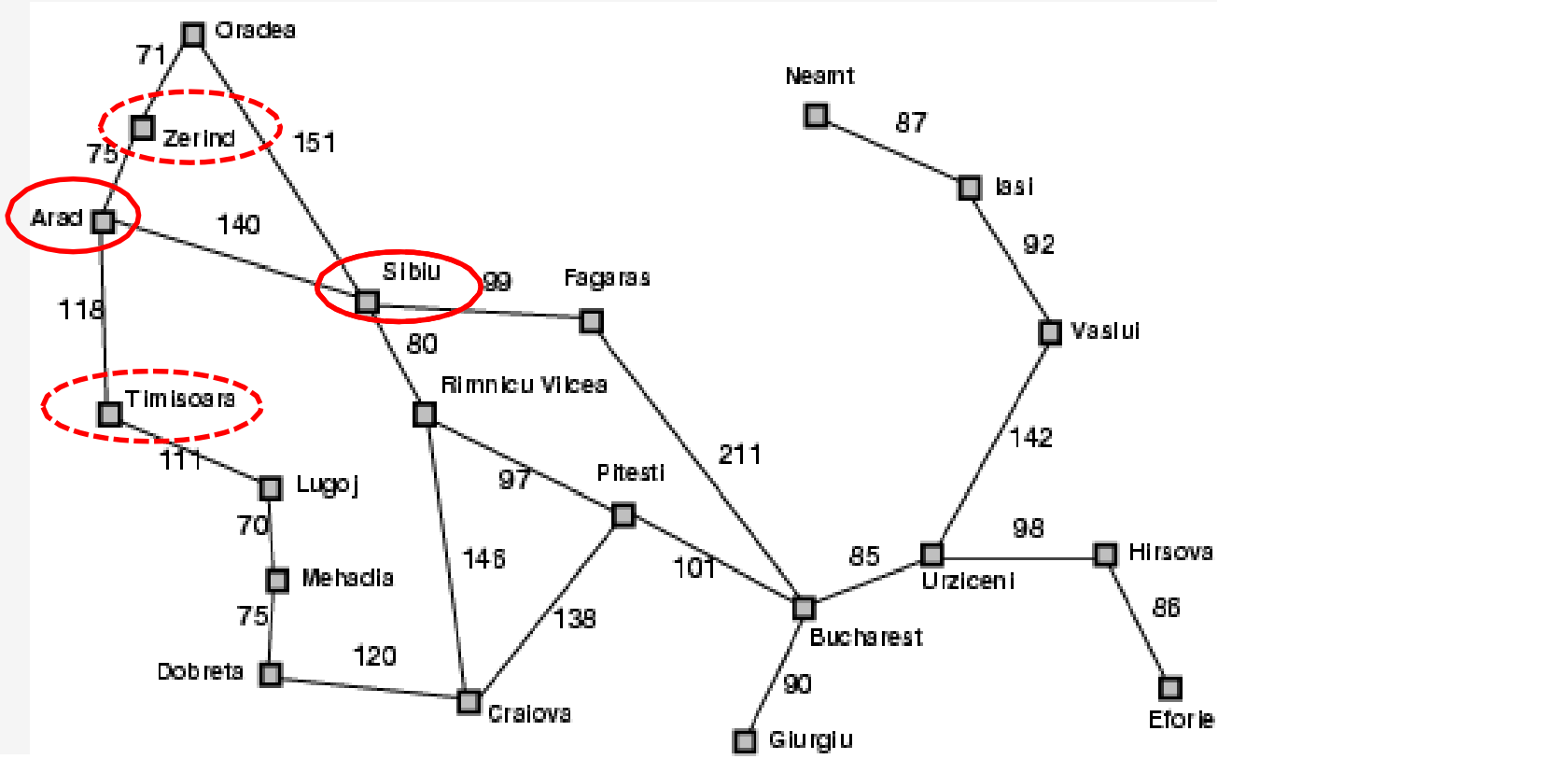
Start: **Arad**
Goal: **Bucharest**



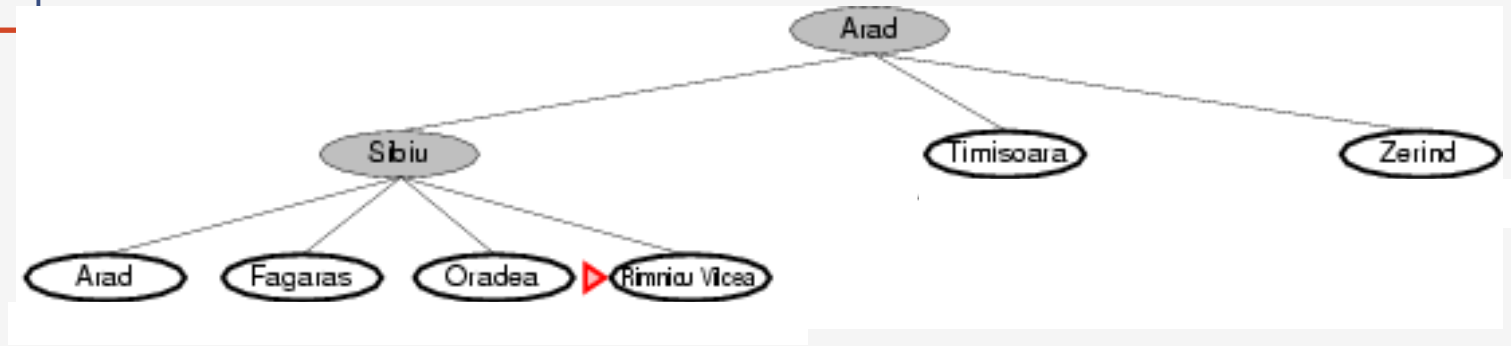
Tree Search Example

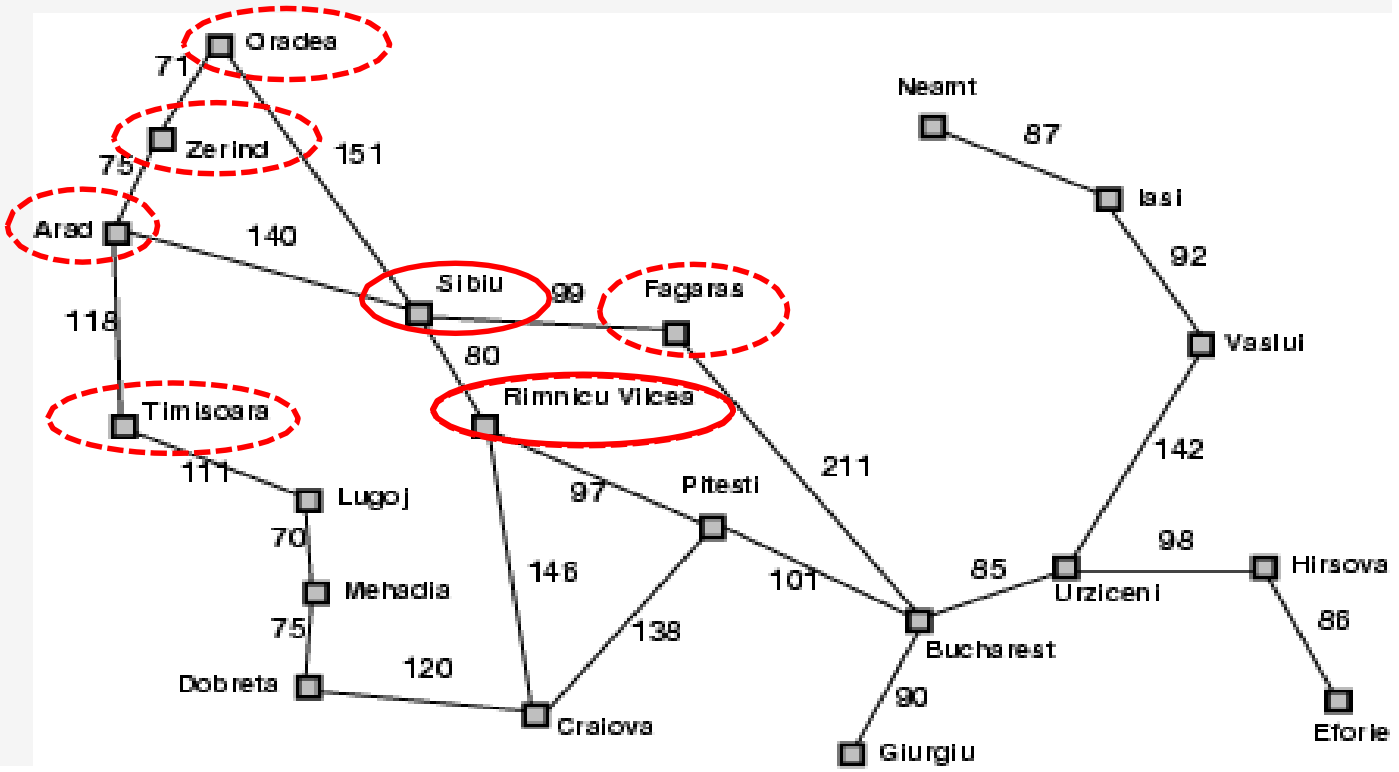


Start: **Arad**
Goal: **Bucharest**



Tree Search Example





Start: **Arad**
Goal: **Bucharest**

Handling Repeated States

To handle repeated states:

- Every time you expand a node, add that state to the **explored set**; do not put explored states on the frontier again.
- Every time you add a node to the frontier, check whether it already exists with a higher path cost, and if yes, replace that node with the new one.

General search algorithm

```
Open = initial state                    // open list is all generated states
                                        // that have not been "expanded"
While open not empty                    // one iteration of search algorithm
   state = First(open)                  // current state is first state in open
   Pop(open)                            // remove new current state from open
   if Goal(state)                       // test current state for goal condition
      return "succeed"                  // search is complete
                                        // else expand the current state by
                                        // generating children and
                                        // reorder open list per search strategy
   else open = QueueFunction(open, Expand(state))
Return "fail"
```

Note: Search strategies differ only in QueuingFunction

General search algorithm

The ***frontier*** is the set of nodes (and corresponding states) that have been reached but not yet expanded; the ***interior*** is the set of nodes (and corresponding states) that have been expanded.

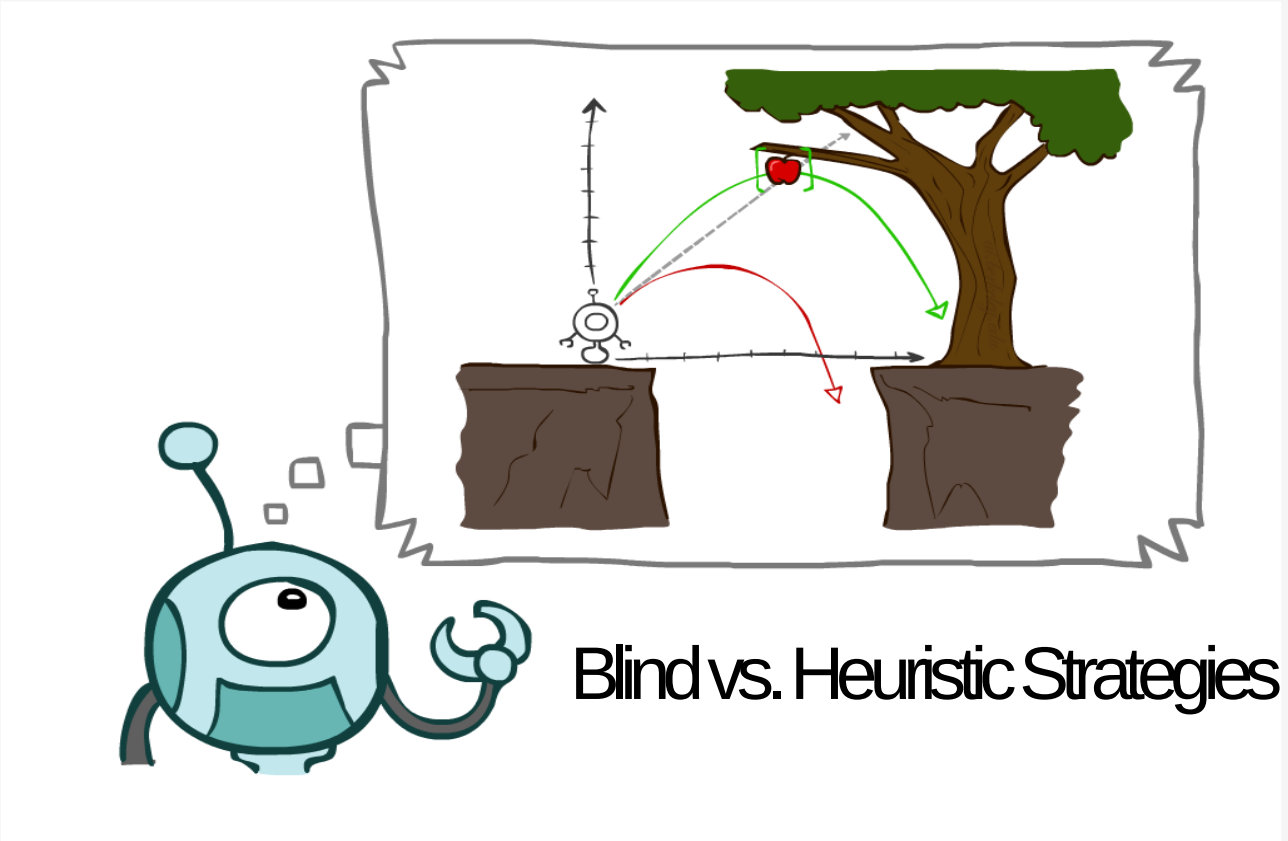
Note that the frontier separates two regions of the state-space graph: an interior region where every state has been expanded and an exterior region of states that still need to be reached.

To understand completeness, consider a search problem with a single goal. That goal could be anywhere in the state space; therefore, a complete algorithm must be capable of systematically exploring every state that is reachable from the initial state.

Search

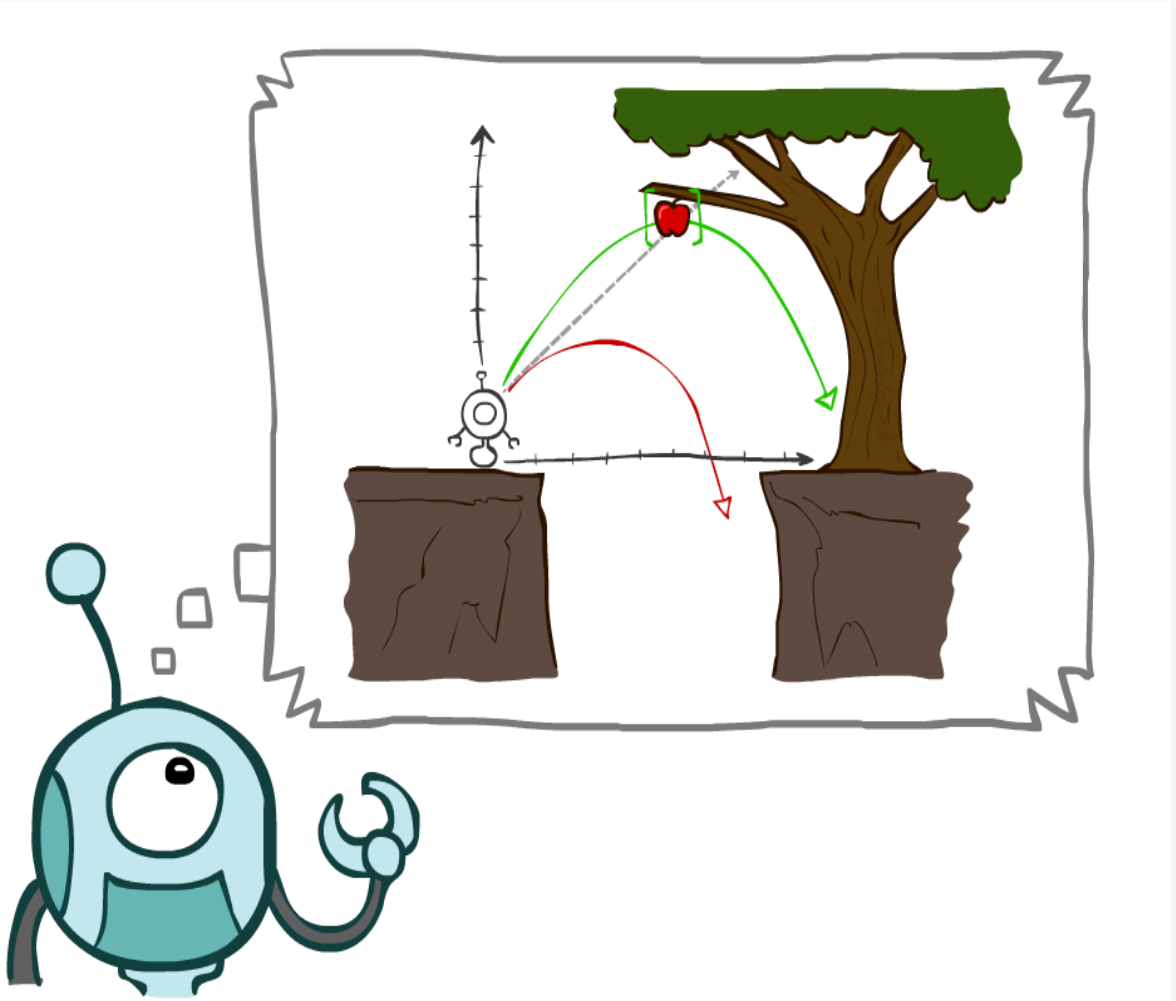
What is search problem?

- **An intelligent agent is trying to find a set or sequence of actions to achieve a goal**
- **This is a goal-based agent**



Today's class

- ***Search***
  - **Uninformed (blind) search method**
    - **Depth first search**
      - **Depth limited**
      - **Iterative deeping**
    - **Breadth first search**
      - **Bidirectional search**
    - **Uniform cost search**



Blind vs. Heuristic Strategies

- **Blind (or Un-Informed / Exhaustive / Brute-Force) strategies** do not exploit any of the information contained in a state.

- **Heuristic (or Informed) strategies** exploits such information to assess that one node is “more promising” than another.

General search algorithm

There are search strategies that come under the heading of **uninformed search.** The term means that they have no information about the number of steps or the path cost from the current state to the goal—all they can do is distinguish a goal state from a non goal state.

Uninformed search is also sometimes called **blind search.**

Strategies that use such considerations are called **informed search** strategies or **heuristic search** strategies.

An uninformed search is less effective than an informed search. An uninformed search is still important, however, because there are many problems for which there is no additional information to consider.

Depth-first search

Depth-first search

Depth-first search always expands one of the nodes at the deepest level of the tree. Only when the search hits a dead end (a non-goal node with no expansion) does the search go back and expand nodes at shallower levels.

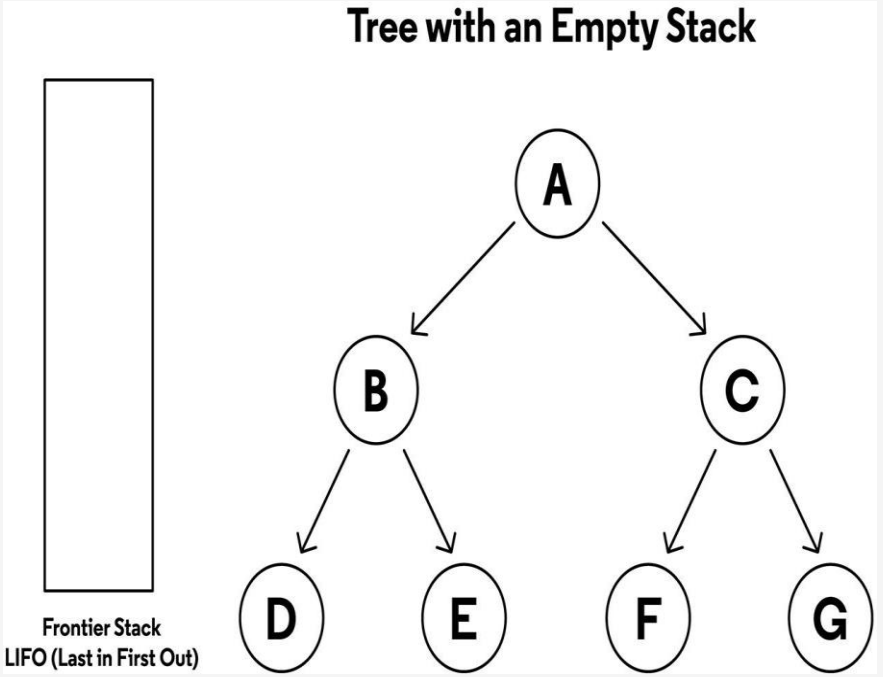
GENERAL-SEARCH can implement this strategy with a queuing function that always puts the newly generated states at the front of the queue. Because the expanded node was the deepest, its successors will be even deeper and are now the deepest.

Depth-first search has very modest memory requirements. As the figure shows, it needs to store only a single path from the root to a leaf node, along with the remaining unexpanded sibling nodes for each node on the path.

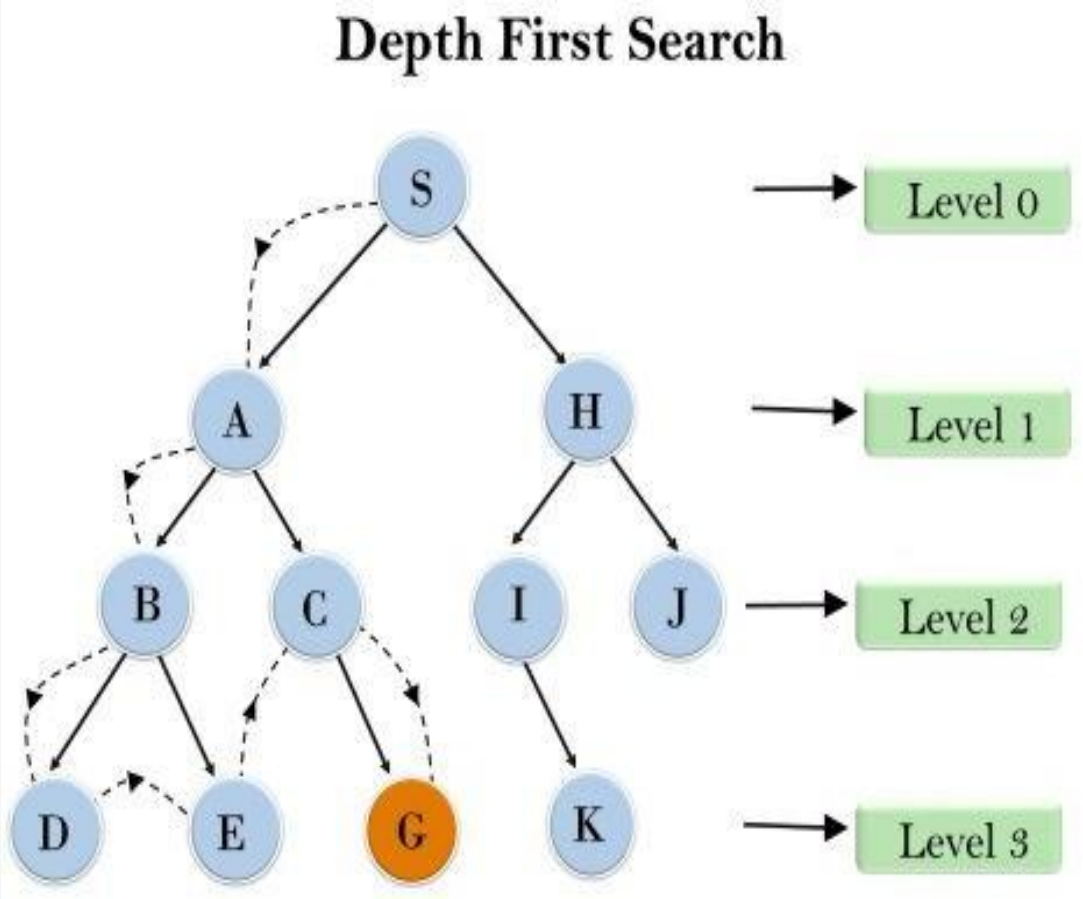
depth-first search has much smaller needs for memory.

Depth First Search

- DFS is recursive a recursive algo for tree traversing
- It starts from the root node and follows each path to its greatest depth node before moving to the next path.
- DFS uses stack for its implementation.



Diagram


Depth First Search
S
A
H
B
C
I
J
D
E
G
K
Level 0
Level 1
Level 2
Level 3

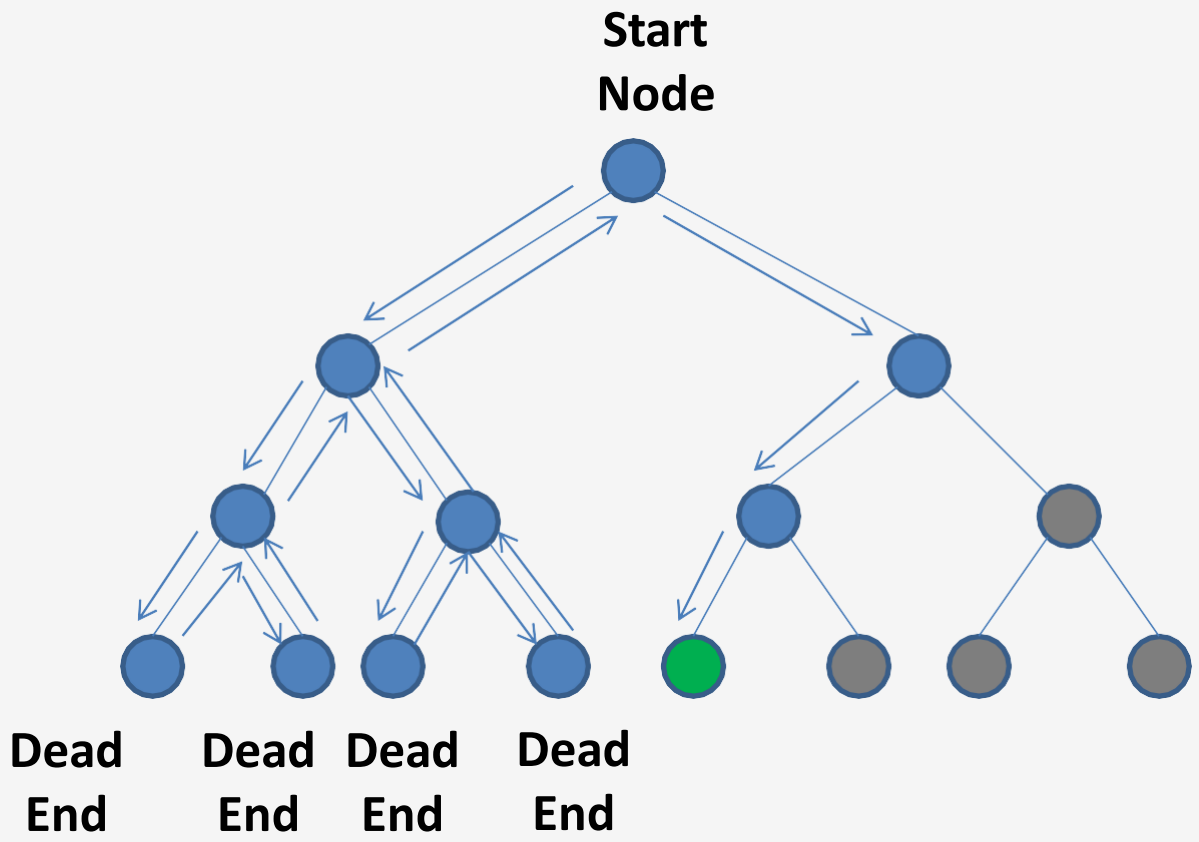
Depth-first search

The time complexity for a depth-first search for problems with many solutions may actually be faster than breadth-first because it has a good chance of finding a solution after exploring only a small portion of the whole space **(finite state)**.
The **drawback** of depth-first search is that it can get stuck if it goes down the wrong path. Many problems have very deep or even infinite search trees, so depth-first search will never be able to recover from an unlucky choice at one of the nodes near the top of the tree (incomplete).
Depth-first search is not cost-optimal; it returns the first solution it finds, even if it is not the cheapest. **For finite state spaces,** it is efficient and complete; **in infinite state spaces**, it is not systematic: it can get stuck going down an infinite path, even if there are no cycles. Thus, the depth-first search is **incomplete.**
In general depth-first search should be avoided for search trees with large or infinite maximum depths.

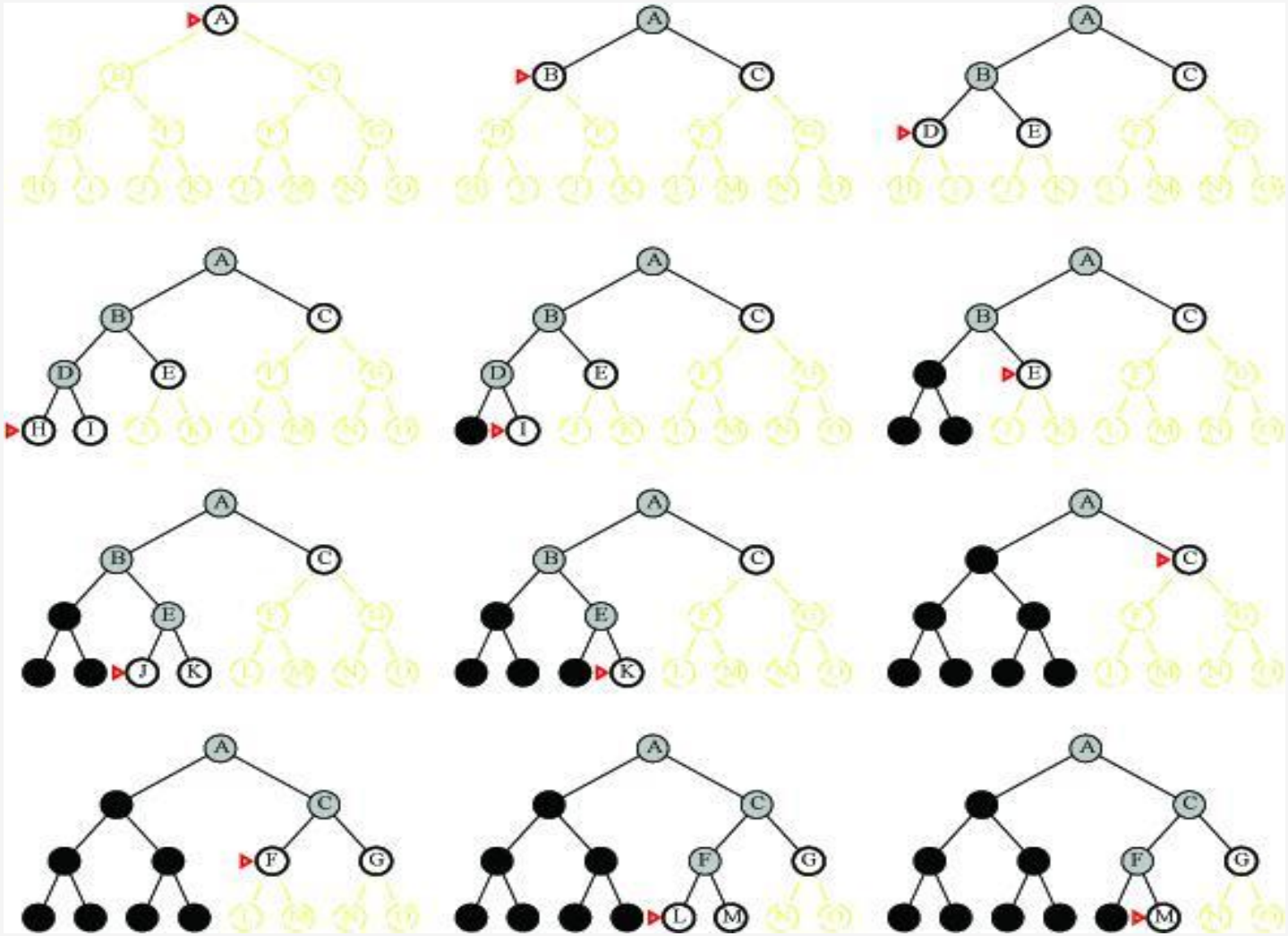
Depth-First Search

- QueueingFn adds the children to the front of the open list
- BFS emulates FIFO queue
- DFS emulates LIFO stack
- Net effect
    - Follow leftmost path to the bottom, then backtrack
    - Expand deepest node first

Backtracking Search

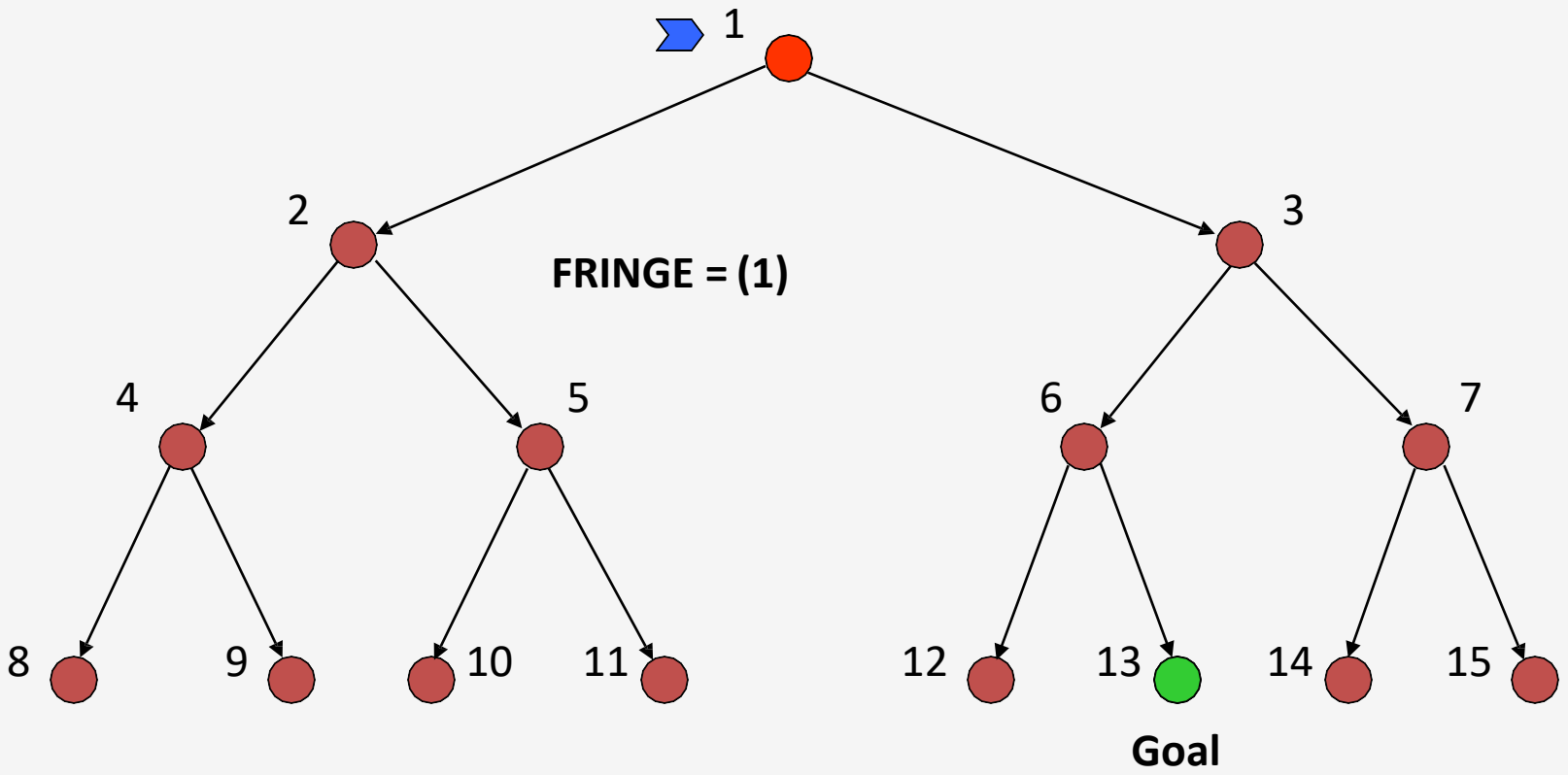


DFS Examples



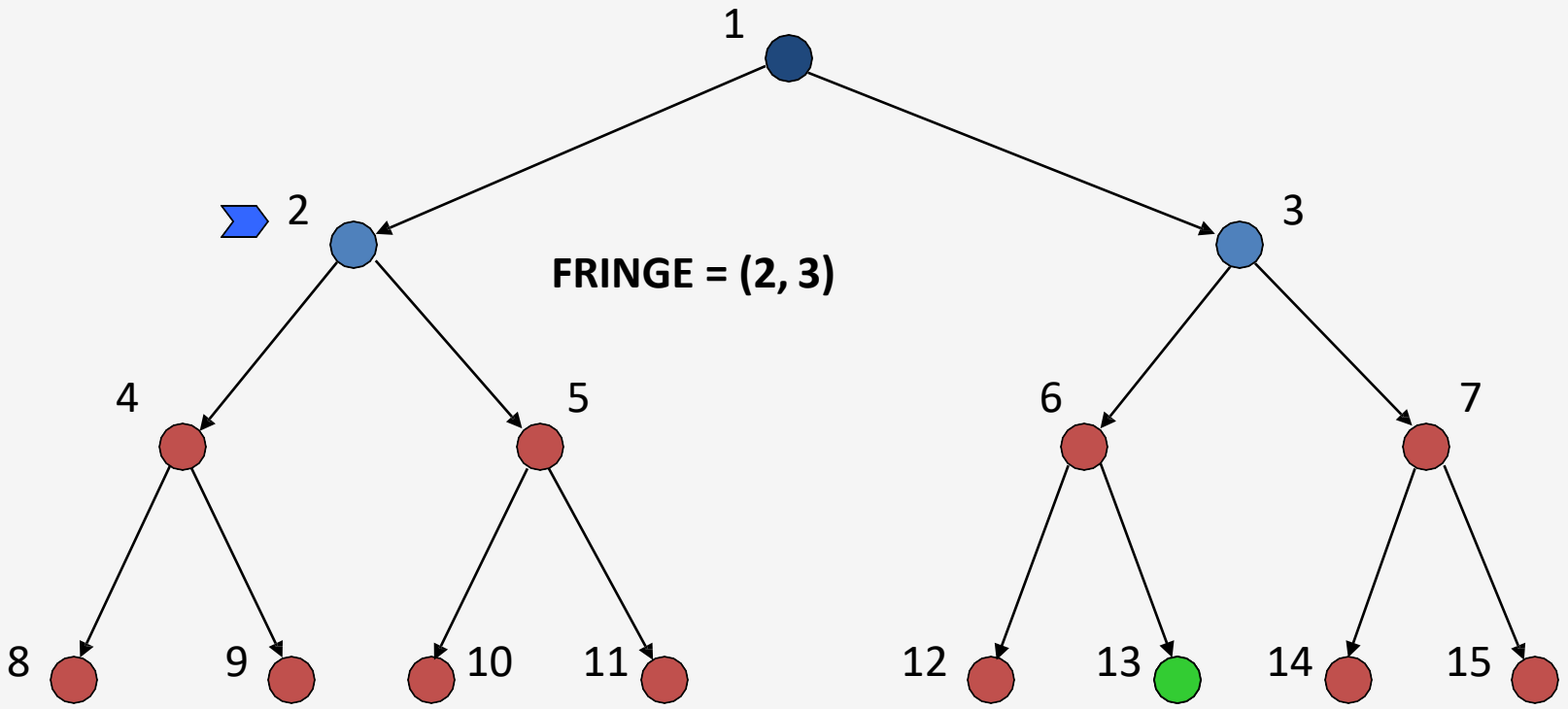
Depth-FirstStrategy

New nodes are inserted at the front of the FRINGE.



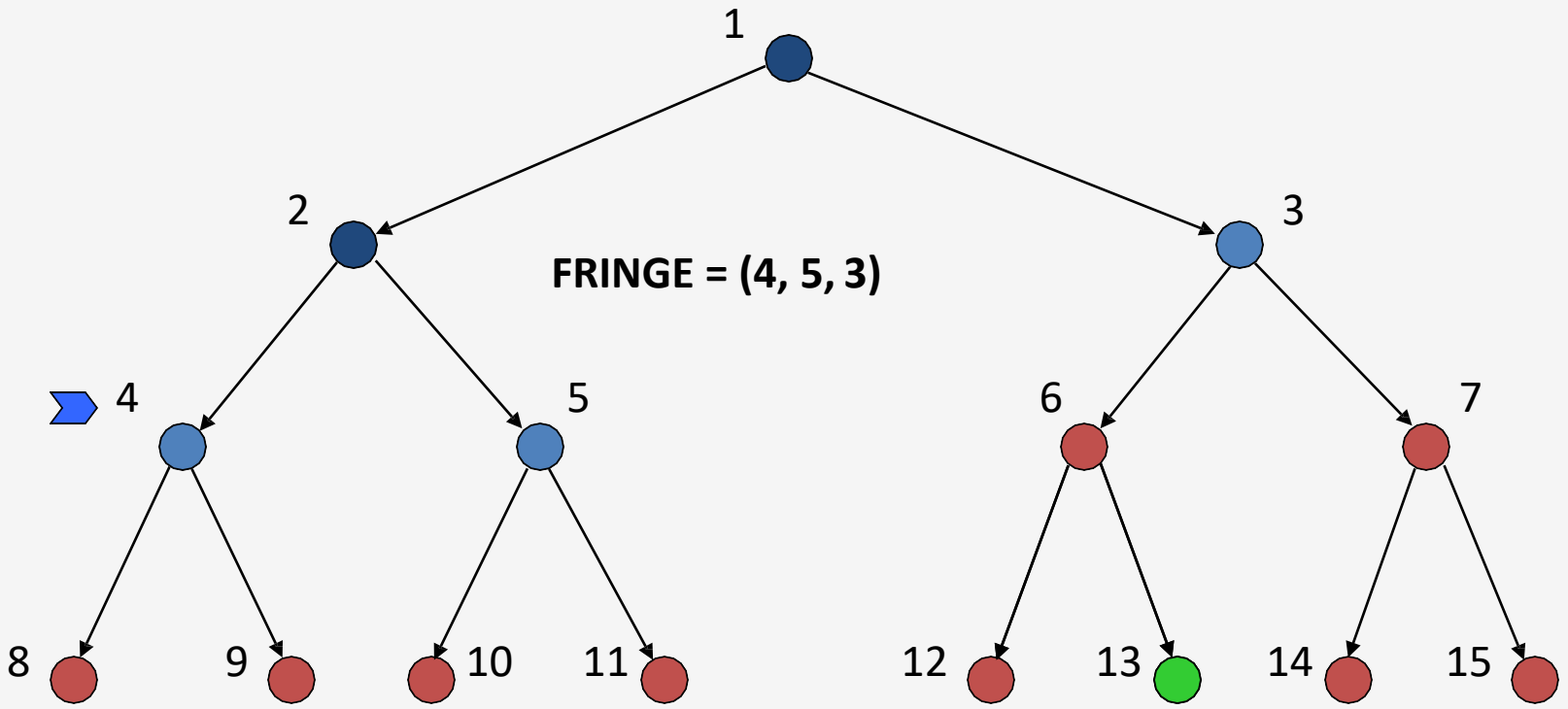
Depth-FirstStrategy

New nodes are inserted at the front of the FRINGE.



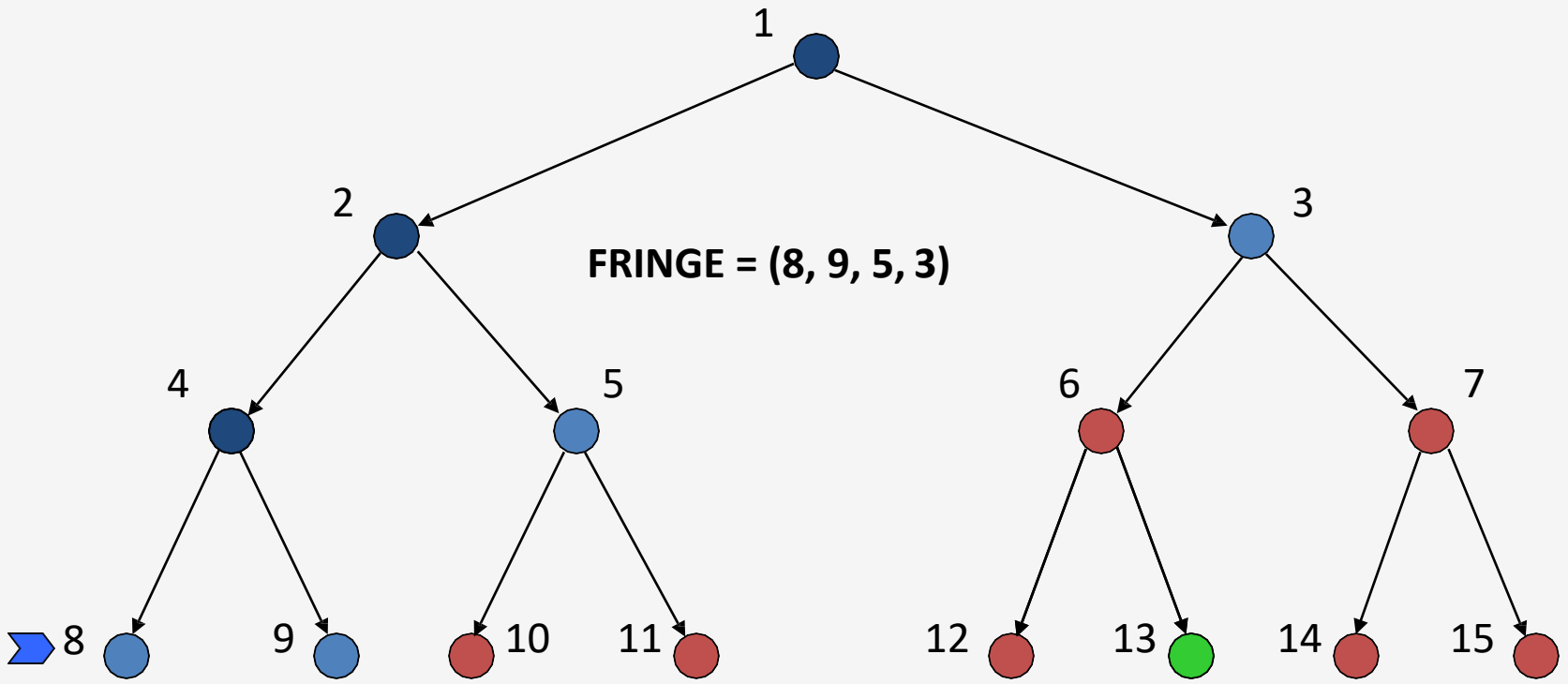
Depth-FirstStrategy

New nodes are inserted at the front of the FRINGE.



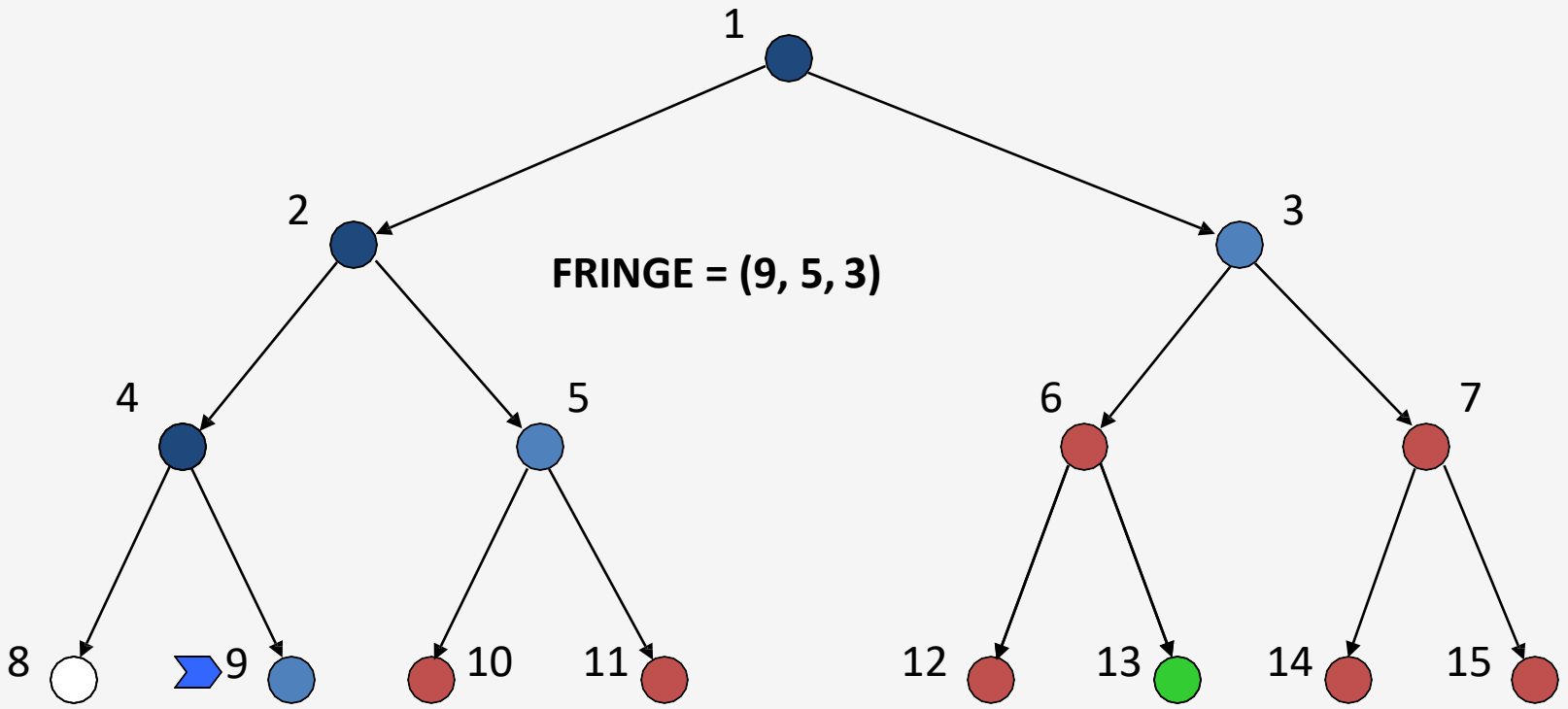
Depth-FirstStrategy

New nodes are inserted at the front of the FRINGE.



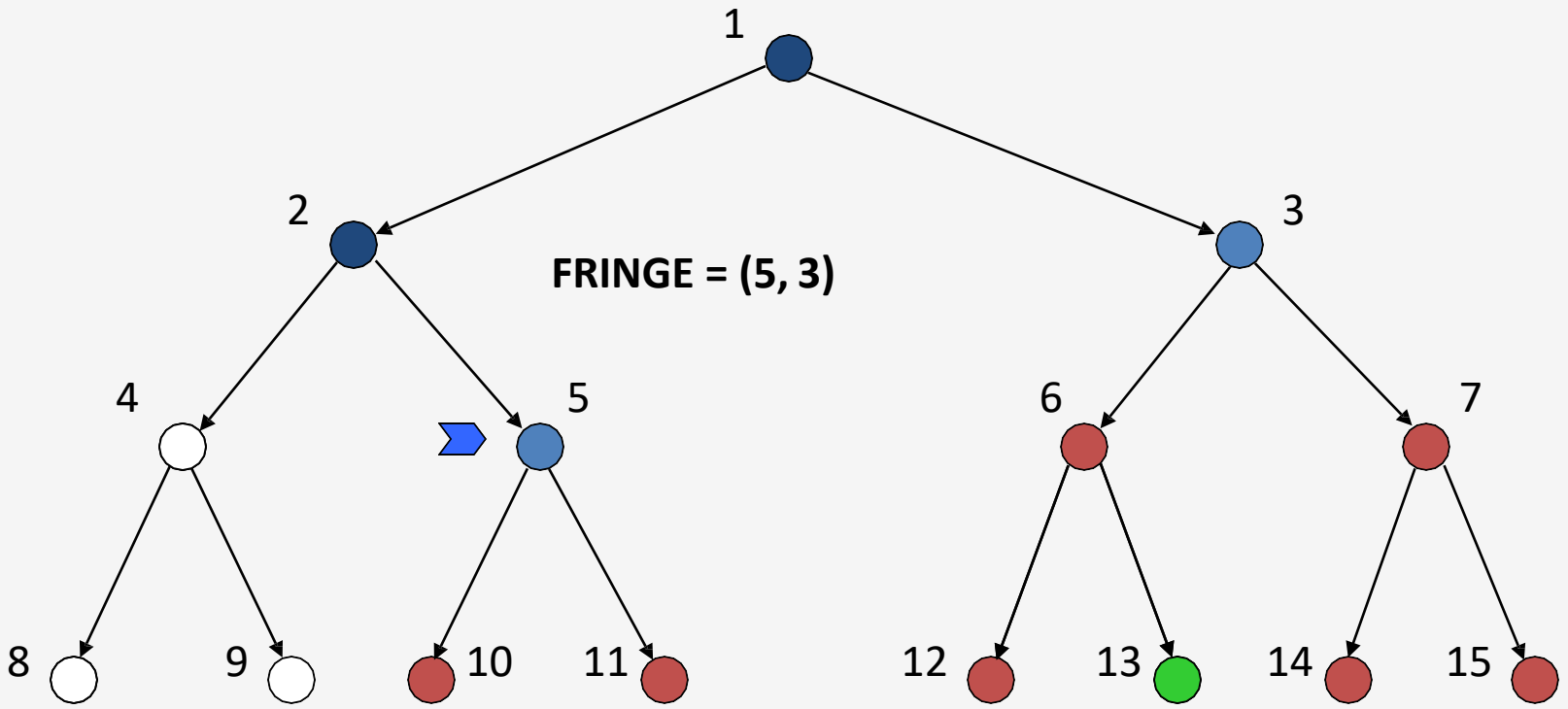
Depth-FirstStrategy

New nodes are inserted at the front of the FRINGE.



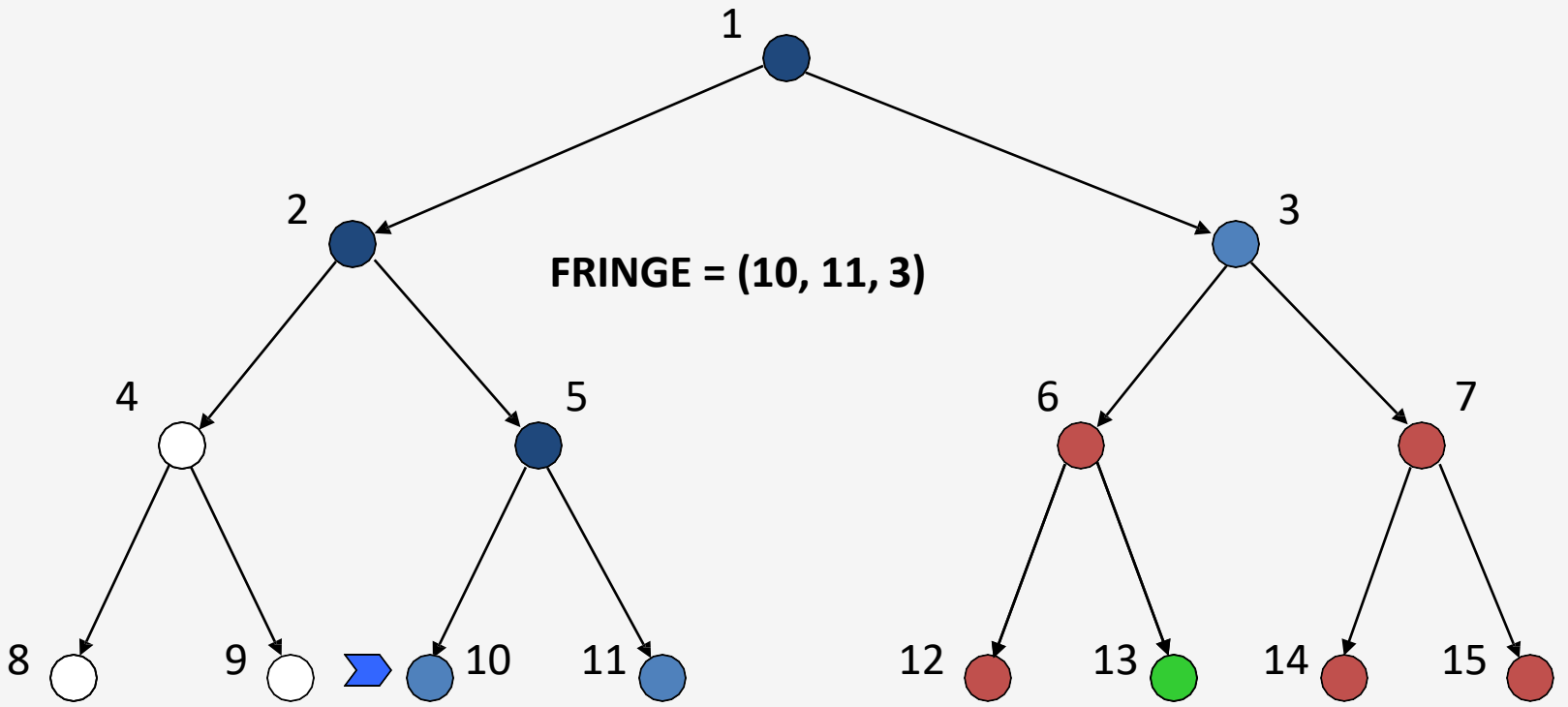
Depth-FirstStrategy

New nodes are inserted at the front of the FRINGE.



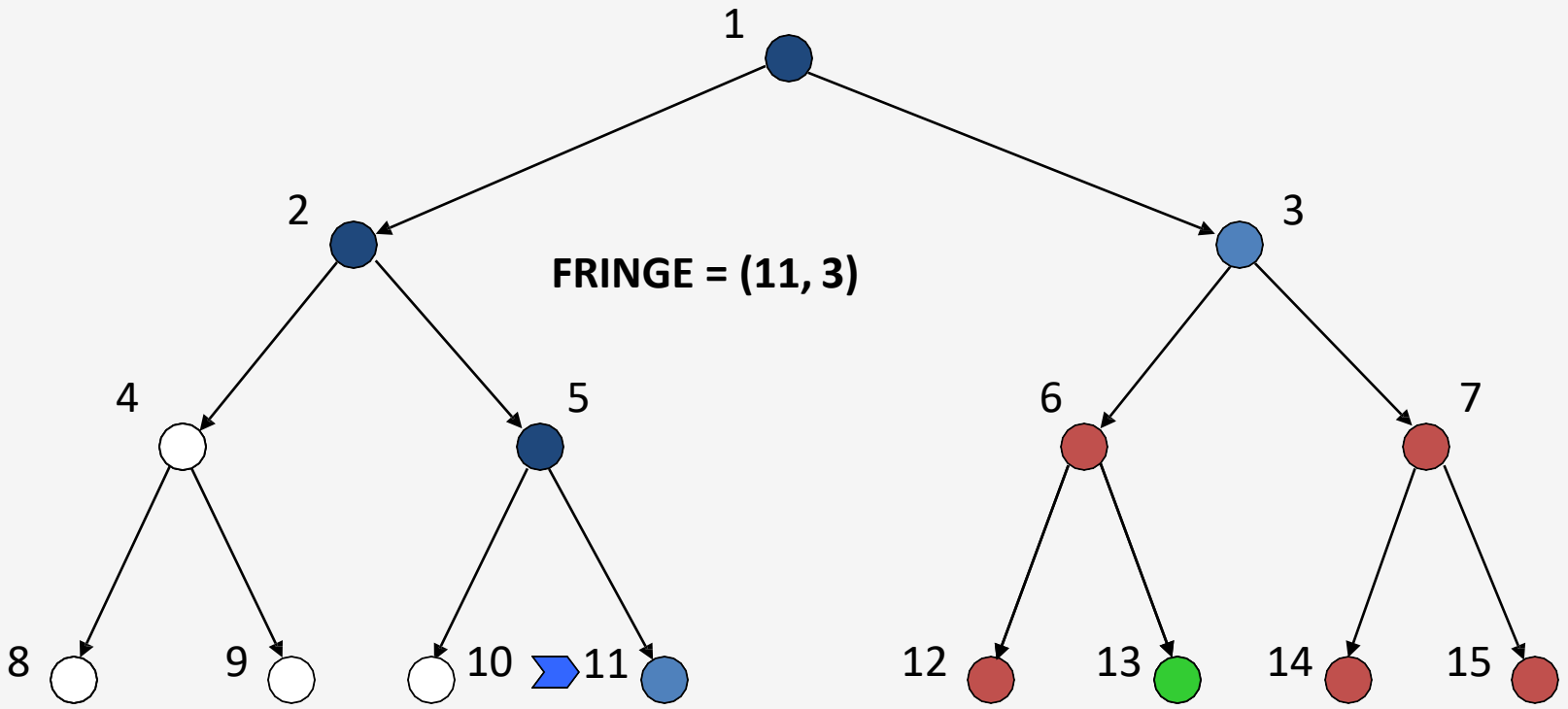
Depth-FirstStrategy

New nodes are inserted at the front of the FRINGE.



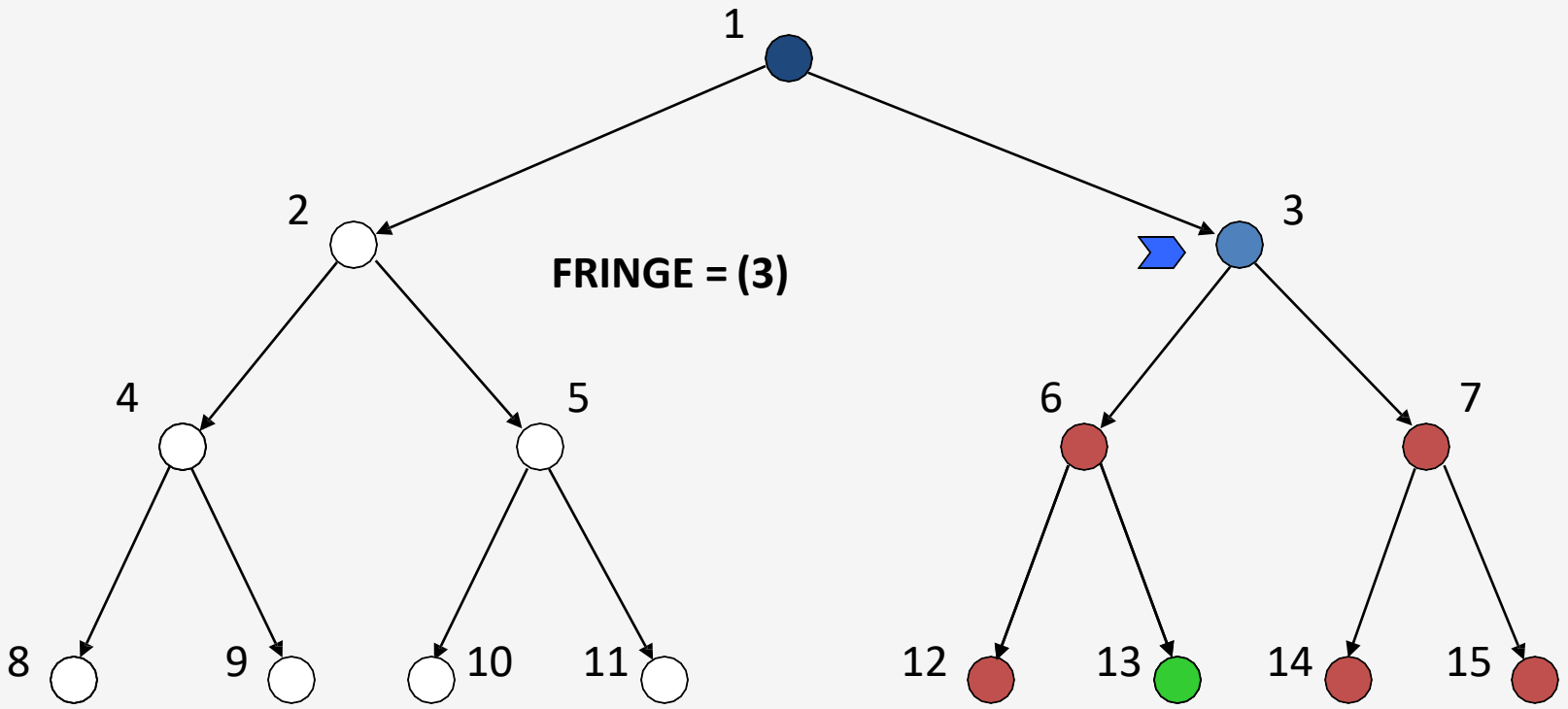
Depth-FirstStrategy

New nodes are inserted at the front of the FRINGE.



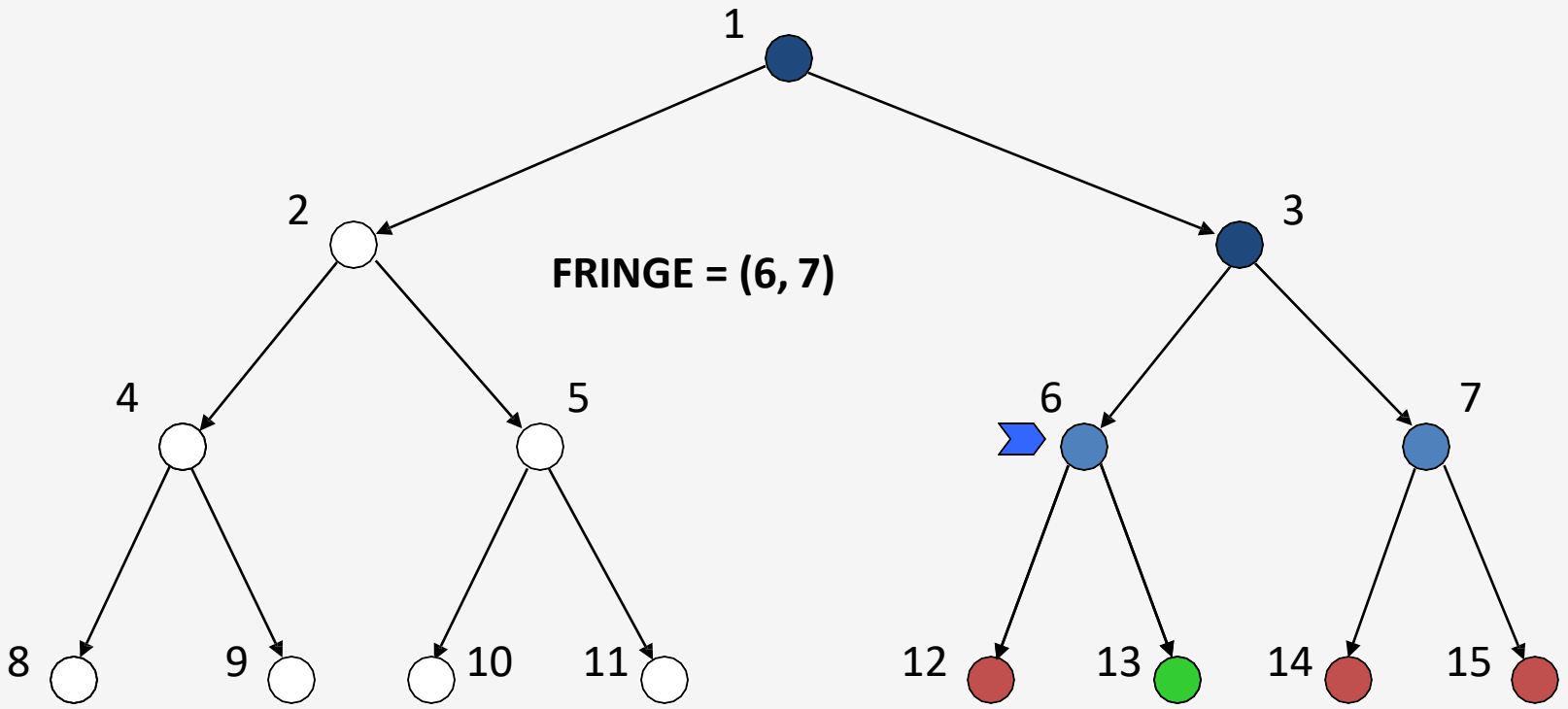
Depth-FirstStrategy

New nodes are inserted at the front of the FRINGE.



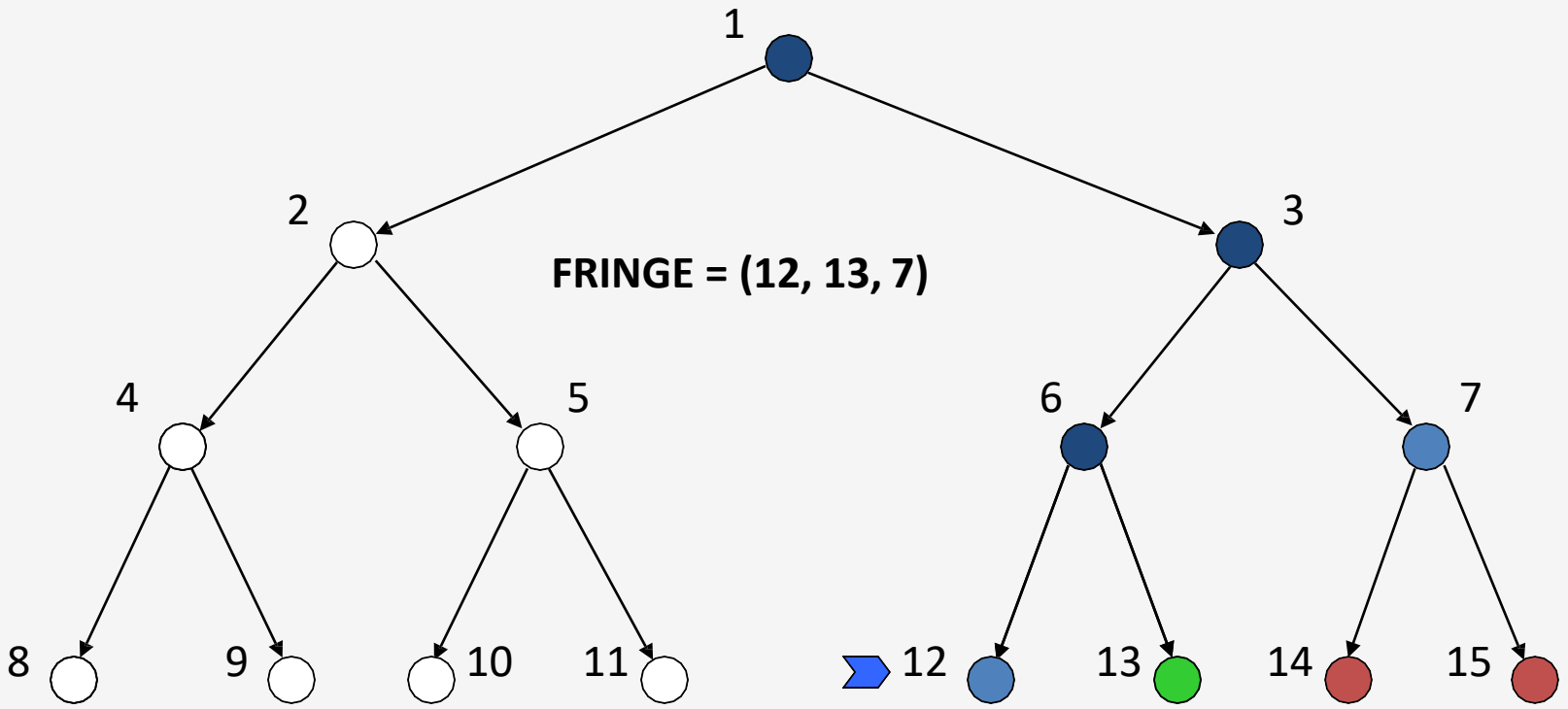
Depth-FirstStrategy

New nodes are inserted at the front of the FRINGE.



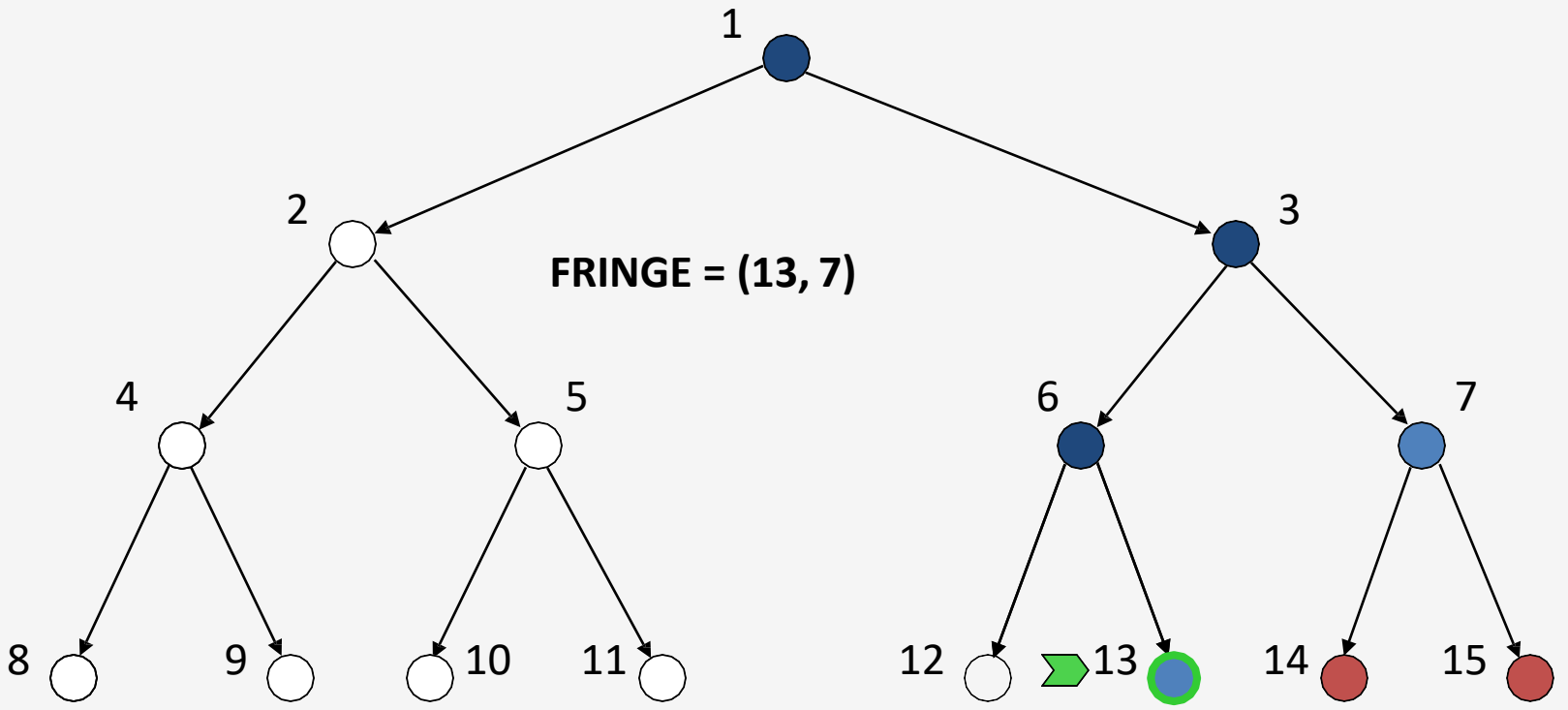
Depth-FirstStrategy

New nodes are inserted at the front of the FRINGE.



Depth-FirstStrategy

New nodes are inserted at the front of the FRINGE.



Analysis

Time complexity

- In the worst case, search entire space
- Goal may be at level d but tree may continue to level m, m>=d, $O(b^m)$
- Particularly bad if tree is infinitely deep

Space complexity

- Only need to save one set of children at each level
- 1 + b + b + … + b (m levels total) = O(bm)
- For previous example, DFS requires 118kb instead of 10 petabytes for d=12 (10 billion times less)

Optimal: it is nonoptimal as it may generate many steps or high costs to reach the goal state.

Completeness: Complete with in finite state space

Depth First Search.

Advantages:

- Requires very less memory as it only store a stack of nodes on the path from root node to the current node.
- Takes less time to reach to the goal node than BFS if it traverses in the right path

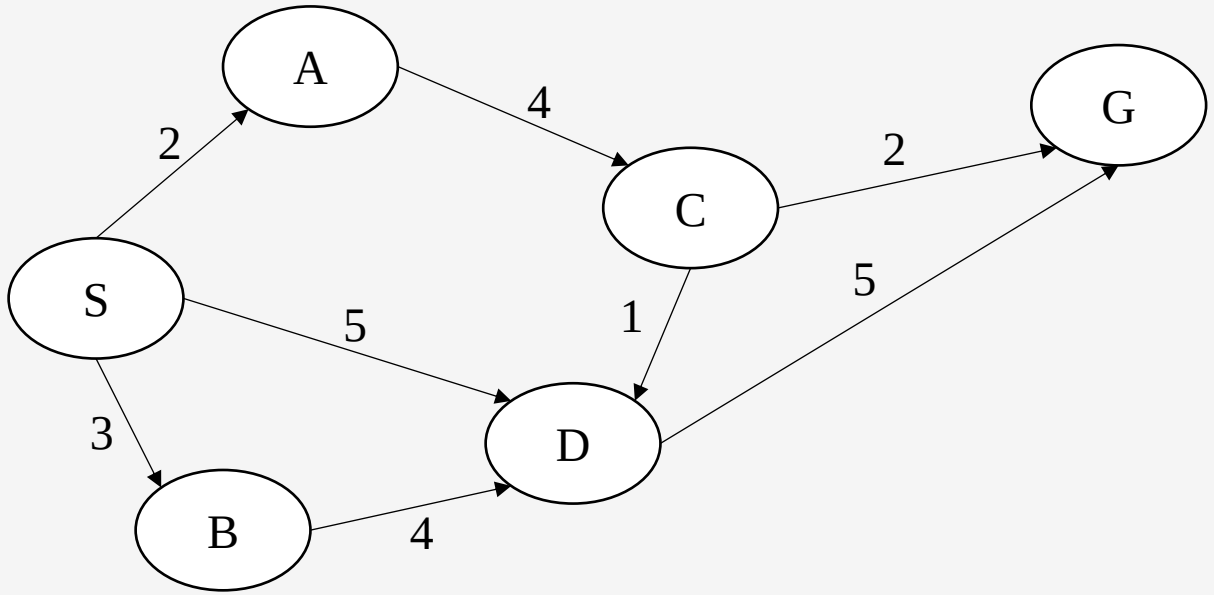
Disadvantages

- Possibility that many states keep reoccurring and there is no guarantee of finding the solution.
- DFS goes deep down searching and sometime it may go to the infinite loop

Depth Limited Search

- To overcome the problem of infinite depth in DFS, it can be limited to predetermined depth.
- Node at the depth limit will treat as it has no successor nodes further.
- It can be terminated with two conditions of failure:
  - Standard value Failure: the problem does not have a solution
  - Cutoff failure value: no solution for the problem within a given depth limit.

Depth-first search DFS

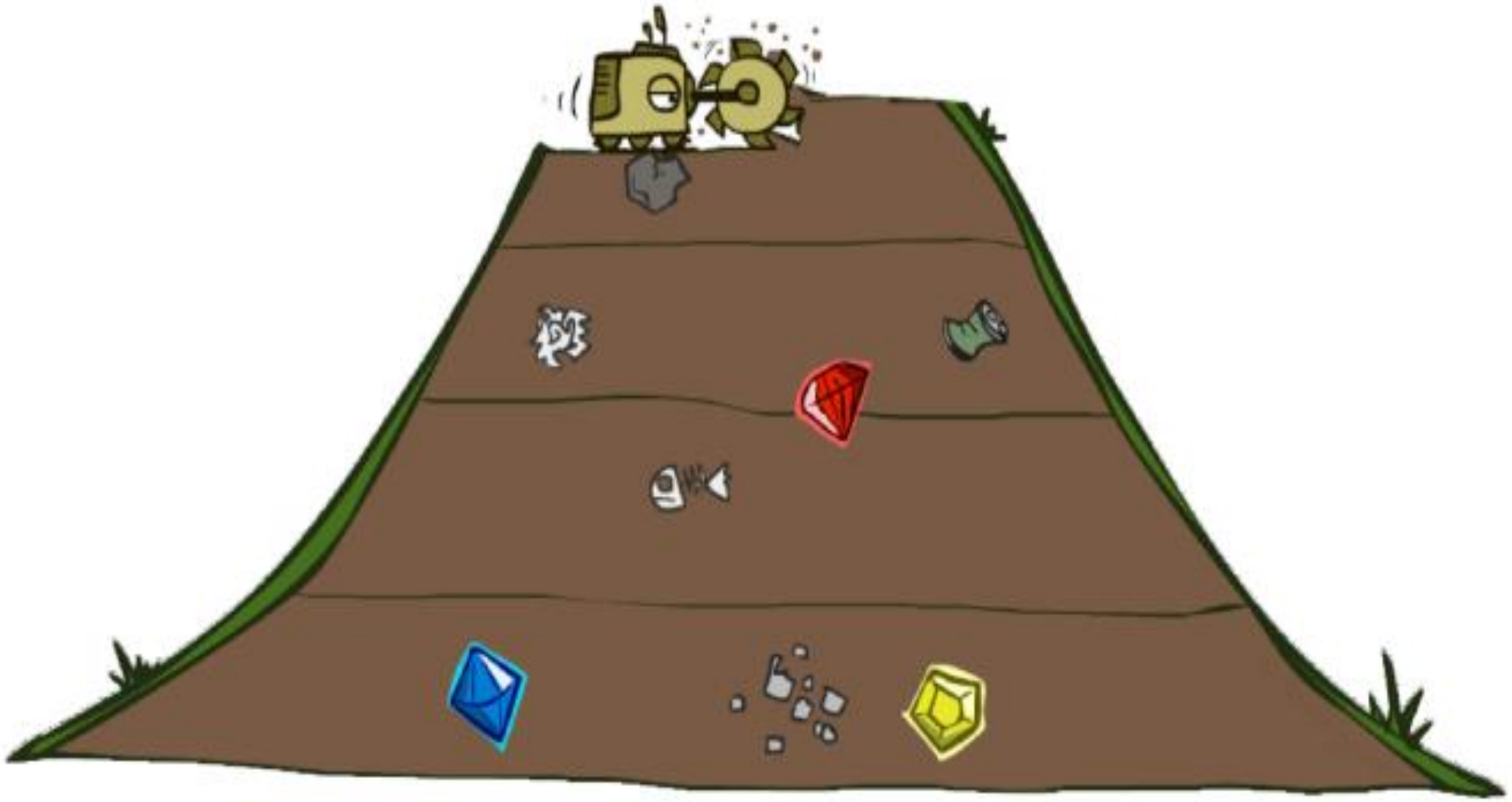


Fringe is a LIFO

Current	Fringe
-	S (0)
S (0)	A B D 1 1 1
D 1	A B G 1 1 2
G 2	

Path: S -> D -> G

Breadth-first search



Breadth-first search

In this strategy, the root node is expanded first, and then all the nodes generated by the root node are expanded next, including *their* successors, and so on.

In general, all the nodes at depth *d* in the search tree are expanded before the nodes at depth *d* + 1

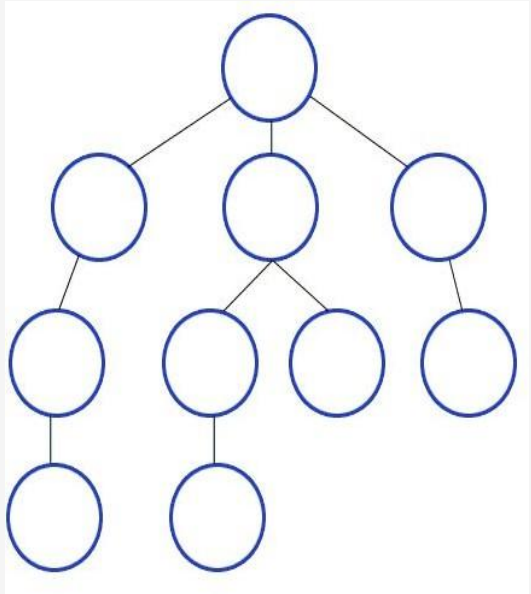
Breadth-first search can be implemented by calling the GENERAL-SEARCH algorithm with a queuing function that puts the newly generated states at the end of the queue after all the previously generated states.

Breadth-first search is a very systematic strategy because it considers all the paths of length 1 first, then all those of length 2, and so on.

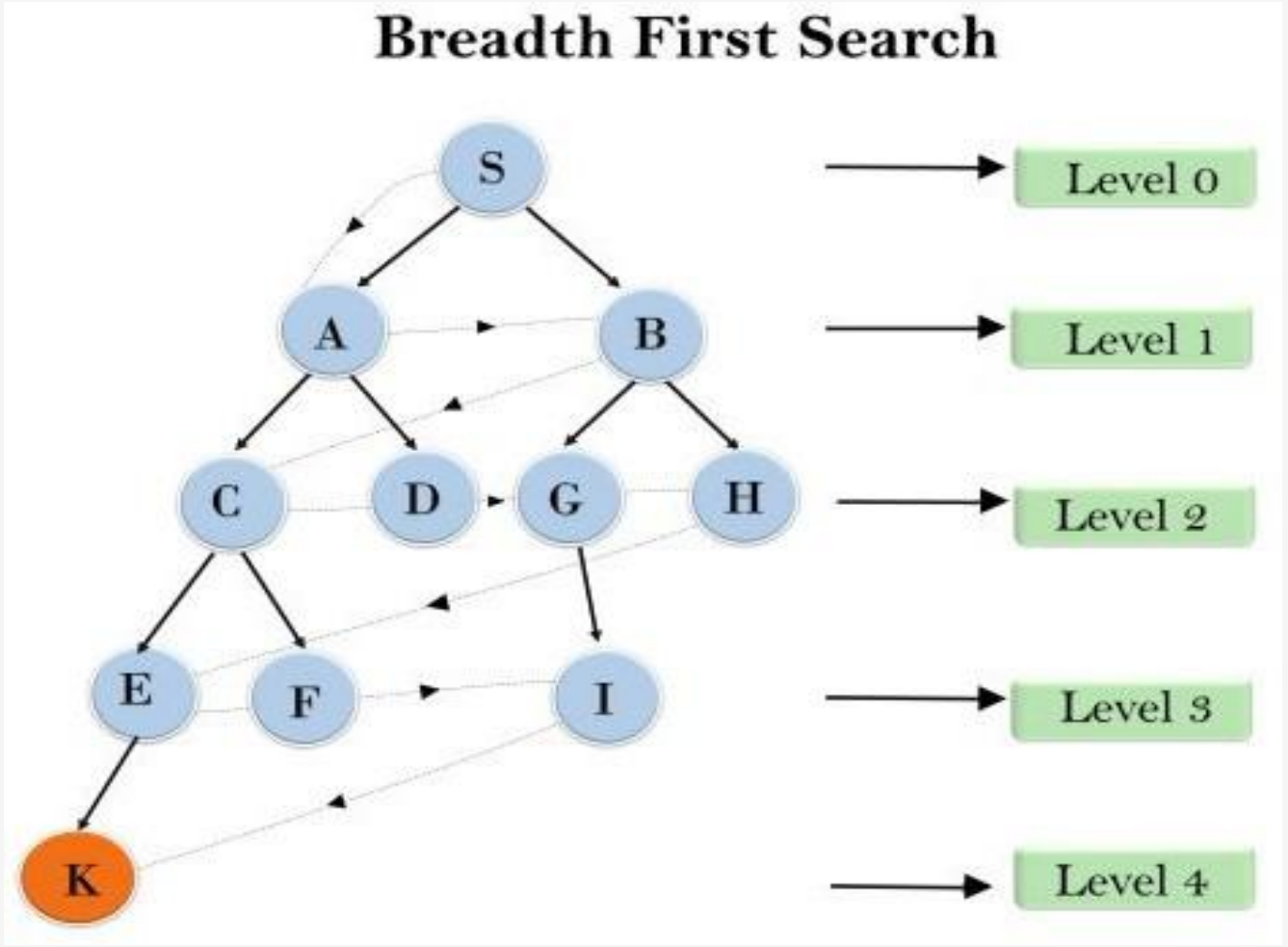
If there is a solution, breadth-first search is guaranteed to find it, and if there are several solutions, breadth-first search will always find the shallowest goal state first.

Breadth First Search (BFS)

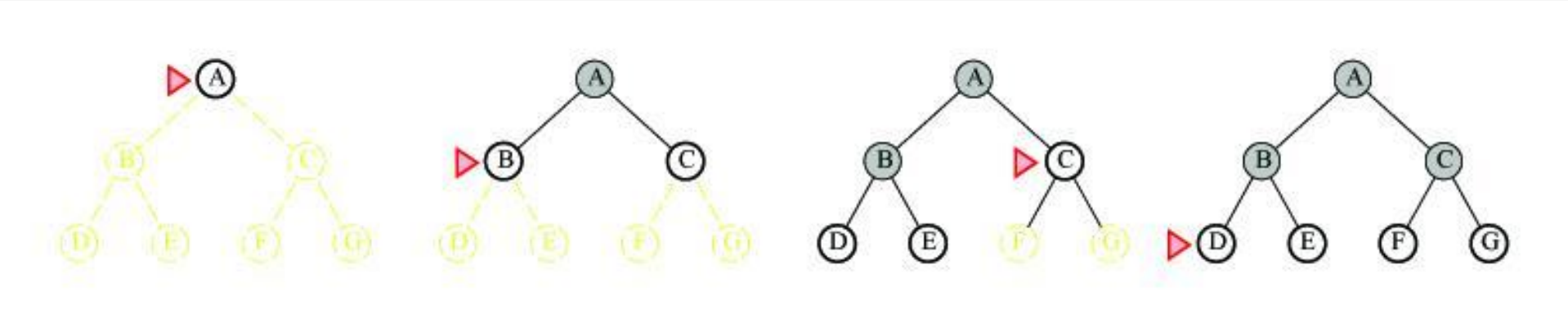
- Searches breadthwise in tree or graph, so it is called BFS
- Starts searching from root node of the tree and expands all the successor node at the current level before moving
- to next nodes
- It is implemented using queue data structure.



Diagram


Breadth First Search
S
A
B
C
D
G
H
E
F
I
K
Level 0
Level 1
Level 2
Level 3
Level 4

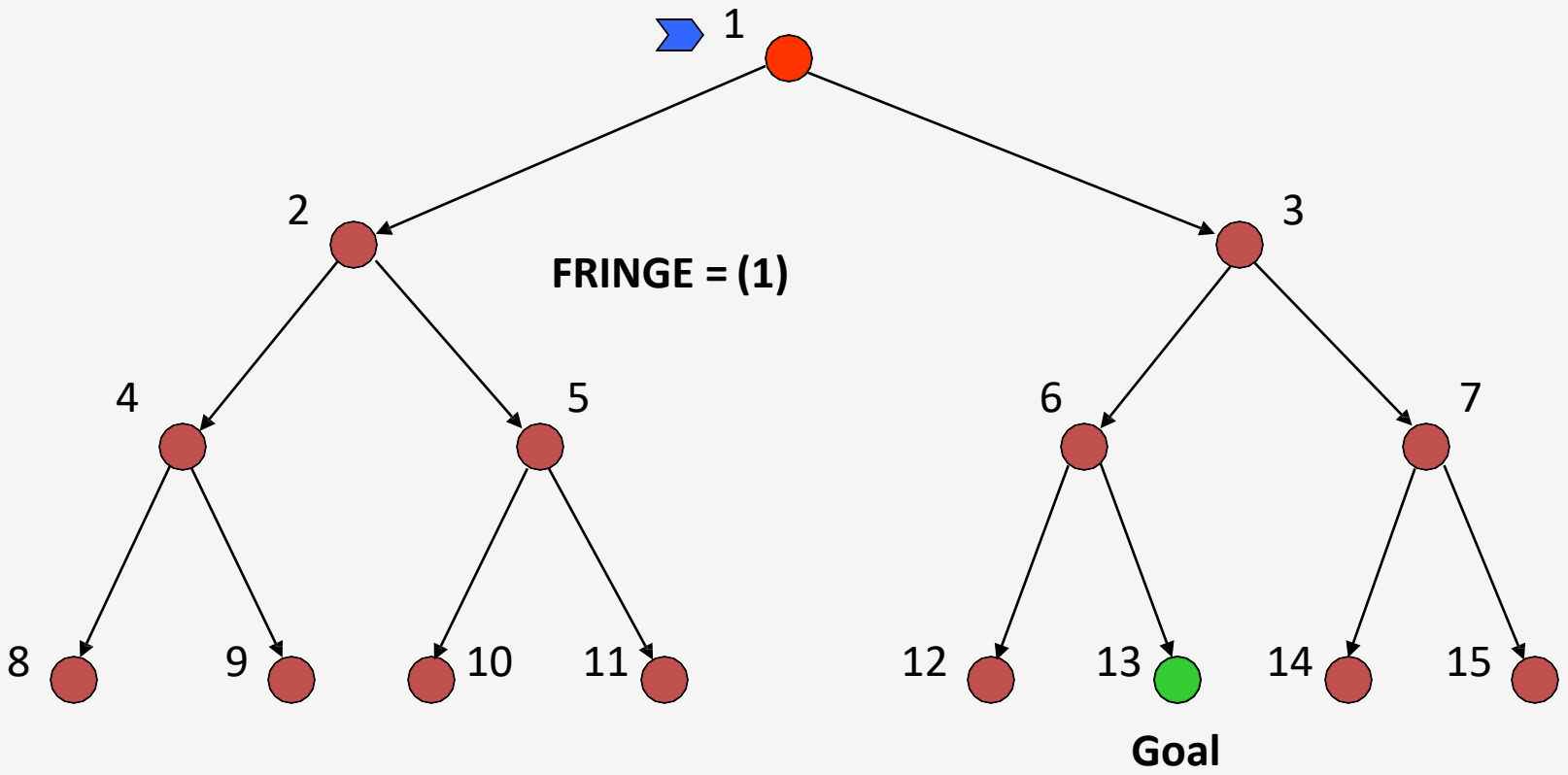
BFS Examples



b = 2

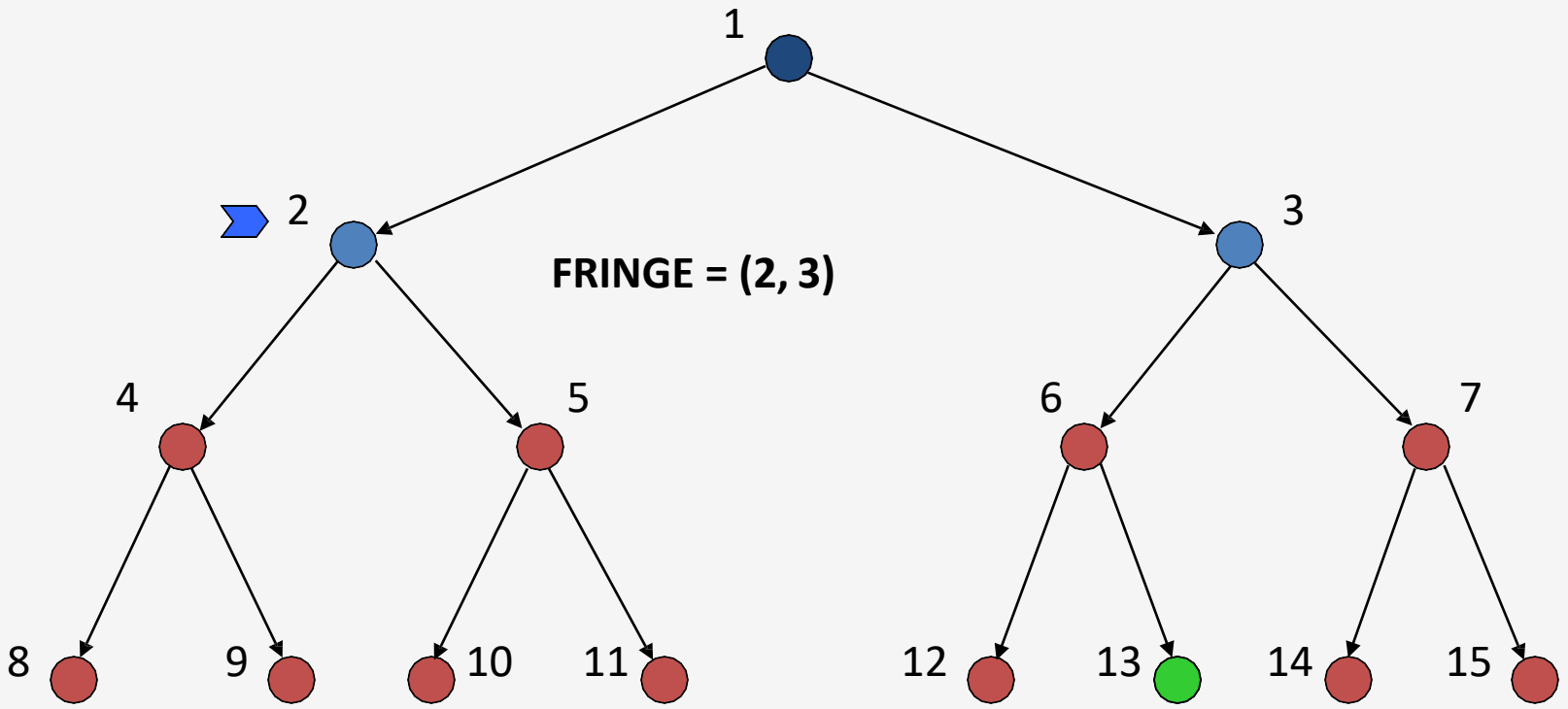
Breadth-FirstStrategy

New nodes are inserted at the end of the FRINGE.



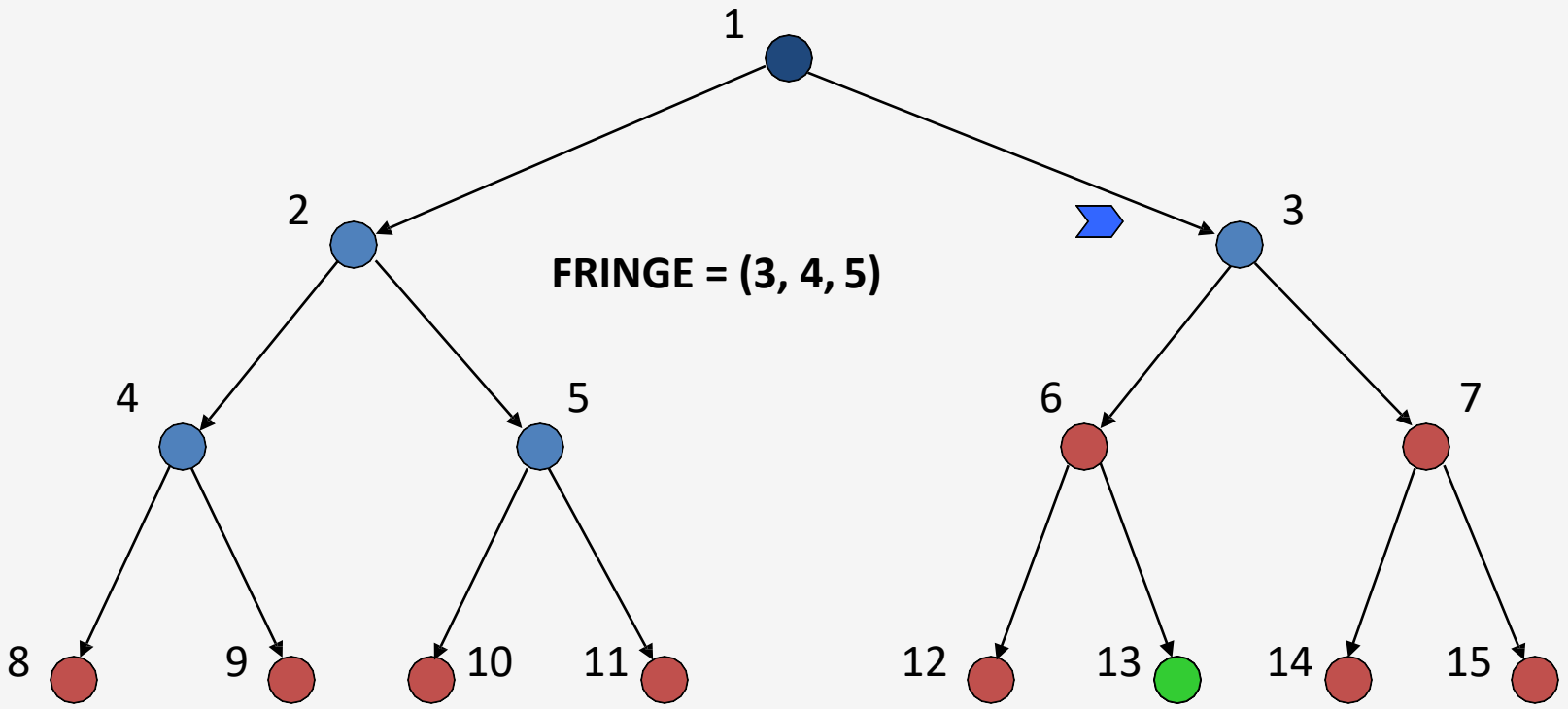
Breadth-FirstStrategy

New nodes are inserted at the end of the FRINGE.



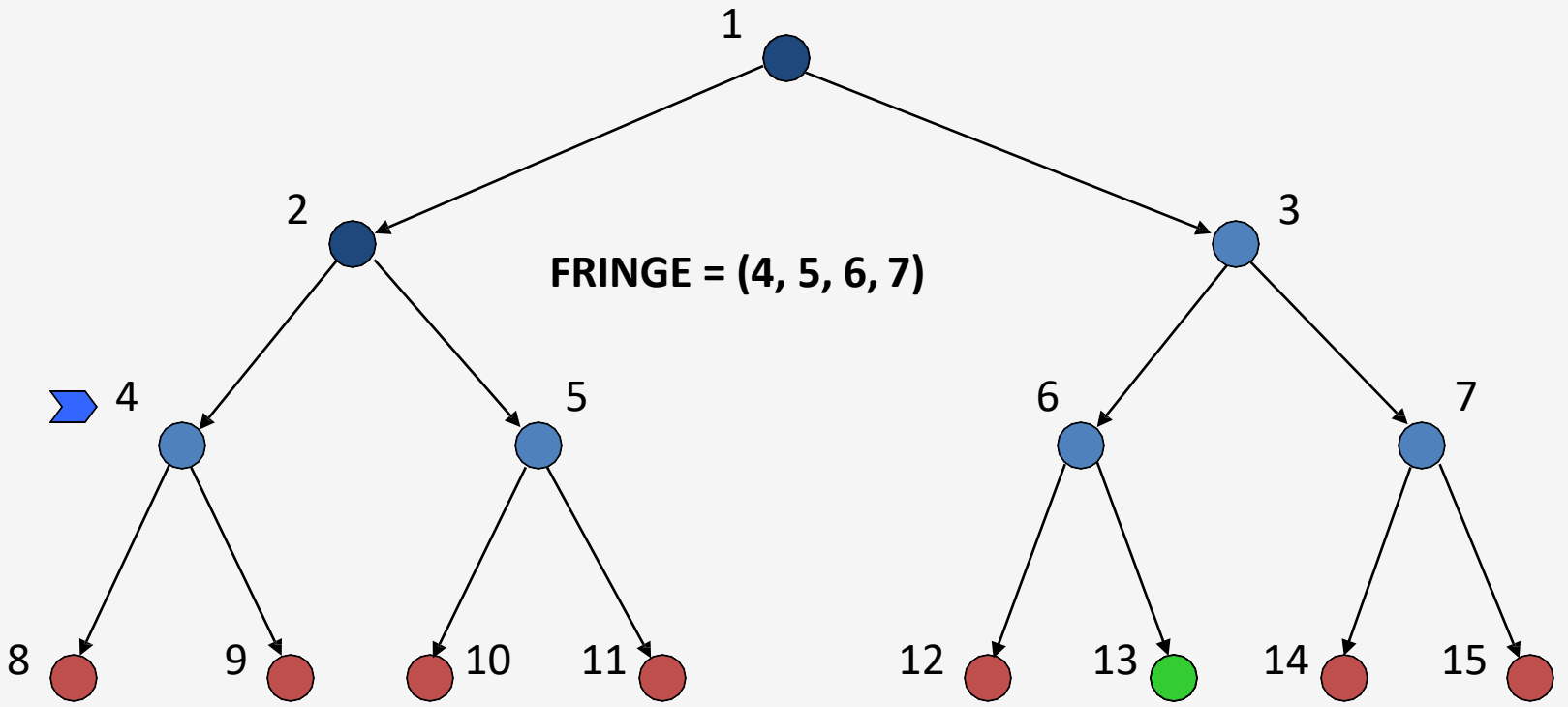
Breadth-FirstStrategy

New nodes are inserted at the end of the FRINGE.



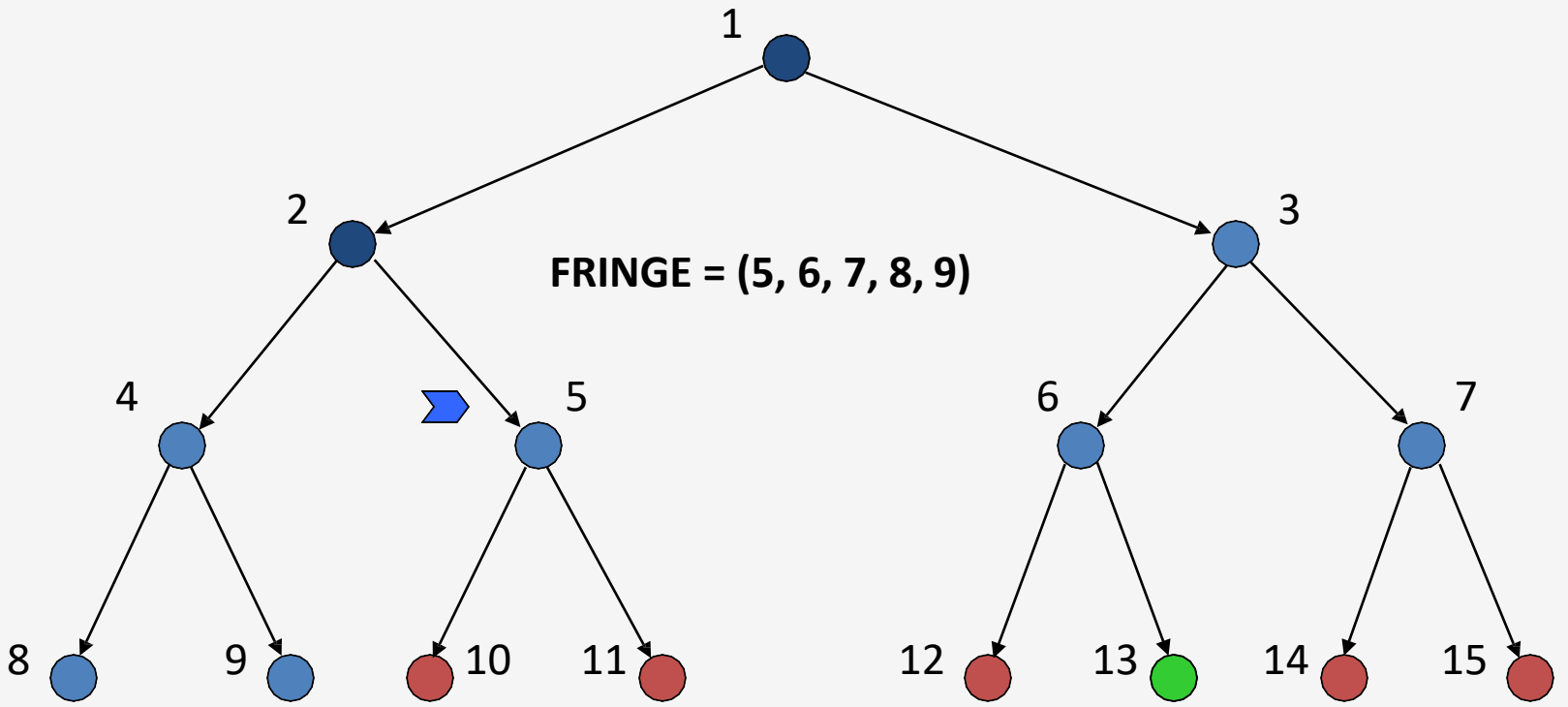
Breadth-FirstStrategy

New nodes are inserted at the end of the FRINGE.



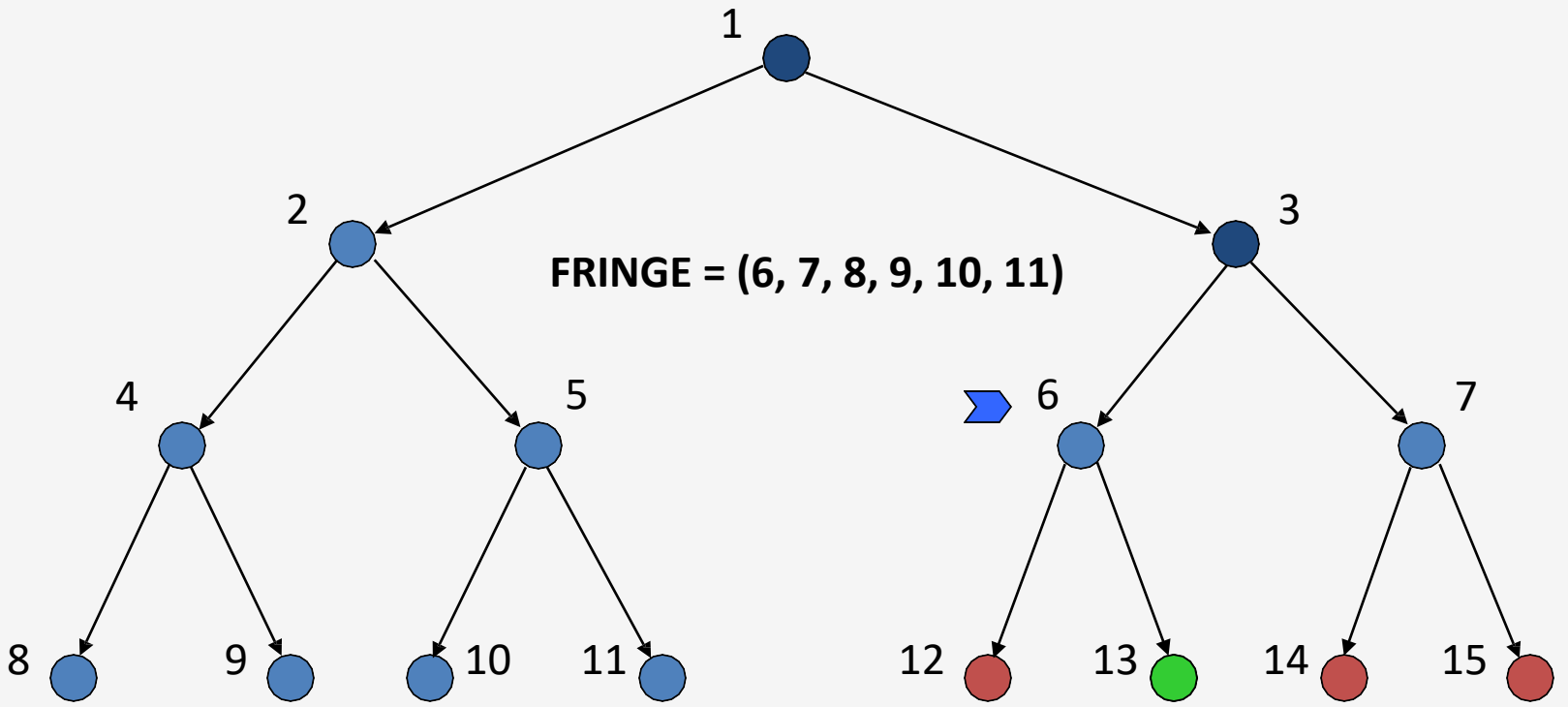
Breadth-FirstStrategy

New nodes are inserted at the end of the FRINGE.



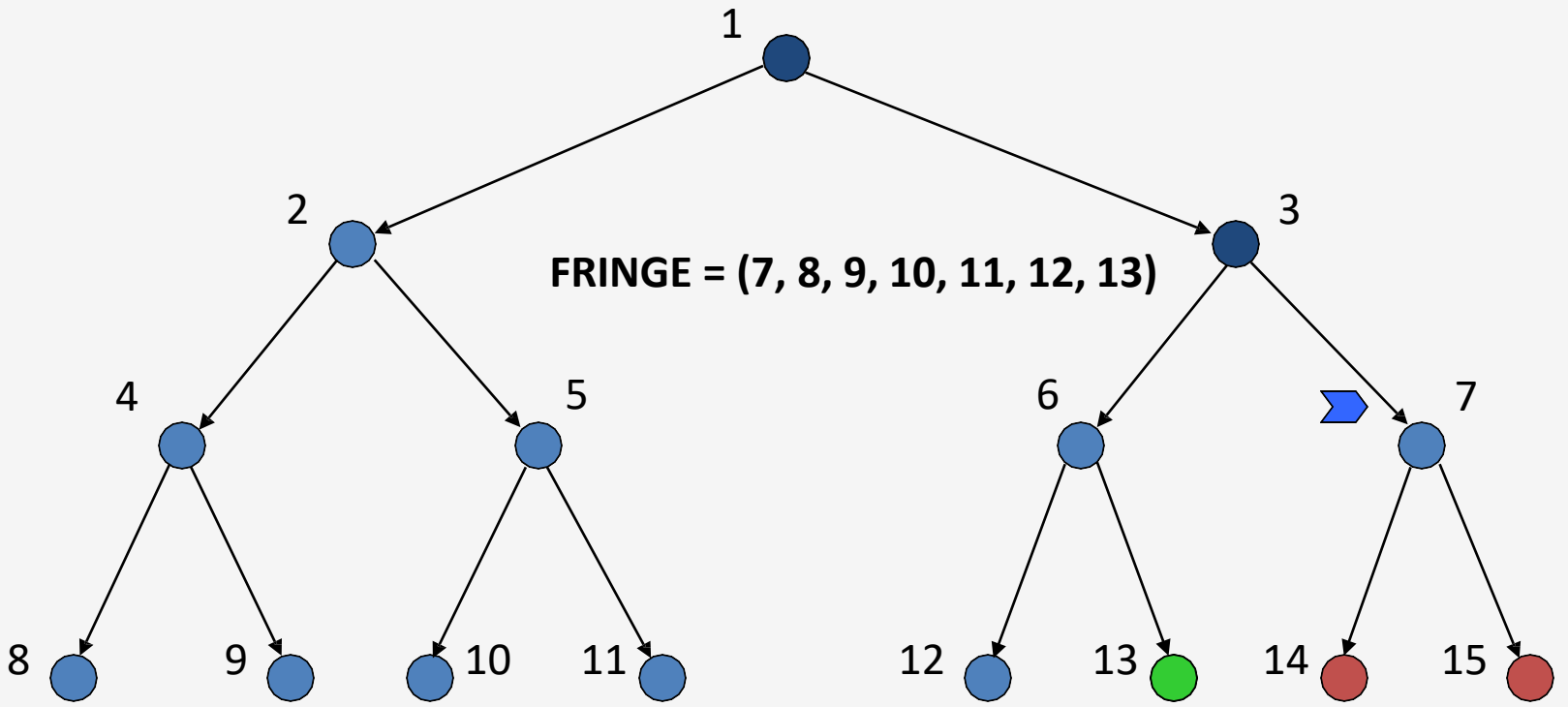
Breadth-FirstStrategy

New nodes are inserted at the end of the FRINGE.



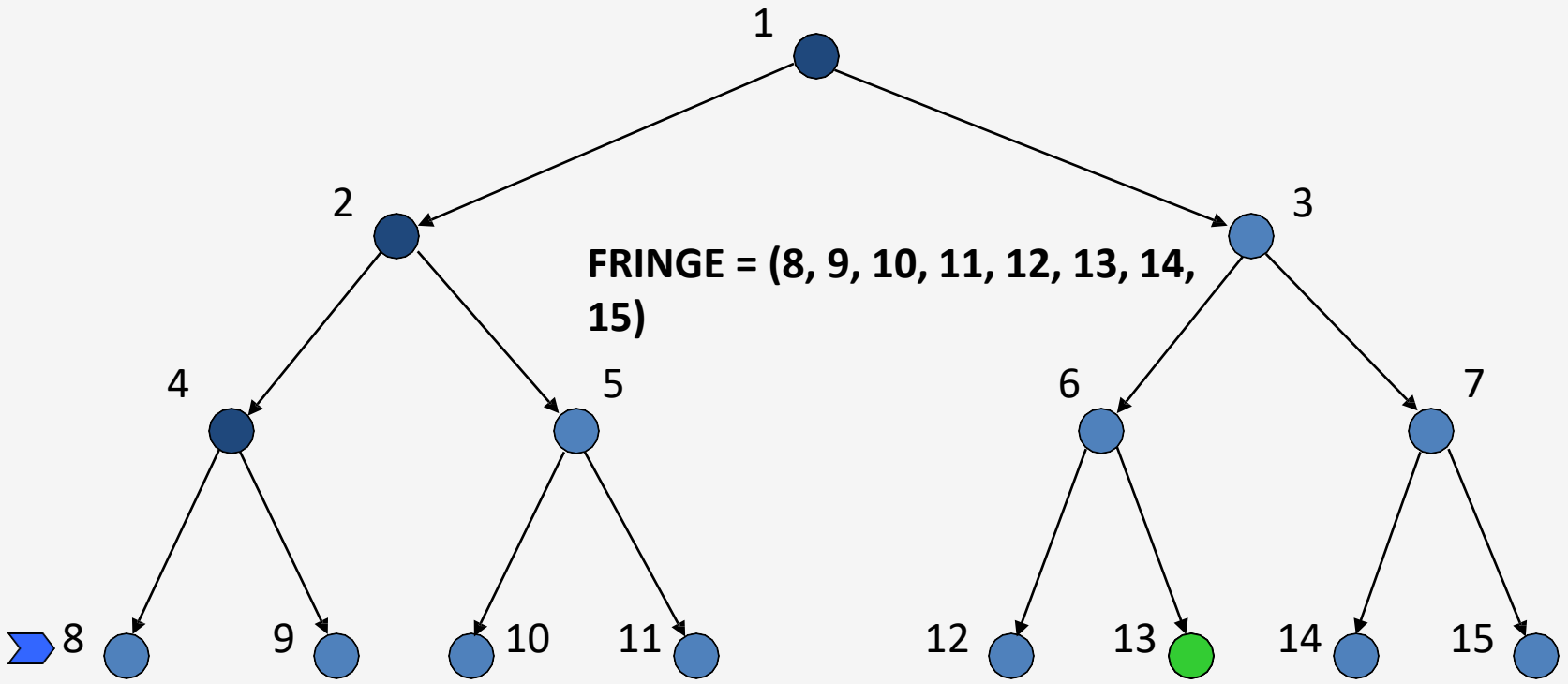
Breadth-FirstStrategy

New nodes are inserted at the end of the FRINGE.



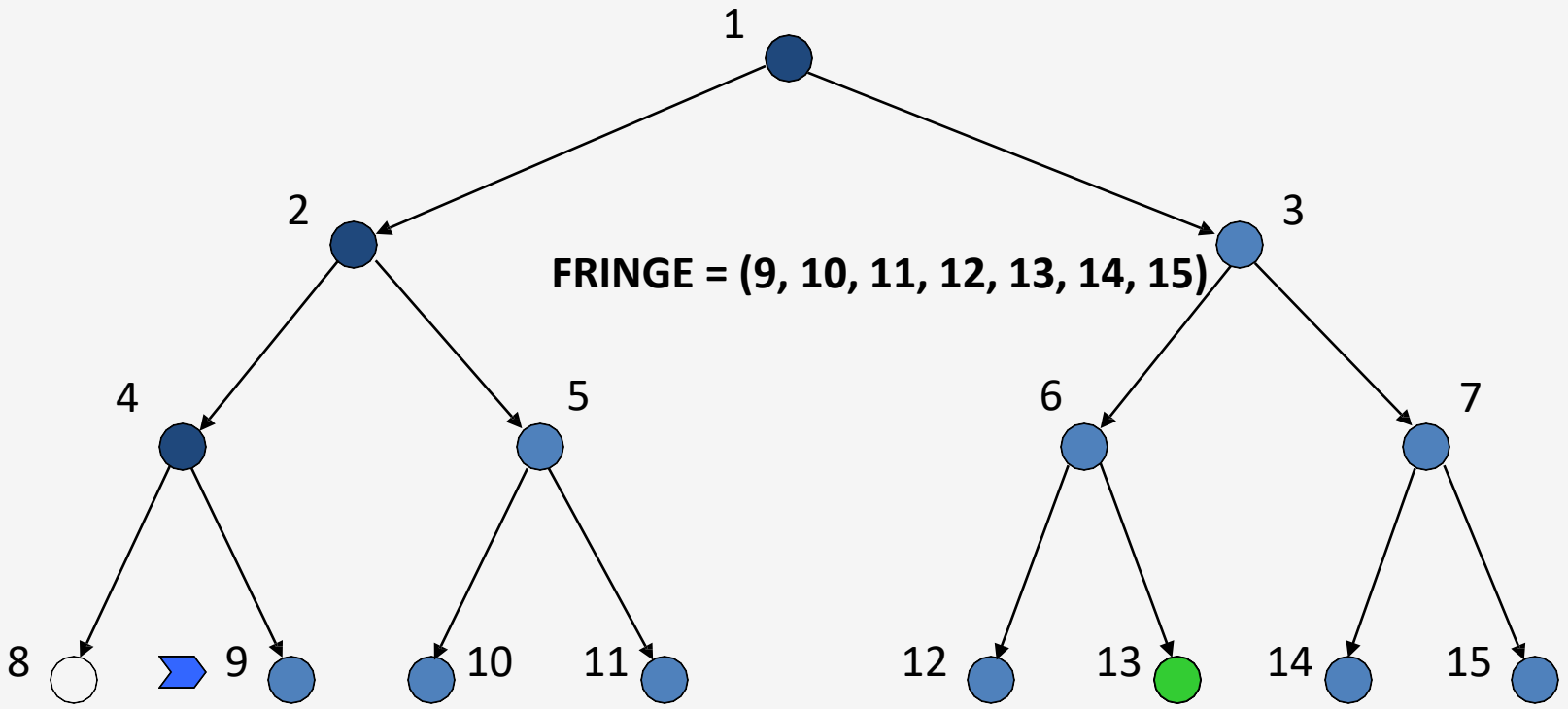
Breadth-FirstStrategy

New nodes are inserted at the end of the FRINGE.



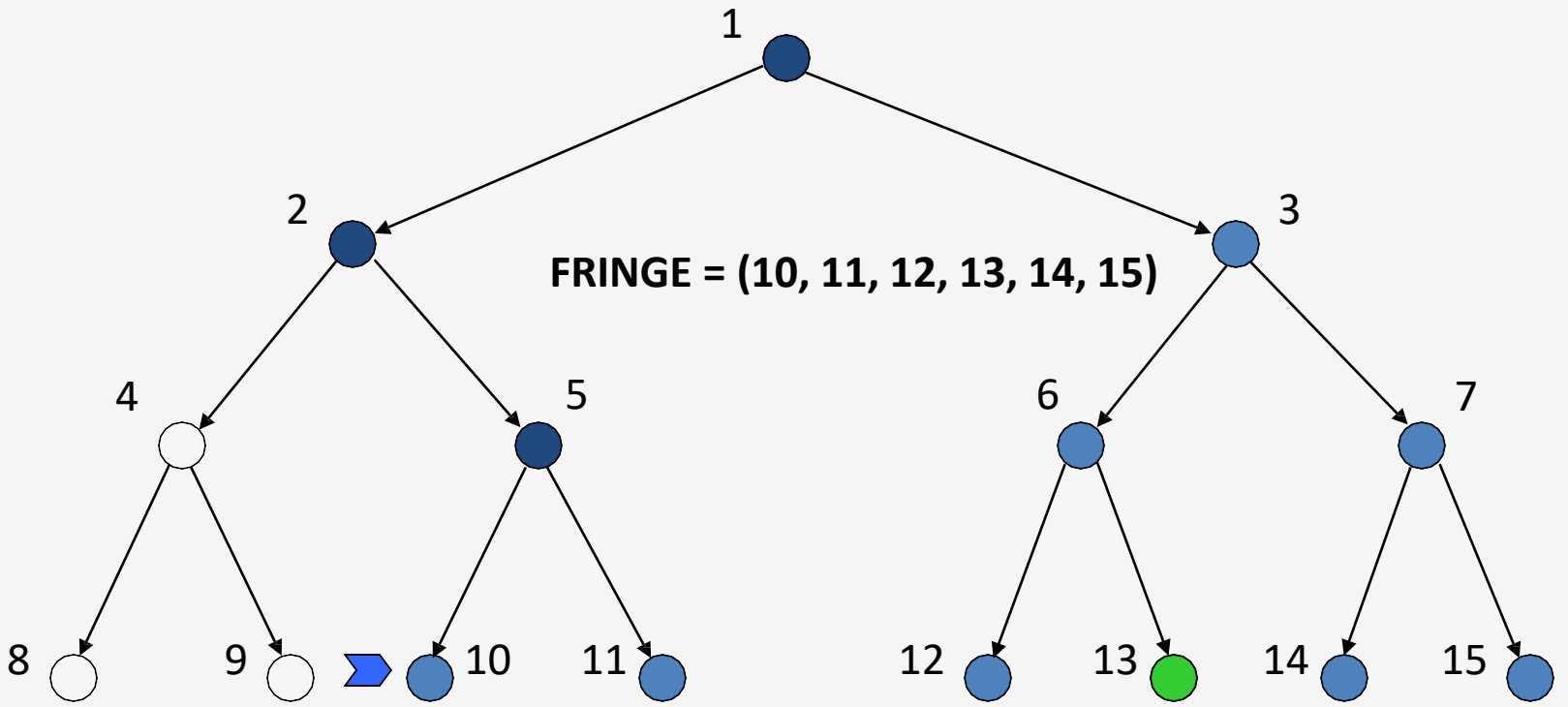
Breadth-FirstStrategy

New nodes are inserted at the end of the FRINGE.



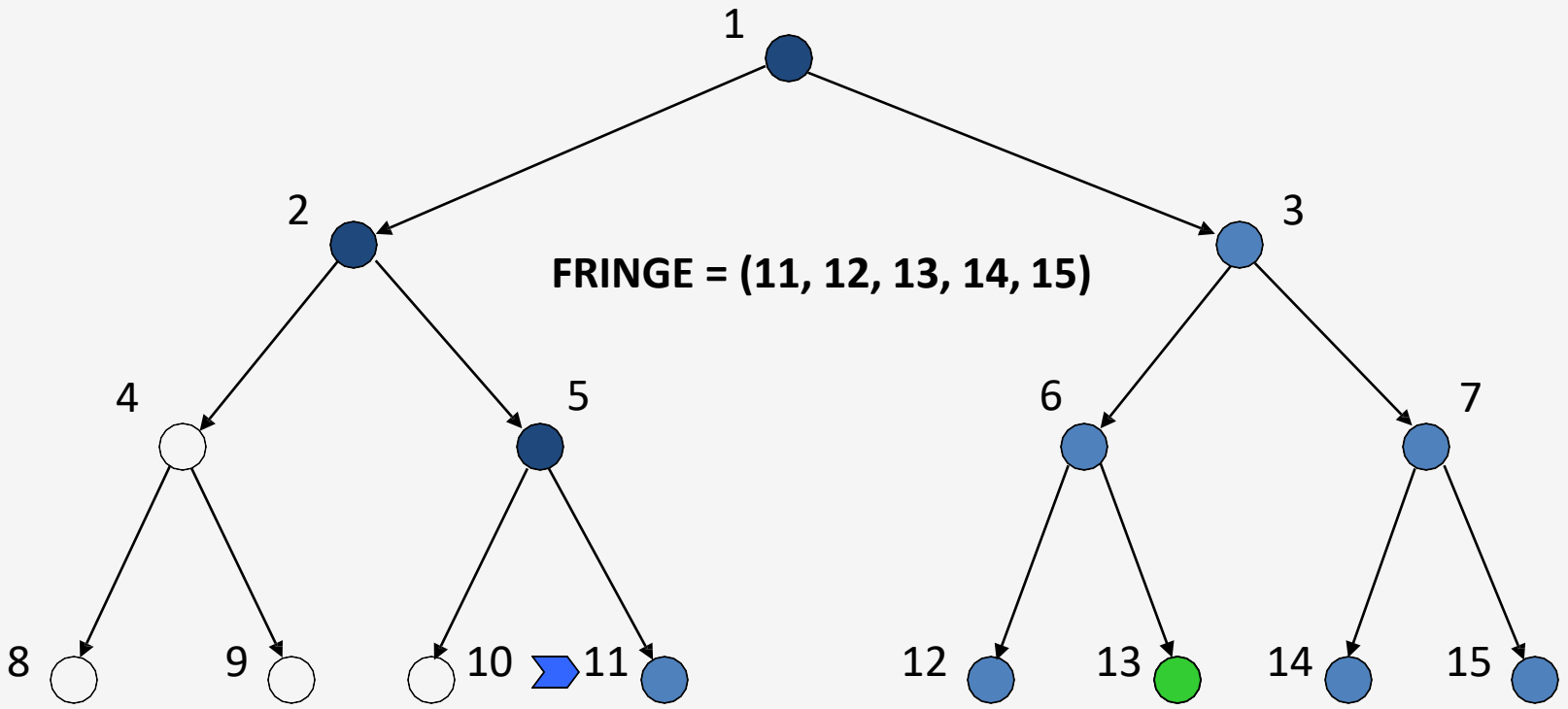
Breadth-FirstStrategy

New nodes are inserted at the end of the FRINGE.



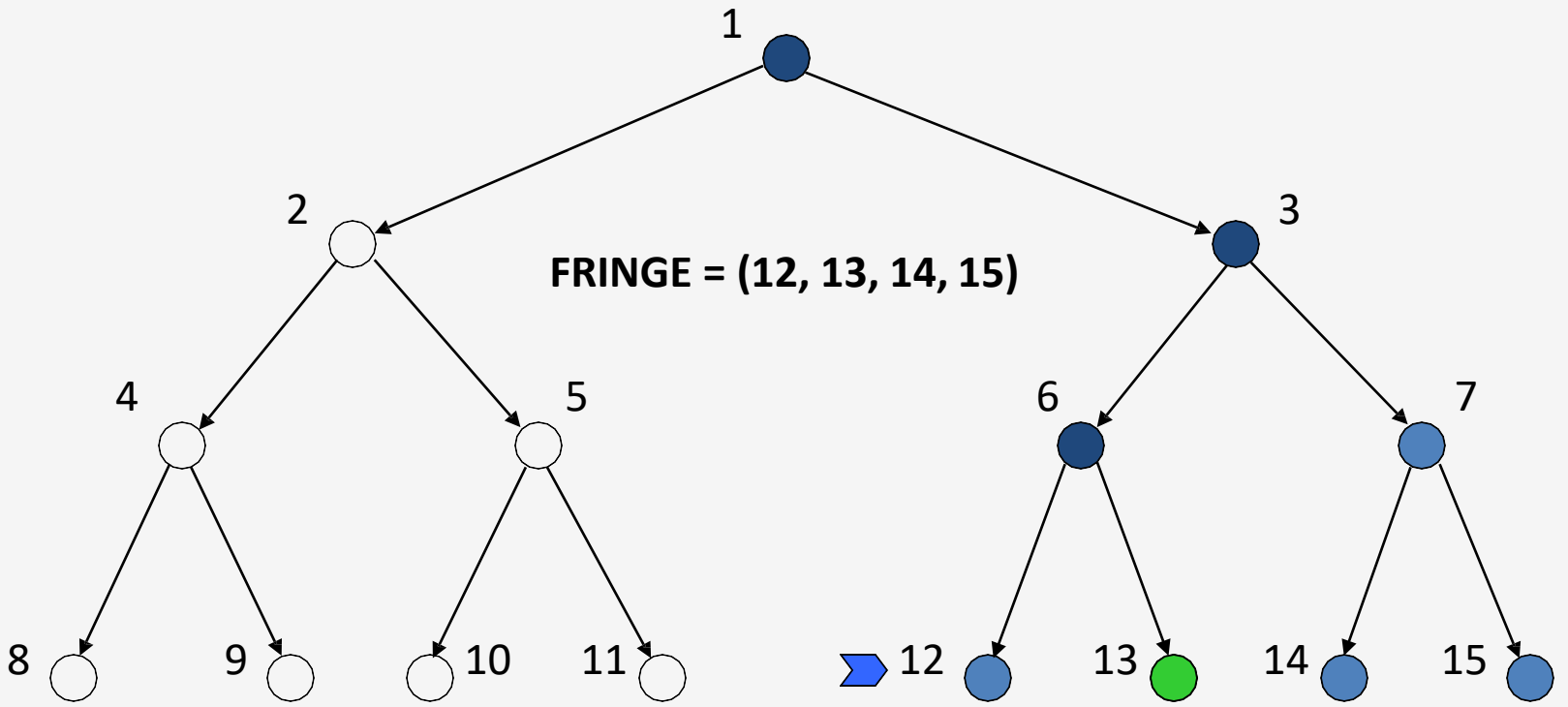
Breadth-FirstStrategy

New nodes are inserted at the end of the FRINGE.



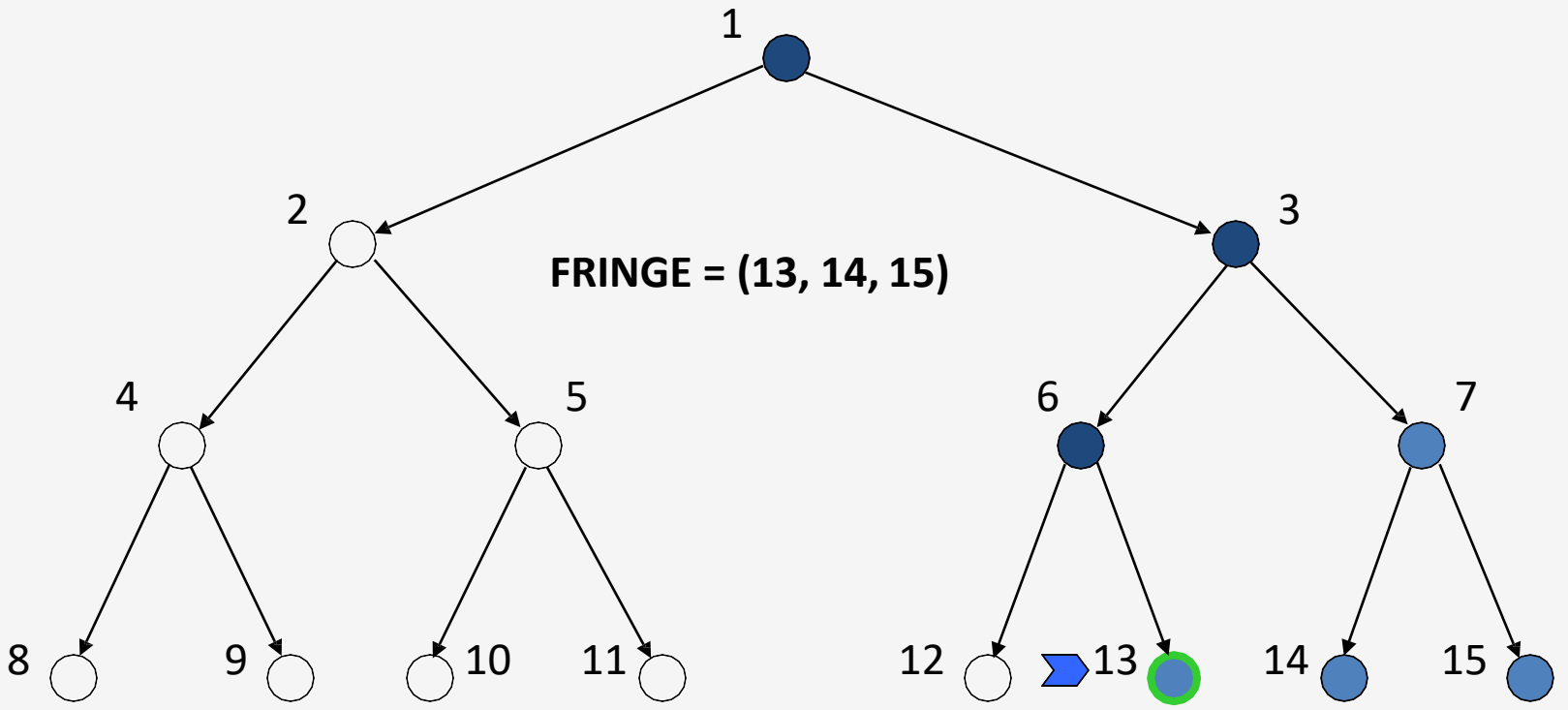
Breadth-FirstStrategy

New nodes are inserted at the end of the FRINGE.



Breadth-FirstStrategy

New nodes are inserted at the end of the FRINGE.



Breadth-first search

In terms of the four criteria

- breadth-first search is **complete**, and it is **optimal**

Drawbacks

- the memory requirements are a bigger problem for breadth-first search
- time requirements are still a major factor.

In general,

exponential complexity search problems cannot be solved for any but the smallest instances.

Analysis

Assume goal node at level d with constant branching factor b

Time complexity (measured in #nodes generated)

➢1 (1st level) + b (2nd level) + b^2 (3rd level) + … + b^d (goal level) + (b^{d+1} – b) = O(b^{d+1})

This assumes goal on far right of level

Space complexity

➢At most majority of nodes at level d + majority of nodes at level d+1 = O(b^{d+1})

➢Exponential time and space

Completeness: If the shallowest goal node is at some finite depth then BFS will find a solution.

Optimality: If the past cost is non-decreasing function of the depth of the node.

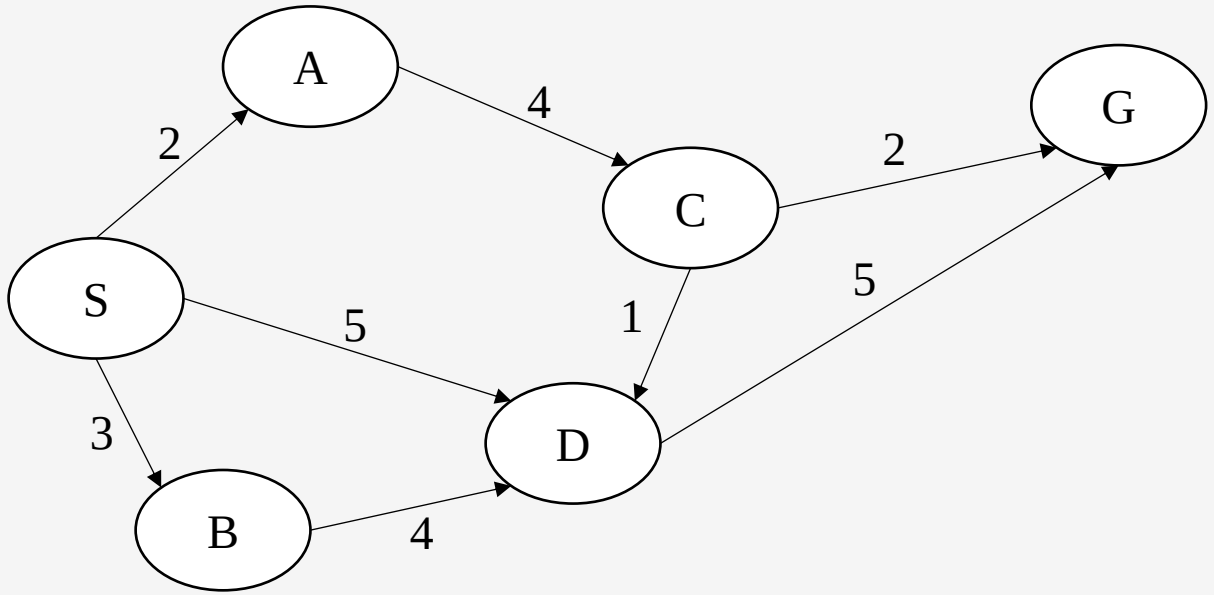
Analysis

See what happens with b=10

- expand 10,000 nodes/second
- 1,000 bytes/node

Depth	Nodes	Time	Memory
2	10^3	.11 seconds	1 megabyte
4	10^5	11 seconds	106 megabytes
6	10^7	19 minutes	10 gigabytes
8	10^9	31 hours	1 terabyte
10	10^{11}	129 days	101 terabytes
12	10^{13}	35 years	10 petabytes
15	10^{16}	3,523 years	1 exabyte

Breadth-first search BFS



Path: S -> D -> G

Fringe is a FIFO

Current	Fringe
-	S (0)
S (0)	SA SB SD (1) (1) (1)
SA (1)	SB SD SAC (1) (1) (2)
SB (1)	SD SAC SBD (1) (2) (2)
SD (1)	SAC SBD SDG (2) (2) (2)
SAC (2)	SBD SDG SACD SACG (2) (2) (3) (3)
SDG (2)	

Comparison of Search Techniques

	DFS	BFS
Complete	N	Y
Optimal	N	N
Heuristic	N	N
Time	b^m	b^{d+1}
Space	bm	b^{d+1}

Depth-Limited Strategy

Depth-first with **depth cutoff *k*** *(maximal depth below which nodes are not expanded)*

Three possible outcomes:

- Solution.
- Failure *(no solution)*.
- Cutoff *(no solution within cutoff)*.

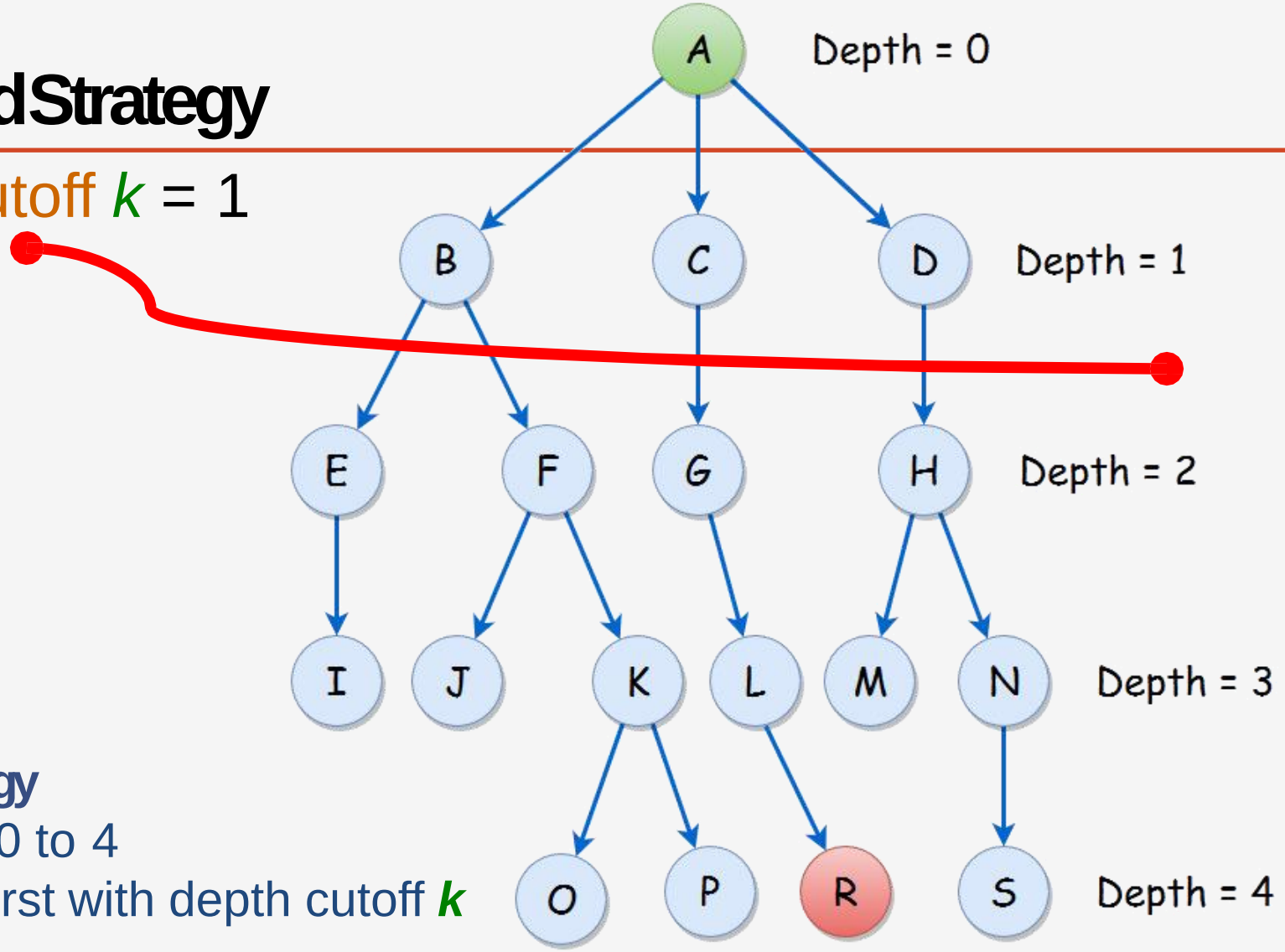
Iterative- Deepening Strategy

Repeat for *k* = 0, 1, 2, .. :

- Perform depth-first with depth cutoff ***k***

Depth-Limited Strategy

E.g., depth cutoff *k* = 1



Iterative-Deepening Strategy

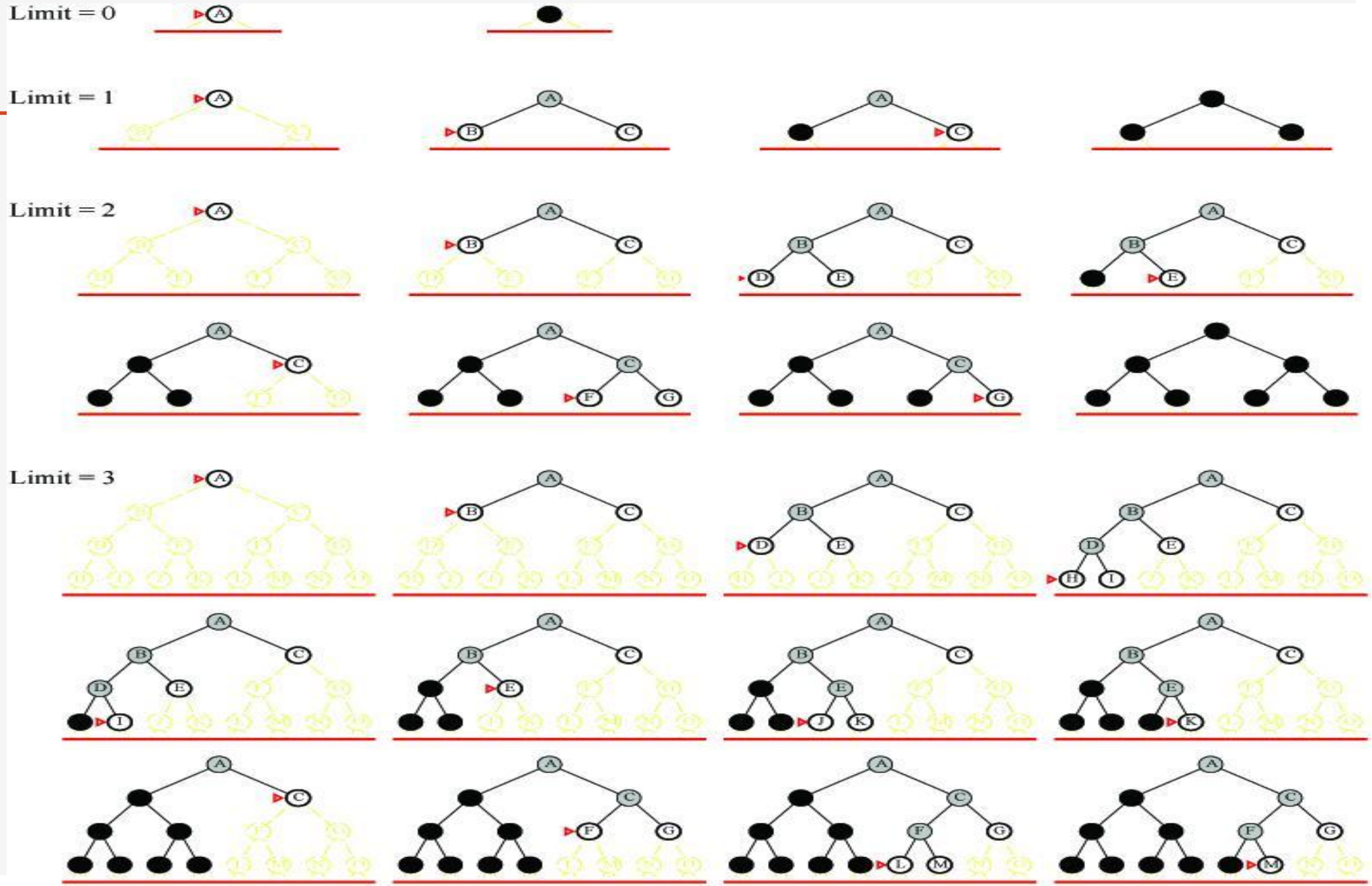
E.g., Repeat for *k* = 0 to 4

- Perform depth-first with depth cutoff ***k***

Iterative Deepening Search

- DFS with depth bound
- QueuingFn is enqueue at front as with DFS
  - Expand(state) only returns children such that depth(child) <= threshold
  - This prevents search from going down infinite path
- First threshold is 1
  - If do not find solution, increment threshold and repeat

Examples



Analysis

What about the repeated work?

Time complexity (number of generated nodes)

- $[b] + [b + b^2] + .. + [b + b^2 + .. + b^d]$
- $(d)b + (d-1)\ b^{2\ +}\ \ldots\ +\ (1)\ b^d$
- $O(b^d)$

Comparing Depth, Breadth, & Iterative Deepening Search Algorithms

- Breadth-first and iterative deepening guarantee shortest solution.
- Breadth-first: high space complexity.
- Depth-first: low space complexity.
- Iterative deepening: best performance in terms of orders of complexity.

Bidirectional Search

Bidirectional search is a graph search algorithm that finds the shortest path from an initial start state to a goal state in a directed graph.

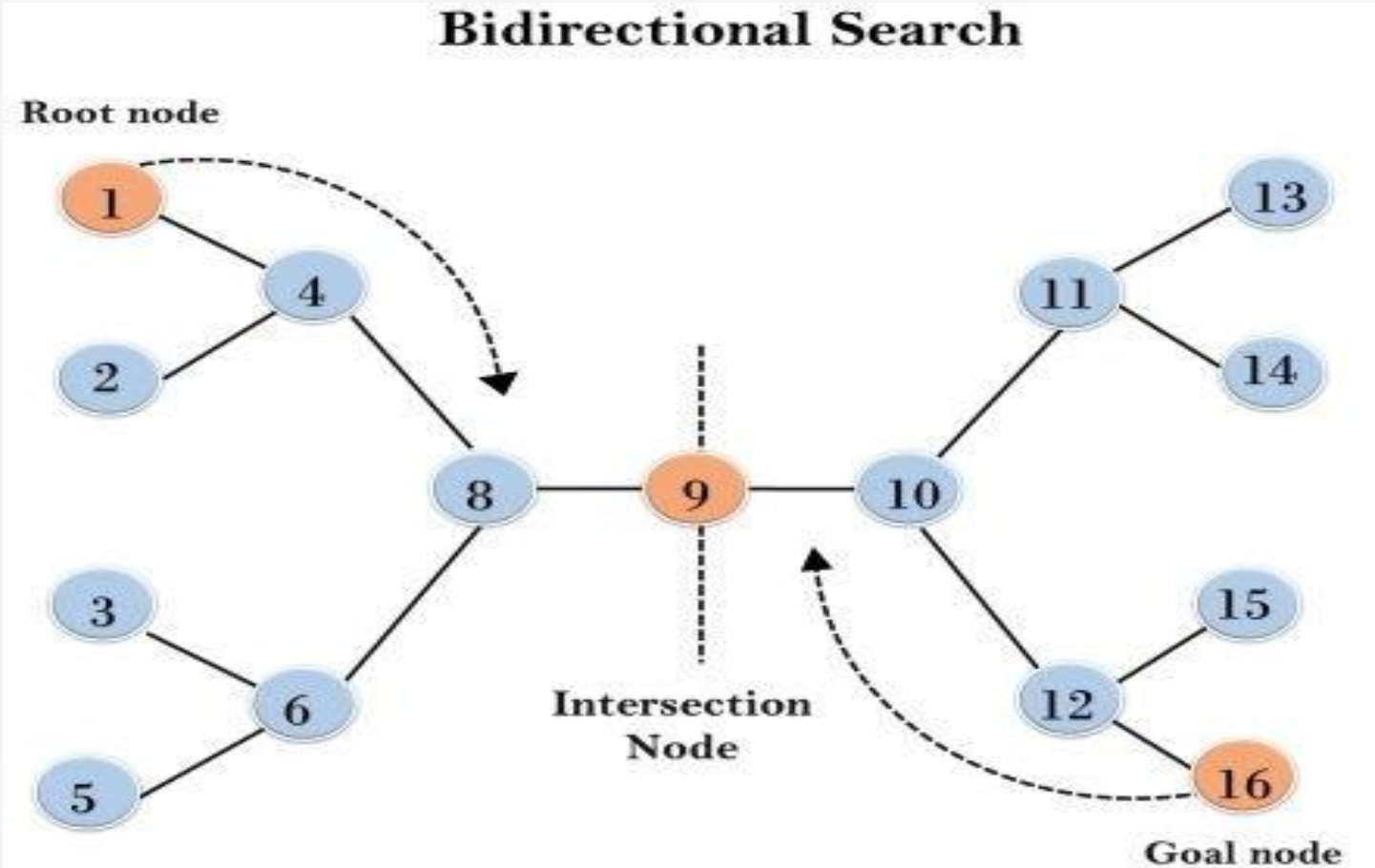
It **runs two simultaneous searches**: one forward from the initial state and one backward from the goal, stopping when they meet.

The **reason** for this approach is that **in many cases, it is faster**: for instance, in a simplified model of search problem complexity in which both searches expand a tree with branching factor b, and the distance from start to goal is d, each of the two searches has complexity $O(b^{d/2})$ (in Big O notation), and the sum of these two search times is much less than the $O(b^d)$ complexity that would result from a single search from the beginning to the goal.

Bidirectional Search

- Runs two simultaneous searches, one from initial state called as forward search and other from the goal state called as backward search.
- Replaces one simple search graph with two small subgraph.
- The search stops when these two graphs intersect each other.
- It can use search techniques such as BFS, DFS, DLS etc.

Diagram


Bidirectional Search
Root node
1
2
3
4
5
6
8
9
10
11
12
13
14
15
16
Intersection Node
Goal node

Bidirectional Search.

Advantages:

- It is fast
- Requires less memory

Disadvantages:

- DIfficult to implement
- Should know the goal state in advance

Analysis

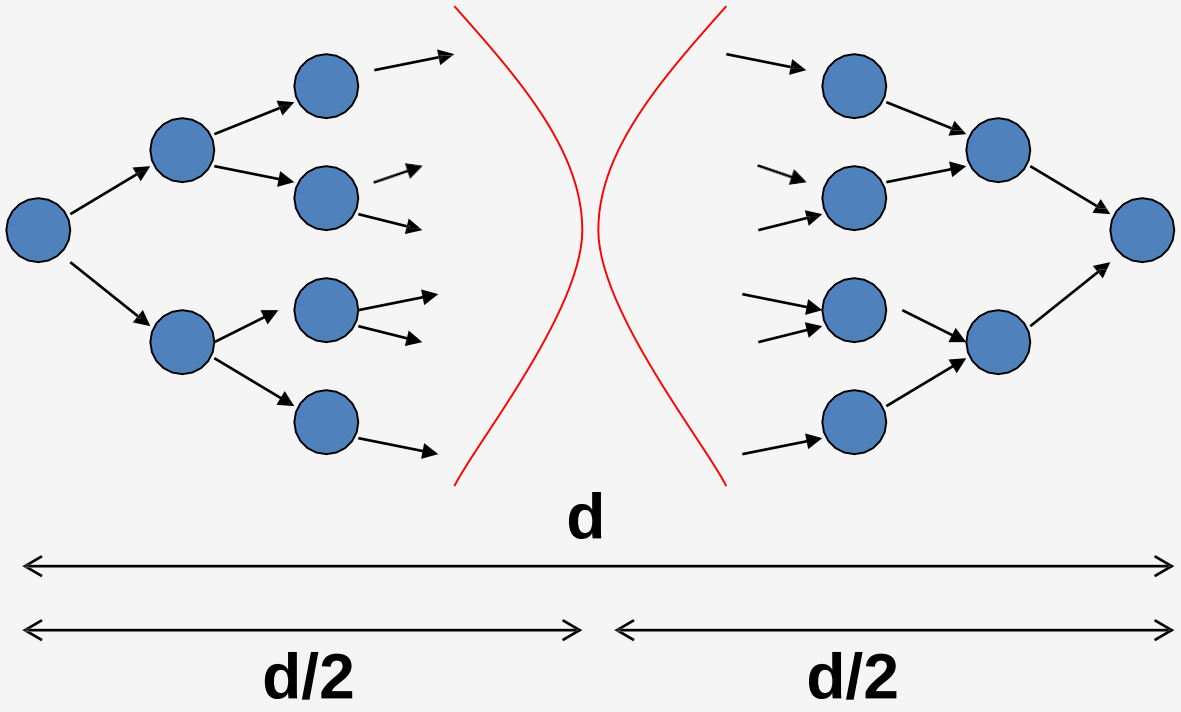
- **Time Complexity:** bidirectional using BFS is **O($b^{d/2}$)**
- **Space Complexity: O($b^{d/2}$)**
- **Optimal:** It is optimal
- **Completeness:** complete if we use BFS in both searches

Bidirectional Search

Complexity of Bidirectional Search

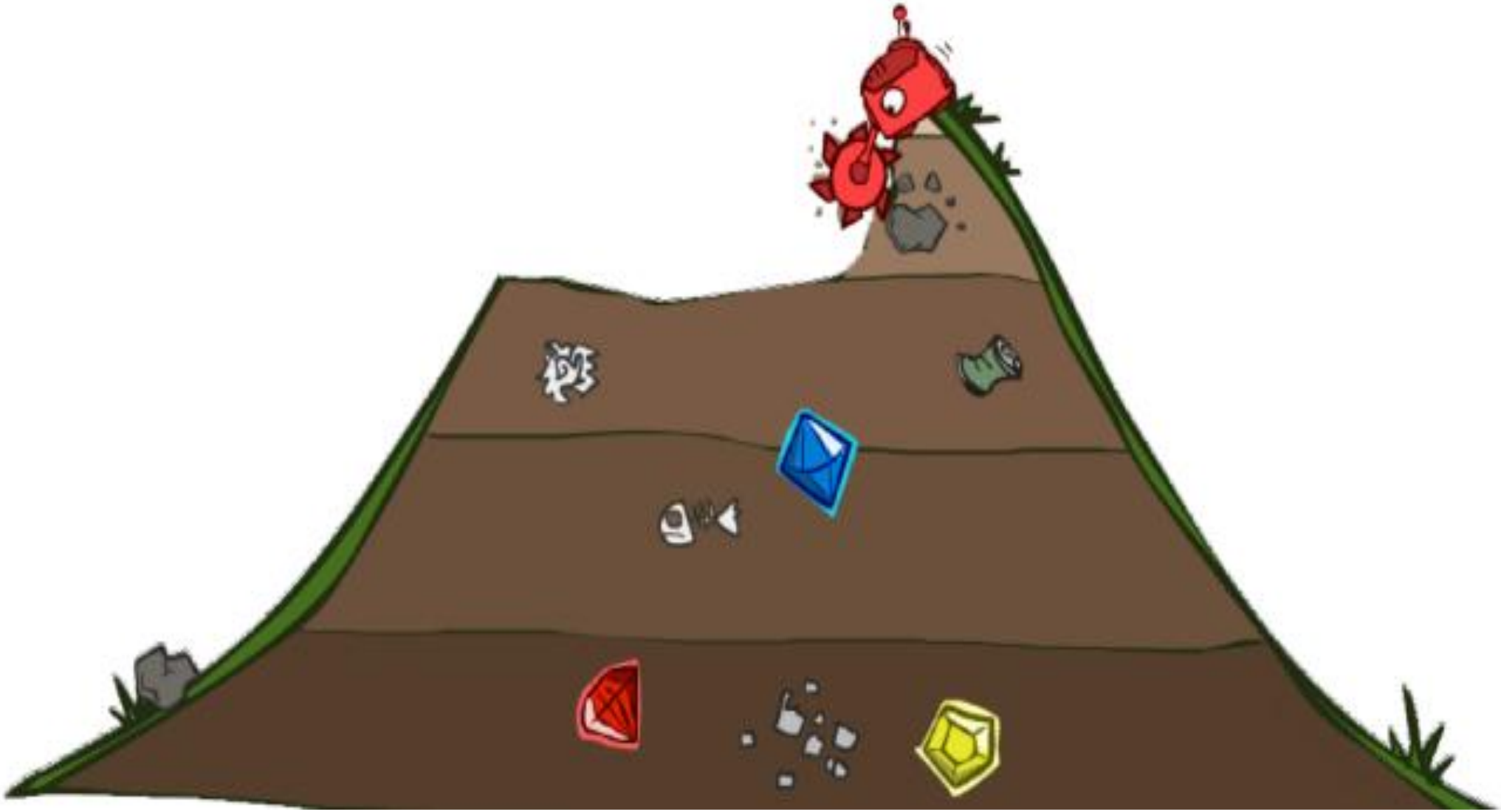
Consider the following case:

- forward and backward branching both *b*, uniform ..



Time ~ $b^{d/2} + b^{d/2} < b^{d}$

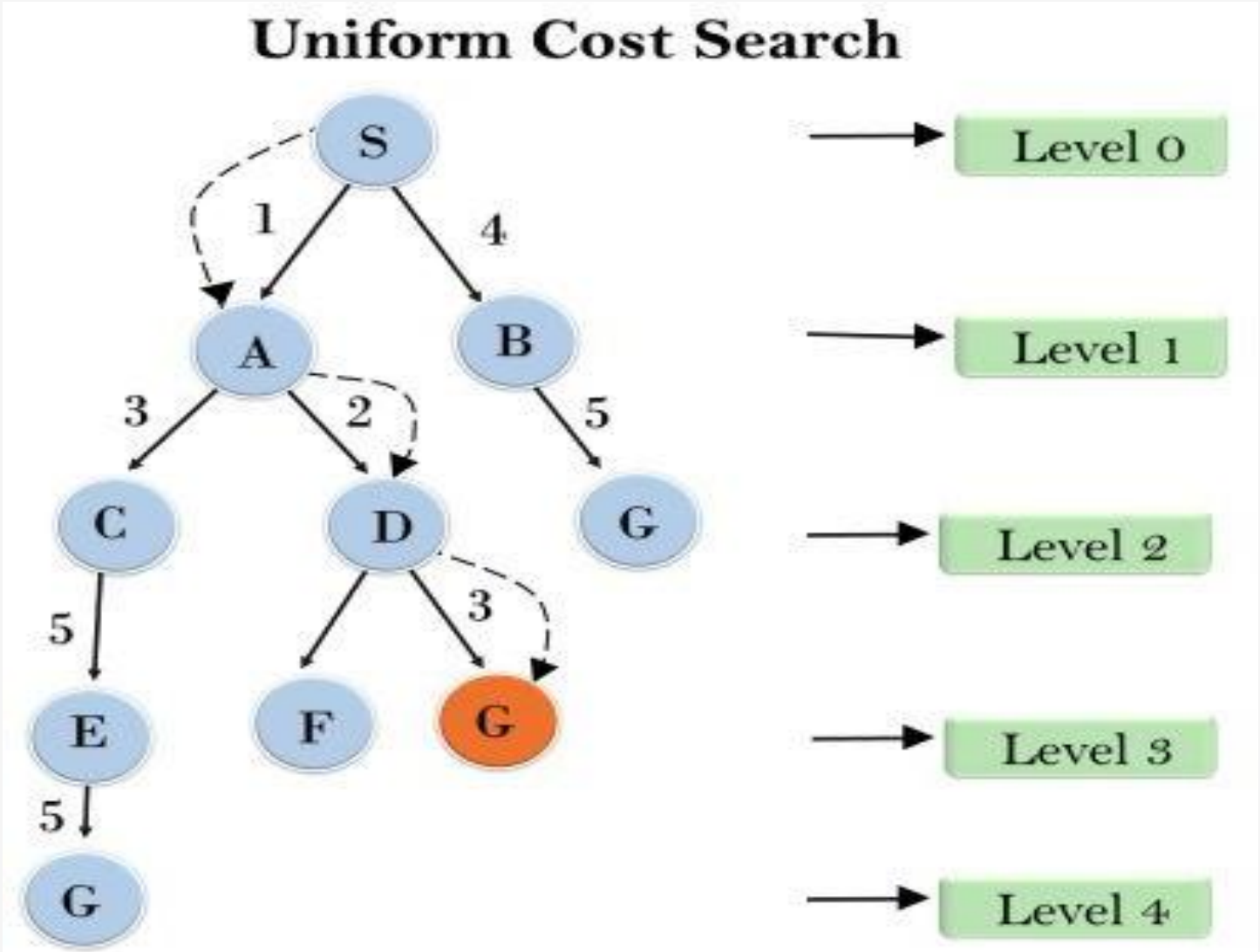
Uniform cost search



Uniform Cost Search

- Search algorithm used for traversing a weighted tree or graph
- Goal: to find path which has lowest cumulative cost to reach the goal
- Expands node according to their path cost from the root node.
- Priority queue is used to implement UCS
- It gives maximum priority to lowest cumulative cost.
- SImilar to BFS if path cost of all edge is same.

Diagram


Uniform Cost Search
S
A
B
C
D
G
E
F
G
G
1
4
3
2
5
5
3
5
Level 0
Level 1
Level 2
Level 3
Level 4

uniform-cost search

Uniform cost search modifies the breadth-first strategy by always expanding the lowest-cost node on the fringe (as measured by the path cost g(n)), rather than the lowest-depth node.

When certain conditions are met, the first solution that is found is guaranteed to be the cheapest solution because if there were a cheaper path that was a solution, it would have been expanded earlier and thus would have been found first.

Uniform Cost Search (Branch&Bound)

QueueingFn is SortByCostSoFar

Cost from root to current node n is g(n)

- Add operator costs along path

First goal found is least-cost solution

Space & time can be exponential because large subtrees with

inexpensive steps may be explored before useful paths with costly steps

If costs are equal, time and space are $O(b^d)$

- Otherwise, complexity related to cost of optimal solution

Uniform Cost Search

Advantages:

- Optimal because at every state path with the least cost is chosen

Disadvantages:

- Does not cares about the number of steps involved in searching and only concerned about path cost. Due to which this algorithm may be stuck in an infinite loop.

uniform-cost search

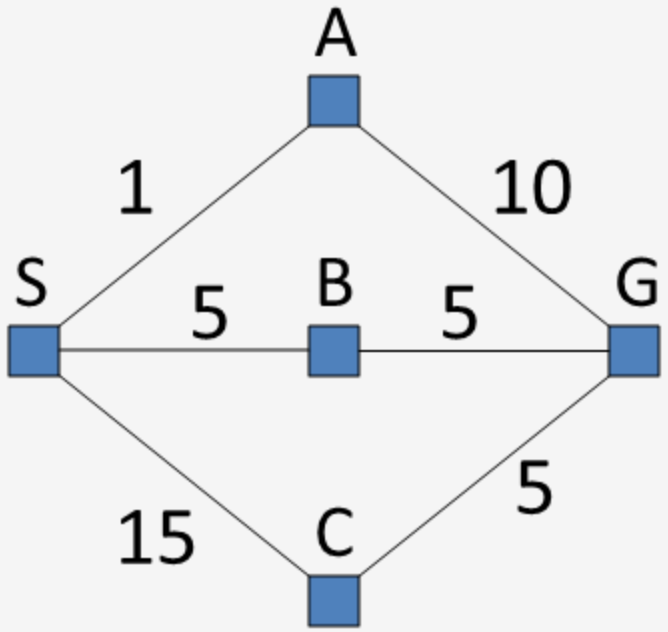
- Each step has some cost $\geq \varepsilon > 0$.
- The cost of the path to each fringe node N is

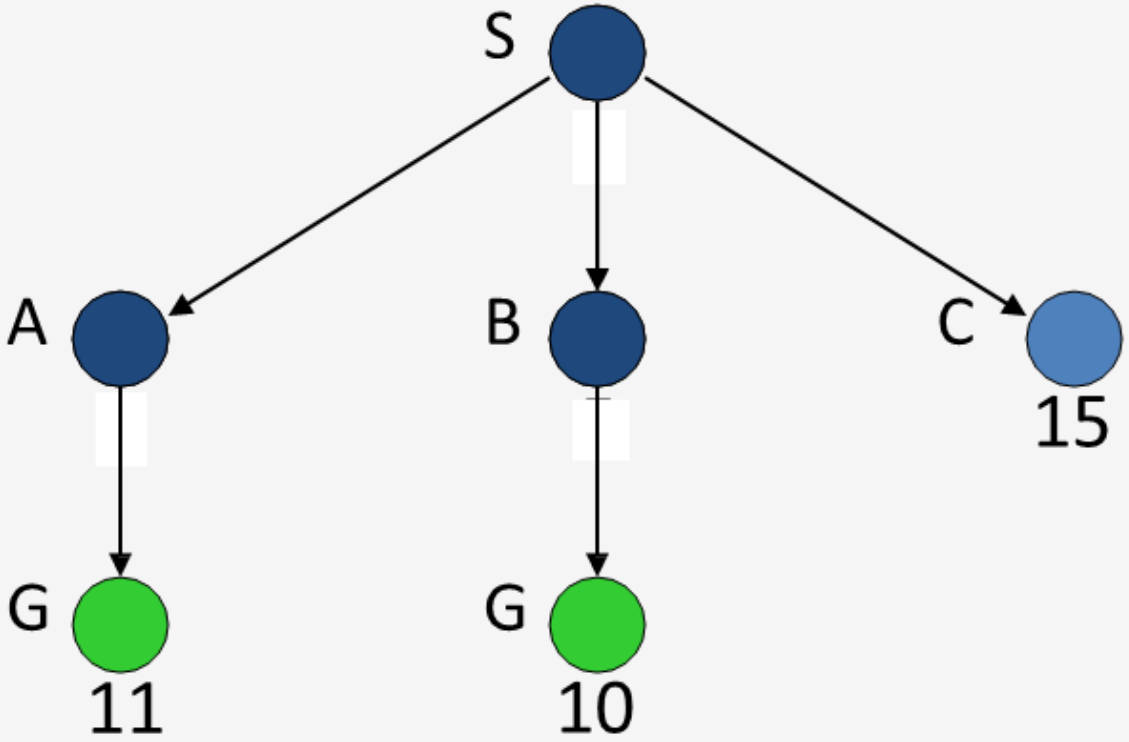
$$g(N) = \Sigma \text{ costs of all steps.}$$

- The goal is to generate a solution path of minimal cost.
- The queue FRINGE is sorted in increasing cost.

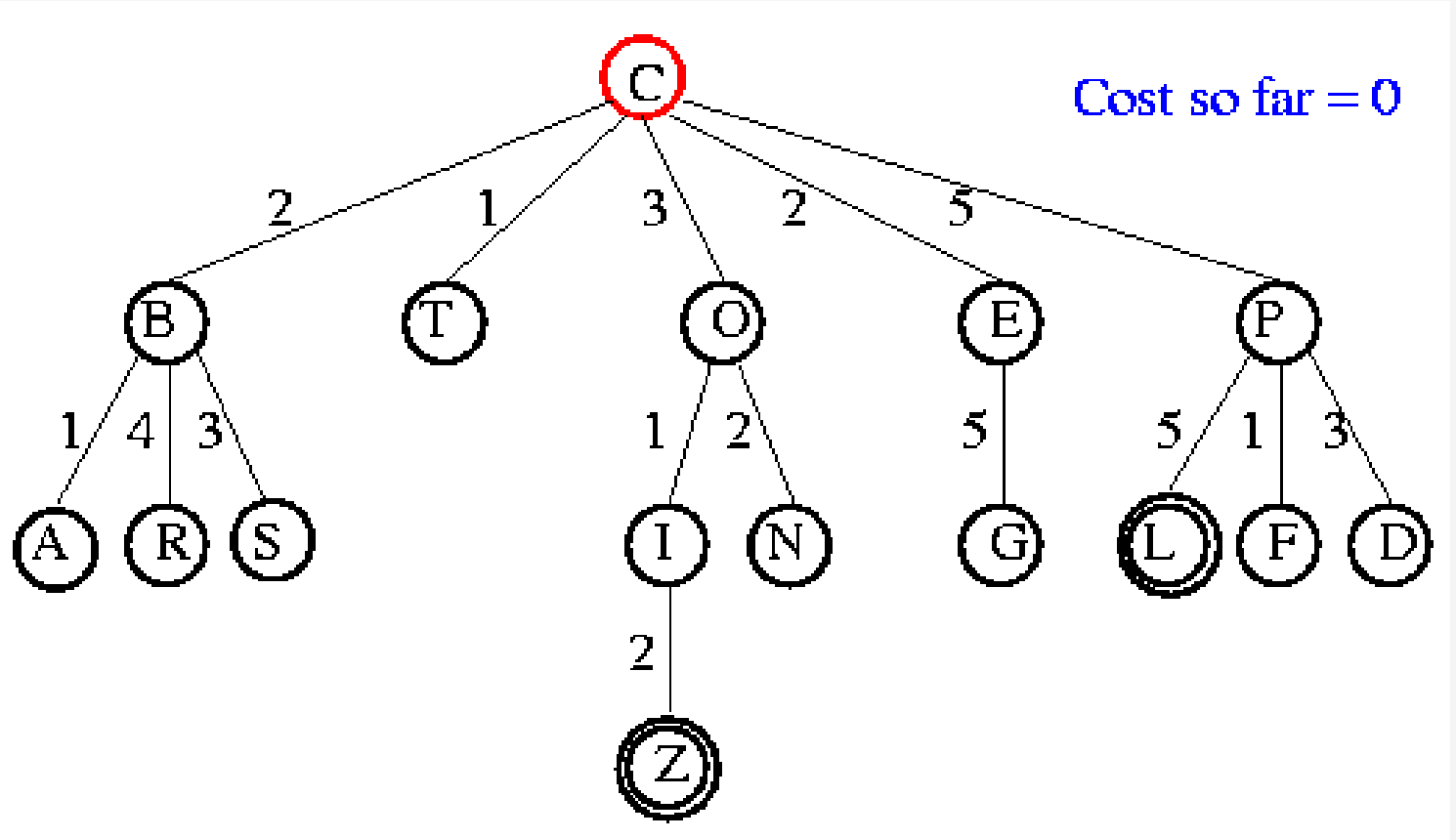
Uniform Cost Search

Nodes List = (S0)
Nodes List = (A1, B5, C15)
Nodes List = (B5, G11, C15)
Nodes List = (G10, G11, C15)



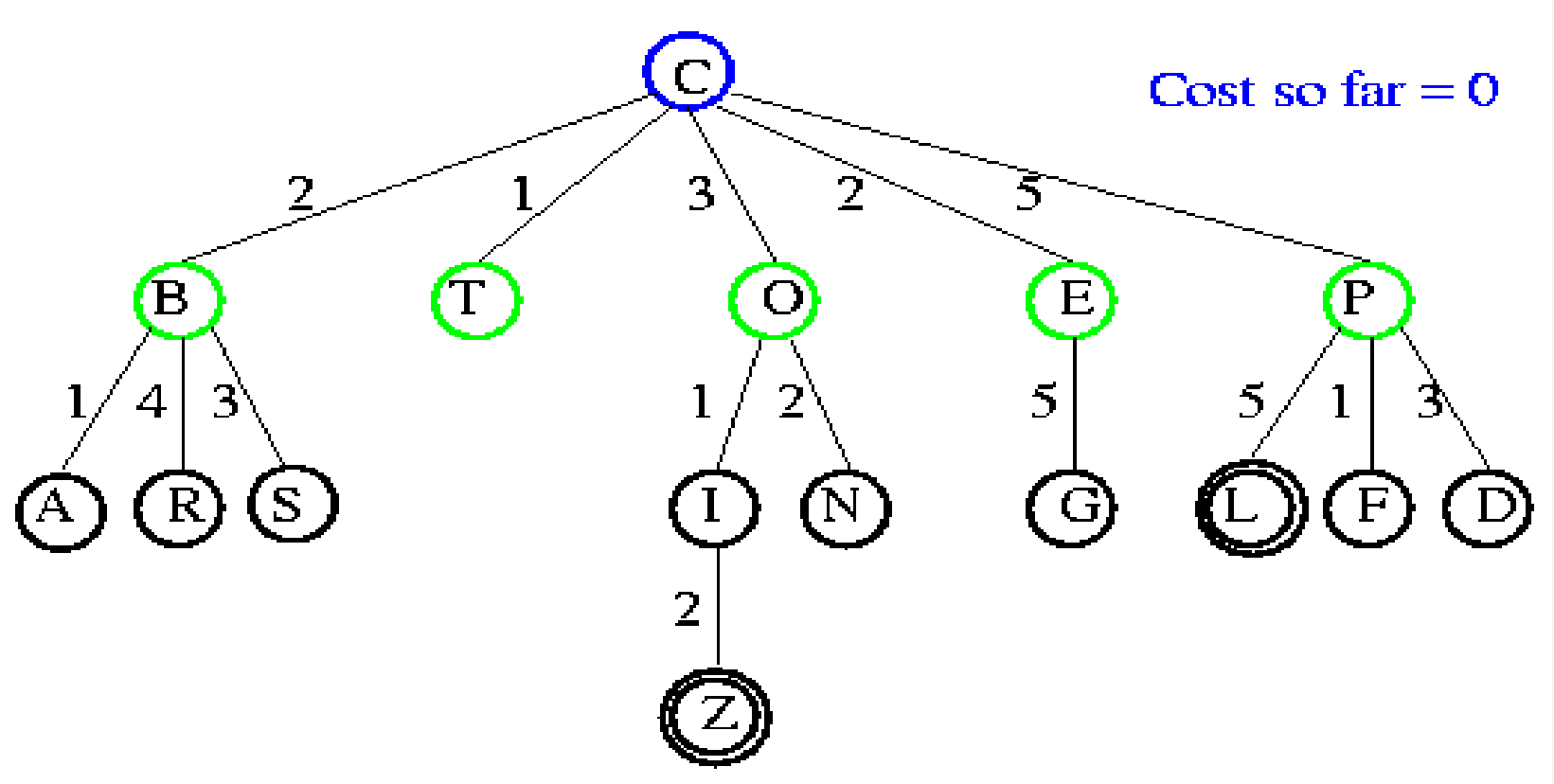


Example



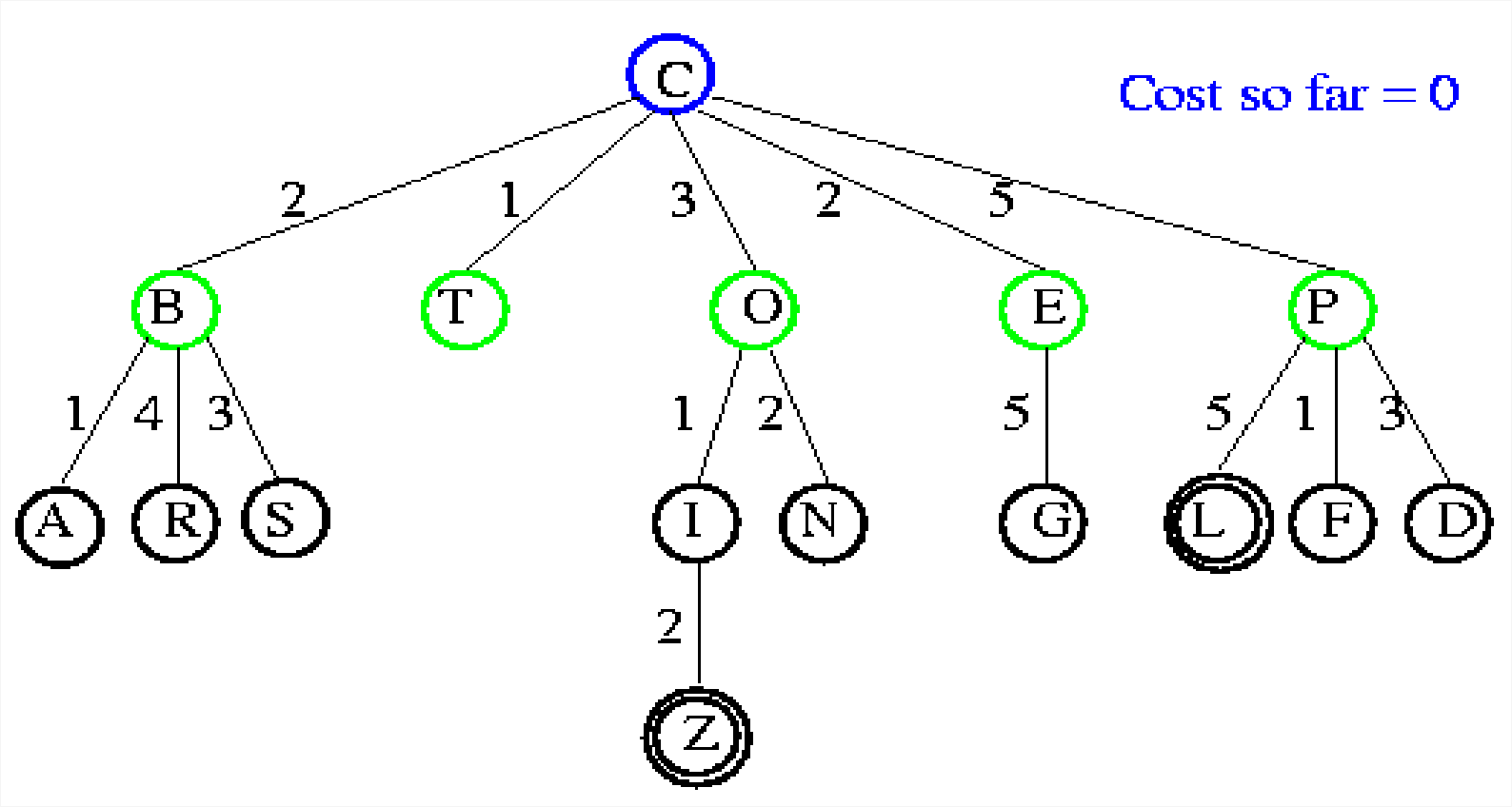
Frontier list: C

Example



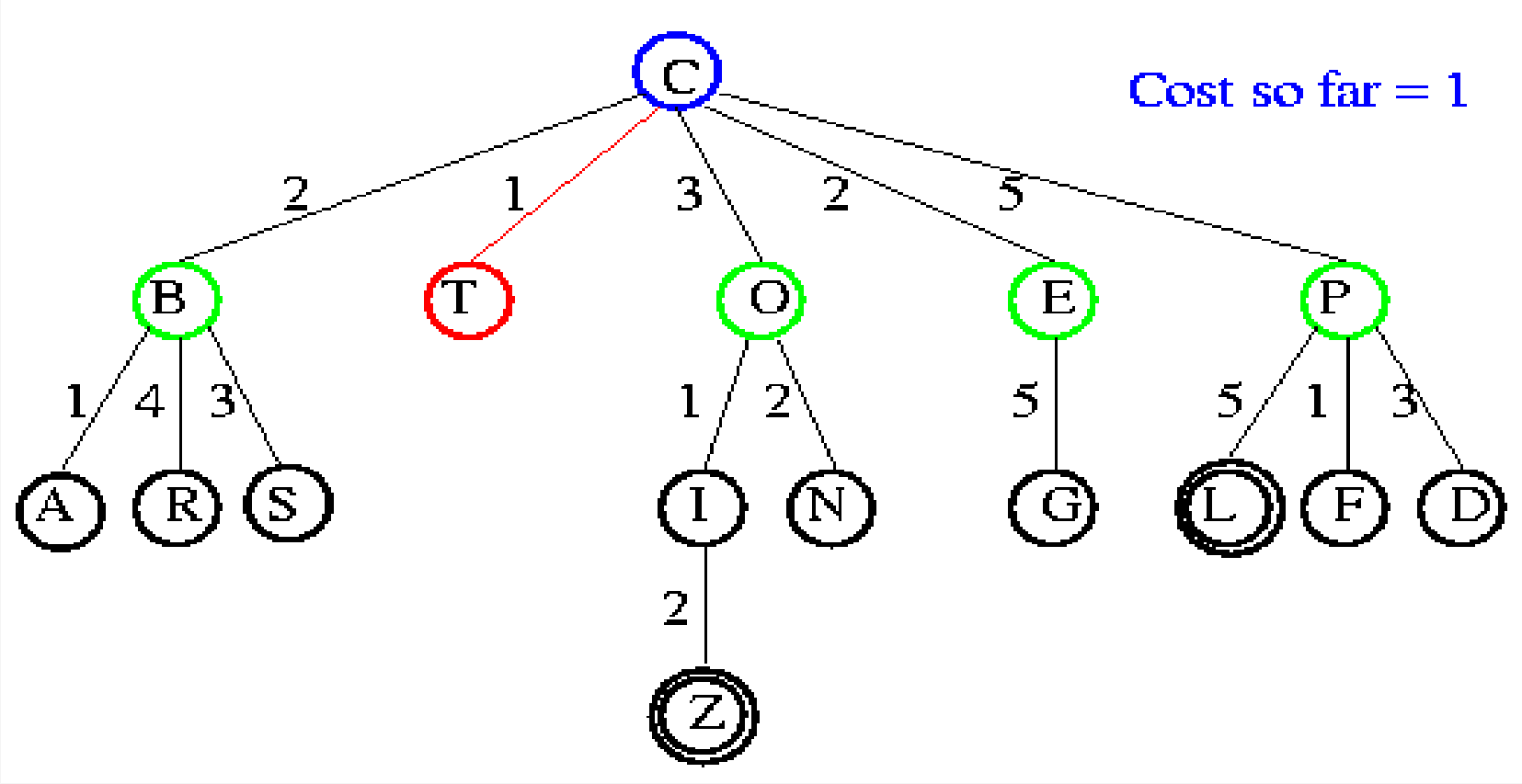
Frontier list: B(2) T(1) O(3) E(2) P(5)

Example



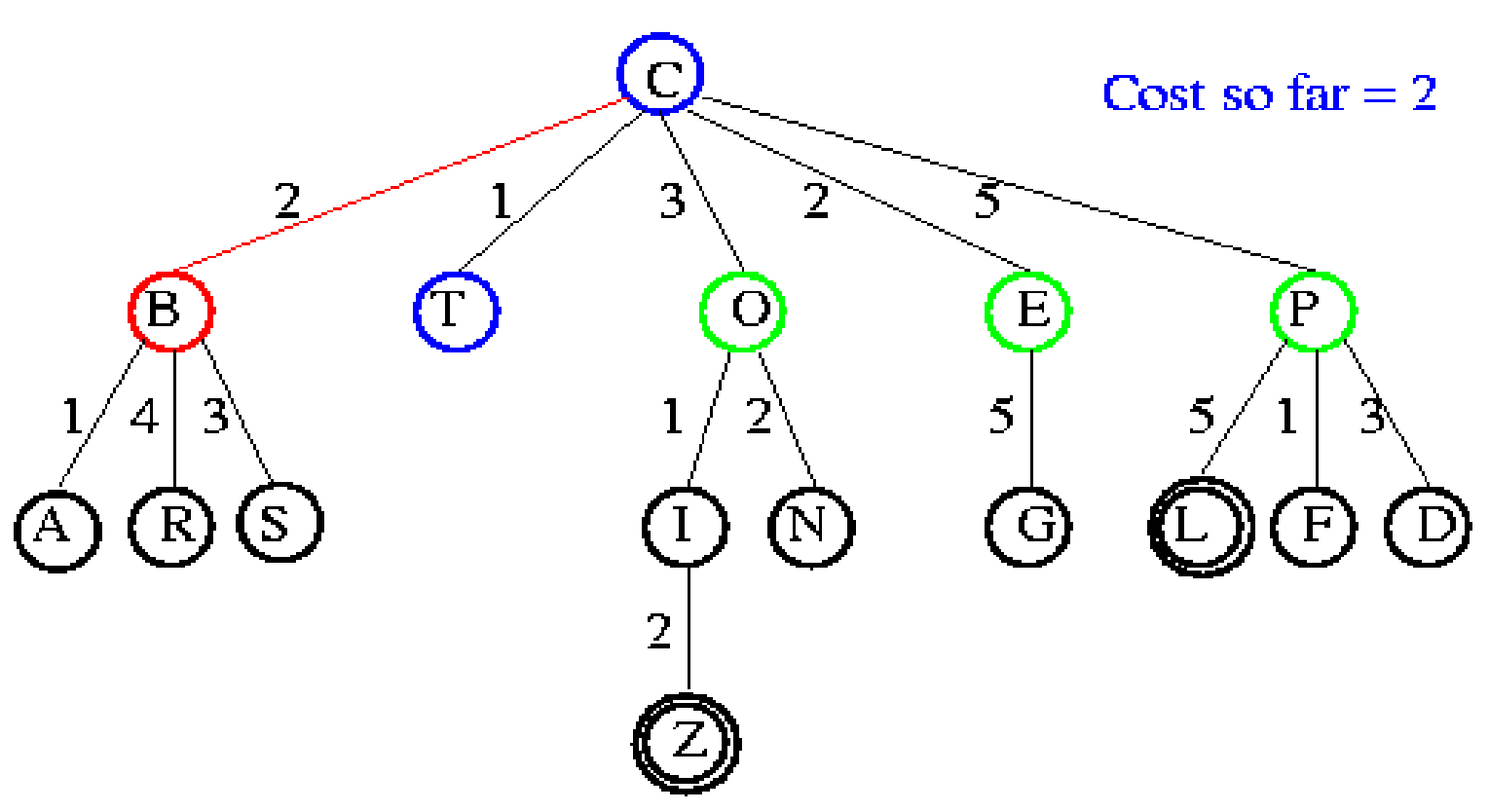
Frontier list: T(1) B(2) E(2) O(3) P(5)

Example



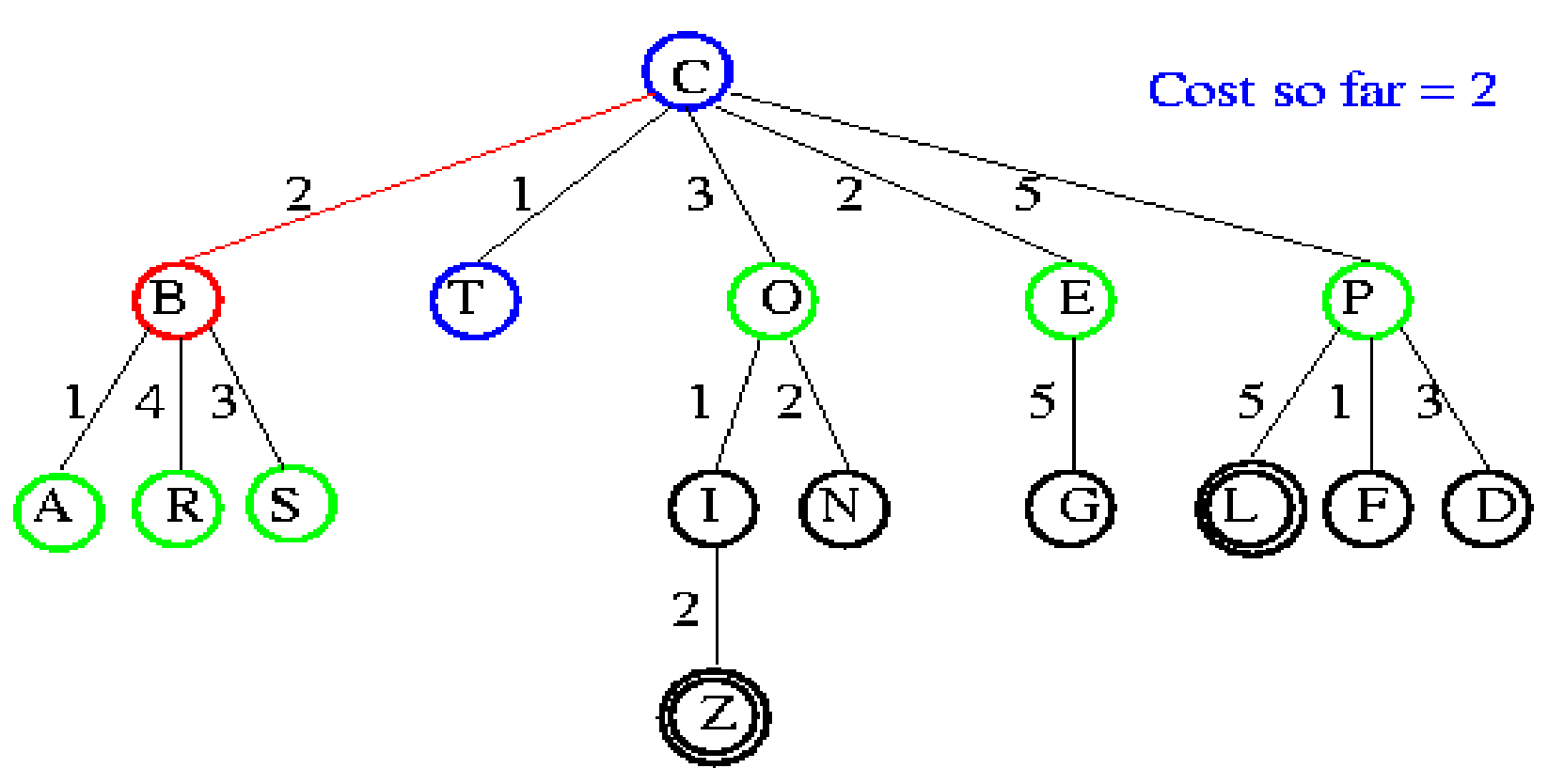
Frontier list: B(2) E(2) O(3) P(5)

Example



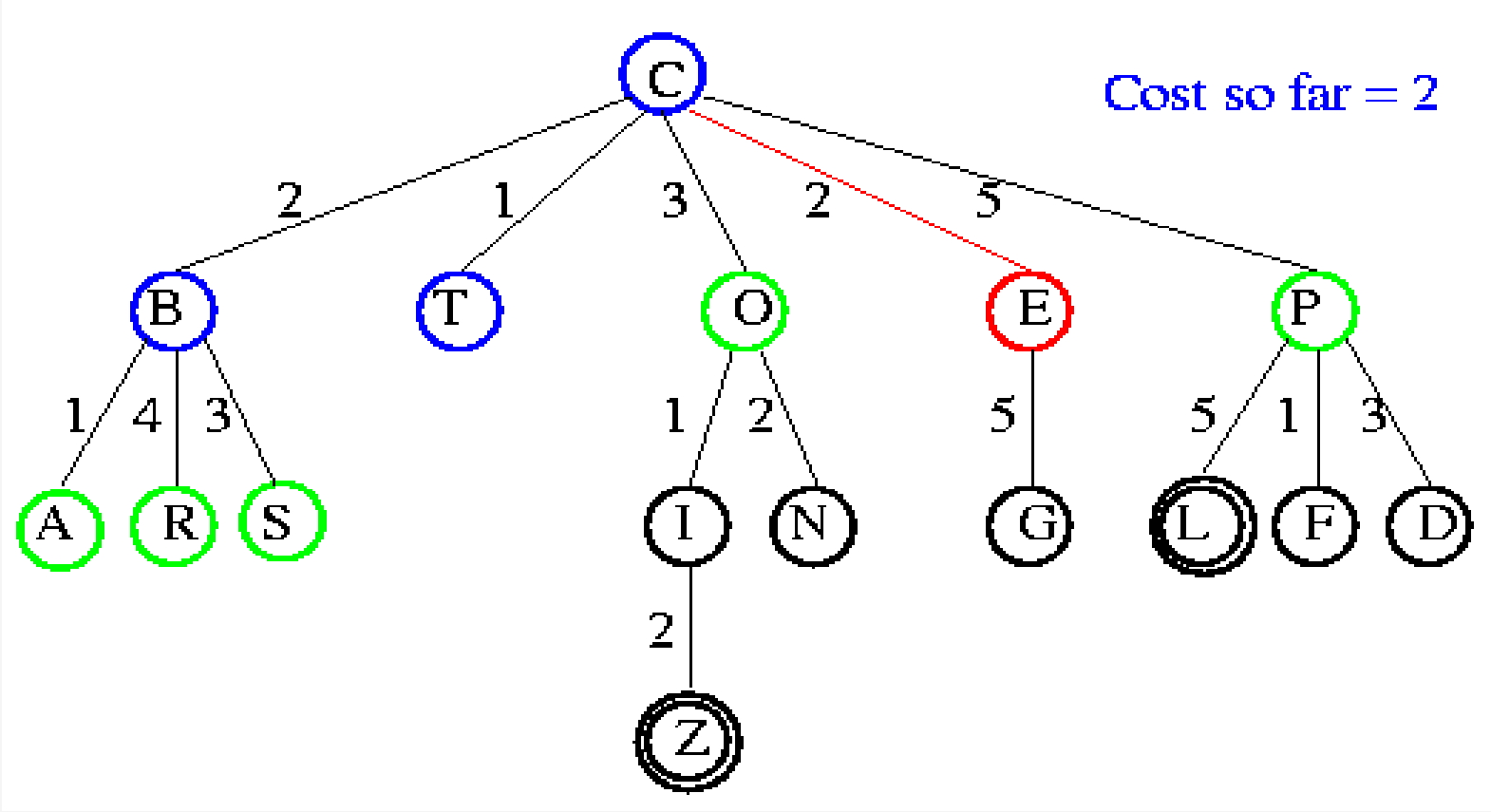
Frontier list: E(2) O(3) P(5)

Example



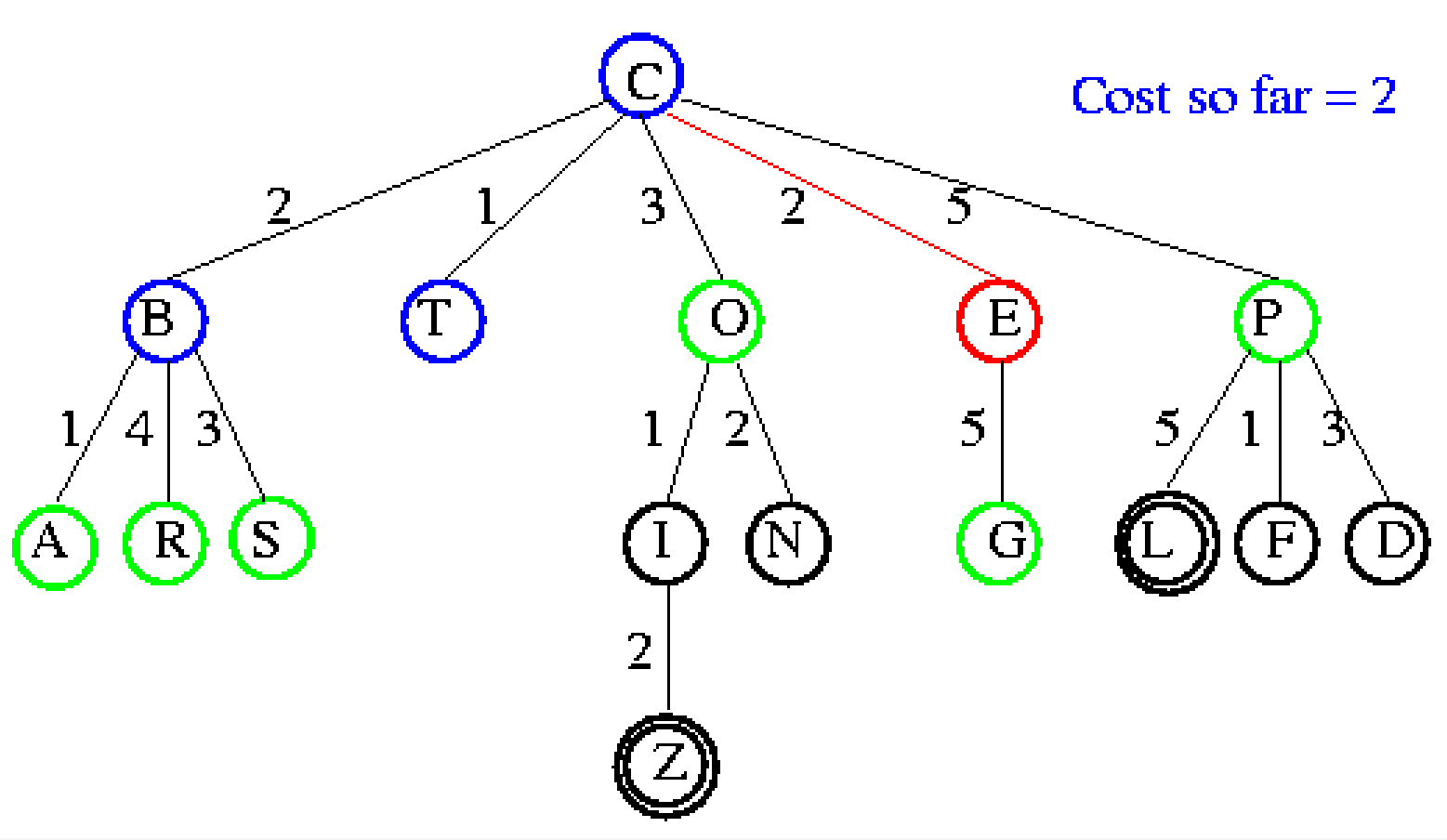
Frontier list: E(2) A(3) O(3) P(5) S(5) R(6)

Example



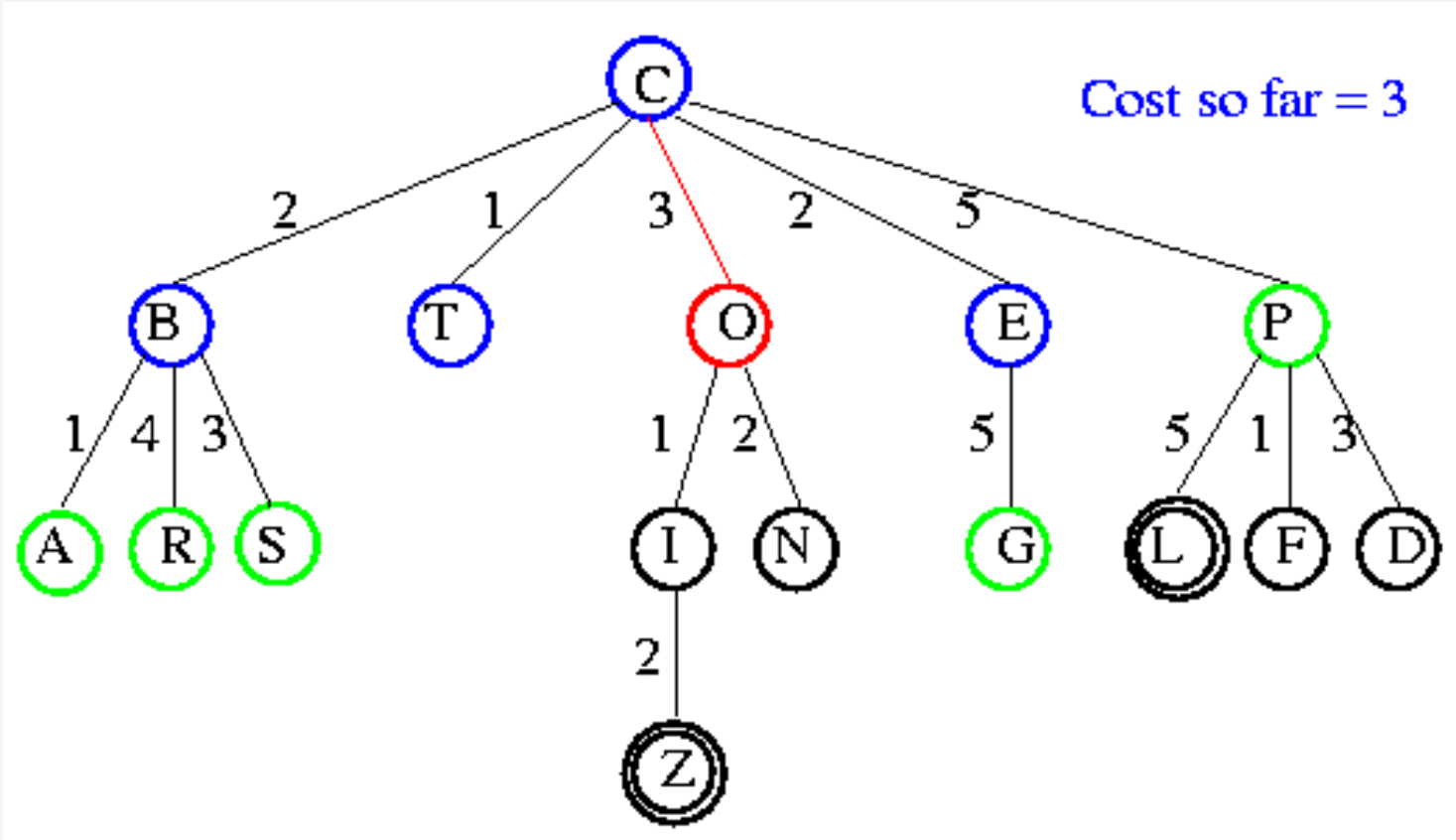
Frontier list: A(3) O(3) P(5) S(5) R(6)

Example



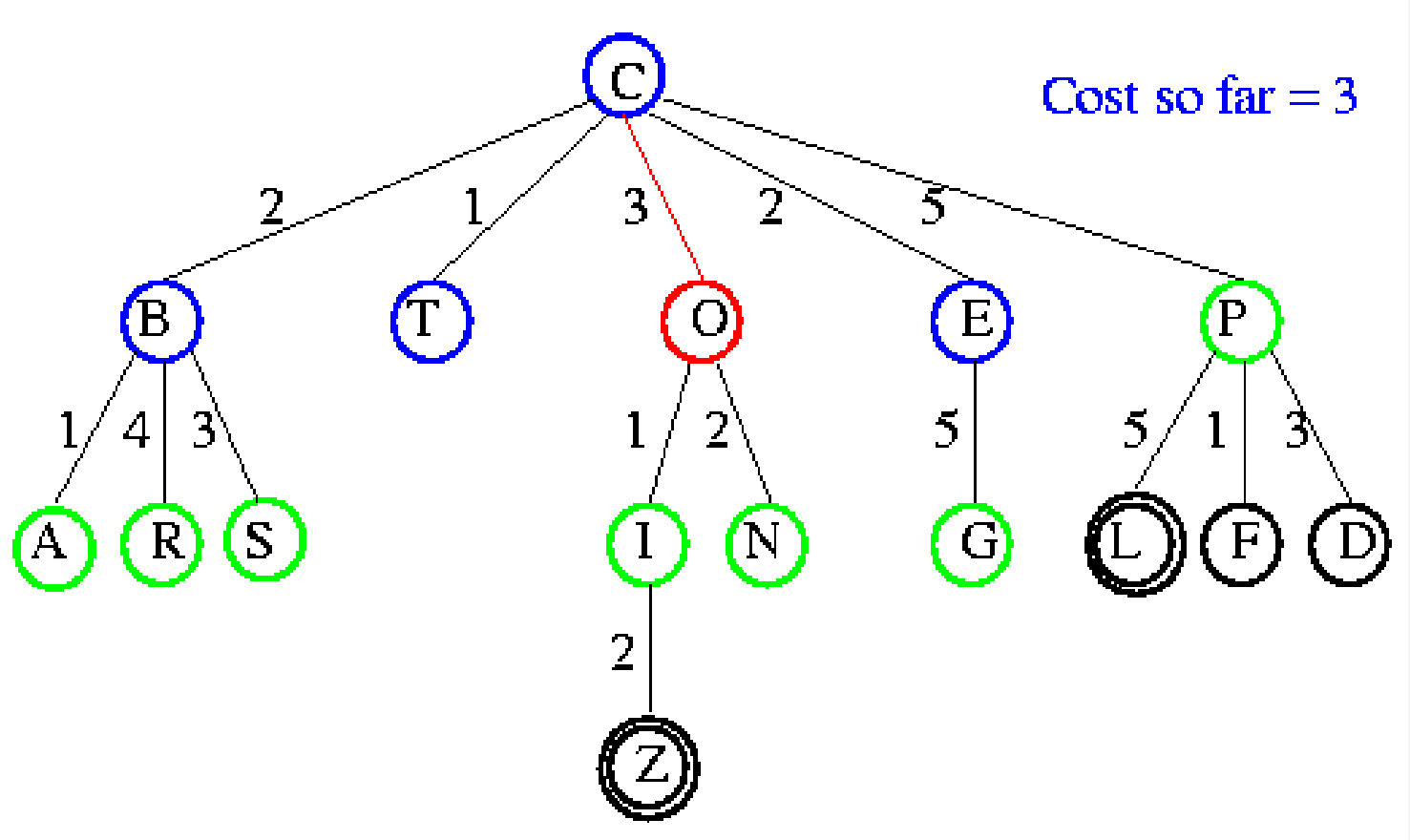
Frontier list: A(3) O(3) P(5) S(5) R(6) G(7)

Example



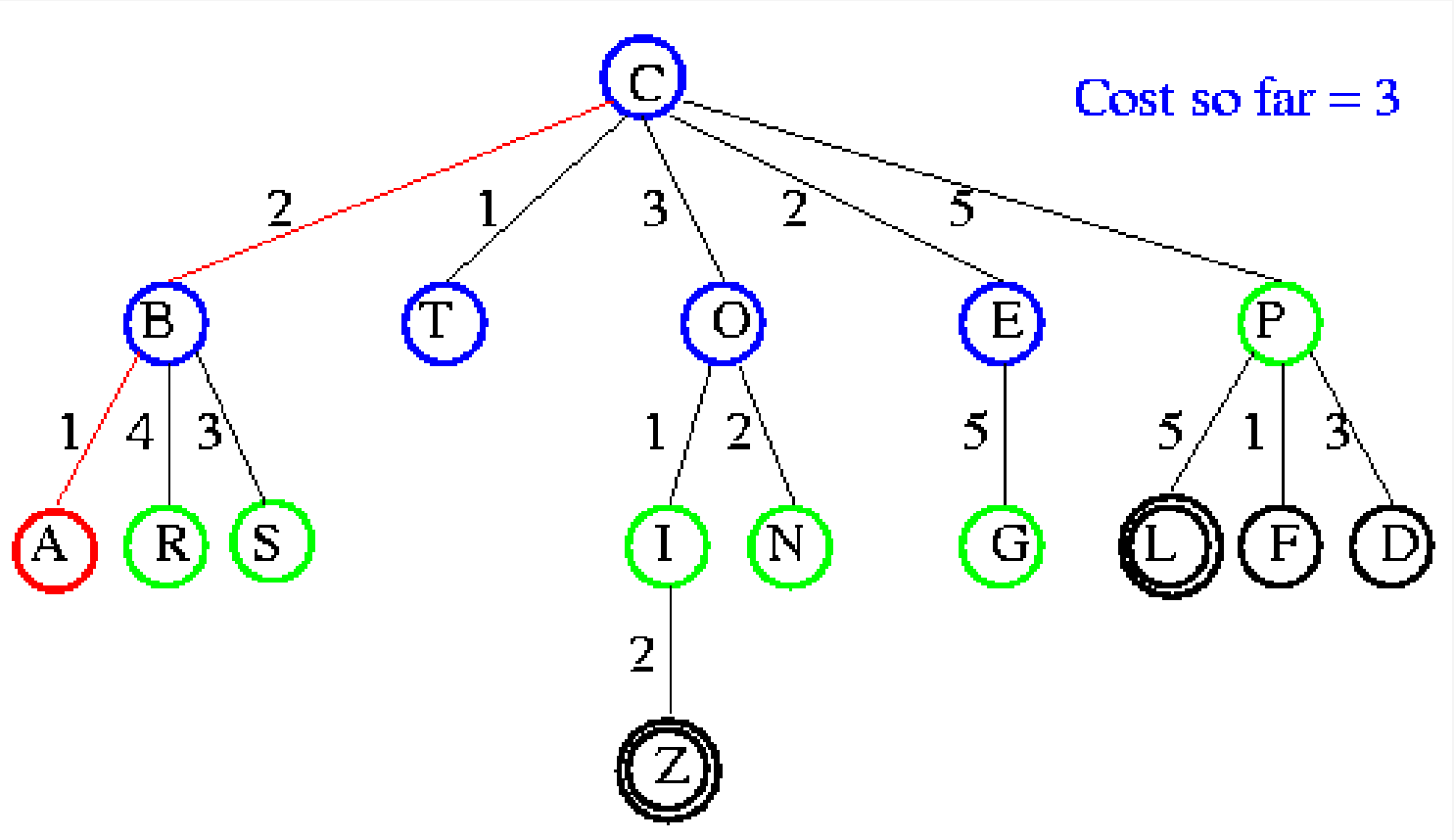
Frontier list: O(3) P(5) S(5) R(6) G(7)

Example



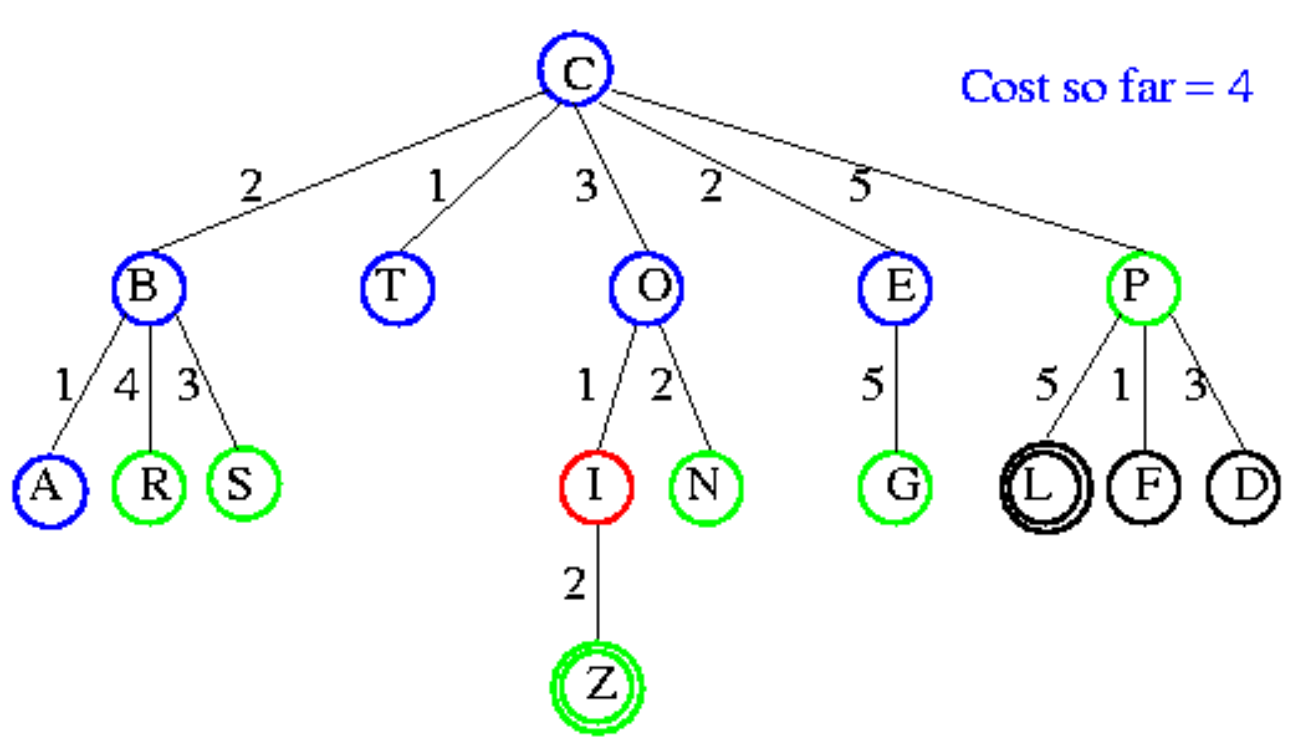
Frontier list: I(4) N(5) P(5) S(5) R(6) G(7)

Example



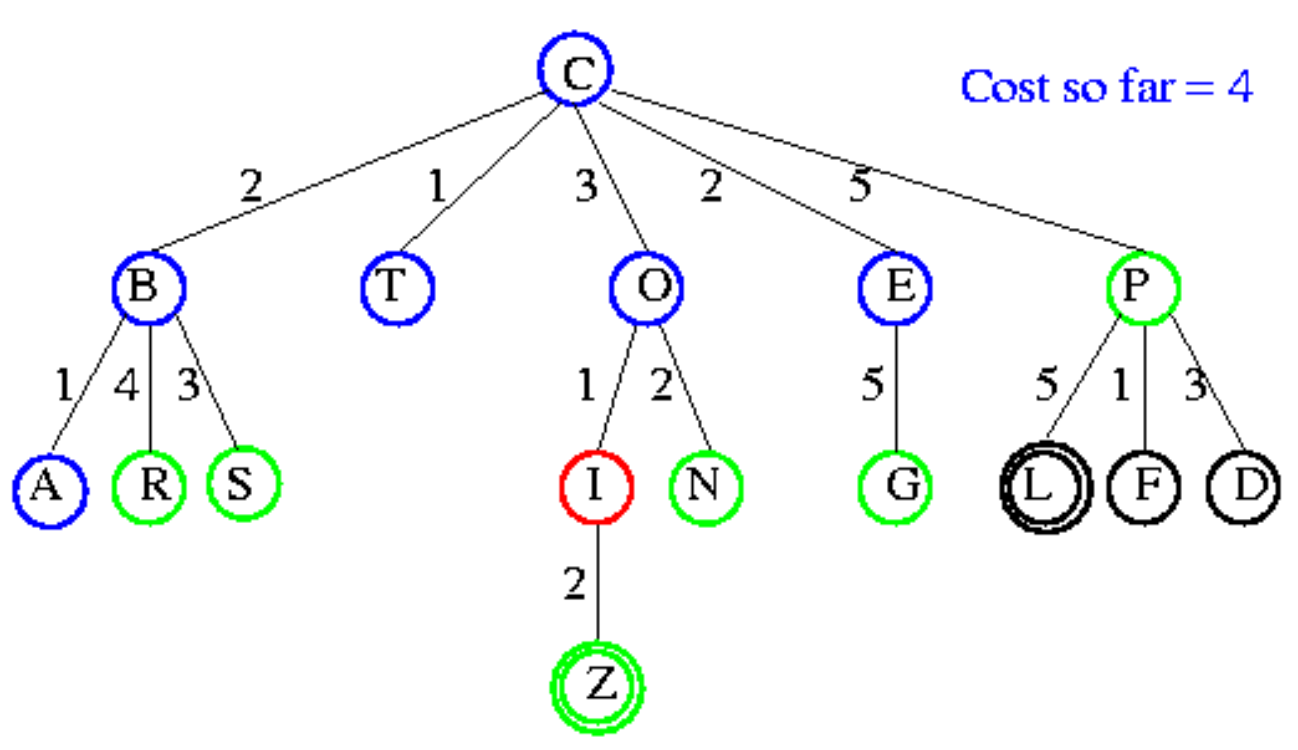
Frontier list: N(5) P(5) S(5) R(6) G(7) Z(6)

Example



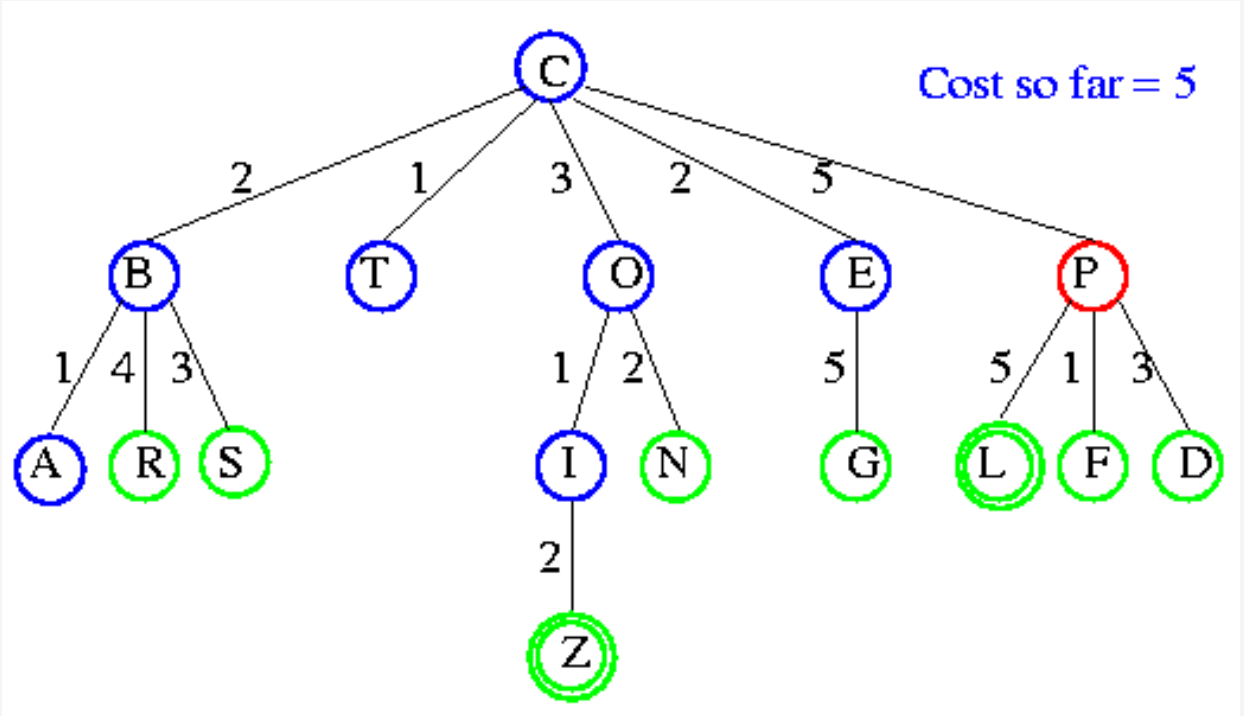
Frontier list: N(5) P(5) S(5) R(6) Z(6) G(7)

Example



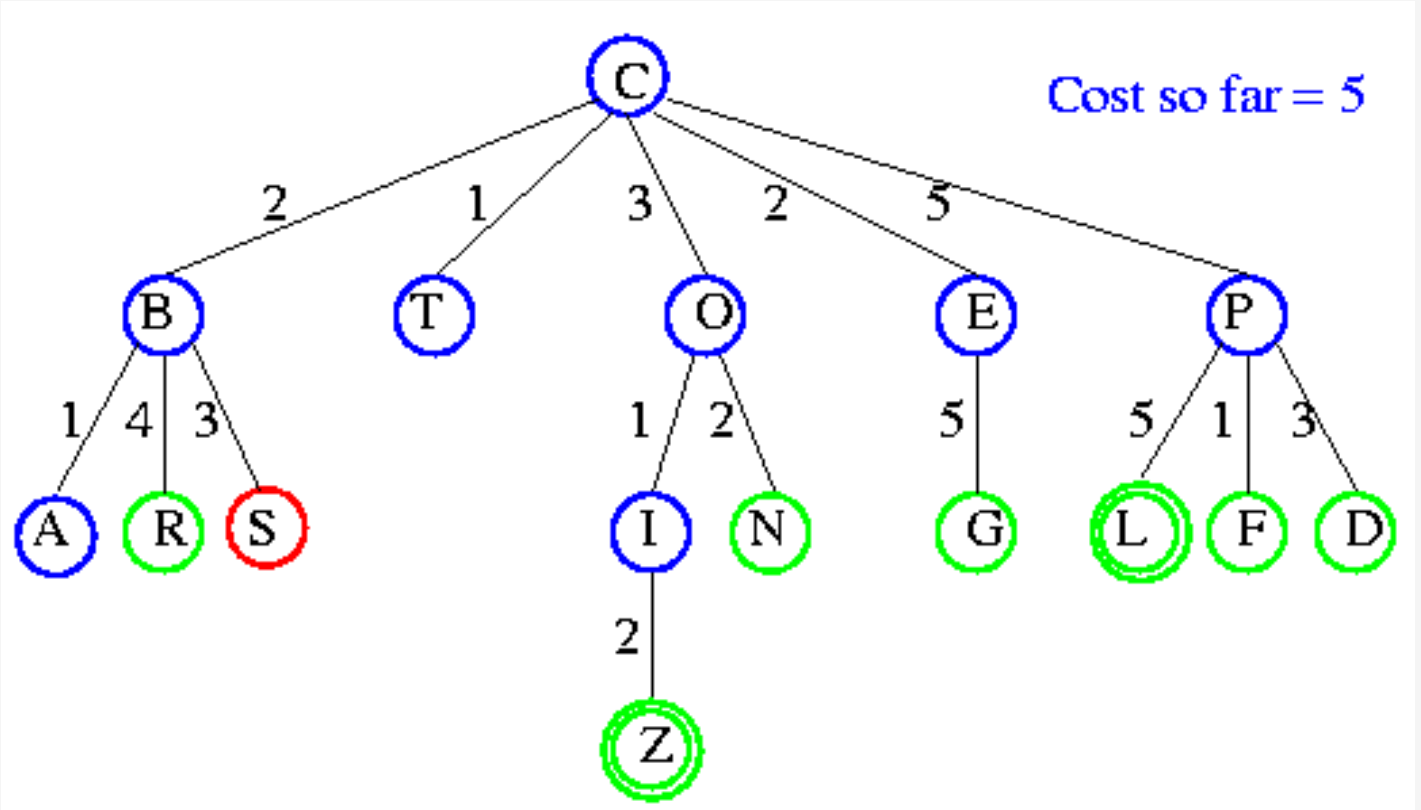
Frontier list: P(5) S(5) R(6) Z(6) G(7)

Example



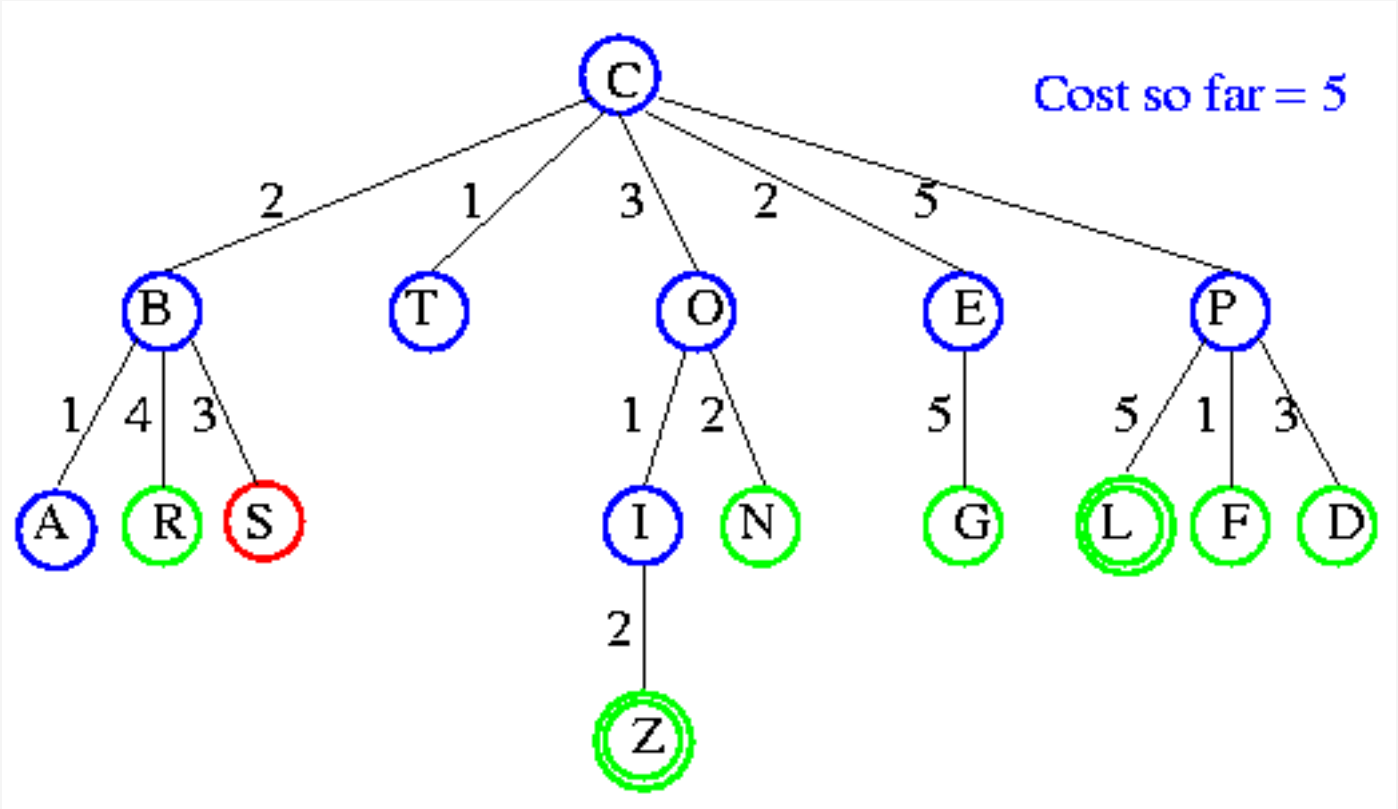
Frontier list: S(5) F(6) R(6) Z(6) G(7) D(8) L(10)

Example



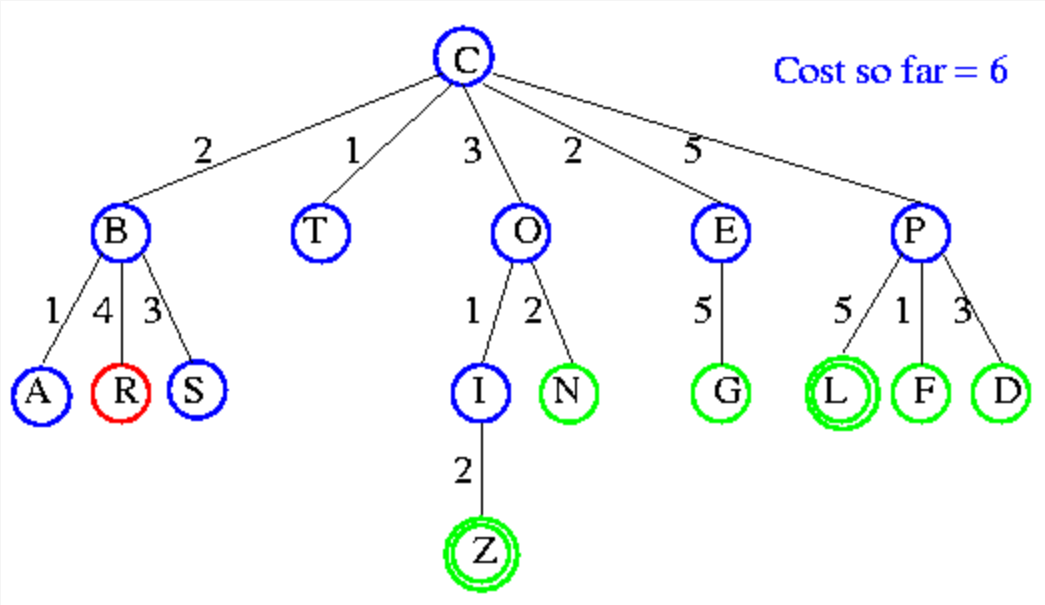
Frontier list: F(6) R(6) Z(6) G(7) D(8) L(10)

Example



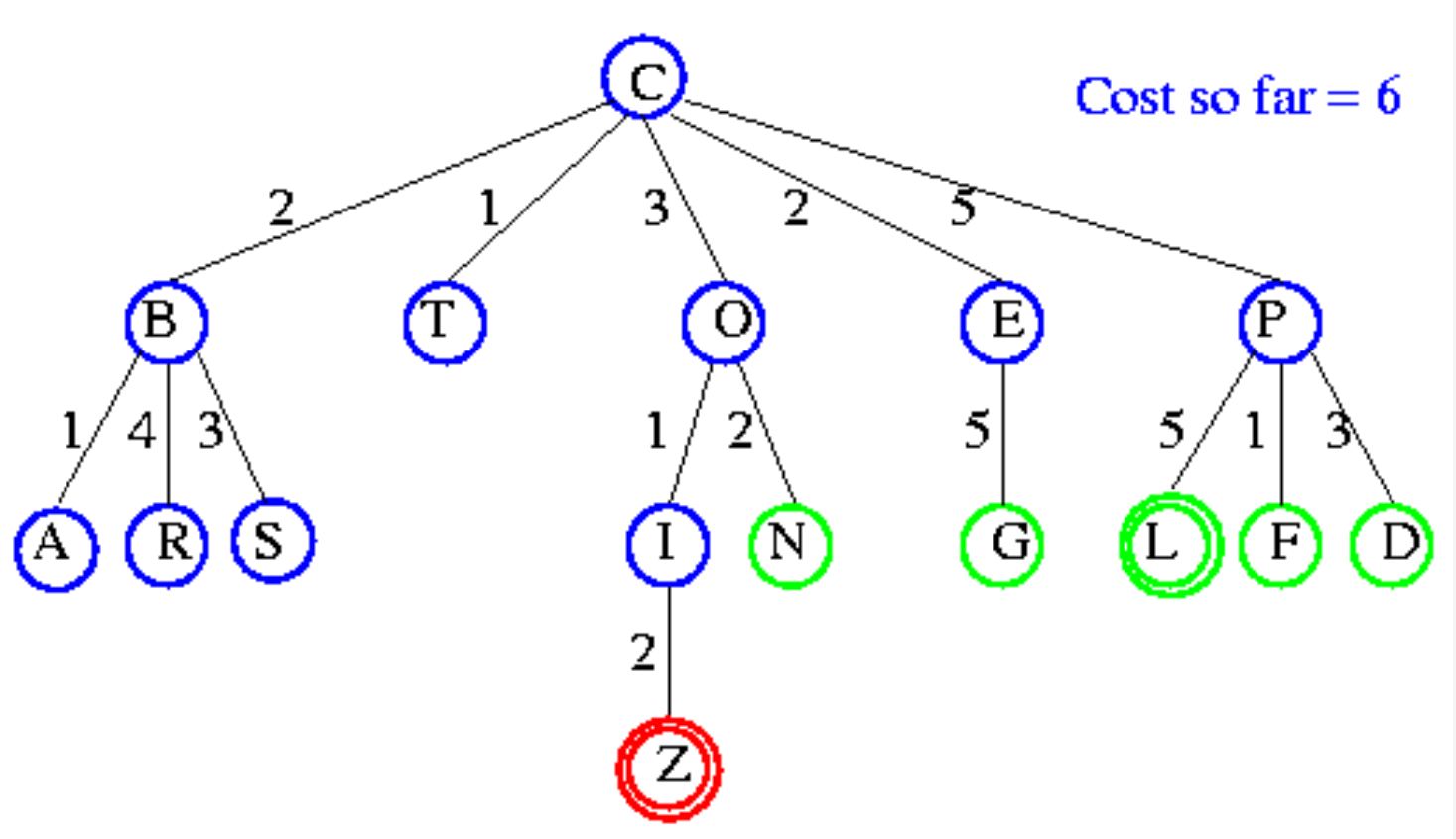
Frontier list: R(6) Z(6) G(7) D(8) L(10)

Example



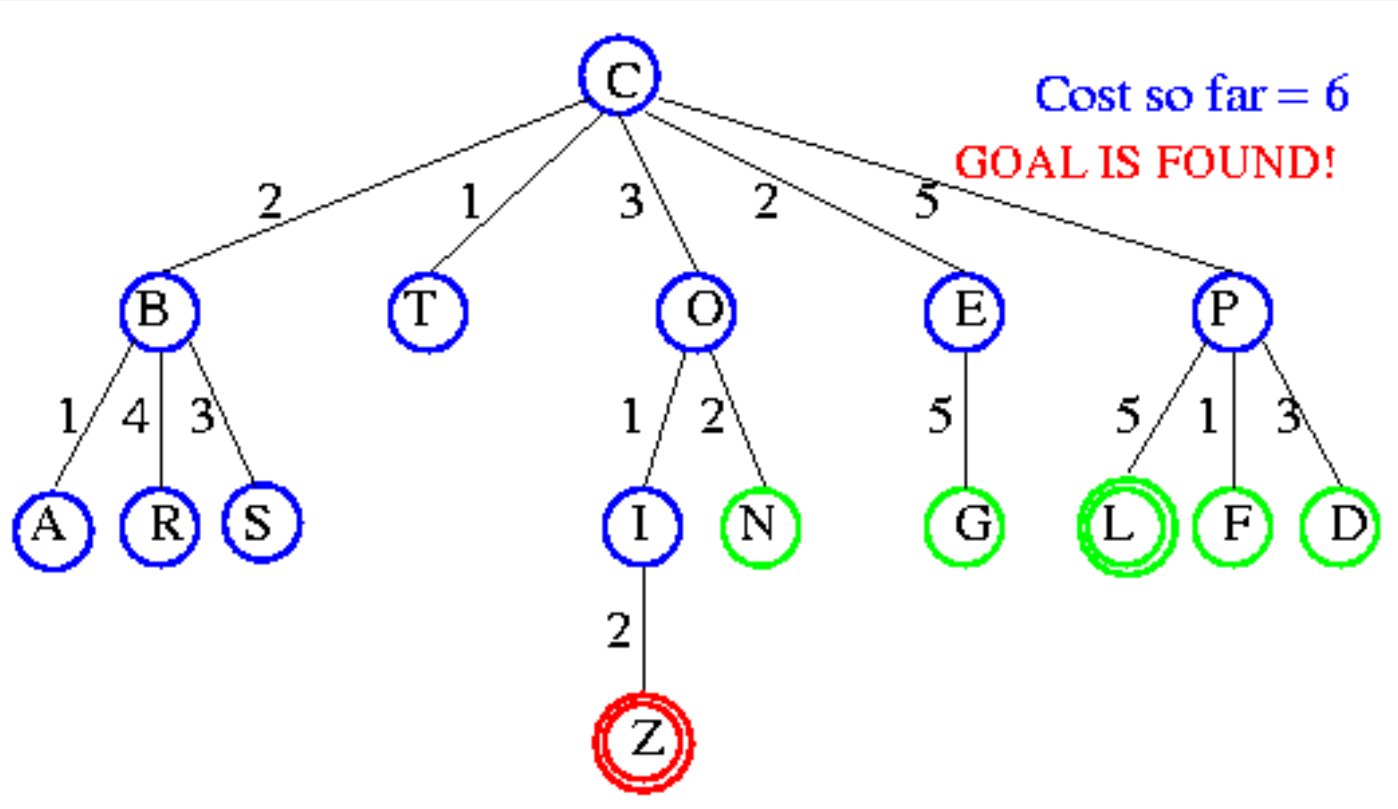
Frontier list: Z(6) G(7) D(8) L(10)

Example



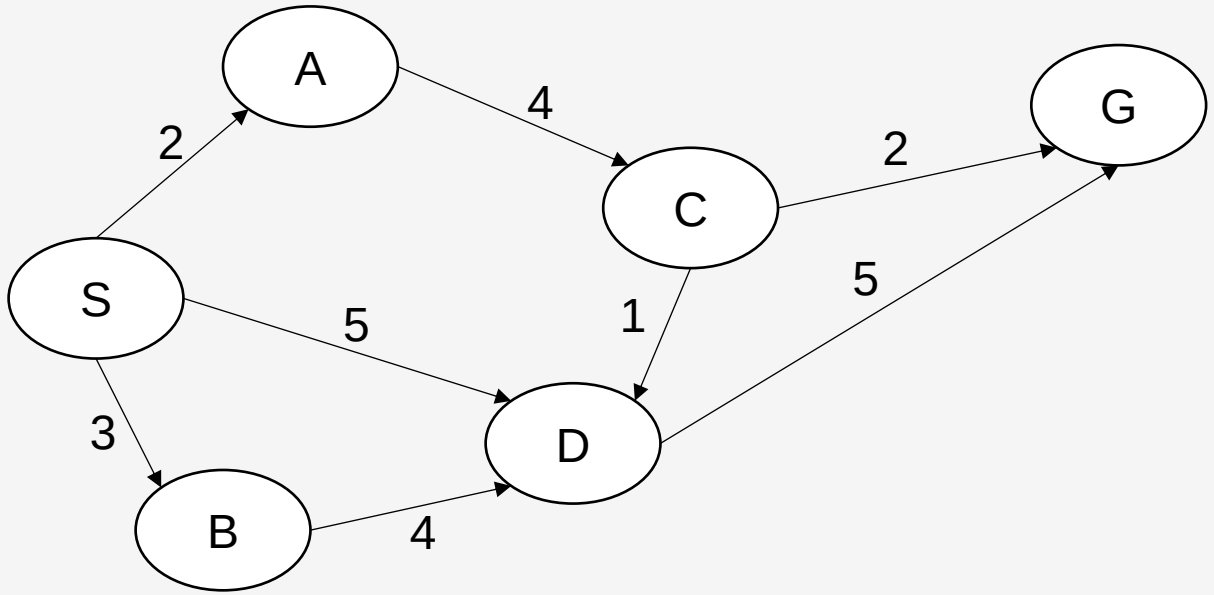
Frontier list: Z(6) G(7) D(8) L(10)

Example



Path: C, O, I, Z

Uniform cost search “path cost”



Path: S -> A -> C -> G
Cost = 2+4+2= 8

Current	Fringe
-	S (0)
S (0)	A B D (2) (3) (5)
A (2)	B D C (3) (5) (6)
B (3)	D C D (5) (6) (7)
D (5)	C D G (6) (7) (10)
C (6)	D D G G (7) (7) (8) (10)
G (8)	

Analysis

- **Time Complexity**: Let **C*** be the cost of the optimal solution, and **ε** be the each step to get closer to the goal node. Then the number of steps is **C*/ε+1**. Here +1 is taken as we start from state **0** and end to **C*/ε**. Hence **O(b^(1+[C*/ε]))**
- **Space complexity:** The same logic is for space complexity. So the worst case space complexity is **O(b^(1+[C*/ε]))**
- **Optimal:** optimal as it selects the path with lowest path cost
- **Completeness:** it is complete, if there is solution, it will find it.

Comparison of Search Techniques

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]

Figure 3.21 Evaluation of tree-search strategies. 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 $\geq \epsilon$ for positive ϵ; [c] optimal if step costs are all identical; [d] if both directions use breadth-first search.

Thank you

Sara Sweidan
PhD, Artificial Intelligence
Assistant Professor
Faculty of Computers & AI
Benha University, Egypt
Sweidan_ds@fci.bu.edu.eg