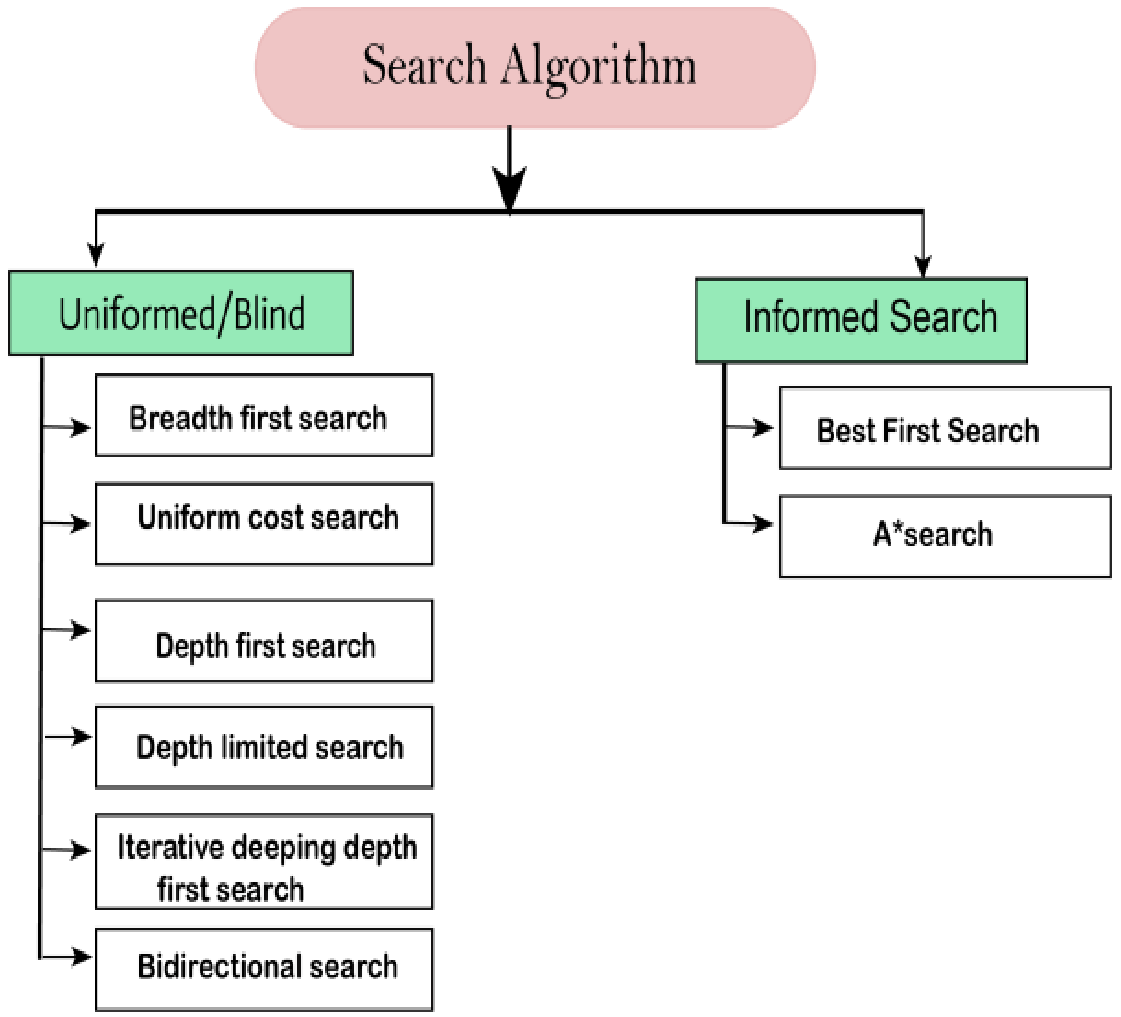
Chapter Three

Solving Problems by Searching

Basic Topics of Chapter 3

- **Solving Problems by Searching:**
  - What is searching
  - Problem Solving Agents
  - Problem Types
  - Problem Formulation.



Solving Problems by Searching

- **Solving problem-** is commonly known as the method to reach the *desired goal* *or* finding a solution to a given situation.
- **Problem-solving** refers to AI techniques, including various techniques such as *forming efficient algorithms, heuristics, and performing root cause analysis to find desirable solutions.*
- Therefore, some of the most prevalent problems that AI has resolved are the following:
  - *8 Puzzle, Vacuum World, Chess, N-Queen problem,*
  - *Travelling Salesman Problem*
  - *Water Jug problem, Tower of Hanoi etc.*

What is searching?

- **Searching means finding**
  - In general finding a *solution*
  - In a website finding *the answer of a question*
  - In a city finding *the correct path*
  - In chess finding *the next best step to move*
- **In terms of AI**
  - ***Search*** is the process of considering various possible sequences of operators applied to the ***initial state*** and ***finding out a sequence*** which culminates in a ***goal state***

Problem-solving agents

- **Problem** is really a collection of information that the agent will use to decide what to do.
- **Problem solving agents** decide what to do by finding sequences of actions that lead to desirable states.
  - **Problem –solving agents** are one kind global based agent
  - *It uses **atomic** representations that is state of the world are considered as wholes, with no internal structure visible to the problem solving.*
- In **AI** mostly used these ***search strategies or algorithms*** to solve a specific problem and provide the best result.

Problem-solving agents(1)

- The basic algorithm for ***problem-solving agents*** consists of 3 phases:
  - ***Formulate*** the problem,
  - ***Search*** for a solution and
  - ***Execute*** the solution.
- **Note: In solving problems**, it is important to understand the concept of a *state space*.
- *The aim of the problem-solving agent is to perform a sequence of actions that change the environment so that it ends up in one of the goal states.*

Problem Formulation

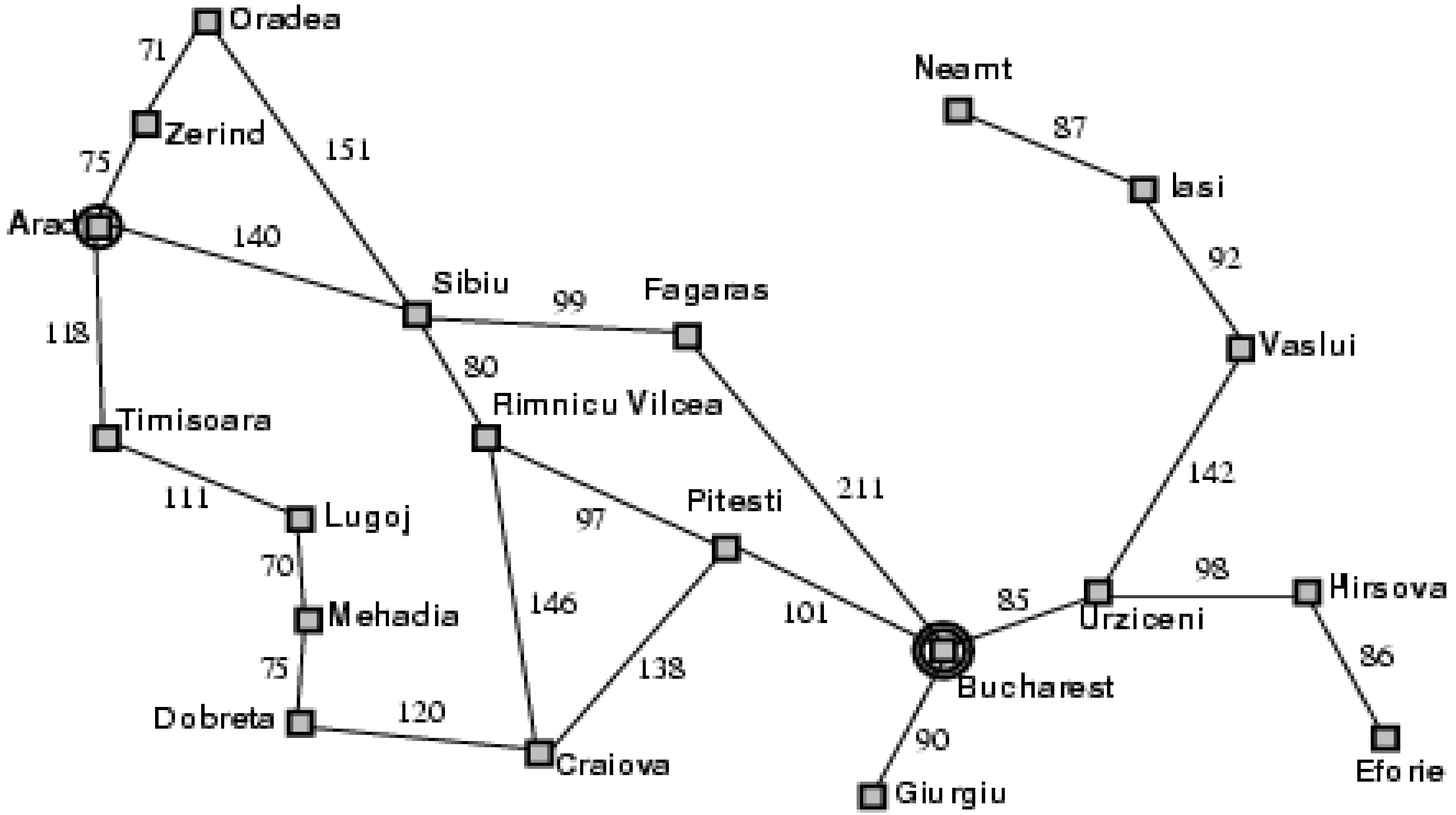
- **Problem formulation-** is the process of deciding *what actions* and *states to take given the goal.*
- If the agents lacks any other information the environment is unknown.
- **Ex. A self-driving car with(known) or without (unknown) a map.**
    - *With the map, agent can decide what to do, can consider future actions to decide among many possible immediate actions.*
- If an environment is ***observable, discrete, known and deterministic*** then the solution to any problem in that environment will be a fixed sequence of actions.
- Ex. **Plot** a course from one city to another

Example #1: A Real-world Scenario

- **A touring agent problem**
  - Imagine an agent in the city of **Arad, Romania** enjoying a touring holiday. Now, suppose the agent has a ***non-refundable tickets*** to fly out of **Bucharest** the following day.
  - In that case, it makes sense for the agent to adopt the goal of getting to **Bucharest.**

Q. What sequence of actions will lead to the agent achieving its goal? (*As shown Romania map in the next slide*)

Figure 3.1: Map of Romania



Possible Solution

- Let’s say that an agent is in the city of ***Arad***, and has to goal of getting to ***Bucharest***.
- Consider the agent currently in the city of **Arad**, toward the end of touring holiday in Romania; Flight leaves tomorrow from **Bucharest**.
- **Formulate goal**: Be in Bucharest
- **Formulate problem**:
  - States: various cities
  - Actions: drive between cities
- **Find solution**:
  - Sequence of cities, e.g., *Arad*, *Sibiu*, *Fagaras*, *Bucharest.*

Well-defined problems and solution

- A problem can be defined formally by five components:

1. **Initial state**
2. A description of possible **actions**
3. The **transition model**
4. The **goal test**
5. A path **cost function**

Well-defined problems and solution...

1. **initial state**
   - The **initial state** that the agent starts in.
   - For example, the initial state for our agent in Romania might be described as **In(Arad)**.
2. **Action:**
   - A description of the possible **actions** available to the agent.
   - Given a particular state s, **ACTIONS(s)** returns the set of actions that can be executed in s.
   - For example, from the state **In(Arad),** the possible actions are **{Go(Sibiu), Go(Timisoara), Go(Zerind)}.**

Well-defined problems and solution …

3. Transition model

- A description of what each action does
- Specified by a function **RESULT(s, a)** that returns the state that results from SUCCESSOR doing action **a** in state **s**.
- **Successor** = refer to any state reachable from a given state by a single action.
- For example, we have RESULT(In(Arad), Go(Zerind)) = In(Zerind).

Note: Together, ***the initial state, actions, and transition model*** implicitly define the **state space** of the problem—

- the set of all states reachable from the initial state by any sequence
- The **state space** forms a directed network or **graph** in which the nodes are states and the links between nodes are actions.

Well-defined problems and solution …

4. Goal test

- The **goal test**, which determines whether a given state is a goal state.
- The agent’s goal in Romania is the singleton set

  {In(Bucharest)}.
- Sometimes the goal is specified by an abstract property rather than an explicitly enumerated set of states.
- For example,
  - in chess, the goal is to reach a state called ***“checkmate***,” where the opponent’s king is under attack and can’t escape.

Well-defined problems and solution …

5. Path cost

- A **path cost** function that assigns a numeric cost to each path.
- The problem-solving agent chooses a cost function that reflects its own performance measure.
- For the agent trying to get to **Bucharest,** time is of the essence, so the cost of a path might be its length in kilometres.
- the cost of a path can be described as the *STEP COST sum* of the costs of the individual actions along the path.
- The **step cost** of taking action a in state s to reach state s′ is denoted by **c(s, a, s′).**

* Optimal Solution

- We have defined the problem(including goals)
  - How do we know that we achieved the goals?
- A **solution** to a problem is:
  - an action sequence that leads from the initial state to a goal state.
- Solution quality is measured by the path cost function,
- An **optimal solution** has the lowest path cost among all solutions.

Single-state Problem Formulation

- **For example**, Let's say that an agent is in the town of ***Arad***, and has to goal of getting to ***Bucharest***.

A problem is defined by five items:

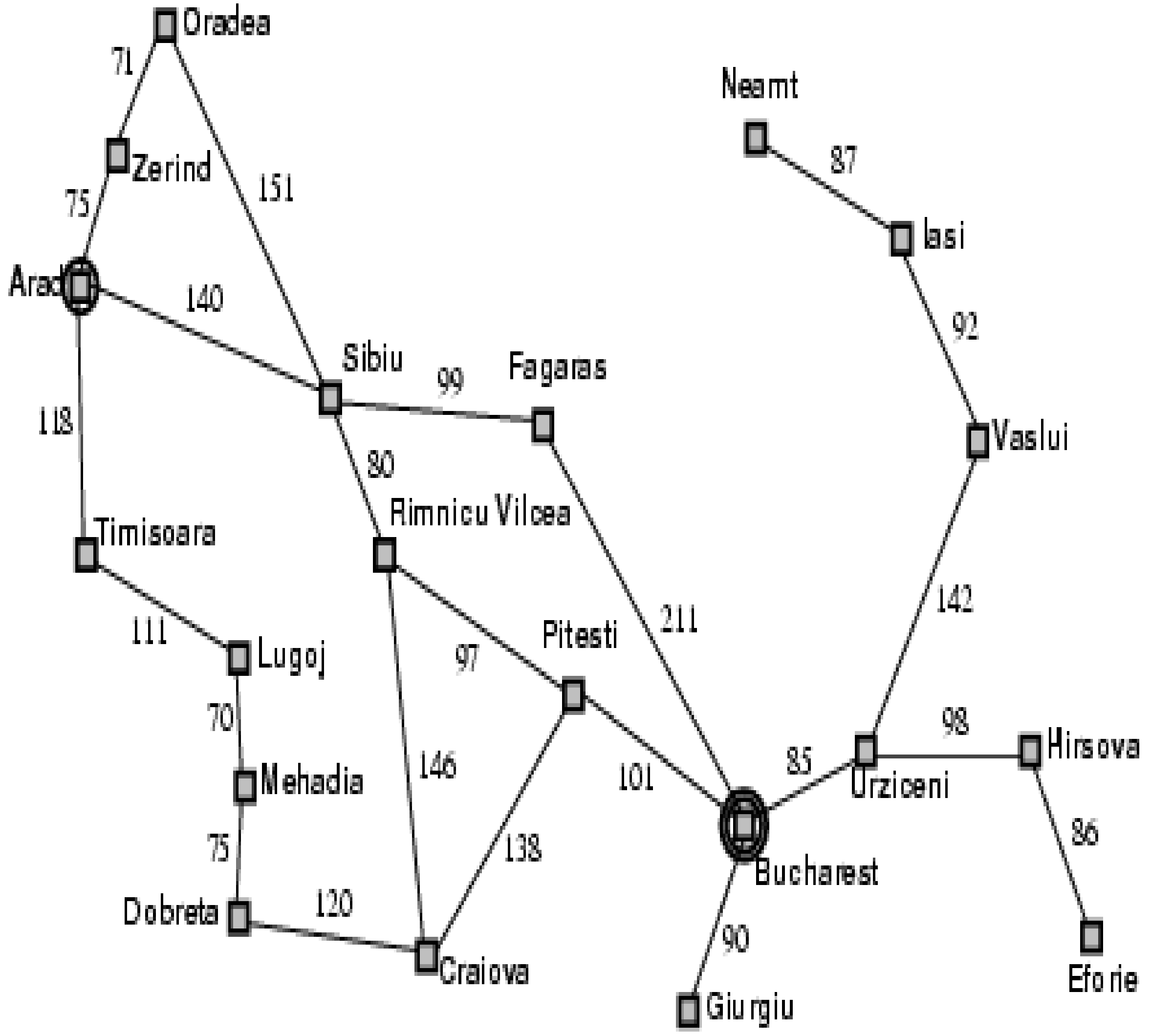
1. Initial state e.g., "at Arad"
2. Actions or successor function *S(x)* = set of action->state pairs
   - e.g., *S(Arad) = {<Arad →Zerind, Zerind>,*
     <Arad →Sibiu,Sibiu>,
     <Arad →Timisoara, Timisoara}
3. Goal test, can be
   - explicit, e.g., *x* = "at Bucharest"
   - implicit, e.g., *Checkmate(x)*
4. Path cost (additive)
   - e.g., sum of distances, number of actions executed, etc.
   - *c(x,a,y)* is the step cost, assumed to be ≥ 0
   - *c(Arad, Arad →Zerind, Zerind) = 75,…etc*

5. Goal test: A solution is a **sequence of actions** leading from the **initial state** to **a goal state.**

Single-state problem formulation

The single-state problem formulation is therefore:

- Initial state: *at Arad*
- Actions: *the successor function S:*
  - *S(Arad) = {<Arad→Zerind, Zerind>,*
    <Arad→Sibiu, Sibiu>,
    <Arad→Timisoara, Timisoara}
  - *S(Sibiu) = {<Sibiu→Arad, Arad>,*
    <Sibiu→Oradea, Oradea>,
    <Sibiu→Fagaras, Fagaras>,
    <Sibiu→Rimnicu Vilcea, Rimnicu Vilcea>}
  - *etc.*
- Goal test: *at Bucharest*
- Path cost:
  - *c(Arad, Arad→Zerind, Zerind) = 75,*
    c(Arad, Arad→Sibiu, Sibiu) = 140,
    c(Arad, Arad→Timisoara, Timisoara) = 118, etc.



State Space Representation

- State space Representation consists of defining an INITIAL state (from where to start), the Goal state(the destination) and then we follow a certain set of sequences of steps(called states).
- Let’s define each of them separately.
  - **State:** AI problem can be represented as a well-formed set of possible states.
  - *The state can be the initial state i.e. starting point, Goal state i.e. destination point, and various other possible states between them which are formed by applying a certain set of rules.*
  - **Space:** In an AI problem the exhaustive set of all possible states is called **space**.
  - **Search:** In simple words, search is a technique that takes the initial state to the goal state by applying a certain set of valid rules while moving through space of all possible states.
  - So we can say that to do a search process we need the following: Initial state, Set of valid rules, Goal state

Problem Types

Based on the environment types discussed in Chapter 2, we can identify a number of common problem types:

Environment Type	Problem Type
i. Deterministic, fully-observable	• Single-state problem
ii. Non-observable, known state space	• Sensorless/conformant problem
iii. Nondeterministic and/or partially-observable	• Contingency problem
iv. Unknown state space	• Exploration problem

Problem Types(Description)

i. Deterministic, Full Observable →Single state problems

- Each state is fully observable and it goes to one definite state after any action
- i.e. Agent knows exactly which state it will be in; the solution is a sequence.
- Here, the goal state is reachable in one single action or sequence of actions.
- Deterministic environments ignore uncertainty.
- Ex Vacuum cleaner with the sensor.

ii. **Non-observable, known state space → sensorless problem (conformant problem)**

- Problem-solving agent does not have any information about the state
- i.e. agent may have no idea where it is. The solution may or may not be reached
- Ex. In the case of a vacuum cleaner, the goal state is to clean the floor.
- **Action** is to suck if there is dirt.
- So, in non-observable conditions, as there is no sensor, it will have to suck the dirt, irrespective of whether it is towards the right or left. Here, the solution is space is the states specifying its movement across the floor.

Problem Types(Description)...

iii. **Nondeterministic, partially observable →** contingency problem

– Percepts provide new information about the current state.

– Often interleave search and execution.

– The effects of action are not clear.

– Ex: if we take a vacuum cleaner and now assume that the sensor is attached to it, then it will suck if there is dirt.

– The movement of the cleaner will be based on its current percepts.

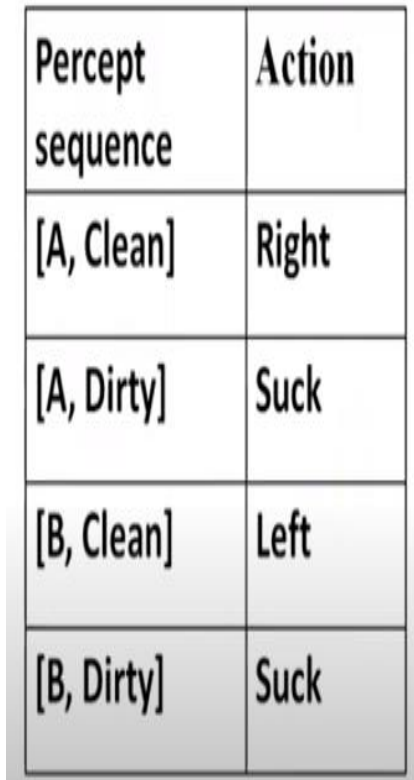
iv. **Unknown state space →** exploration problem

– States and impact of actions are not known.

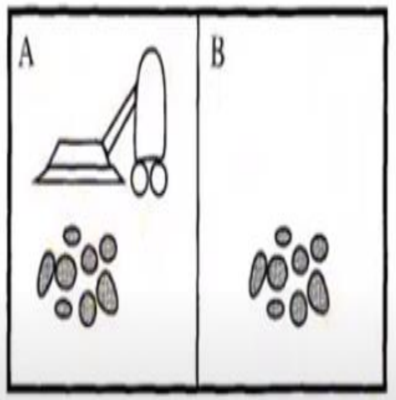
– Ex: Online search that involves acting without complete knowledge of the next state or scheduling without map

Toy problems (Vacuum world Problem)

- The first example we will examine is the **vacuum world** first introduced in Chapter 2 .
- This can be formulated as a problem as follows:
- **States:**
  - The state is determined by both the **agent location** and the **dirt locations.**
  - The **agent** is in one of two locations, each of which might or might not **contain dirt**.
  - Thus there are **$2 \times 2^2 = 8$** possible world states.
  - A larger environment with **n** locations has **$n \cdot 2^n$** states.



Percept sequence	Action
[A, Clean]	Right
[A, Dirty]	Suck
[B, Clean]	Left
[B, Dirty]	Suck

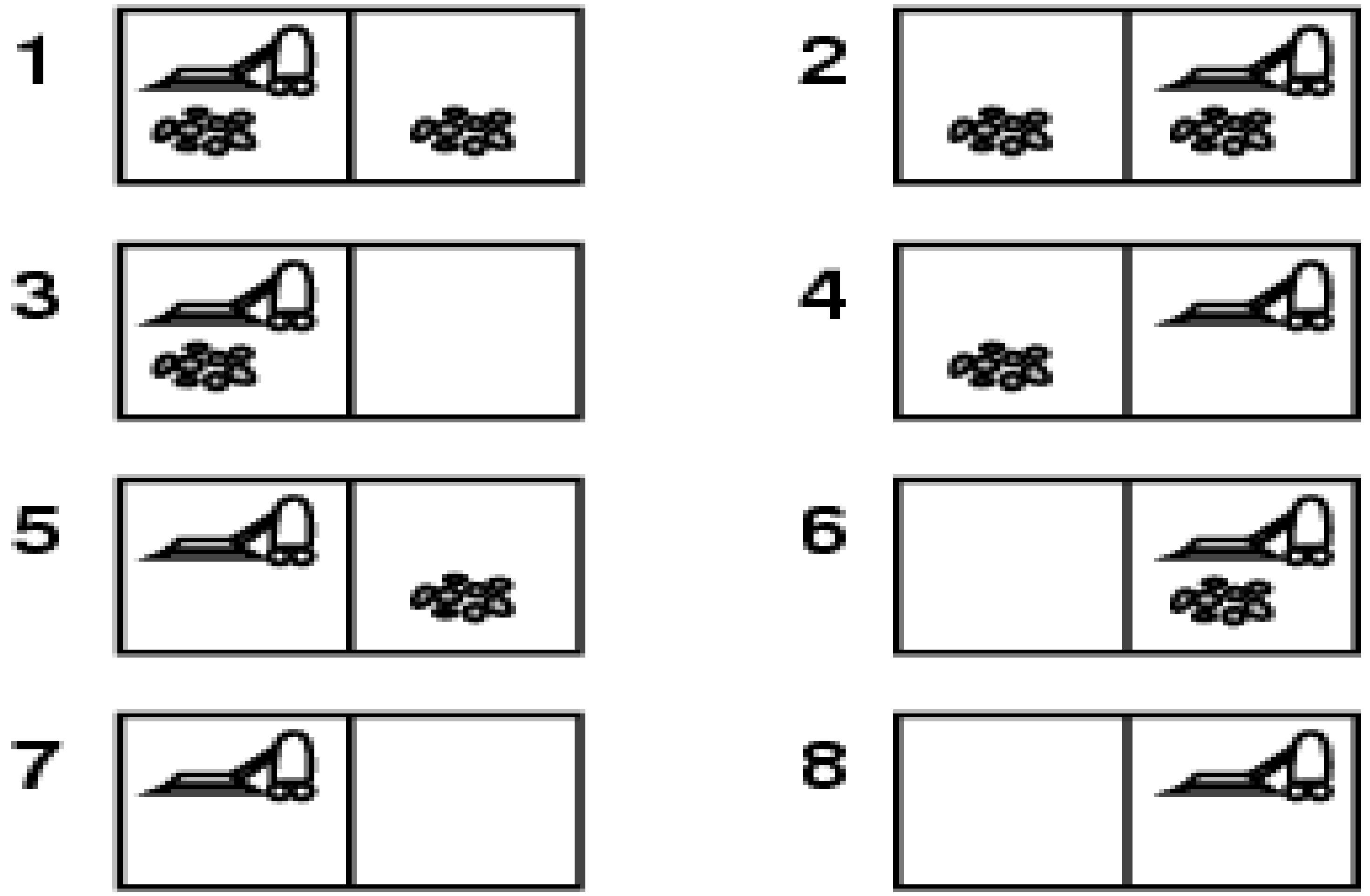


Toy problems (Vacuum world Problem)…

- **Initial state:**
  - Any state can be designated as the initial state.
- **Actions:**
  - In this simple environment, each state has just three actions: ***Left, Right***, and ***Suck***.
- **Transition model:**
  - The actions have their expected effects, except that moving *Left* in the leftmost square, moving *Right* in the rightmost square, and *Sucking* in a clean square have no effect.
- **Goal test**: This checks whether all the squares are clean.
- **Path cost**: Each step costs 1, so the path cost is the number of steps in the path.

Example #2: Vacuum World(State Representation)

- Single-state, start in #5. Solution?



Example #2: Vacuum World(State Representation)..

- Single-state, start in #5.

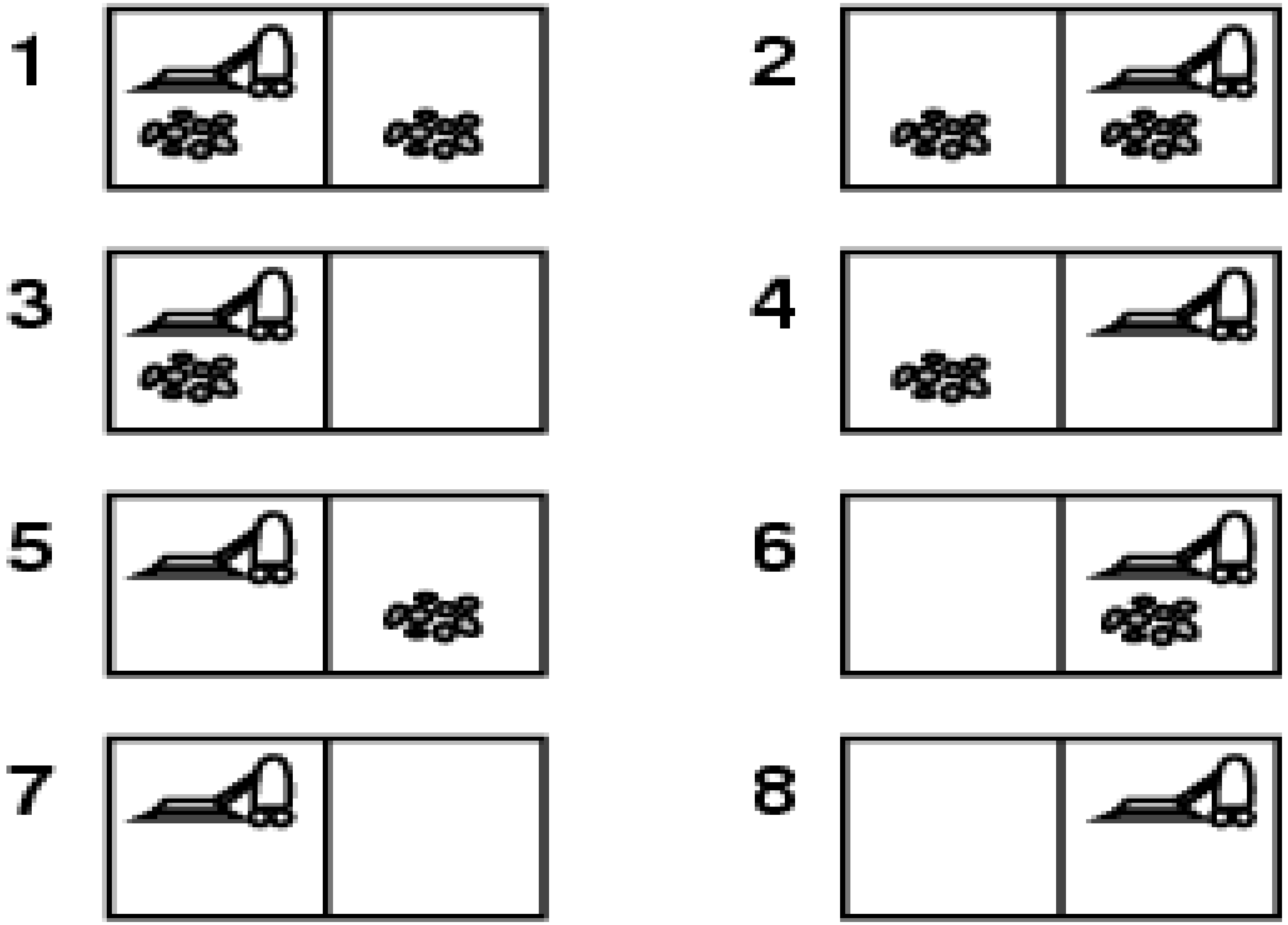
  Solution? *[Right, Suck]*

- Sensorless, start in

  {*1,2,3,4,5,6,7,8* } e.g.,

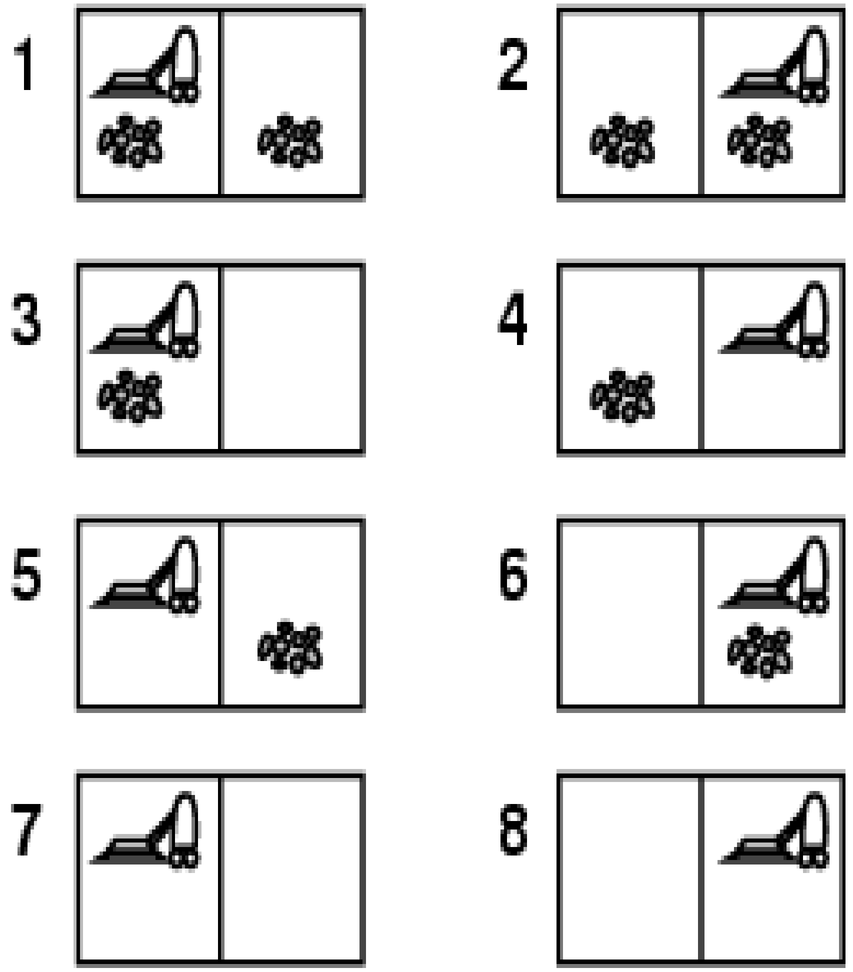
  Right goes to {*2,4,6,8* }

  Solution?



Example #2: Vacuum World(State Representation)

- Sensorless, start in {*1,2,3,4,5,6,7,8*} e.g., *Right* goes to {*2,4,6,8*} Solution? *[Right,Suck,Left,Suck]*
- Contingency
  - Nondeterministic: *Suck* may dirty or a clean carpet
  - Partially observable: just know location & dirt at current location.
  - Percept: *[L, Clean]*, i.e., start in #5 or #7 Solution?



Example #2: Vacuum World(State Representation)

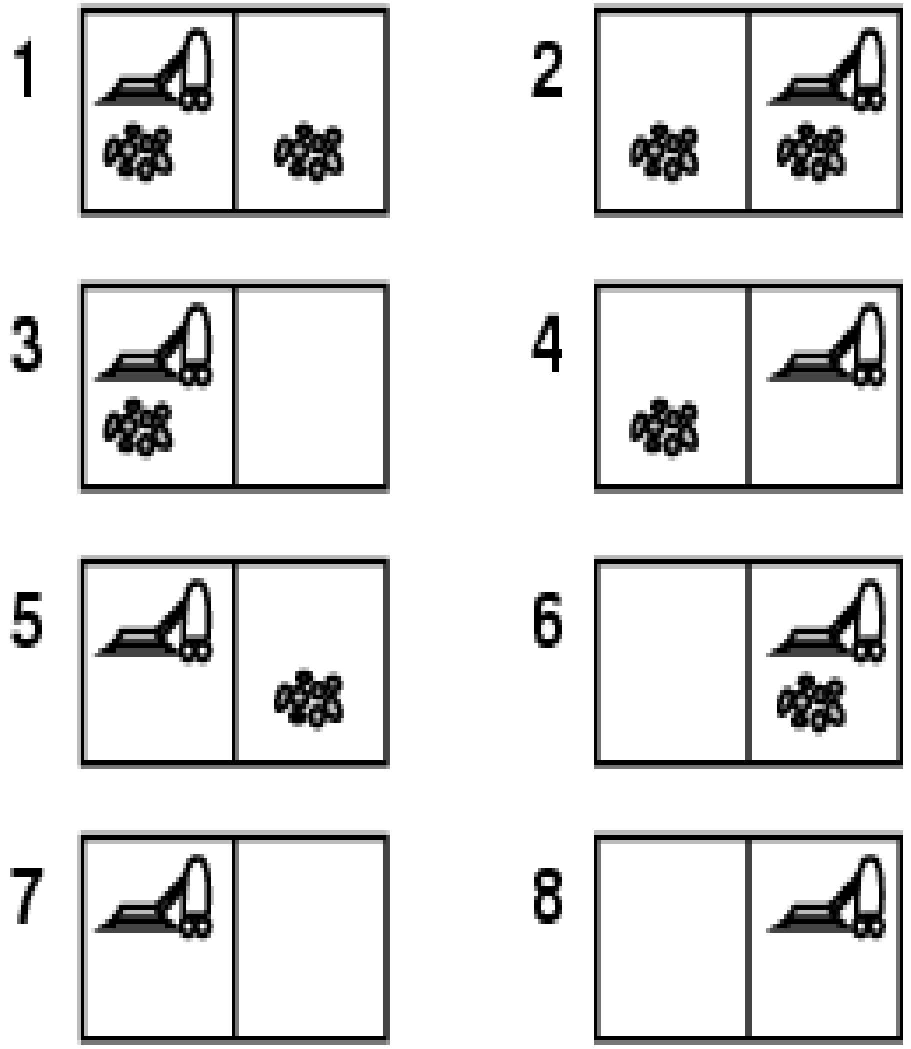
- Sensorless, start in

  {*1,2,3,4,5,6,7,8*} e.g.,

  Right goes to {*2,4,6,8*}

  Solution?

  [Right, Suck, Left, Suck]
- Contingency
  - Nondeterministic: *Suck* may dirty a clean carpet
  - Partially observable: location, dirt at current location.
  - Percept: *[L, Clean]*, i.e., start in #5 or #7
    Solution? *[Right, **if** dirt **then** Suck]*
- ***Note:** Formally, the goal is {state 7, state 8}*



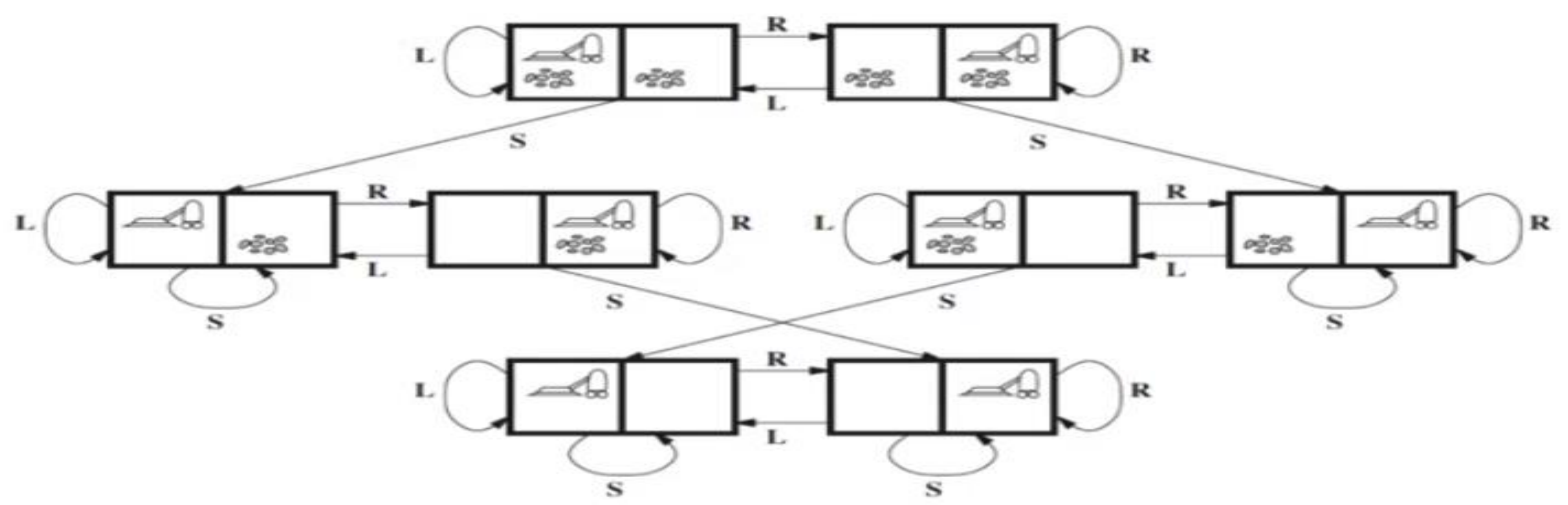
Selecting a state space – “**Abstraction**”

- **Abstraction** refers to the process of **removing unnecessary or irrelevant details** from the problem definition.

  → **State space** must be **abstracted** for problem solving

  - (Abstract) state = set of real states
  - (Abstract) action = set of real actions
    - e.g., "Arad → Zerind" represents a complex set of possible routes, detours, rest stops, etc.
  - (Abstract) solution = set of real paths that are solutions in the real world
- To be valid,
  - The abstract solution **must be a solution** for all the real-world problems it was an **abstraction**.
- To be useful,
  - Each abstract action **should be "easier"** than the original problem.

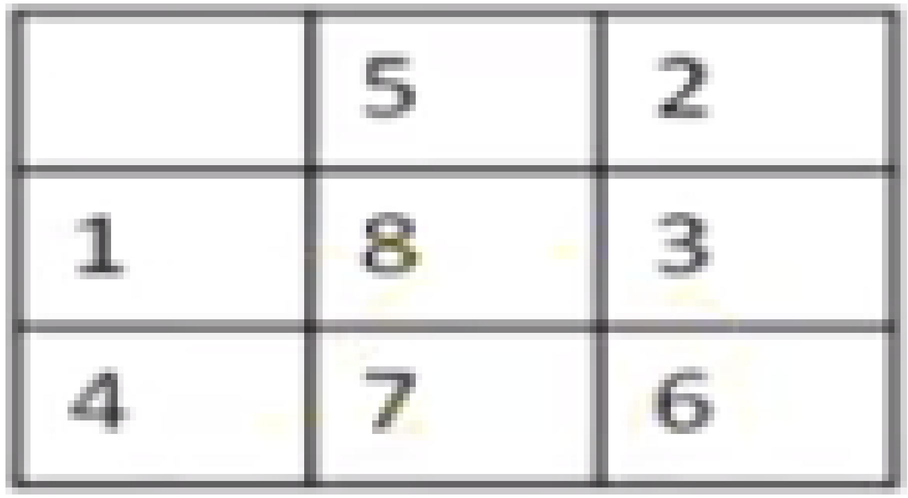
Example: Vacuum world state space graph



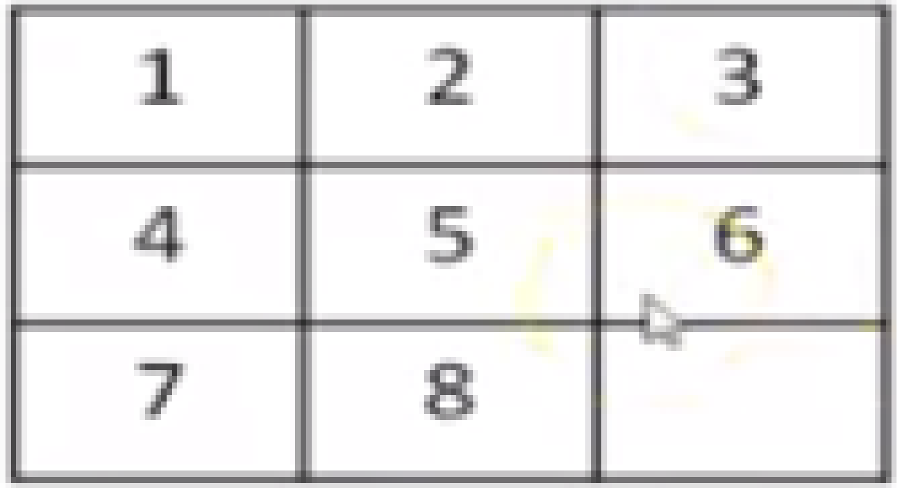
- states? One of two locations, each of which may contain dirt => $2x2^2$=8 possible states
- actions? *Left*, *Right*, *Suck*
- goal test? No dirt at all locations
- path cost? 1 per action

#2: Toy Problem-8 Puzzle problem

- The 8-puzzle problem consists of a 3 x 3 board with eight numbered tiles and a blank space
- A tile adjacent to the blank space can slide into the blank.



	5	2
1	8	3
4	7	6



1	2	3
4	5	6
7	8	

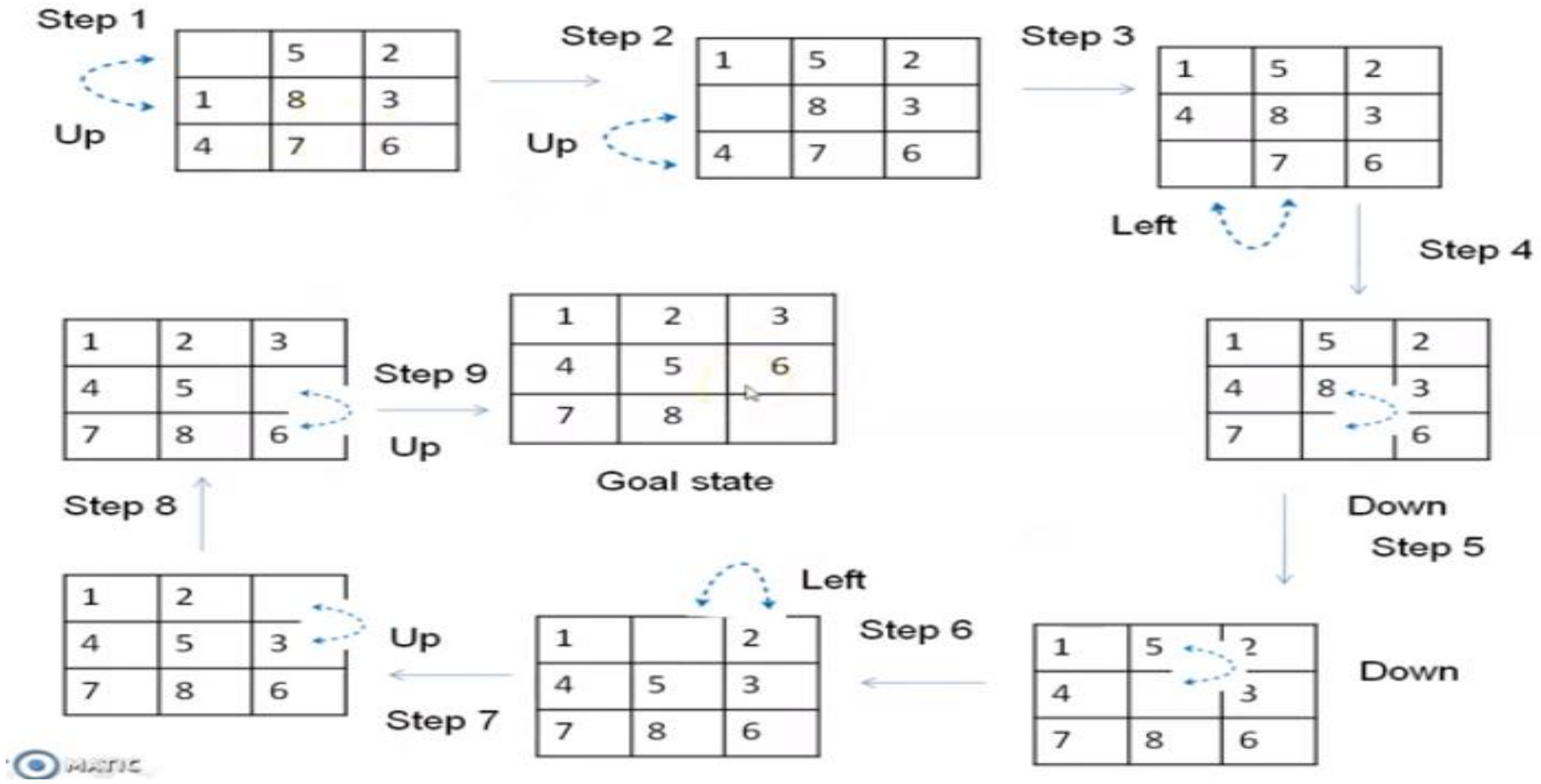
Start State

Goal State

8-Puzzle problem- How it works

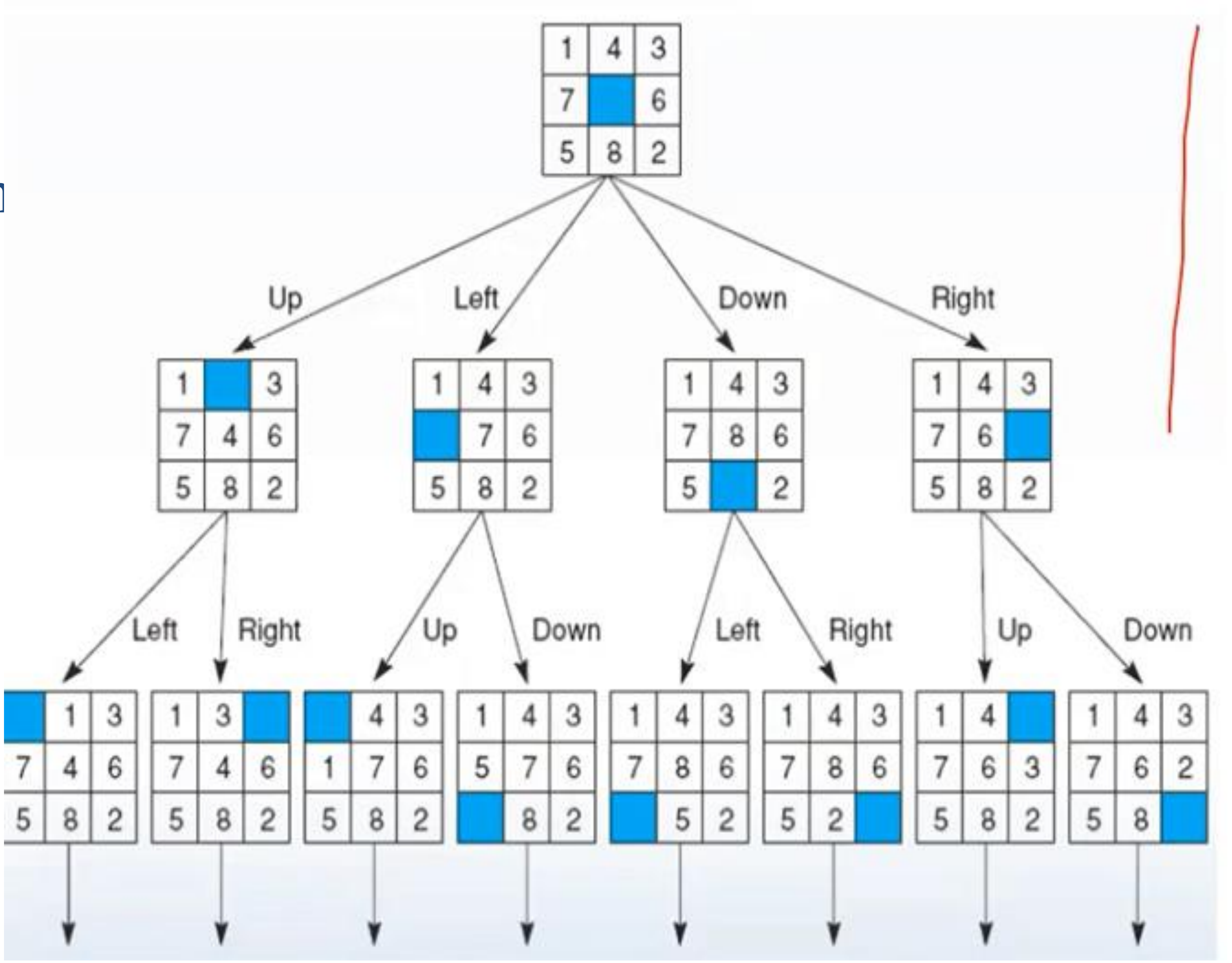
- The object is to reach a specified goal state
- **States:**
  - A state description specifies the location of each of the eight tiles and the blank in one of the nine squares.
- **Initial State:**
  - Any state can be designated as the initial state (*No fixed state for initial state*)
- **Successor function:**
  - This generates the legal states that result from trying the four actions(blank moves left, Right, Up, or Down).
- **Goal test:**
  - This checks whether the state matches the goal configuration (other goal configurations are possible).
- **Path Cost:** Each step costs 1, so the path cost is the number of steps in the path.

8-Puzzle problem- How it works...



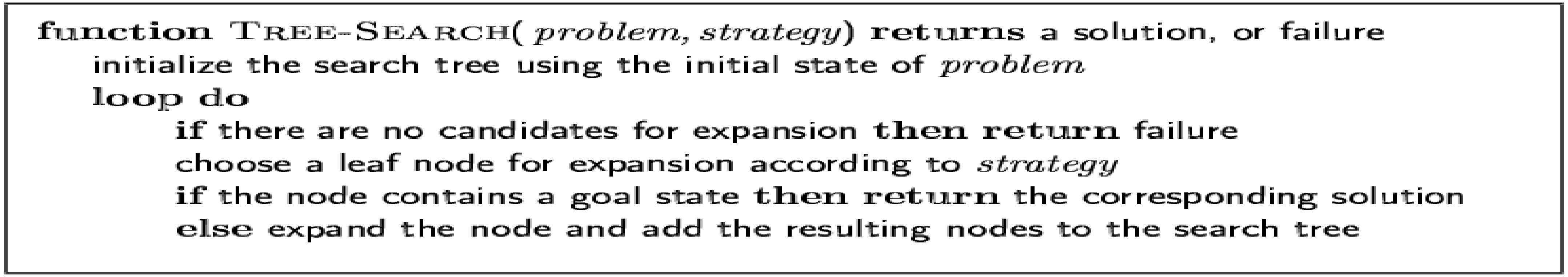
8-puzzle problem

State space



Tree search algorithms

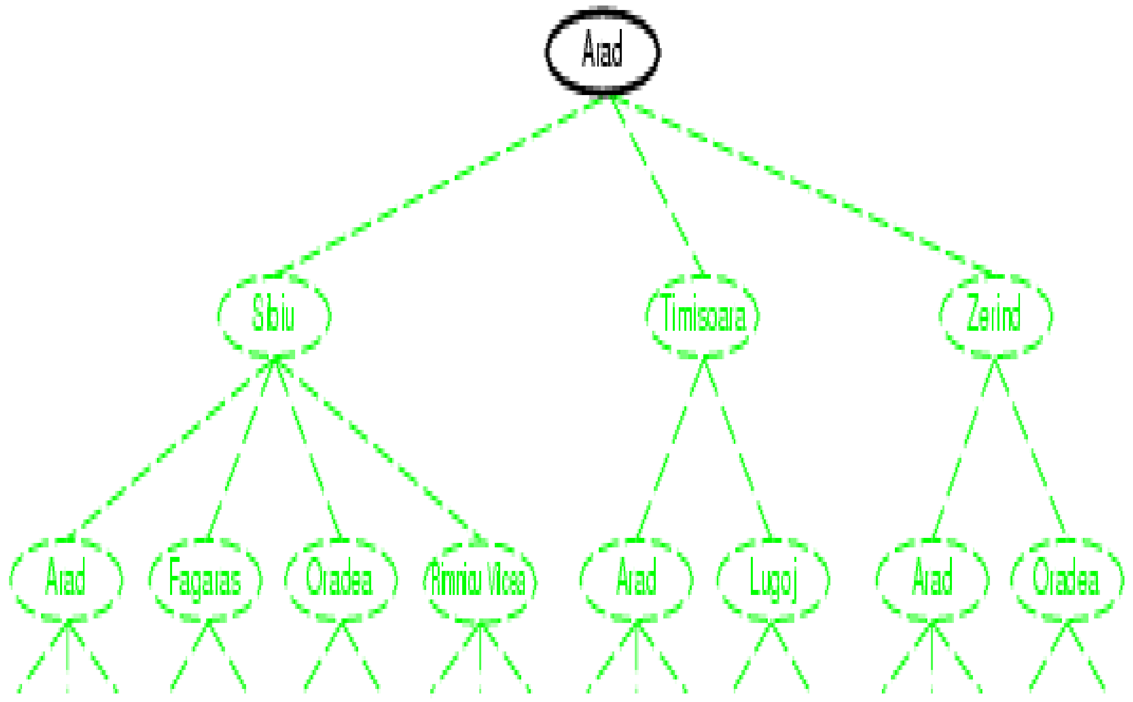
- Basic idea:
  - offline, simulated exploration of state space by generating successors of already-explored states (**expanding states**).
  - If the **node represents a goal state** we **stop searching**.
  - Otherwise we "**expand**" **the selected node** (i.e. generate its possible successors using the successor function) and **add the successors as child nodes** of the selected node.

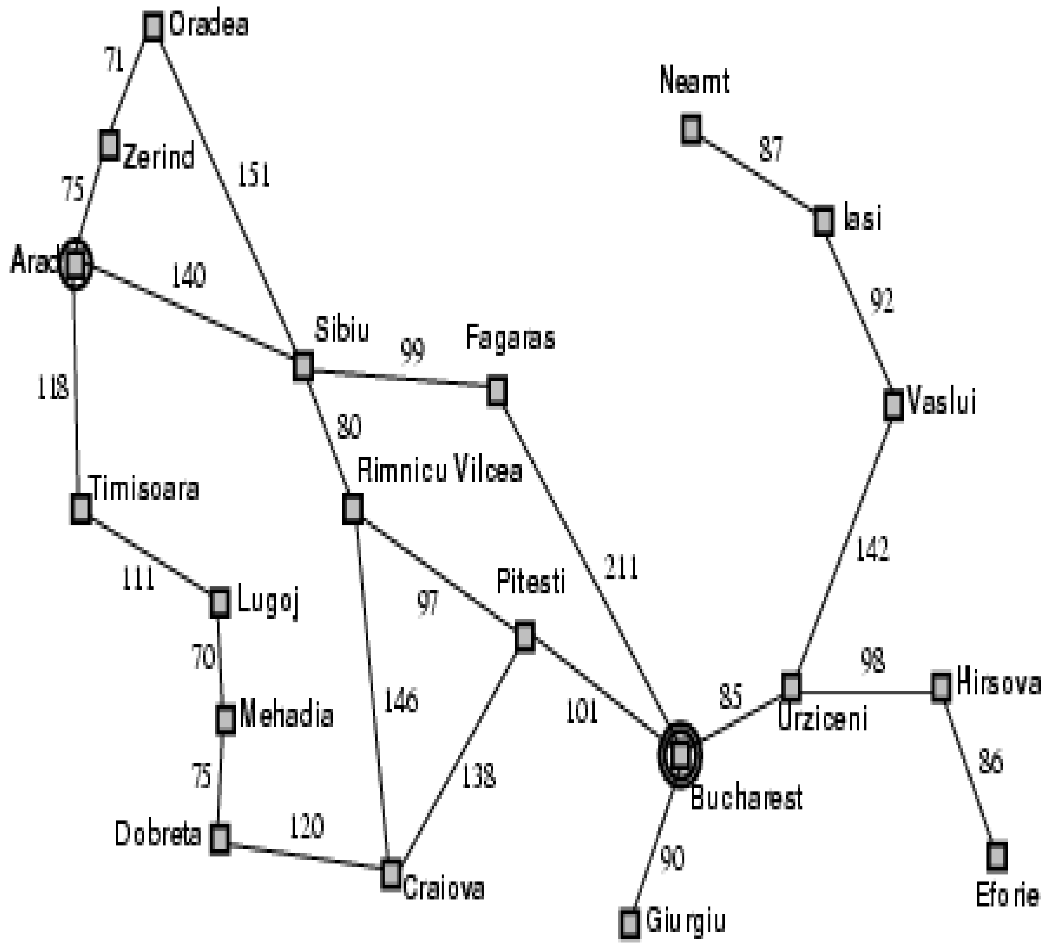


```
function TREE-SEARCH( problem, strategy) returns a solution, or failure
    initialize the search tree using the initial state of problem
    loop do
        if there are no candidates for expansion then return failure
        choose a leaf node for expansion according to strategy
        if the node contains a goal state then return the corresponding solution
        else expand the node and add the resulting nodes to the search tree
```

Tree search example (1)

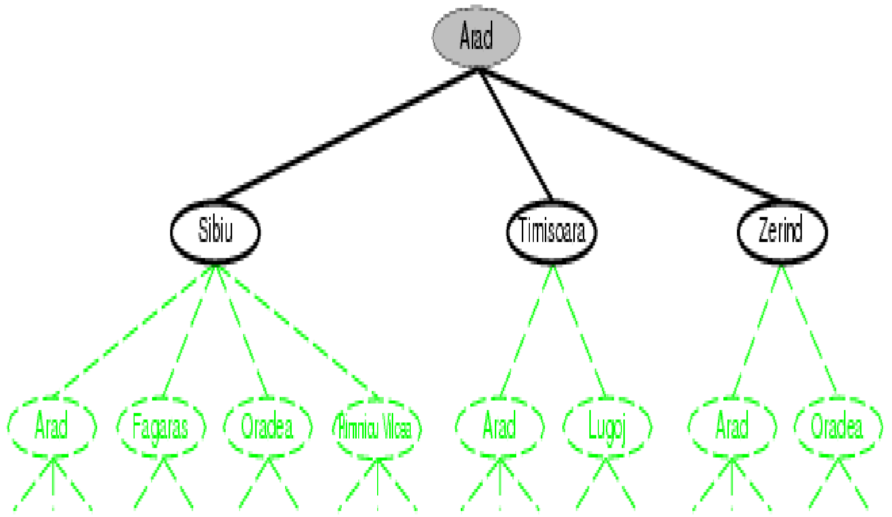
- **Partial search trees** for finding a route from **Arad to Bucharest.**
  - **Nodes** that **have been expanded** are **shaded**;

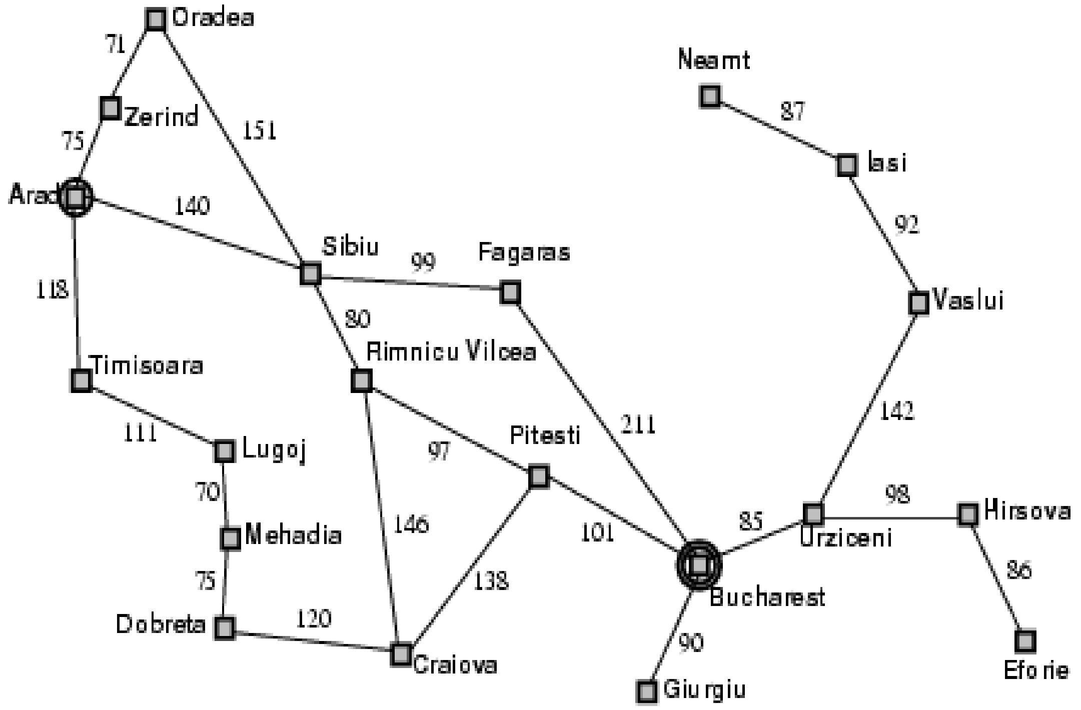




Tree search example (2)

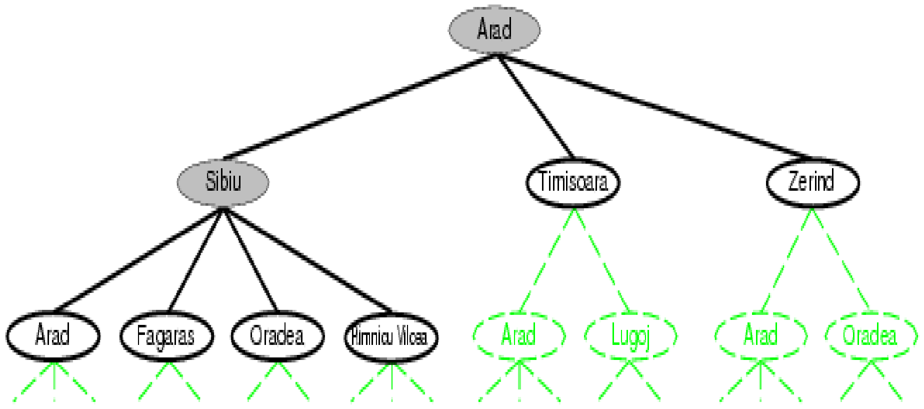
- **Partial search trees** for finding a route from **Arad to Bucharest.**
  - **Nodes** that have been expanded are **shaded**;
  - **Nodes** that **have been generated** but **not yet expanded** are **outlined in bold**;

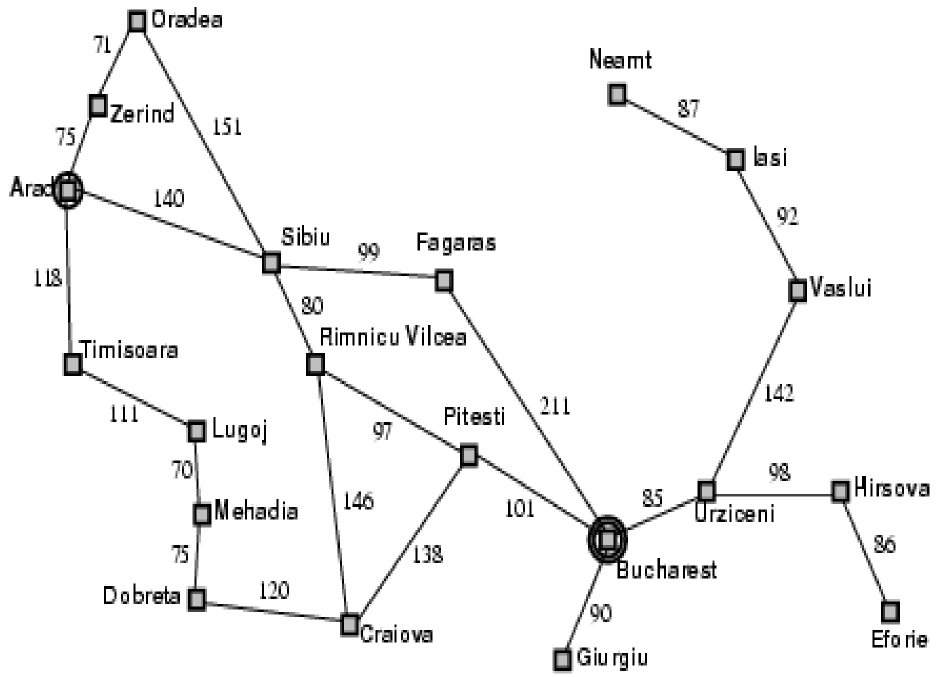




Tree search example (3)

- **Partial search trees** for finding a route from **Arad to Bucharest.**
  - **Nodes** that have been expanded are **shaded**;
  - **Nodes** that **have been generated** but **not yet expanded** are **outlined in bold**;
  - **Nodes** that **have not yet been generated** are shown in **faint dashed lines.**





Searching Strategies

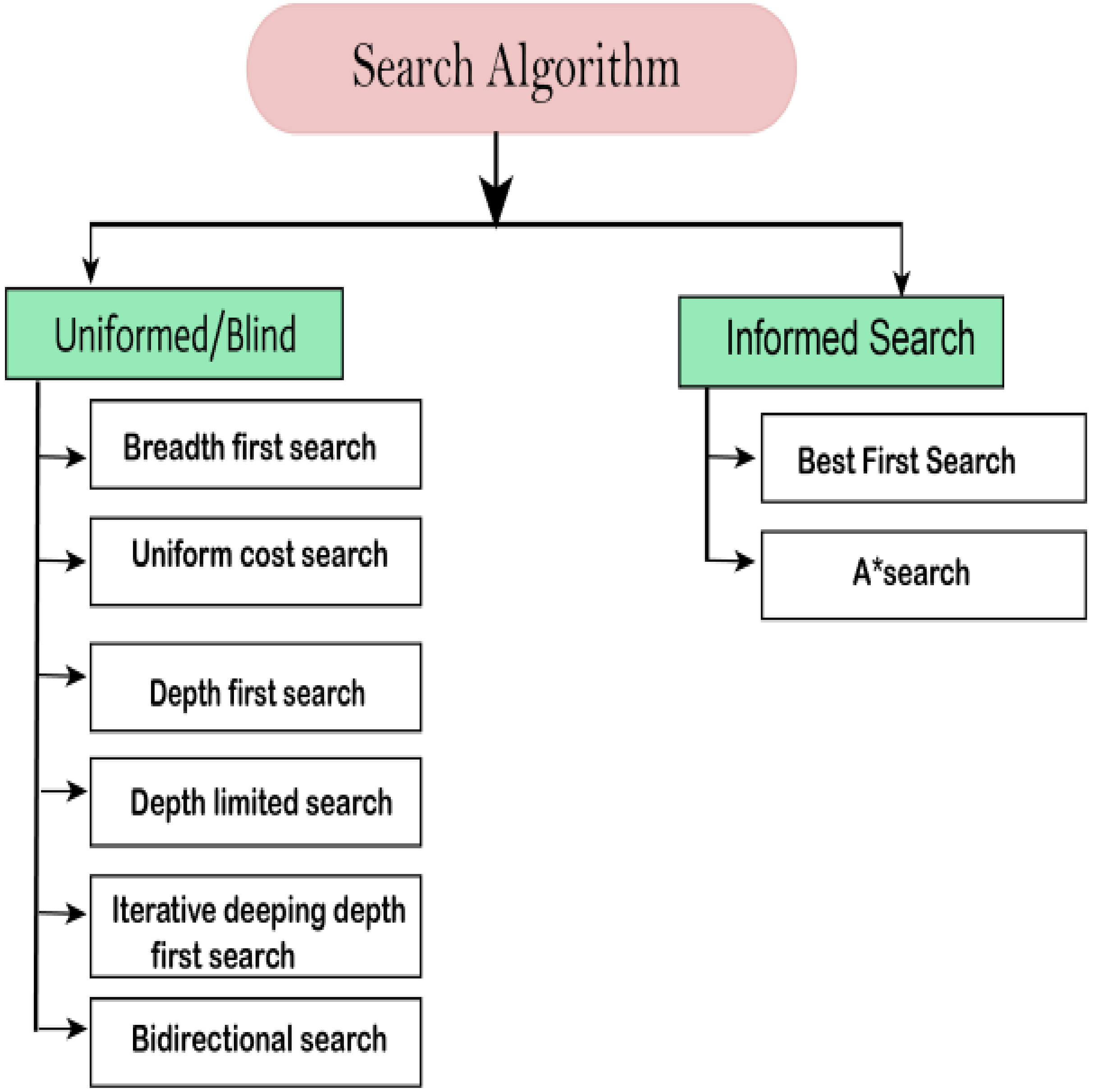
- There are four properties of search algorithms
  - **Completeness:** A search algorithm is said to be complete if it guarantees to return a solution if at least any solution exists for any random input.
  - **Optimality:** If a solution found for an algorithm is guaranteed to be the best solution (lowest path cost) among all other solutions, then such a solution for is said to be an optimal solution.
  - **Time Complexity:** is a measure of time for an algorithm to complete its task.
  - **Space Complexity:** It is the maximum storage space required at any point during the search, as the complexity of the problem.

Searching Strategies

- Time and space complexity are measured in terms of:
  - ***b*:** maximum branching factor of the search tree.
  - *E.g. the branching factor from Arad is 3.*
  - ***d*:** depth of the least-cost solution.
  - ***m*:** maximum depth of the state space (may be ∞).

Types of Search Algorithms

- Based on the search problems we can classify the search algorithms into
  - ✓ **uninformed (Blind search) search** and
  - ✓ **informed search (Heuristic search)** algorithms.



Uninformed/Blind Search:

- They do not contain any domain knowledge such as closeness, the location of the goal.
- It operates in a ***brute-force*** way as it only includes information about how to traverse the tree and how to identify ***leaf*** and ***goal*** nodes.
- Have no information about the number of steps or the path cost from the current state to the goal.
- It examines each node of the tree until it achieves the goal node.
- It is important for problems for which there is no additional information to consider.

Informed (= heuristic) search

- Have problem-specific knowledge (knowledge that is true from experience).
- In an informed search, problem information is available which can guide the search.
- Can find solutions more efficiently than uninformed search
- ***Informed search*** is also called a **Heuristic search.**
  - *A **heuristic** is a way which might not always be guaranteed for best solutions but guaranteed to find a good solution in reasonable time.*
- ***Informed search*** can solve much complex problem which could not be solved in another way.

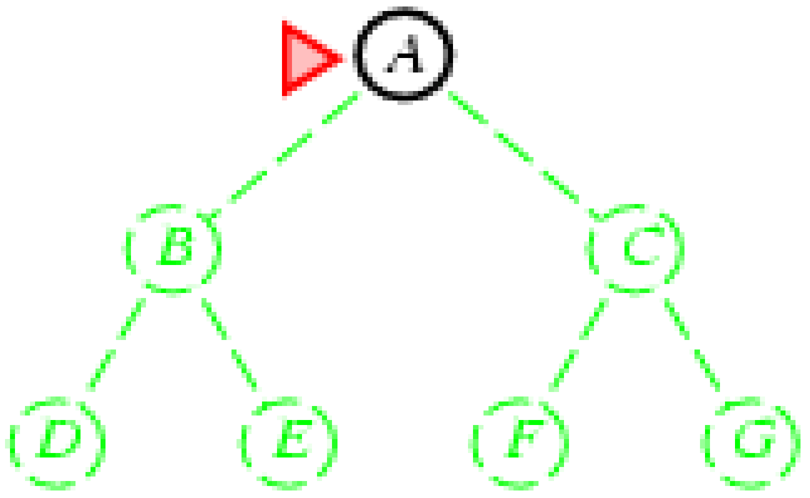
a. Uninformed search Algorithm

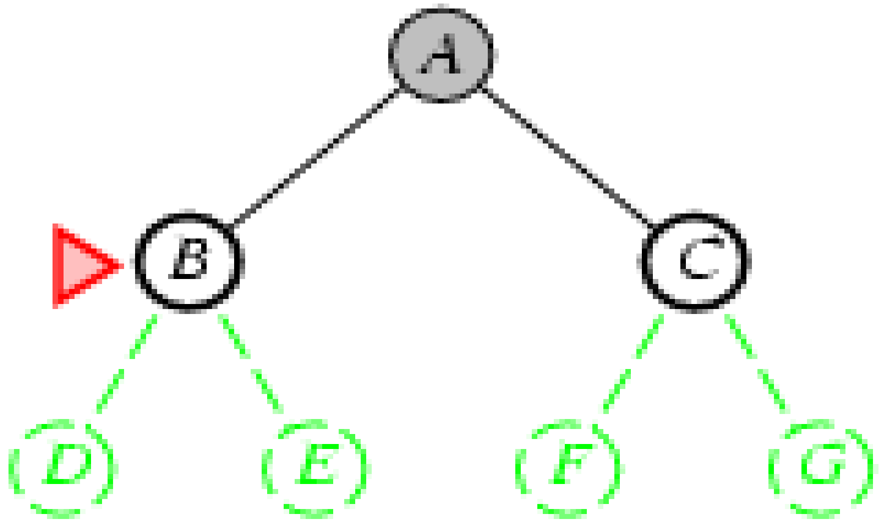
- The **simplest type of tree search** algorithm is called ***uninformed*, or *blind*, tree search**.
- Uninformed search strategies **use only the information available** in the problem definition.
  - *Breadth-first search*
  - *Uniform-cost search*
  - *Depth-first search*
  - *Depth-limited search*
  - *Iterative deepening search*
  - *Bidirectional search*

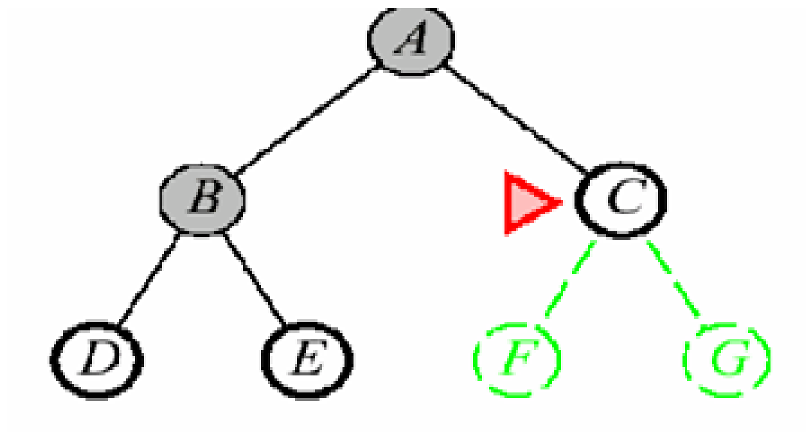
1. Breadth-First Search (BFS)

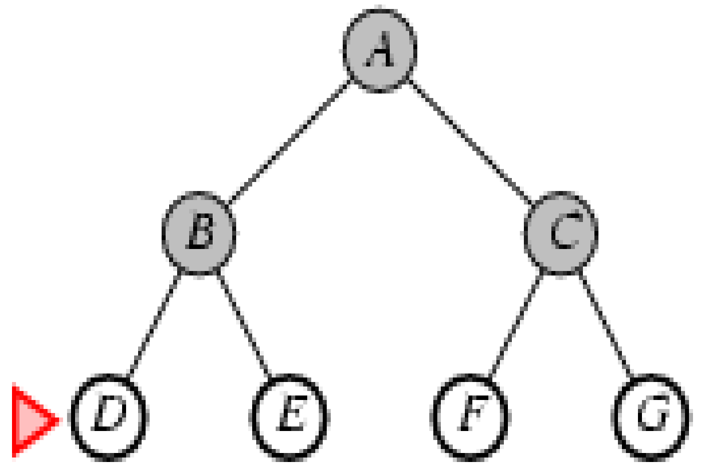
- BFS-is the most common search strategy for traversing a **tree** or **graph**.
- BFS algorithm starts searching from the **root node** of the tree and expands all successor node at the current level before moving to ***nodes of next level.***
- In BFS search we **always select the minimum depth node for expansion.**
- **Expand shallowest unexpanded node.**
- **Implementation:**
  – *fringe* is a **FIFO queue**, i.e., **new successors go at end**

Example #1: Breadth-first search







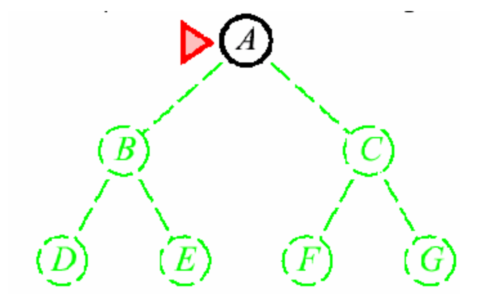


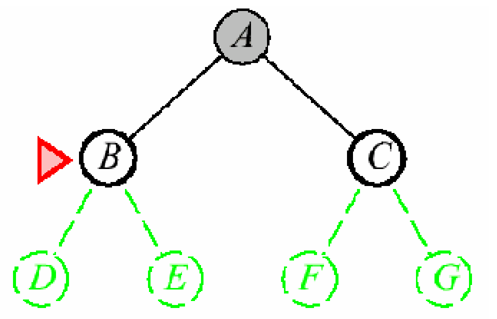
Properties of BFS(Describe more…)

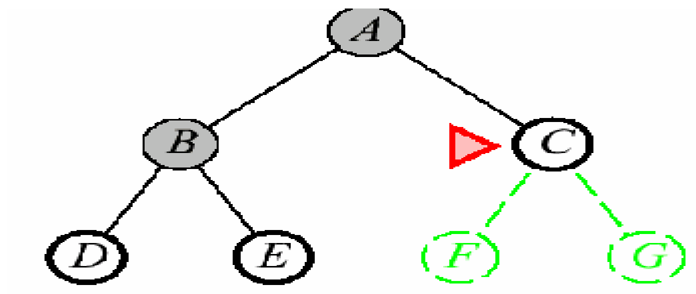
- Expand shallowest unexpanded node,
  - i.e. expand all nodes on a given level of the search tree before moving to the next level
- **Implementation**: **use FIFO queue** data structure to store the list:
  - Expansion: **put successors at the end of queue**.
  - **Pop nodes** from the **front of the queue** .
- **Properties:**
  - Takes space: keeps every node in memory.
  - Optimal and complete: guarantees to find solution.
    - Optimal -**Yes**, if step costs identical
    - Complete -**Yes** (if *b* is finite)

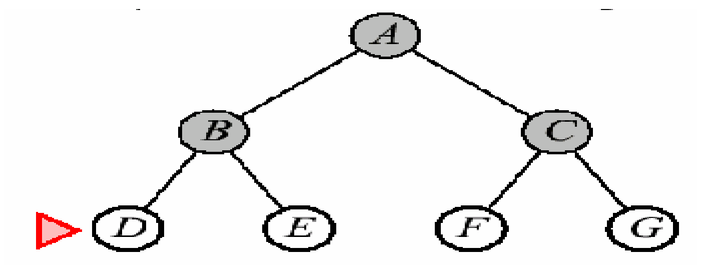
▪ Time/Space complexity is exponential!

▪ Space is the bigger problem (more than time)









Breadth-First Search (BFS)…

- ✓ **What does this mean?** From the four criteria, it means:
  - ✓ **BFS is complete**. If there exists an answer, it will be found.
  - ✓ **BFS is optimal.** The path from the start state to goal state will be shallow(small distance).

- ✓ **What about time complexity and space complexity?**
  - ✓ If we look at how BFS expands from the root we see that it first expands on a set number of nodes, say b.
  - ✓ On the second level it becomes b^2.
  - ✓ On the third level it becomes b^3.
  - ✓ And so on until it reaches b^d for some depth d.
    - ✓ **$1 + b + b^2 + b^3 + \dots + b^d$ which is $O(b^d)$**
- ✓ Since all leaf nodes need to be stored in memory, space complexity is the same as time complexity.

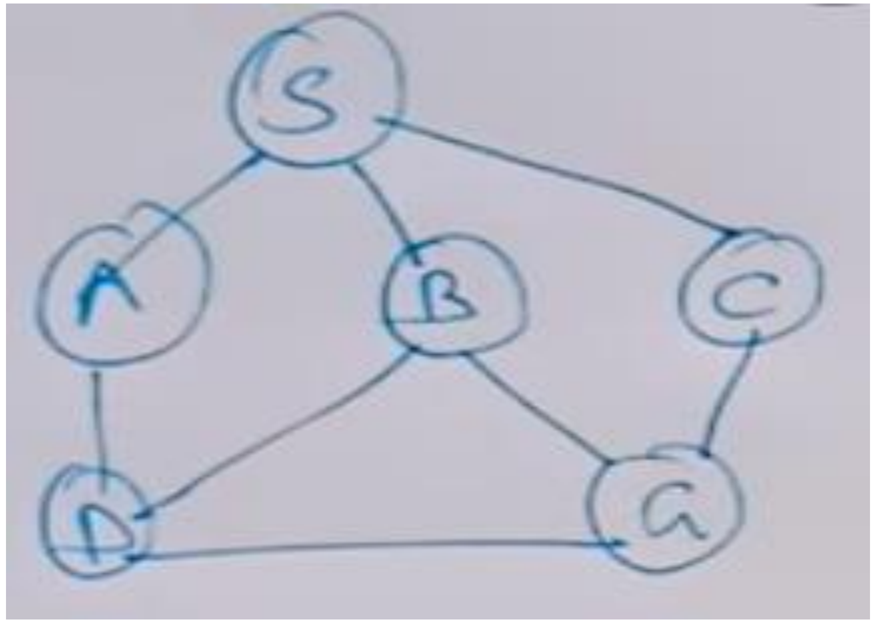
BFS:Advantages and Disadvantages

- **Advantages:**
  - BFS will provide a solution if any solution exists.
  - If there are more than one solutions for a given problem, then BFS will provide the minimal solution which requires the least number of steps.
- **Disadvantages:**
  - It requires lots of memory since each level of the tree must be saved into memory to expand the next level.
  - BFS needs lots of time if the solution is far away from the root node.

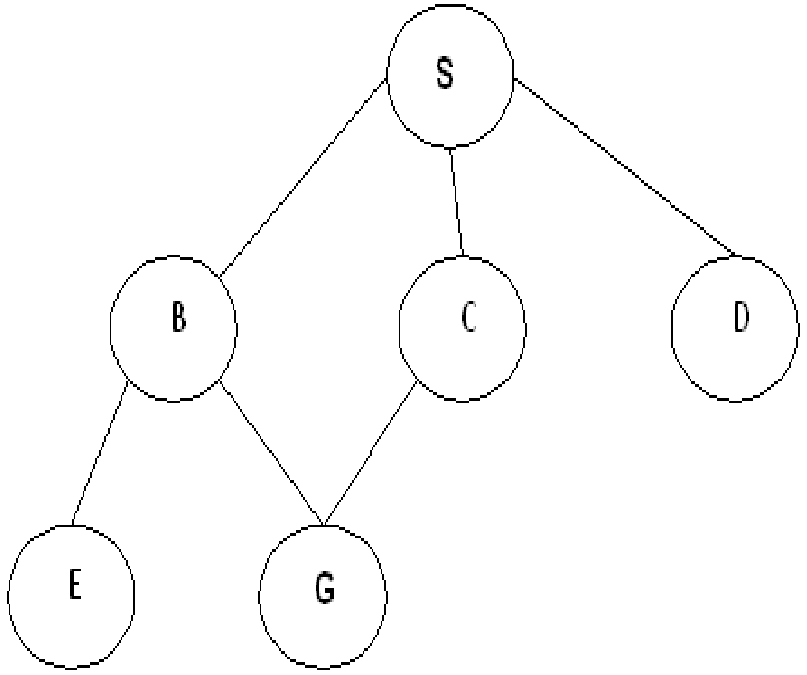
Exercise #1

- Apply BFS to find an optimal path from start node to Goal node.
  - S is Start node and
  - G is Goal node.

a.



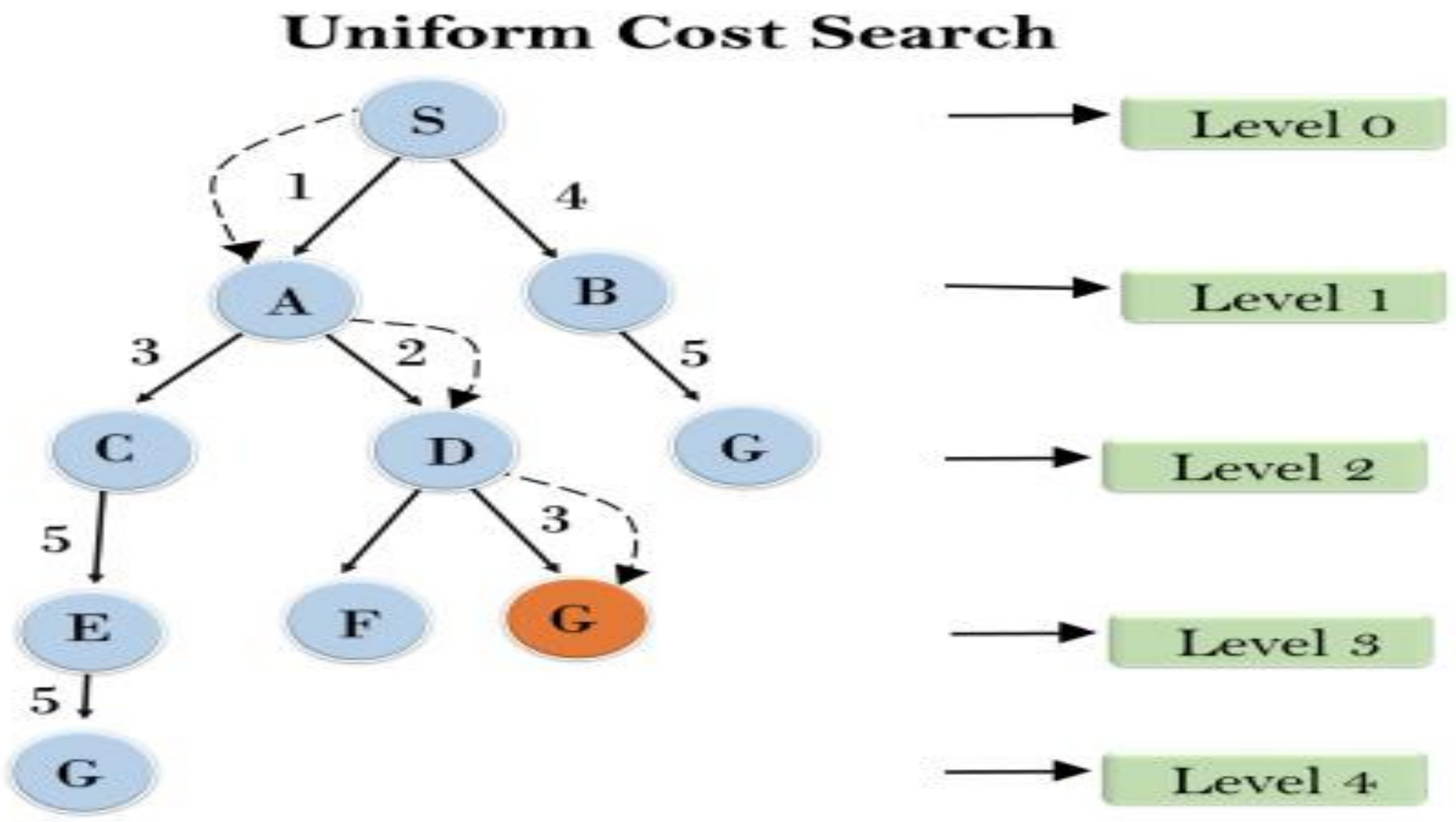
b.



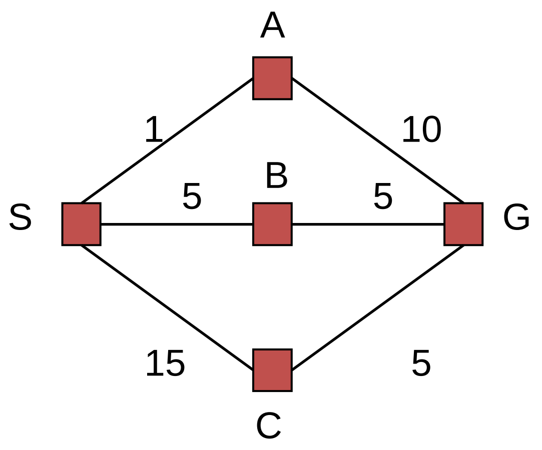
2. Uniform cost Search

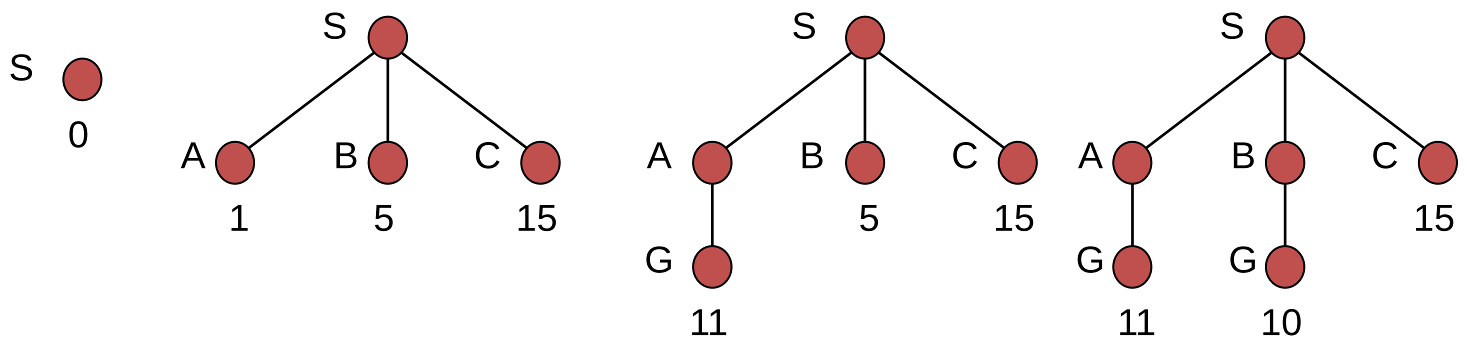
- The goal of this technique is **to find the shortest path to the goal in terms of cost.**
    - It modifies the BFS by always expanding the least-cost unexpanded node.
- **Expand the least-cost unexpanded node first.**
- **Implementation:** Use a priority queue instead of a simple queue.
- **Advantages:** UCS- is optimal because at every state the path with the least cost is chosen.
- **Disadvantages:** It does not care about the number of steps involve in searching and only concerned about path cost.
    - Due to which this algorithm may be stuck in an infinite loop.

Example #1


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

Example #2: Uniform cost Search





- **Properties:**
  - **–Equivalent to breadth-first if step costs all equal.**
  - **–This strategy finds the cheapest solution** provided the cost of a path must never decrease as we go along the path

    g(successor(n)) ≥ g(n), for every node n
  - **–Takes space since it keeps every node in memory.**

2. Uniform cost Search (1)

- **Evaluating UCS with the four criteria:**
  - **Complete?** Yes. Same as BFS.
  - **Optimal?** Yes. More so than BFS, because the shortest path is guaranteed.
  - **Time complexity?** Still $O(b^d)$
  - **Space complexity?** Still $O(b^d)$

3. Depth-first search

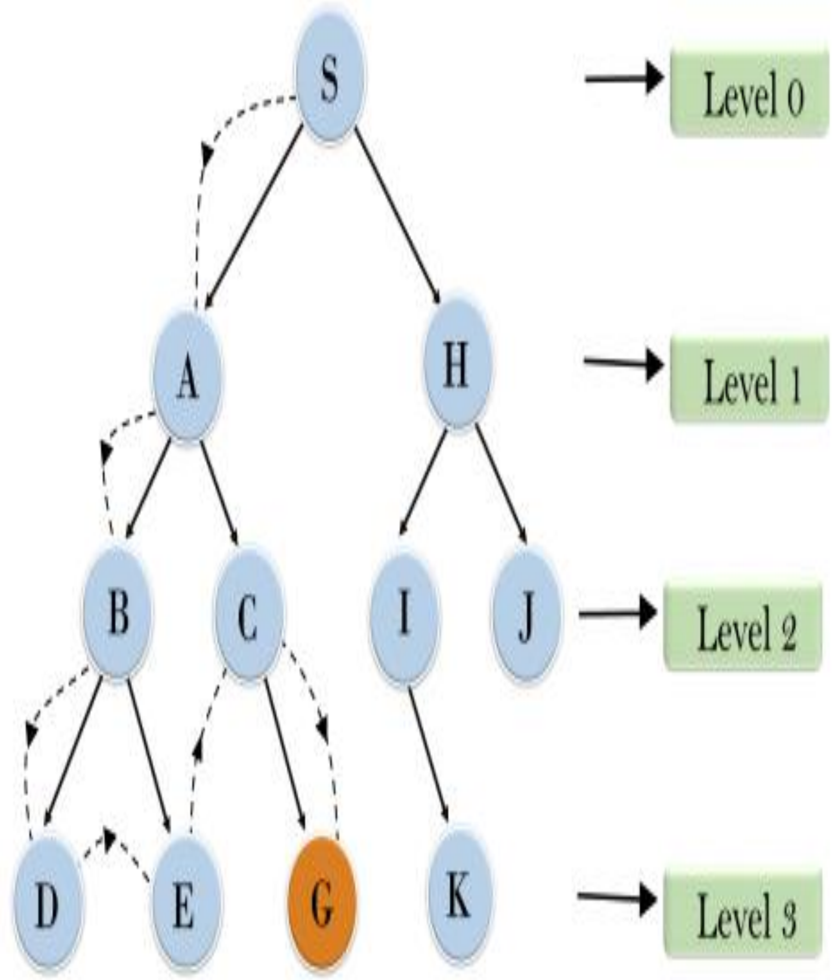
- DFS is a *recursive algorithm* for traversing a tree or graph data structure.
- It is called the ***depth-first search*** because it starts from the root node and follows each path to its greatest depth node before moving to the next path.
- i.e . *Expand deepest unexpanded node.*
- The process of the DFS algorithm is similar to the BFS algorithm
- **Implementation:**
  – *fringe* = **LIFO stack**, i.e., **put successors at front**

Example #1: Depth-First Search

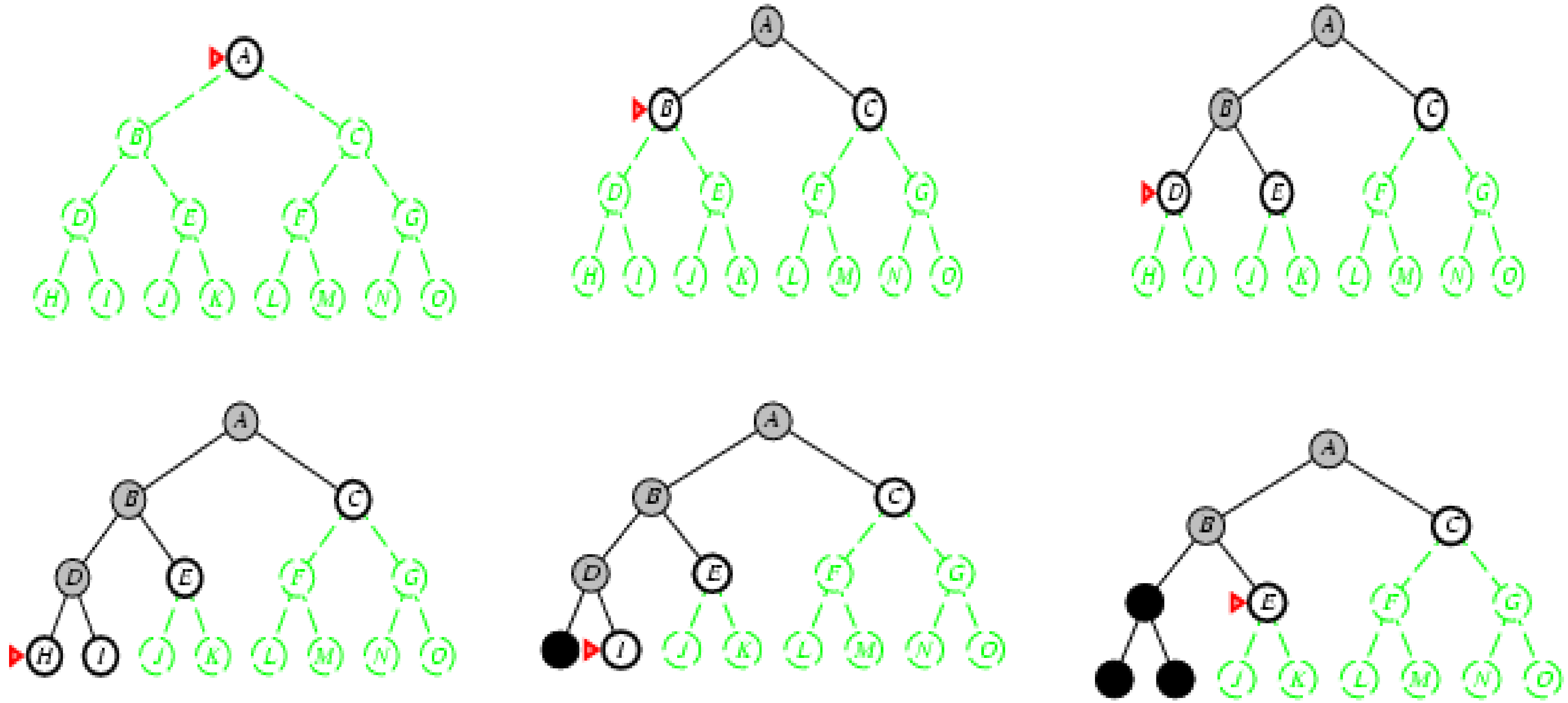
In the below search tree, we have shown the flow of DFS, and it will follow the order as:

Root node--->Left node ----> right node.

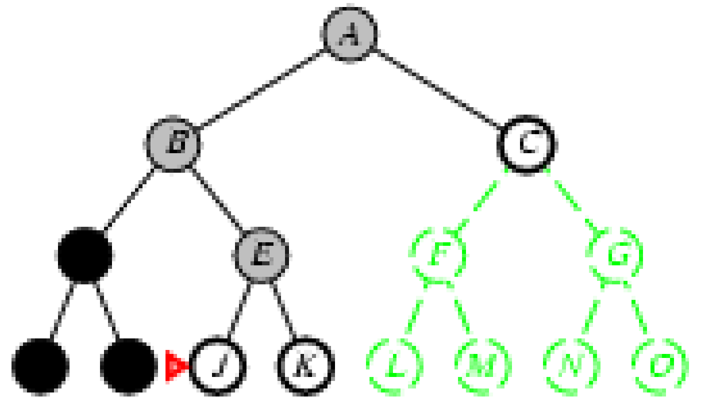
- *It will start searching from root node S, and traverse A, then B, then D and E, after traversing E, it will backtrack the tree as E has no other successor and still goal node is not found.*
- *After backtracking it will traverse node C and then G, and here it will terminate as it found goal node.*

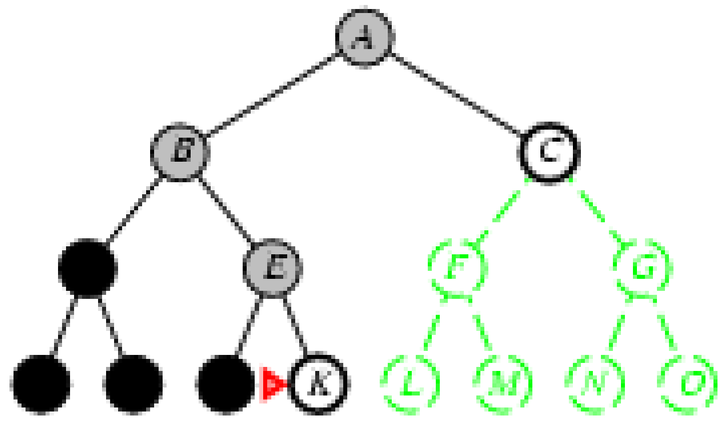


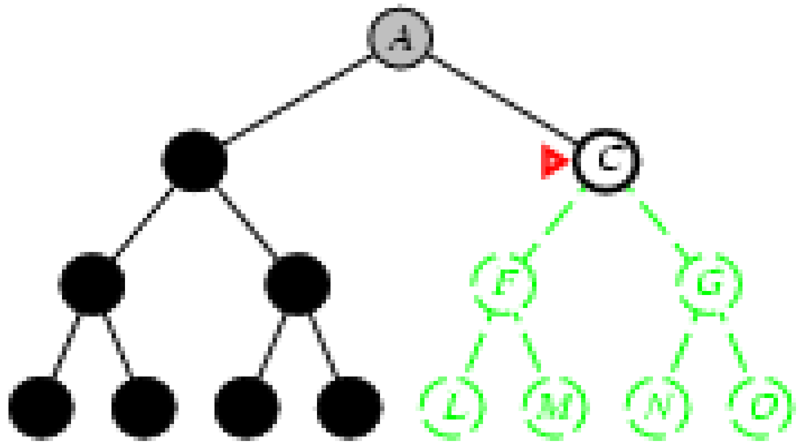
Example #2: Depth-first search

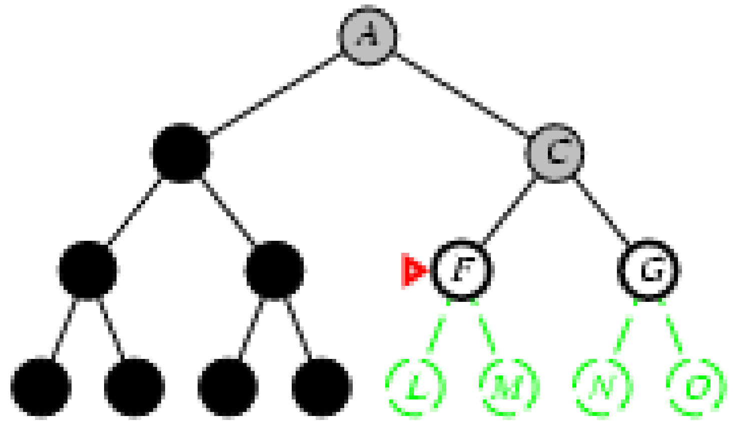


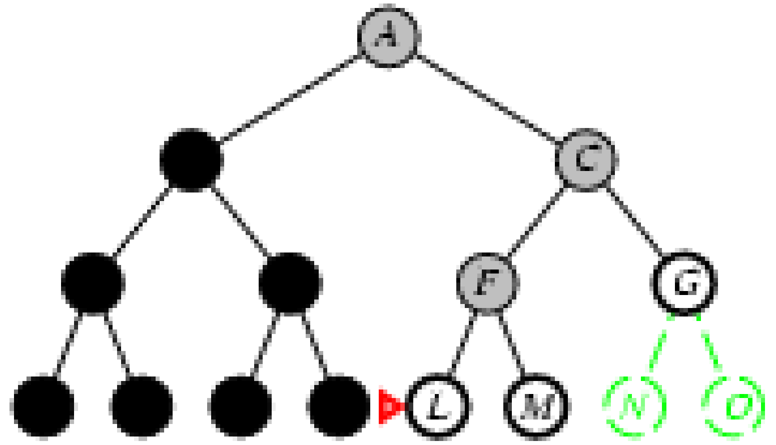
Example #2: Depth-first search(2)

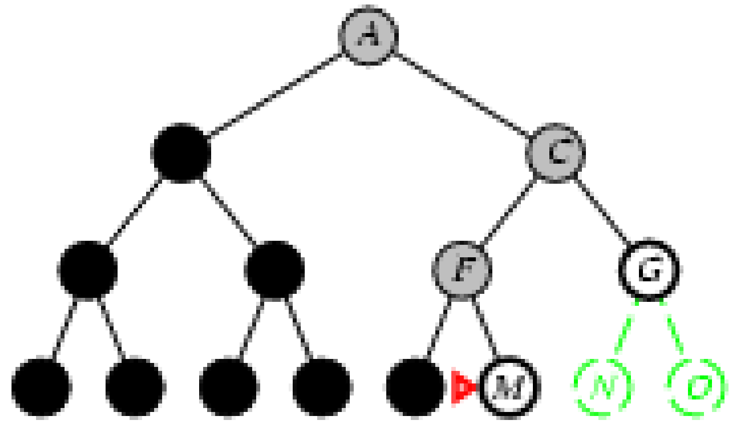








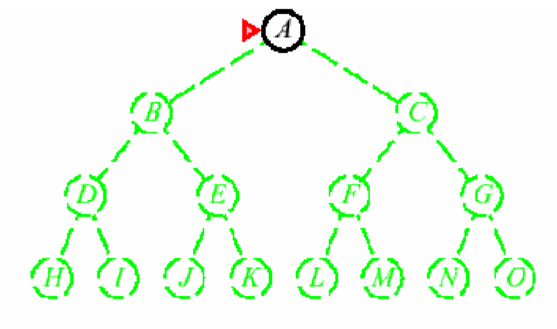


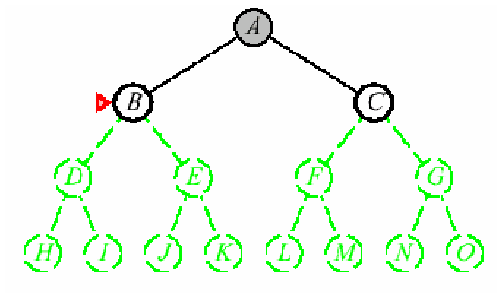


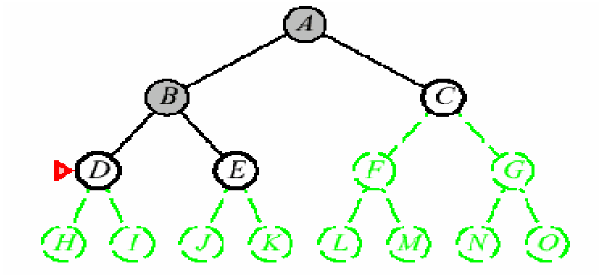
Properties of DFS(Describe more…)

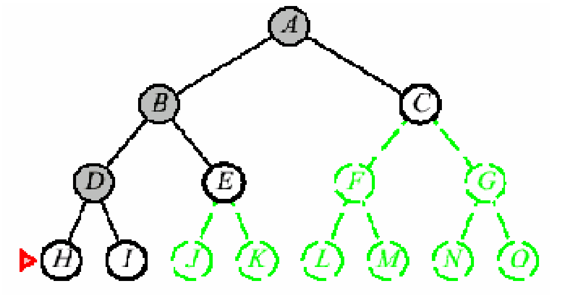
- Expand one of the node at the deepest level of the tree.
  - Only when the search hits a non-goal dead end does the search go back and expand nodes at shallower levels
- **Implementation**: treat the **list as LIFO stack**
  - Expansion: **push successors at the top of stack**
  - **Pop nodes** from the **top of the stack**

- Properties
  - **Incomplete** and **not optimal**: fails in infinite-depth spaces, spaces with loops.
    - Modify to avoid repeated states along the path
  - **Takes less space** (Linear): Only needs to remember up to the depth expanded









DFS:Advantage and Disadvantage

- **Advantage:**
  - DFS requires very less memory as it only needs to store a stack of the nodes on the path from root node to the current node.
  - It takes less time to reach to the goal node than BFS algorithm (if it traverses in the right path).
- **Disadvantage:**
  - There is the possibility that many states keep re-occurring, and there is no guarantee of finding the solution.
  - DFS algorithm goes for deep down searching and sometime it may go to the infinite loop.

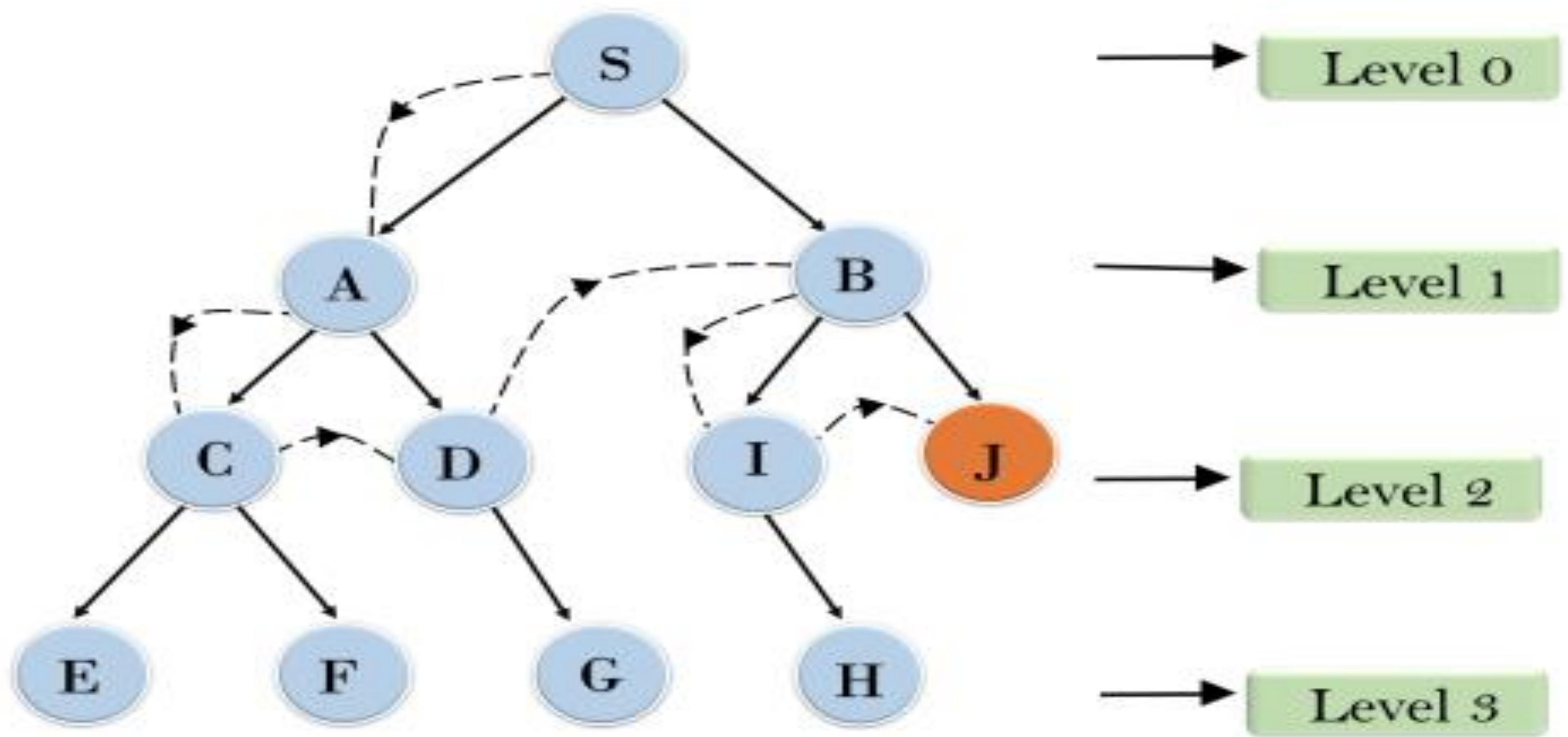
4. Depth-limited search

- A DLS- is similar to DFS with a predetermined limit.
- Depth-limited search can solve
  - *the drawback of the infinite path in the Depth-first search.*
- In this algorithm, the node at the depth limit will treat as it has no successor nodes further.
- DLS can be terminated with two Conditions of failure:
  - ***Standard failure value:*** It indicates that problem does not have any solution.
  - ***Cutoff failure value:*** It defines no solution for the problem within a given depth limit.

4. Depth-limited search(1)

- E.g: we had to find the shortest path to visit 10 cities. If we start at one of the cities, then there are at least nine other cities to visit. So nine is the limit.
- Since we restrict a limit, little changes from DFS with the exception that we will avoid searching an infinite path.
- DLS is now complete if the limit we impose is greater than or equal to the depth of our solution.

Example #1: Depth-limited search



4.Depth-limited search (2)

- **Advantages:**
  - Depth-limited search is Memory efficient.
  - E.g depth-first search with depth limit $l=2$

    i.e., never expand nodes at depth l=2.
- **Disadvantages:**
  - **Incomplete** if solution is below cut-off depth l
  - **Not optimal** if the problem has more than one solution

4. Depth-limited search (3)

Evaluation of DLS

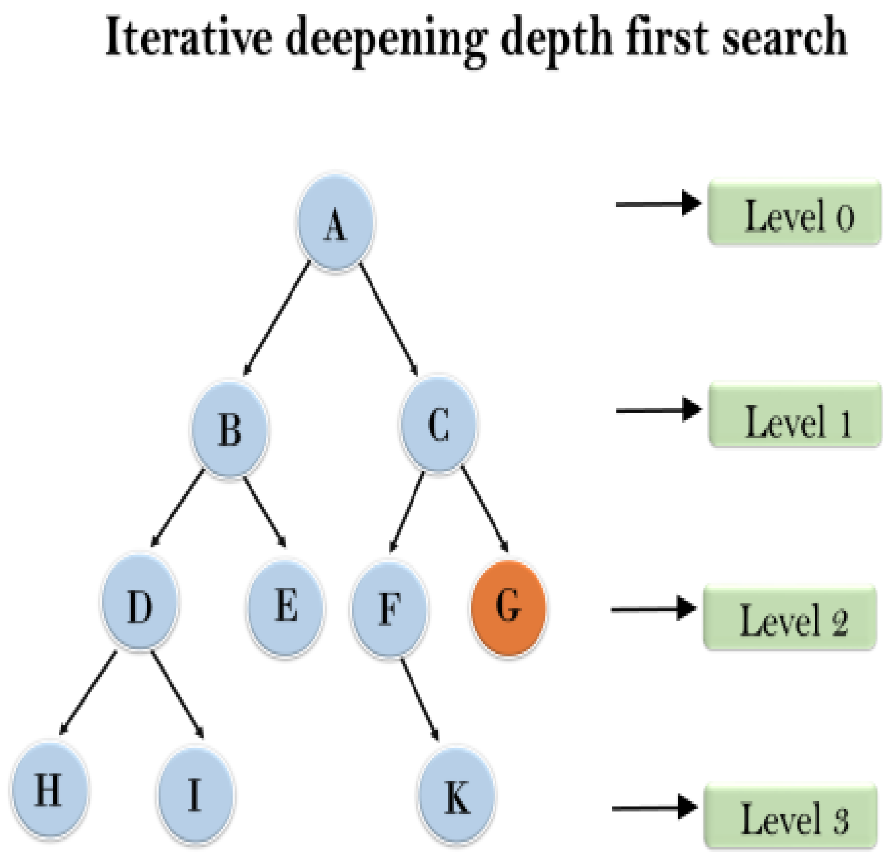
So how is it's time complexity and space complexity?

- DLS is O(b^{l}) where l is the limit we impose.
- Space complexity is O(b^{l}).
- So, space and time are almost identical to DFS.
- What about optimality? No.

5. Iterative Deeping Search (IDS)

- The **IDS**
  - is a combination of DFS and BFS algorithms to find the best solution.
- This algorithm performs DFS up to a certain "*depth limit"*, and it keeps increasing the depth limit after each iteration until the goal node is found.
- It does by gradually increasing the ***limit—first 0, then 1, then 2,*** and so on until a goal is found.
- IDS **uses only linear space** and ***not much more time*** than other uninformed algorithms.

Example #1



1'st Iteration-----> A

2'nd Iteration----> A, B, C

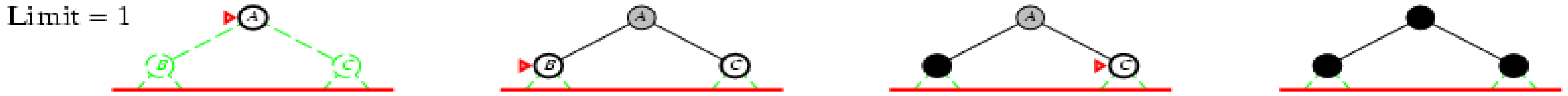
3'rd Iteration------>A, B, D, E, C, F, G

4'th Iteration------>A, B, D, H, I, E, C, F, K, G

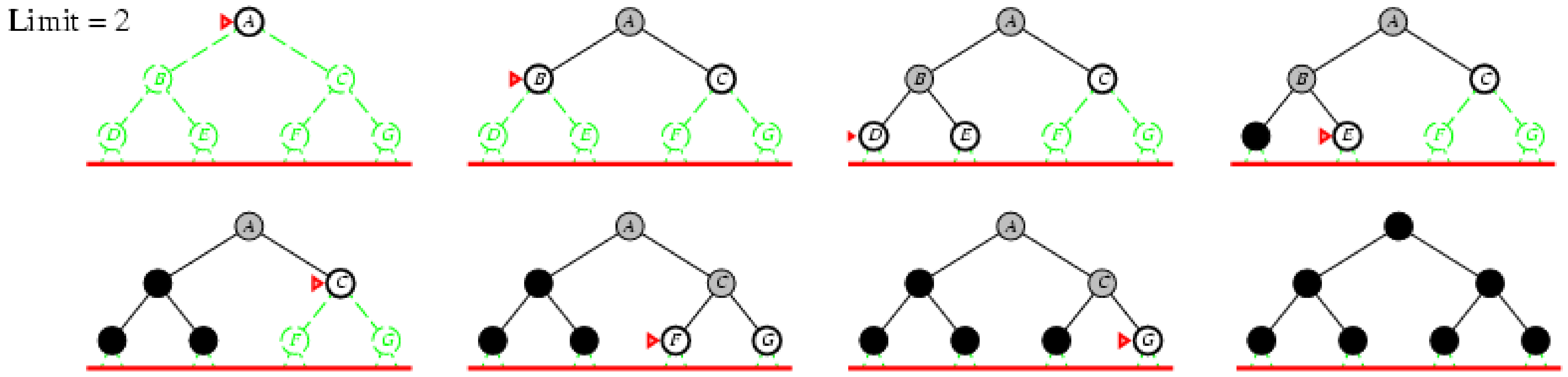
In the fourth iteration, the algorithm will find the goal node.

Example #2: Iterative Deepening search, *l* =0

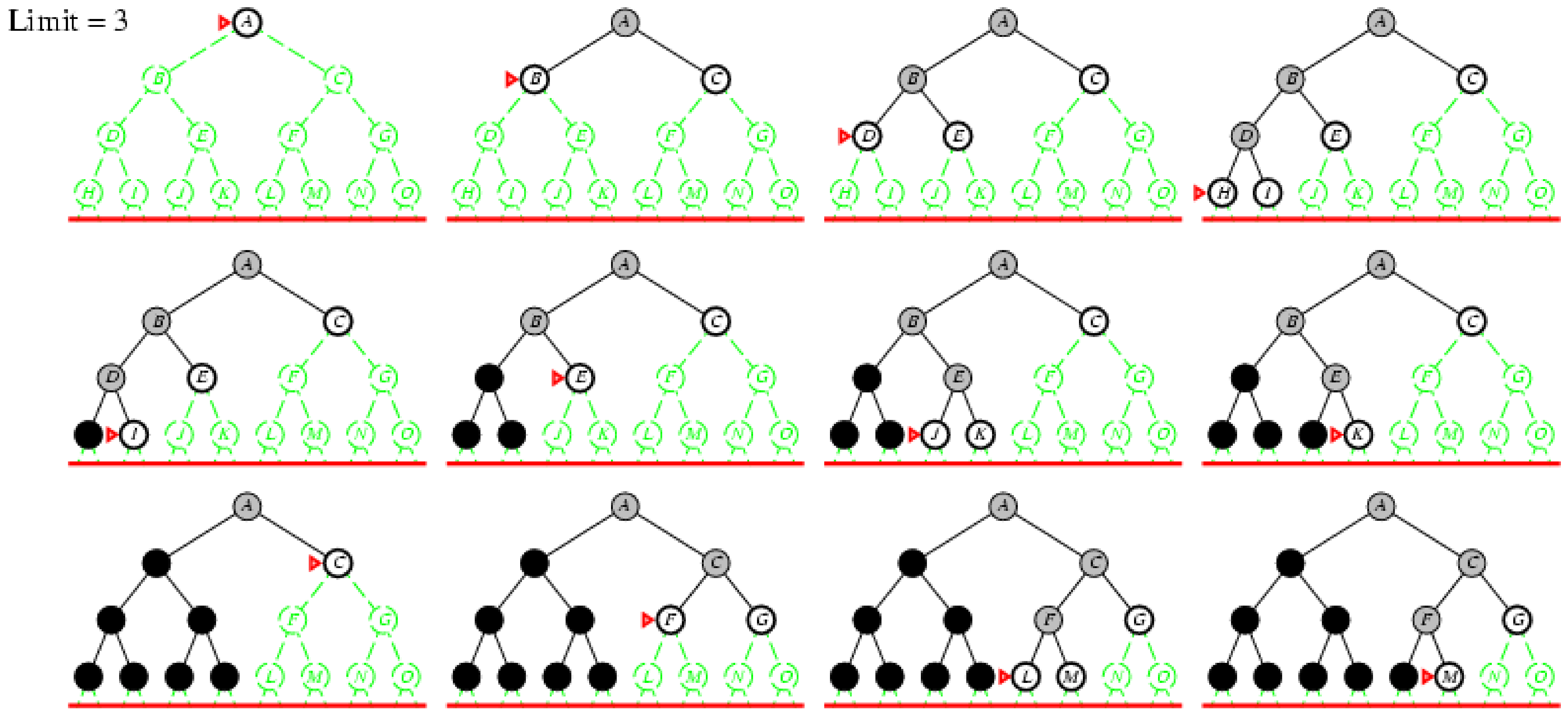
Iterative deepening search *l* =1



Iterative deepening search *l* =2



Iterative deepening search *l* =3



IDS: Advantage and Disadvantage

- **Advantages:**
  - It combines the benefits of BFS and DFS search algorithm in terms of *fast search* and *memory efficiency*.
  - *useful uninformed search when search space is large, and depth of goal node is unknown.*
- **Disadvantages:**
  - *it repeats all the work of the previous phase.*

Evaluating IDS

- ✓ We look at the bottom most nodes once, while the level above twice, and the level above that thrice and so on until the root.
- **Number of nodes** generated in a **depth-limited search** to depth *d* with branching factor *b*: $N_{DLS} = b^1 + b^2 + \ldots + b^d$
- **Number of nodes** generated in an **iterative deepening search** to depth *d* with branching factor *b*: $N_{IDS} = d\, b^1 + (d-1)b^2 + \ldots + 1b^d$
- E.g. for $b = 10$, $d = 5$,
  - $N_{DLS} = 10 + 100 + 1,000 + 10,000 + 100,000 = 111,110$
  - $N_{IDS} = 50 + 400 + 3,000 + 20,000 + 100,000 = 123,450$
- Like DLS, IDS performance is measured as follows:
  - Completeness: Yes
  - Optimality: yes
  - Time complexity: $O(b^d)$
  - Space Complexity: *bd*

6. Bidirectional Search

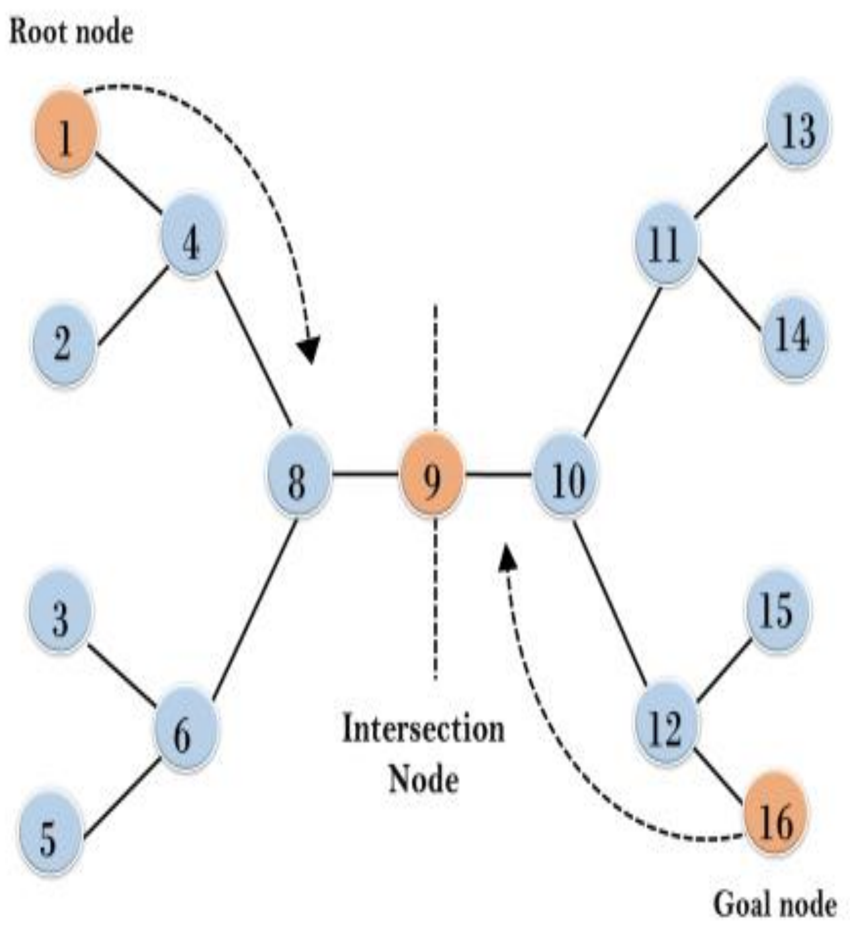
- **Bidirectional search** replaces one single search graph with two small sub-graphs at once:
  - **Forwards from initial state and Backwards from goal**
- The search stops when these two graphs intersect each other.
- **Usually done with *breadth-first searches*.**
- Idea is that $\mathbf{b^{d/2}+b^{d/2} < b^{d}}$ (time/space complexity)
- **Time complexity: $\mathbf{O(2b^{d/2}) = O(b^{d/2})}$**
- **Space complexity** is the same because for the two searches to meet at some point, all nodes need to be stored in memory.

✓ It is complete and optimal.

- Before expanding node, **check if it is in fringe of other search.**

Example #1: Bidirectional Search

- This algorithm divides one graph/tree into two sub-graphs.
- It starts traversing from node 1 in the ***forward direction*** and starts from goal node 16 in the ***backward direction.***
- The algorithm terminates at node 9 where two searches meet.



Bidirectional Search

- **Advantages:**
  - Only need to go to **half depth.**
  - It can enormously **reduce time complexity**, but is not always applicable.
  - Bidirectional search is fast.
  - Requires less memory
- **Difficulties:**
  - Do you really know solution? **Unique**?
  - Can **not reverse operators.**
  - Memory requirements may be important: Record all paths to check they meet
    - **Memory intensive**
- **Note** that if a heuristic function is inaccurate, the two searches might miss one another.

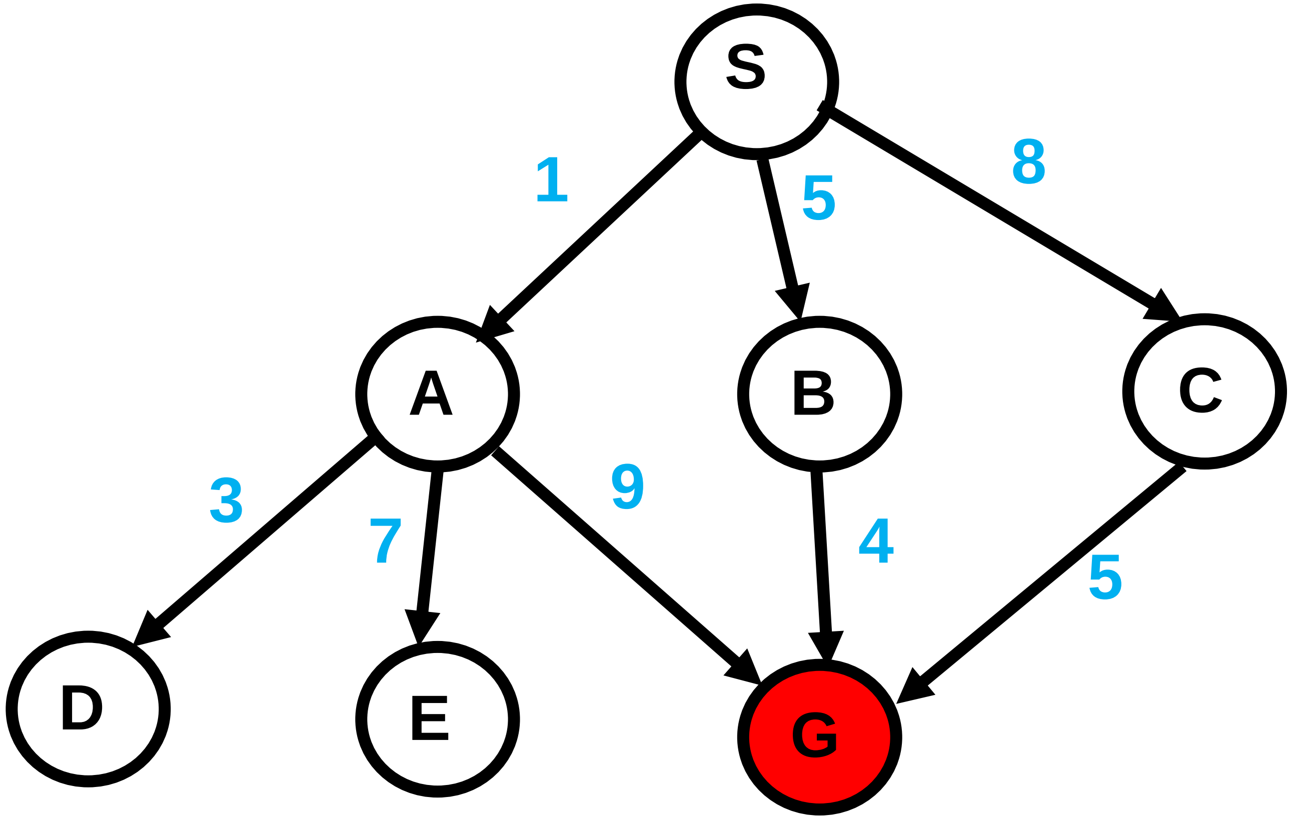
Comparing Uninformed Search

Search Strategies	Complete	Optimal	Time complexity	Space complexity
Breadth first search	yes	yes	$O(b^d)$	$O(b^d)$
Uniform cost search	yes	yes	$O(b^d)$	$O(b^d)$
Depth first search	no	no	$O(b^m)$	O(bm)
Depth limited search	if l >= d	no	$O(b^l)$	O(bl)
Iterative deepening search	yes	yes	$O(b^d)$	O(bd)
Bi-directional search	yes	yes	$O(b^{d/2})$	$O(b^{d/2})$

- **b** - is branching factor,
- **d** - is depth of the shallowest solution,
- **m** - is the maximum depth of the search tree,
- **l** - is the depth limit.

Exercise #1: Uninformed Search Strategies

- Assume that S is Start node and G is Goal node.



Informed Search Algorithms

- ***Uninformed search algorithms:-***
    - *Which looked through search space for all possible solutions of the problem without having any additional knowledge about search space.*
- But **informed search algorithm** contains an array of knowledge such as ***how far we are from the goal, path cost, how to reach to goal node,*** etc.
- This knowledge helps agents to explore less to the ***search space*** and ***find more efficiently the goal node.***
- The **informed search algorithm** is more useful for large search space.
- ***Informed search algorithm*** uses the idea of ***heuristic,*** so it is also called ***Heuristic search.***

Informed Search Algorithms…

- **Heuristic** is a function which is used in Informed Search, and it finds the most *promising path*.
- It takes the current state of the agent as its input and *produces the estimation of how close agent is from the goal.*
- The heuristic method, however, might not always give the ***best solution***, but it guaranteed to find a good solution in a reasonable time.
- Heuristic function estimates how close a state is to the goal.

Informed Search Algorithms ...

- It is represented by **h(n),** and it calculates the cost of an optimal path between the pair of states.
- The value of the heuristic function is always **positive.**
- Admissibility of the heuristic function is given as: h(n) <= h*(n)
- Here **h(n) is heuristic cost**, and **h*(n) is the estimated cost.**
- Hence **heuristic cost** should be less than or equal to the **estimated cost**.
- In the informed search we will discuss two main algorithms which are given below:
  - ***Best First Search Algorithm(Greedy search)***
  - ***A* Search Algorithm***

1.Best-first Search (Greedy Search):

- **BFS algorithm** always selects the path which appears best at that moment.
- It is the combination of DFS and BFS algorithms.
- It uses the heuristic function and search.
- **Best first Search** allows us to take the advantages of both algorithms.
- In the best first search algorithm, we expand the node which is closest to the goal node and the closest cost is estimated by ***heuristic function***
- i.e. **$f(n)= h(n)$.** Were, $h(n)=$ estimated cost from node n to the goal.
- The ***greedy best first algorithm*** is implemented by the ***priority queue.***

1.Best-first Search (Greedy Search):

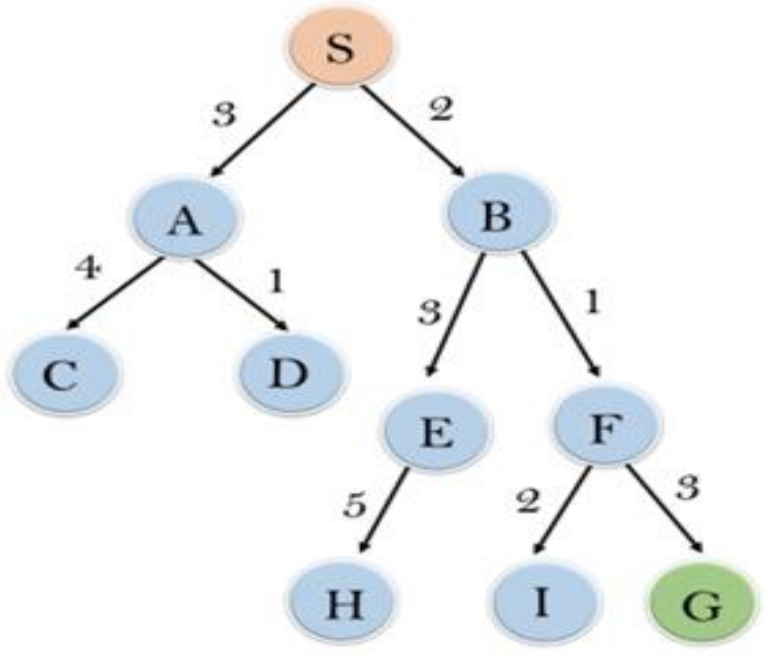
- It expands nodes based on their heuristic value **h(n) ,**we will need to use two lists of nodes:
  - **OPEN-** nodes that have been generated and have had the heuristic function applied to them, but which have not yet been examined (i.e., had their successors generated).
  - **CLOSED-** nodes that have already been examined.
  - We need to keep these nodes in memory if we want to search a graph rather than a tree, since whenever a new node is generated, we need to check whether it has been generated before.

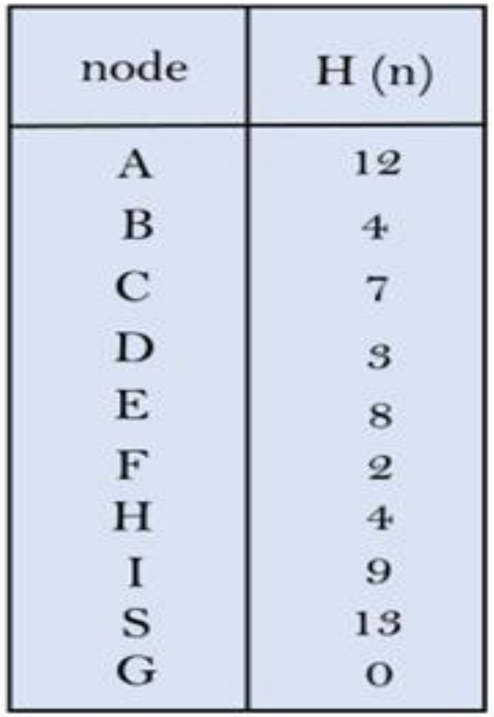
Algorithm: Best first search

- **Step 1:** Place the starting node into the **OPEN list**.
- **Step 2:** If the **OPEN list** is empty, Stop and return failure.
- **Step 3:** Remove the node n, from the **OPEN** list which has the lowest value of h(n), and places it in the **CLOSED** list.
- **Step 4:** Expand the node n, and generate the successors of node n.
- **Step 5:** Check each successor of node n, and find whether any node is a goal node or not. If any successor node is goal node, then return success and terminate the search, else proceed to Step 6.
- **Step 6:** For each successor node, algorithm checks for evaluation function f(n), and then check if the node has been in either **OPEN** or **CLOSED** list.
- If the node has not been in both list, then add it to the OPEN list.
- **Step 7:** Return to Step 2.

BFS: Example #1

- Consider the below search problem, and we will traverse it using ***greedy best-first search.***
- At each iteration, each node is expanded using evaluation function **f(n)=h(n)** , which is given in the below table.

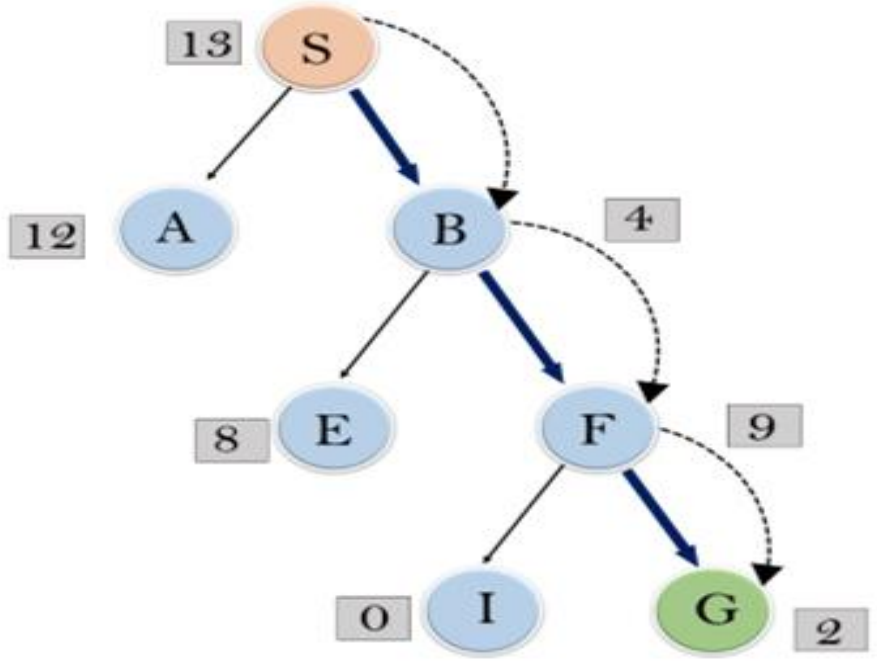




node	H (n)
A	12
B	4
C	7
D	3
E	8
F	2
H	4
I	9
S	13
G	0

BFS: Example #1…

- In this search example, we are using two lists which are **OPEN** and **CLOSED** Lists.
- Following are the iteration for traversing the above example.



- **Expand the nodes of S and put in the CLOSED list**
- **Initialization:** Open [A, B], Closed [S]
- **Iteration 1:** Open [A], Closed [S, B]
- **Iteration 2:** Open [E, F, A], Closed [S, B]
  : Open [E, A], Closed [S, B, F]
- **Iteration 3:** Open [I, G, E, A], Closed [S, B, F]
  : Open [I, E, A], Closed [S, B, F, G]
- Hence the final solution path will be:

  S----> B----->F----> G

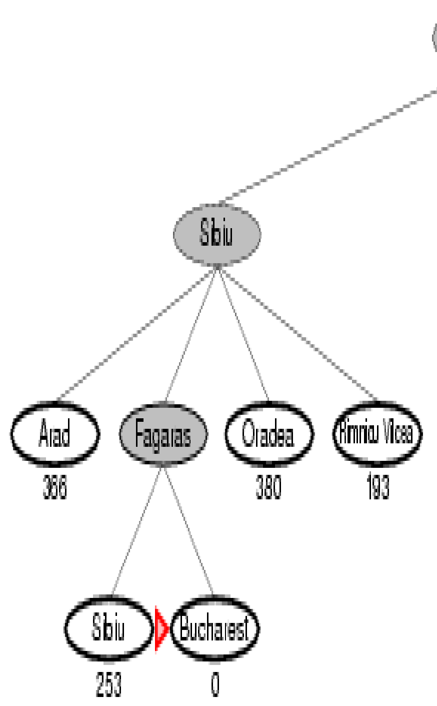
BFS: Advantages and Disadvantages

- **Advantages:**
  - ✓ Best first search can switch between BFS and DFS by gaining the advantages of both the algorithms.
  - ✓ This algorithm is more efficient than BFS and DFS algorithms.
- **Disadvantages:**
  - ✓ It can behave as an unguided depth-first search in the worst case scenario.
  - ✓ It can get stuck in a loop as DFS.
  - ✓ This algorithm is not **optimal**.

Properties of Best-First Search (Greedy)

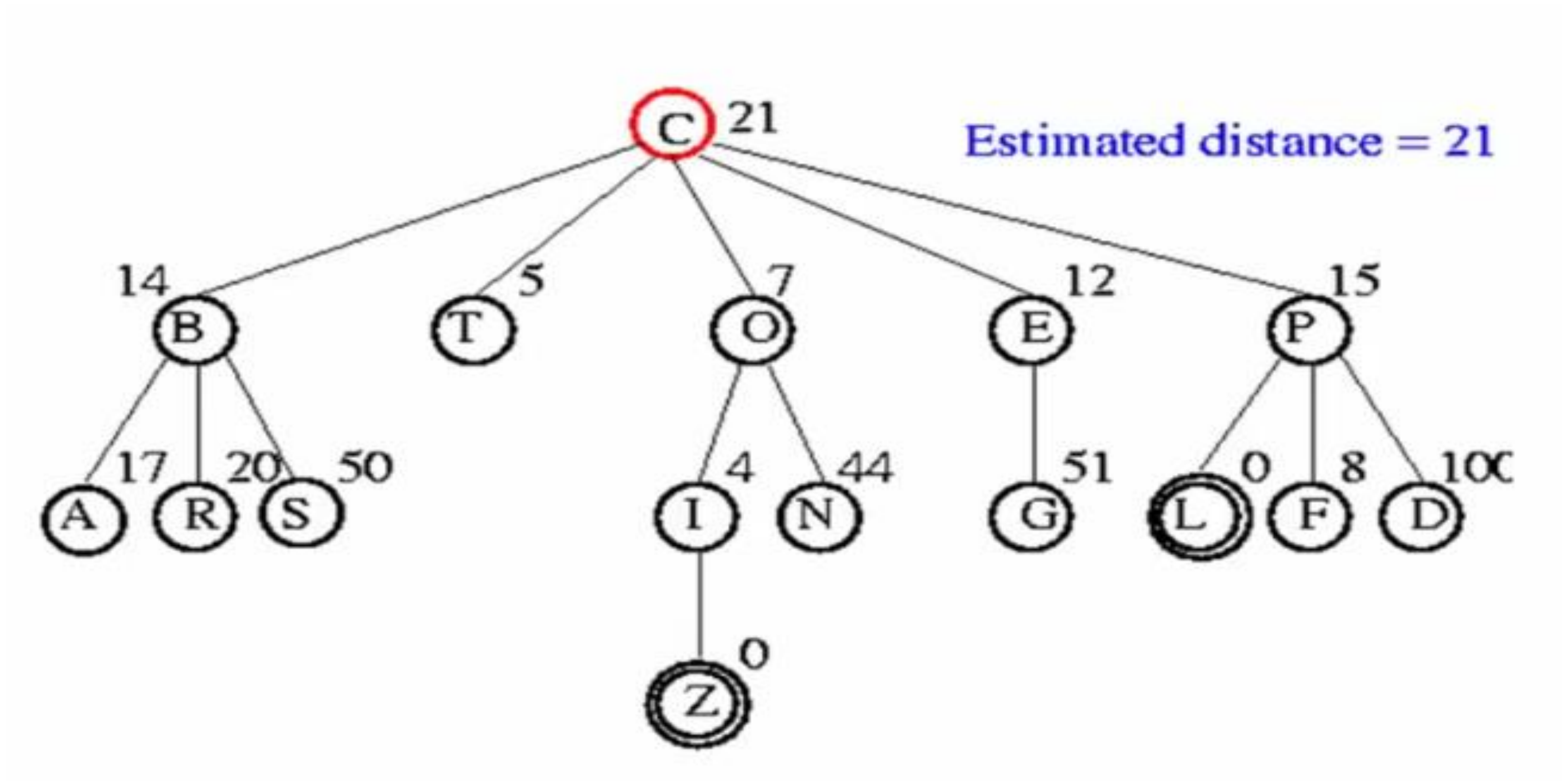
- **Time Complexity:**
  - ✓ **O(b^m)** , but a good heuristic can give dramatic improvement
- **Space Complexity:**
  - ✓ O(b^m). Where, m is the maximum depth of the search space.
  - ✓ keeps all nodes in memory
- **Complete:**
  - ✓ incomplete, even if the given state space is finite.
- **Optimal:**
  - ✓ ***not optimal.***

E.g: hSLD(n) = straight-line distance from *n to Bucharest city.*



BFS–Solved Exercise #1

- Find the path from C to Z or L
- Here C is the initial or Source node and L and Z are goal nodes.

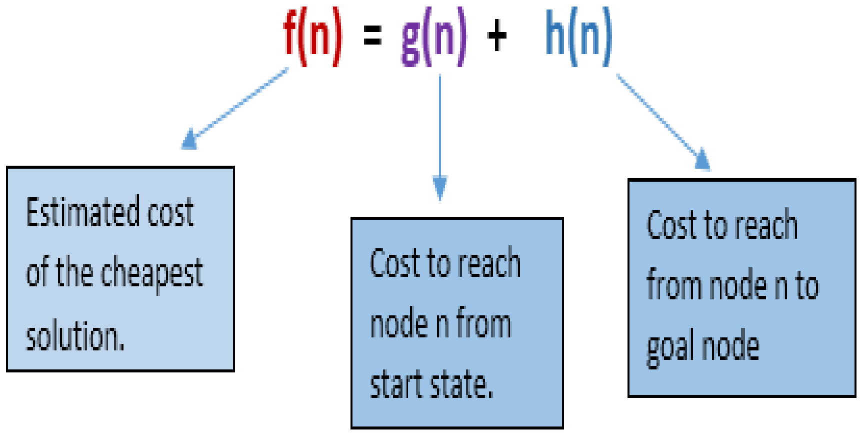


2. A* Search Algorithm

- **A* search** is the most commonly known form of ***best-first search.***
- It uses **heuristic function h(n),** and cost to reach the node n from the start state **g(n).**
- It has combined features of **UCS** and **greedy best-first search**, by which it solve the problem efficiently.
- **A* search algorithm** finds the shortest path through the search space using the heuristic function.
- This search algorithm expands less search tree and provides optimal result faster.
- A* algorithm is similar to UCS except that it uses **g(n)+h(n)** instead of g(n).

2. A* Search Algorithm….

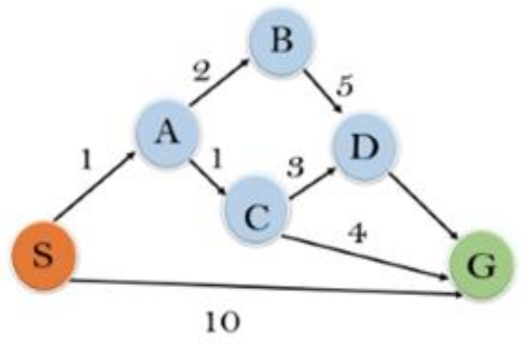
- In A* search algorithm, we use search heuristic as well as the cost to reach the node.
- Hence we can combine both costs as following, and this sum is called as a **fitness number**.

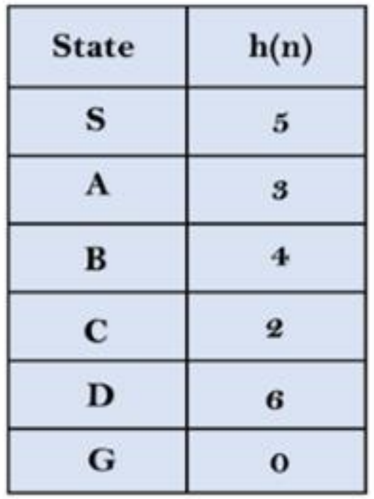

f(n) = g(n) + h(n)
Estimated cost of the cheapest solution.
Cost to reach node n from start state.
Cost to reach from node n to goal node

Note: At each point in the search space, only those node is expanded which have the lowest value of f(n), and the algorithm terminates when the goal node is found.

A* search : Example

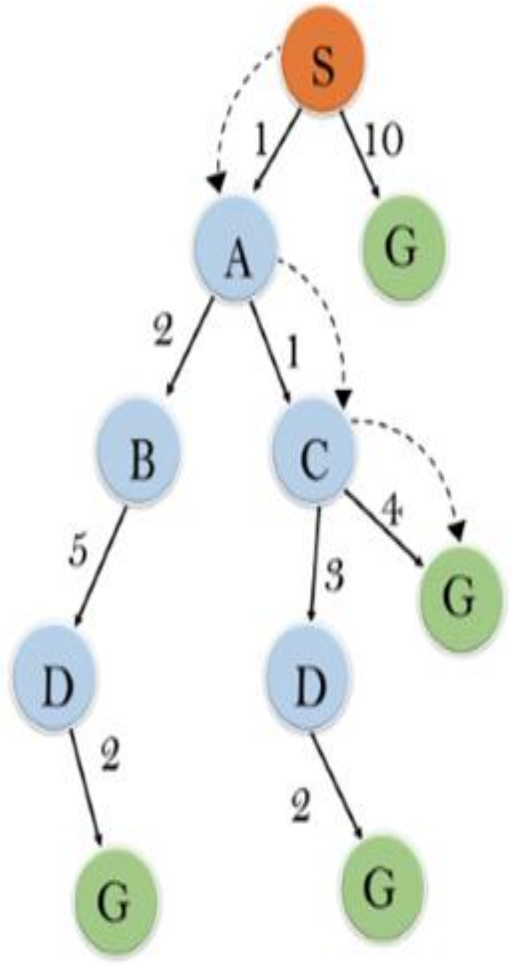
- In this example, we will traverse the given graph using the A* algorithm.
- The heuristic value of all states is given in the below table so we will calculate the f(n) of each state using the formula **f(n)= g(n) + h(n),** where g(n) is the cost to reach any node from start state.
- Here we will use OPEN and CLOSED list.





State	h(n)
S	5
A	3
B	4
C	2
D	6
G	0

A* search : **Solution**



Initialization: {(S, 5)}

Iteration1:

{(S--> A, 4), (S-->G, 10)}

Iteration2:

{(S--> A-->C, 4), (S--> A-->B, 7), (S-->G, 10)}

Iteration3:

{(S--> A-->C--->G, 6), (S--> A-->C--->D, 11), (S--> A-->B, , (S-->G, 10)}

Iteration 4

the final result, as

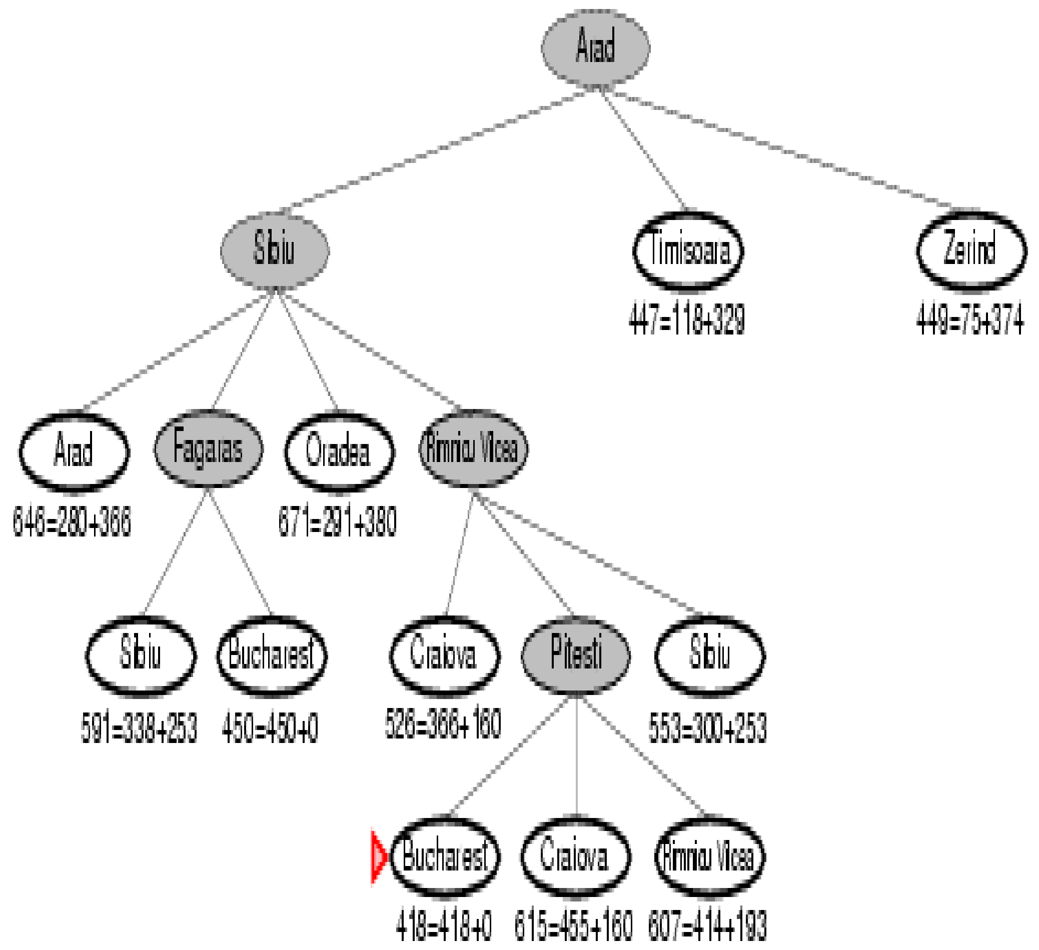
S--->A--->C--->G it provides the optimal path with cost 6.

A* search: Advantages and Disadvantages:

- **Advantages:**
  - A* search algorithm is the best algorithm than other search algorithms.
  - A* search algorithm is optimal and complete.
  - This algorithm can solve very complex problems.
- **Disadvantages:**
  - It does not always produce the shortest path as it mostly based on heuristics and approximation.
  - A* search algorithm has some complexity issues.
  - The memory requirement as it keeps all generated nodes in the memory, so it is not practical for various large-scale problems.

Properties of A* search [Ref:textbook]

- **Complete:** A* algorithm is complete as long as:
  - Branching factor is finite.
  - Cost at every action is fixed.
- **Optimal:** Yes
- **Time Complexity:** Exponential
  - **O(b^d),** where b is the branching factor.
  - depends on heuristic function, and the number of nodes expanded is exponential to the depth of solution d.
- **Space Complexity:**
  - **O(b^d)** Keeps all nodes in memory



Chapter Exercise

1. Explain why problem formulation must follow goal formulation?
2. Define in your own words the following terms: state, state space, search tree, search node, goal, action, transition model, and branching factor.
3. What’s the difference between a world state, a state description, and a search node? Why is this distinction useful?

Assignment #2

- Memory-bounded heuristic search
- Avoiding Repeated States
- Constraint satisfaction problems (CSPs)
- Hill Climbing Algorithm in Artificial Intelligence

The end of Chapter 3!