Cleaver Search Algorithms

Lecture 2

Structures for State Space Search

Graph Theory

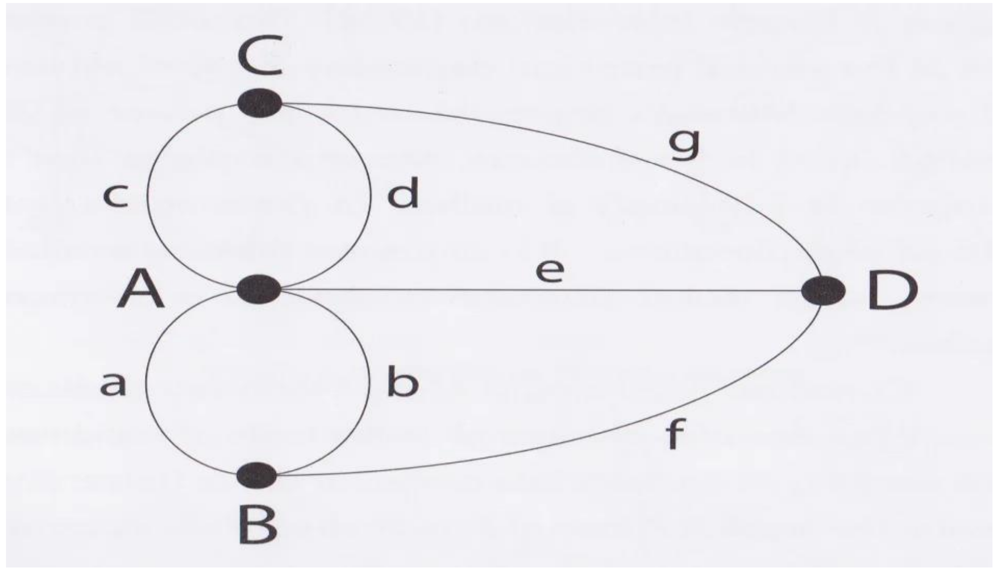
- A *graph* is a set of ***nodes*** or *states* and a set of *arcs* that connect the nodes. **A *labeled* graph** has one or more descriptors (labels) attached to each node that distinguish that node from any other node in the graph.
- In a *state space graph*, these descriptors identify states in a problem-solving process.
- If there are no descriptive differences between two nodes, they are considered the same.
- The arc between two nodes is indicated by the labels of the connected nodes.

Structures for State Space Search

Graph Theory

- The arcs of a graph may also be labeled.
- Arc labels are used to indicate that an arc represents a named relationship (as in a semantic network) or to attach weights to arcs (as in the traveling salesperson problem).

- If there are different arcs between the same two nodes (as in Figure aside), these can also be distinguished through labeling



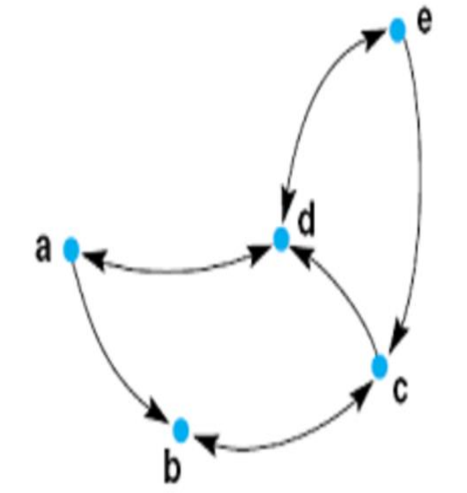
Graph of the Konigsberg bridge system.

Structures for State Space Search

Graph Theory

- A graph is **directed** if arcs have an **associated directionality.**
- The arcs in a **directed graph** are usually drawn as arrows or have an arrow attached to indicate direction.
- Arcs that can be crossed in either direction may have two arrows attached but more often have no direction indicators at all.

- Figure aside is a labeled, directed graph: arc (a, b) may only be crossed from node a to node b, arc (b, c) is crossable in either direction.
- A path through a graph connects a sequence of nodes through successive arcs.
- The path is represented by an ordered list that records the nodes in the order they occur in the path.
- In the Figure, [a, b, c, d] represents the path through nodes a, b, c, and d, in that order



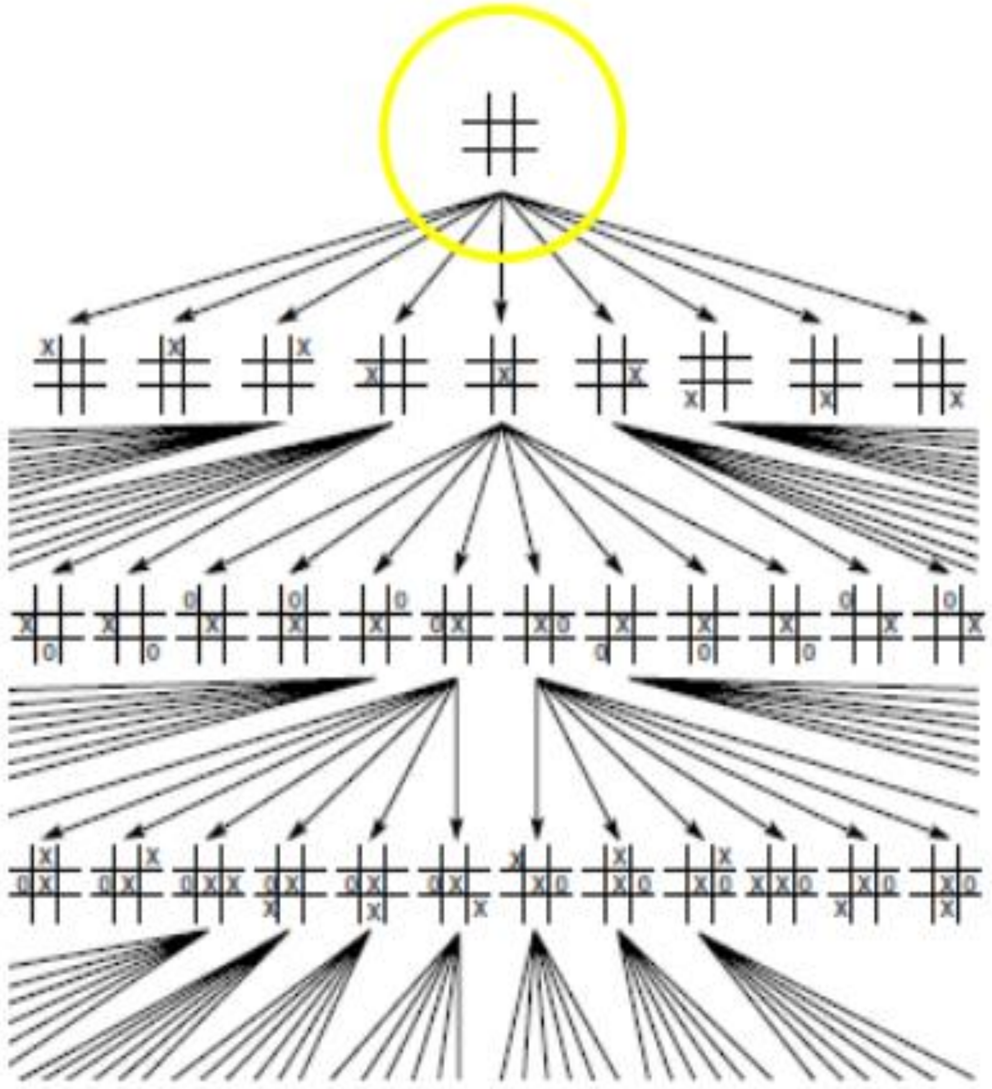
Nodes = {a,b,c,d,e}

Arcs = {(a,b),(a,d),(b,c),(c,b),(c,d),(d,a),(d,e),(e,c),(e,d)}

Structures for State Space Search

Graph Theory

- A *rooted* graph has a unique node, called the *root*, such that there is a path from the root to all nodes within the graph.
- in rooted graph, the root is usually drawn at the top of the page, above the other nodes.
- The state space graphs for games are usually rooted graphs, with the start of the game as the root.
- The initial moves of the tic-tac-toe game graph are represented by the rooted graph of Figure.
- This is a directed graph with all arcs having a single direction.
- this graph contains no cycles; players cannot (as much as they might sometimes wish!) undo a move.



Portion of the state space for tic-tac-toe.

Structures for State Space Search

Graph Theory (Trees)

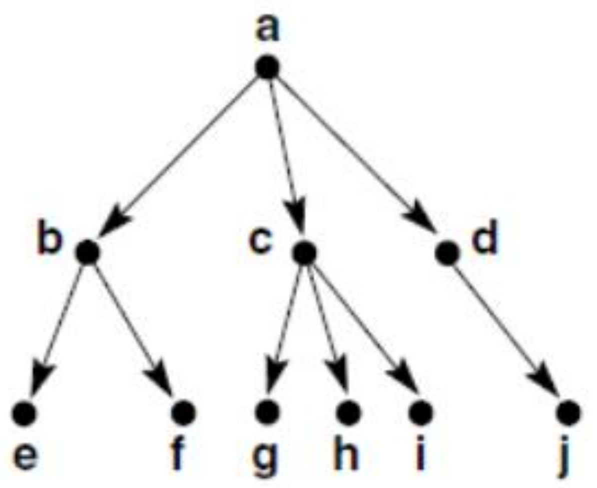
- A ***tree*** is a graph in which two nodes have at most one path between them.
- Trees often have roots, in which case they are usually drawn with the root at the top, like a rooted graph.
- Because each node in a tree has only one path of access from any other node, it is impossible for a path to loop or cycle through a sequence of nodes.
- For rooted trees or graphs, relationships between nodes include parent, child, and sibling.

Structures for State Space Search
Graph Theory (Trees)

- These are used in the usual familial fashion with the parent preceding its child along a directed arc. The children of a node are called siblings.
- Similarly: an ancestor(predecessor) comes before a descendant (successor) in some path of a directed graph.

In the Figure:

- b is a parent of nodes e and f.
- f children of b
- e and f are siblings of each other.
- Nodes a and c are ancestors of states g, h.
- g, h, and i are descendants of a and c.

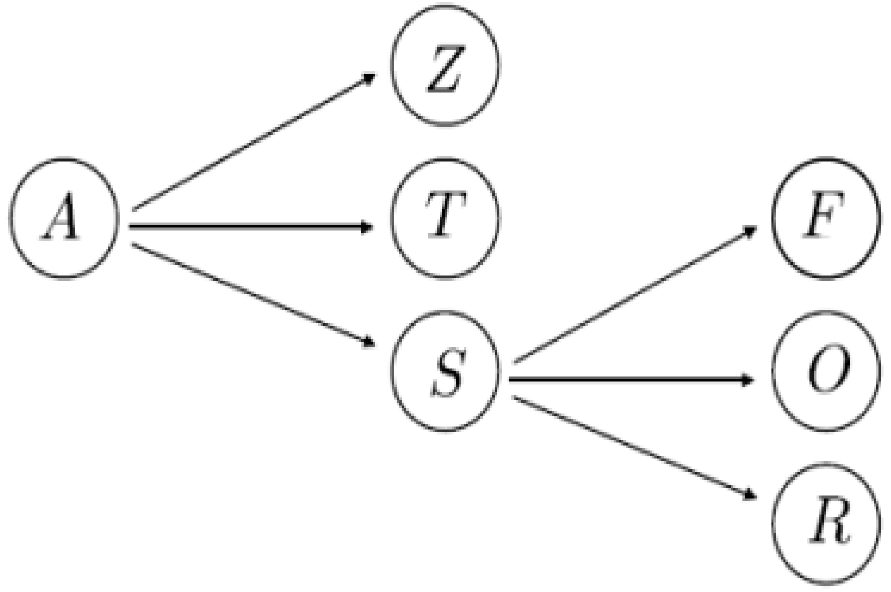


A rooted tree, exemplifying family relationship

Search Tree

- In what order should we expand states?
- Here, we expanded S, but we could also expanded Z and T
- **Different search algorithms expand in different orders.**

- Search algorithms vary primarily according to how they choose which state to expand next so called search strategy.



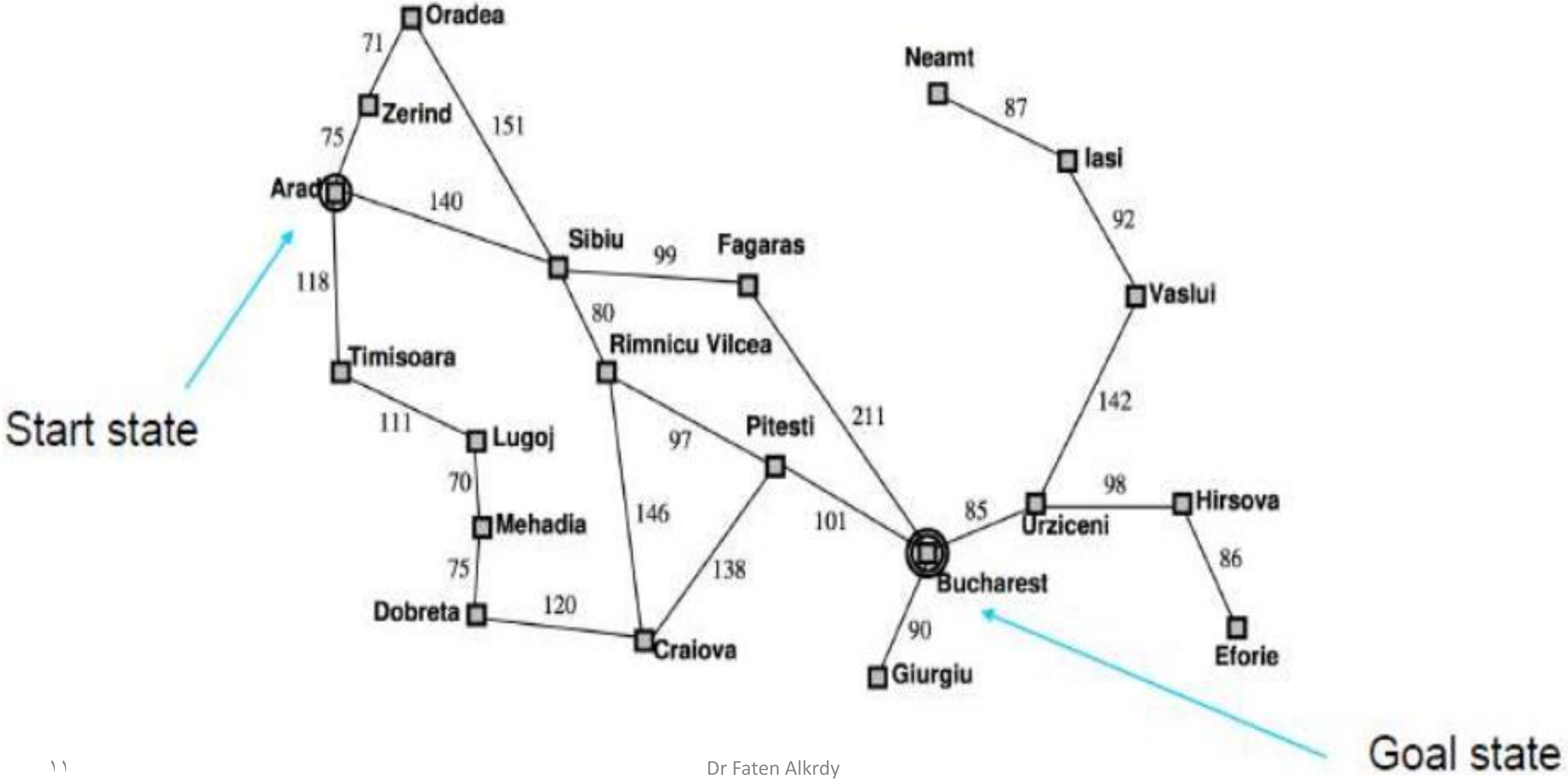
Searching for problems solutions search tree

- A solution is an action sequence, so search algorithms work by considering various possible action sequences.
- SEARCH TREE :The possible action sequences starting at the initial state form a search tree with the initial state NODE at the root
- the branches are actions and the nodes correspond to states in the state space of the problem.
- in Romania Example:
- The root node of the tree corresponds to the initial state, In(Arad).
- The first step is to test whether this is a goal state. (Clearly it is not, but it is important).
- we need to consider taking various actions.

Searching for problems solutions search tree

- We do **expanding** the current state; that is, applying each legal action to the current state, thereby **generating** a new set of states.
- in this case, we add three branches from the **parent node** *In(Arad)* leading to three new **child nodes**: *In(Sibiu), In(Timisoara),* and *In(Zerind).*
- Now we must choose which of these three possibilities to consider further.
- the essence of search following up one option now and putting the others aside for later, in case the first choice does not lead to a solution.
- Suppose we choose Sibiu first. We check to see whether it is a goal state (it is not) and then expand it to get *In(Arad), In(Fagaras), In(Oradea),* and *In(RimnicuVilcea).* We can then choose any of these four or go back and choose Timisoara or Zerind.
- Each of these six nodes is a **leaf node**, that is, a node with no children in the tree.

Problem formulate



Searching for problems solutions search tree

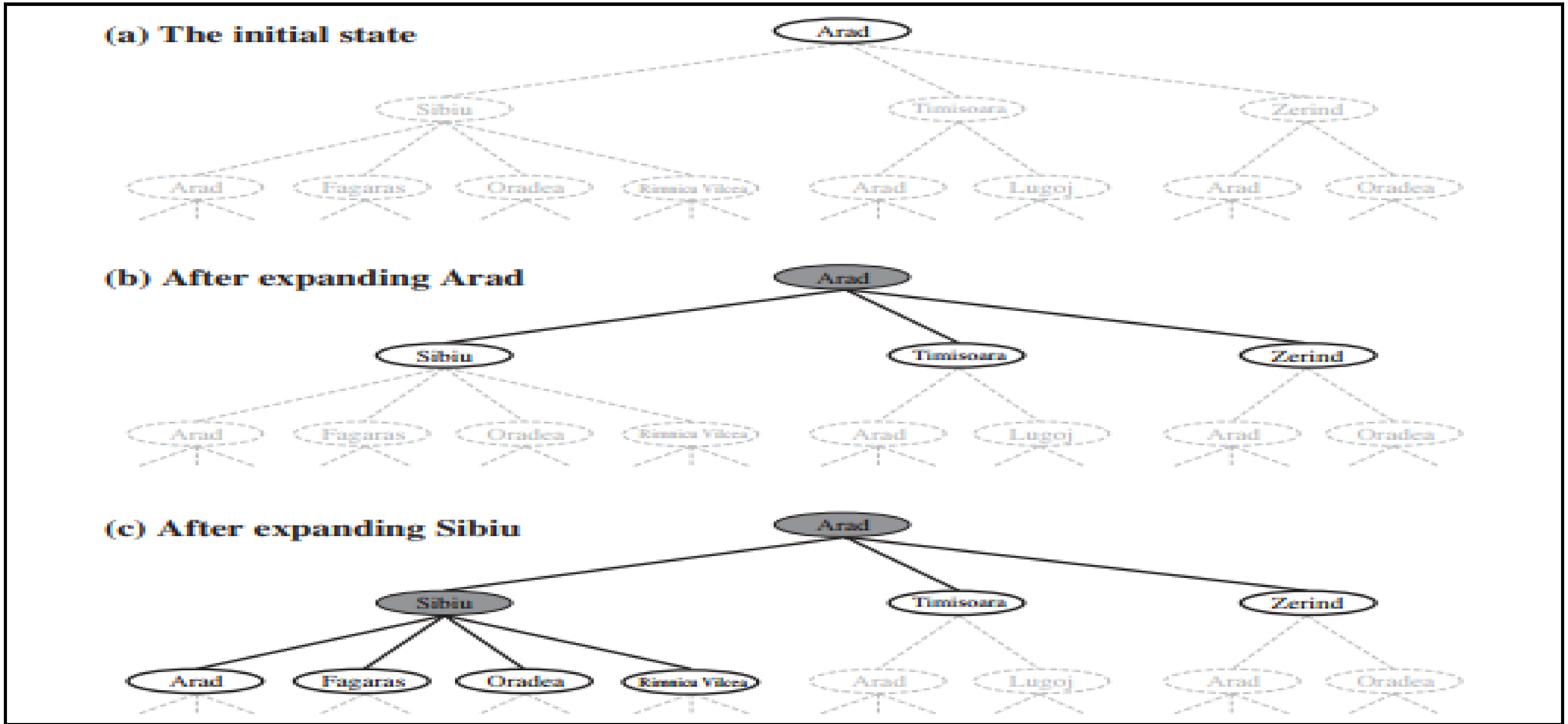


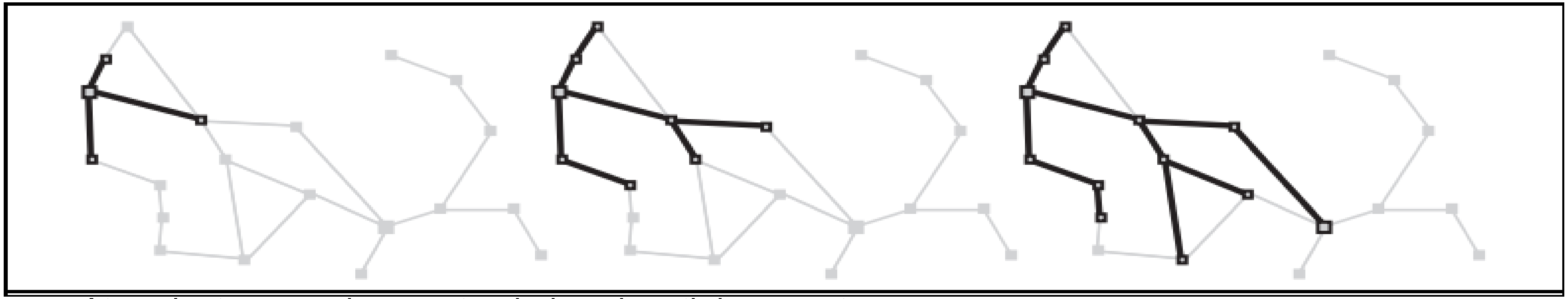
Figure 3.6 Partial search trees for finding a route from Arad to Bucharest. Nodes that have been expanded are shaded; nodes that have been generated but not yet expanded are outlined in bold; nodes that have not yet been generated are shown in faint dashed lines.

Searching for problems solutions
search tree

- The set of all leaf nodes available for expansion at any given point is called the **frontier**.
- In the past figure, the frontier of each tree consists of those nodes with bold outlines.
- Figure 3.6 it includes the path from Arad to Sibiu and back to Arad again! We say that *In(Arad)* is a **repeated state** in the search tree, generated in this case by a **loopy path**.
- loopy paths means that the complete search tree for Romania is *infinite* because there is no limit to how often one can traverse a loop.
- the map shown in Figure 3.2—has only 20 states. loops can cause certain algorithms to fail.
- Loopy paths are redundant paths, which exist whenever there is more than one way to get from one state to another.
- If you are concerned about reaching the goal, there’s never any reason to keep more than one path to any given state, because any goal state that is reachable by extending one path is also reachable by extending the other.

Searching for problems solutions
search tree

- search tree constructed by the GRAPH-SEARCH algorithm contains at most one copy of each state, so we can think of it as growing a tree directly on the state-space graph, as shown in the next Figure .
- the frontier **separates** the state-space graph into the explored region and the unexplored region, so that every path from the initial state to an unexplored state has to pass through a state in the frontier.



At each stage, we have extended each path by one step.

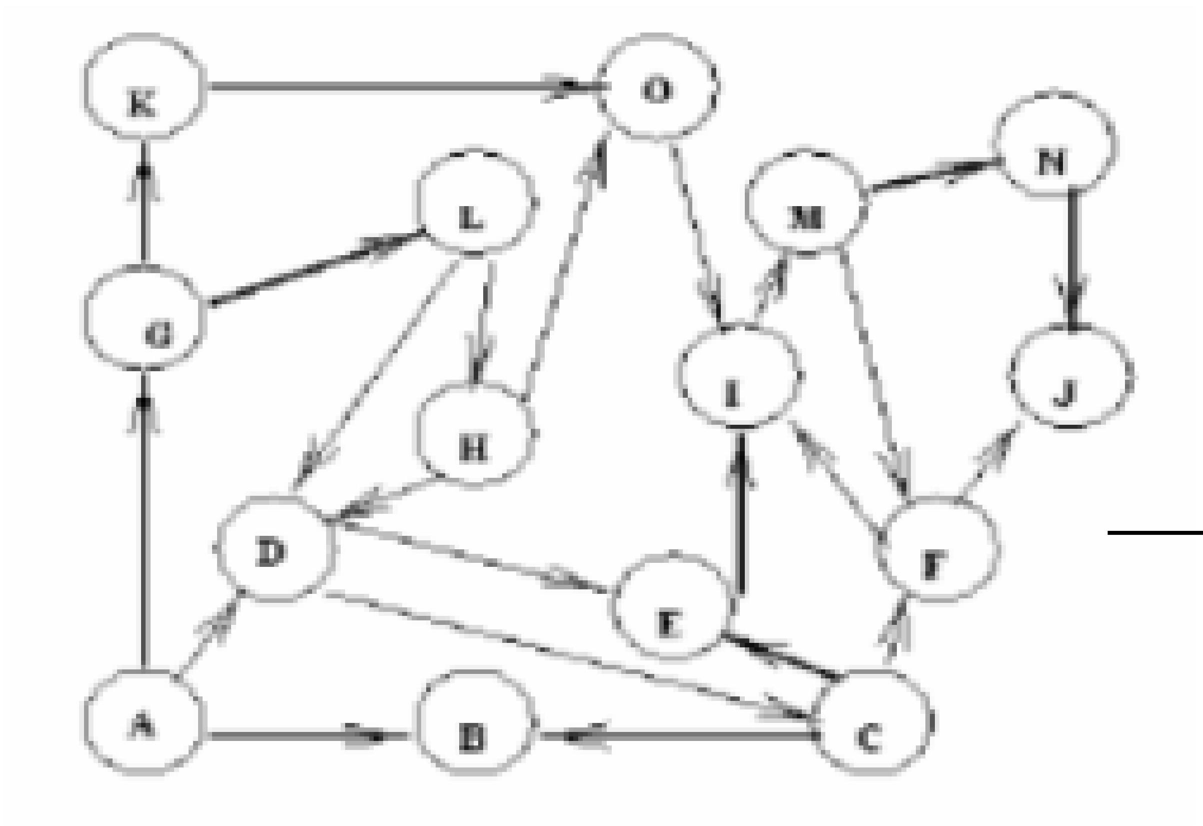
Searching for problems solutions
search tree

- The way to avoid exploring redundant paths is to remember where one has been.
- To do this, we augment the TREE-SEARCH algorithm with a data structure called the **explored set** (also known as the **closed list**), which remembers every expanded node.
- Newly generated nodes that match previously generated nodes—ones in the explored set or the frontier can be discarded instead of being added to the frontier.

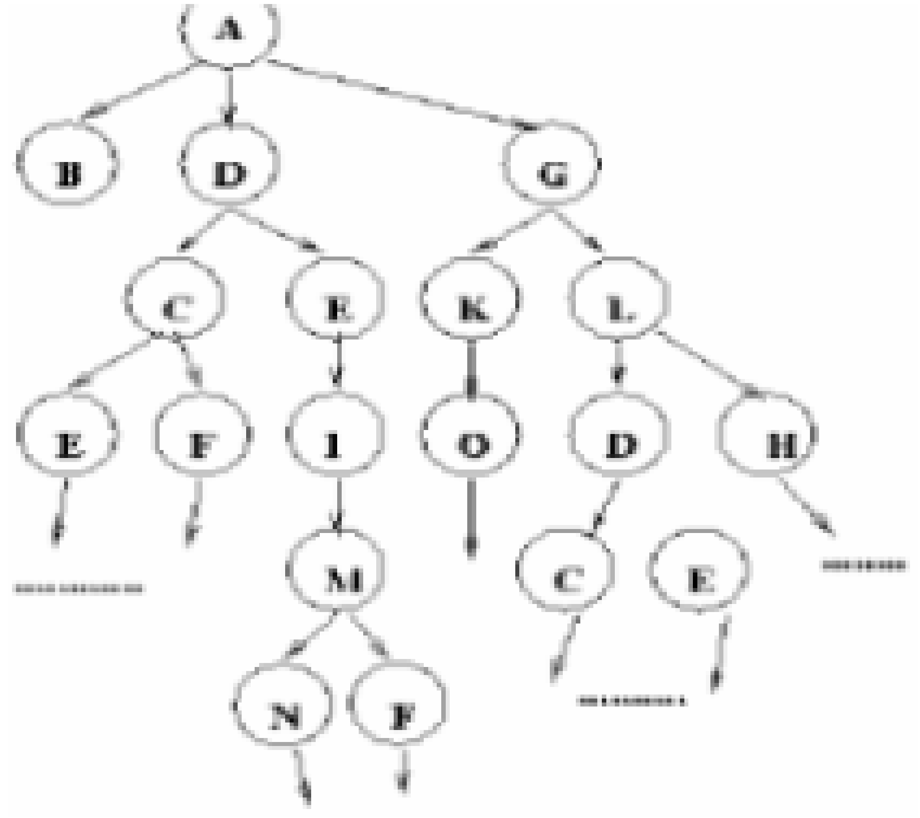
The State Space Representation of Problems

- **It is, therefore, important to distinguish between problems whose state space is a tree and those that may contain loops.**
- General graph search algorithms must detect and eliminate loops from potential solution paths.
- tree searches may gain efficiency by eliminating this test and its overhead.

Searching for problems solutions search tree

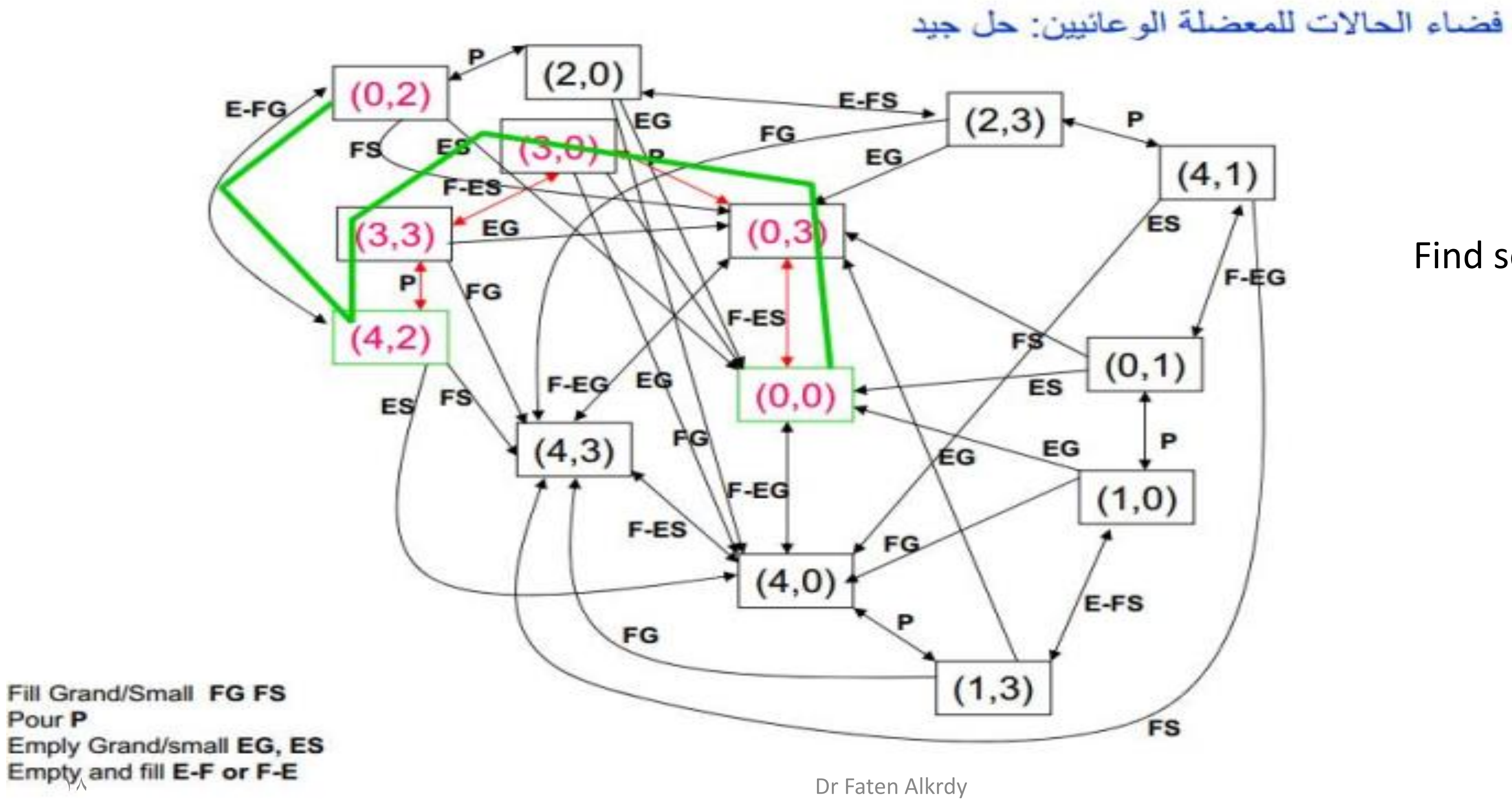


A proplem



Search tree for the proplem

Searching for problems solutions search tree



Find search tree!!!

classes of search

Based on the search problems, we can classify the search algorithm as Four classes of search :

- **uninformed search**
- **informed search**
- **constraint satisfaction**, which tries to find a set of values for a set of variables.
- **adversarial search**, which is used in games to find effective strategies to play and win two-player games.

classes of search
Uninformed search algorithms

- A problem determines the graph and the goal, but not which path to select from the frontier, This is the job of a search strategy.

- A search strategy specifies which paths are selected from the frontier.
- Different strategies are obtained by modifying how the selection of paths in the frontier is implemented.

- **Uninformed Search** algorithms have no additional information on the goal node other than the one provided in the problem definition.

- The plans to reach the goal state from the start state differ only by the order and length of actions.

classes of search
Uninformed search algorithms

- The uninformed search algorithm does not have any domain knowledge such as closeness, location of the goal state, etc.
- it behaves in a brute-force way.

- It only knows the information about how to traverse the given tree and how to find the goal state.

- This algorithm is also known as the Blind search algorithm or Brute -Force algorithm.

classes of search
Uninformed search algorithms

- **brute force algorithm are:**

  It is an intuitive[natural], direct, and straight forward technique of problem-solving in which all the possible ways or all the possible solutions to a given problem are enumerated.

- Many problems solved in day-to-day life using the brute force strategy, for example: exploring all the paths to a nearby market to find the minimum shortest path.

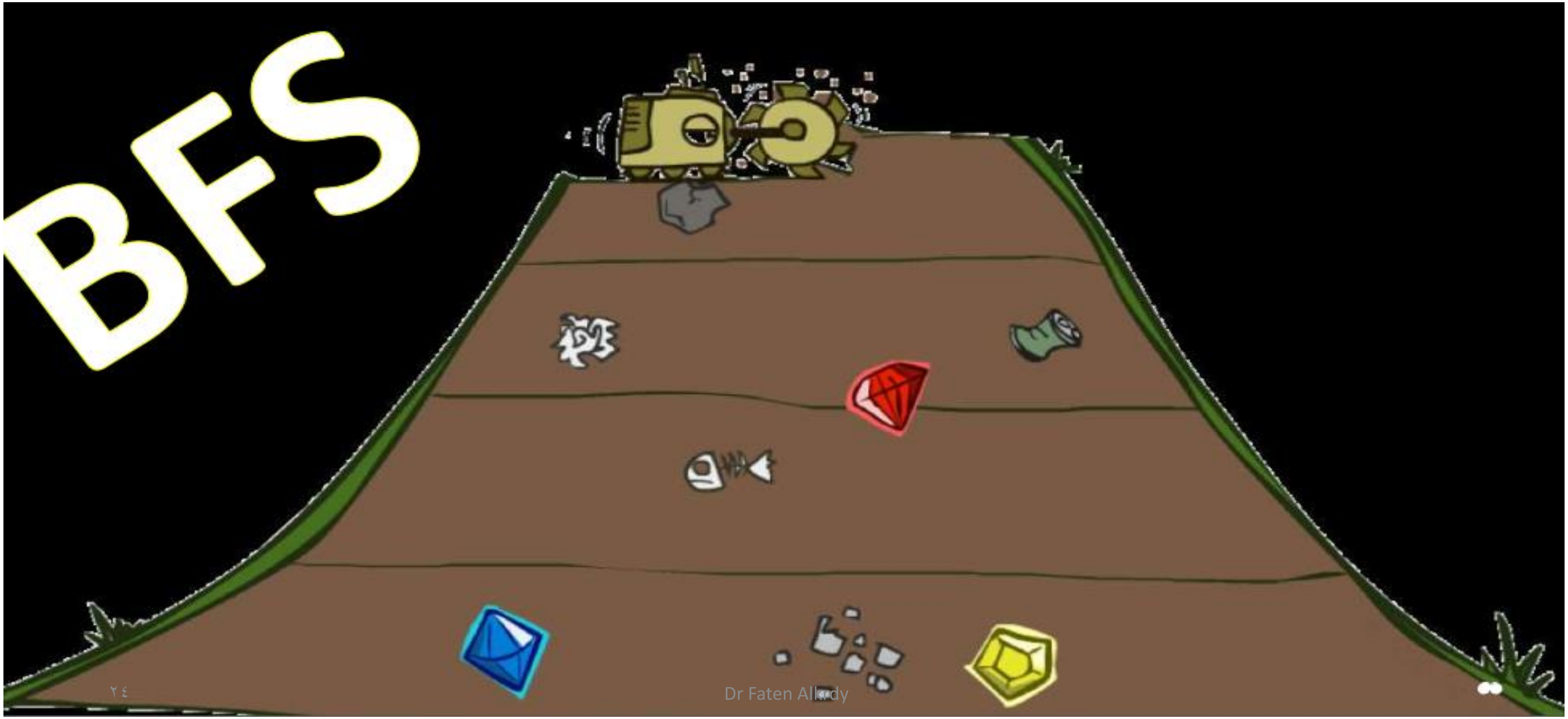
- In fact, daily life activities use a brute force nature, even though optimal algorithms are also possible.

classes of search
Uninformed search algorithms

The uninformed search strategies are of six types. They are-

- Breadth-first search
- Depth-first search
- Depth-limited search
- Iterative deepening depth-first search
- Bidirectional search
- Uniform cost search

Uninformed search algorithms
Breadth First Search Algorithm (BFS)



Uninformed search algorithms

Breadth First Search Algorithm (BFS)

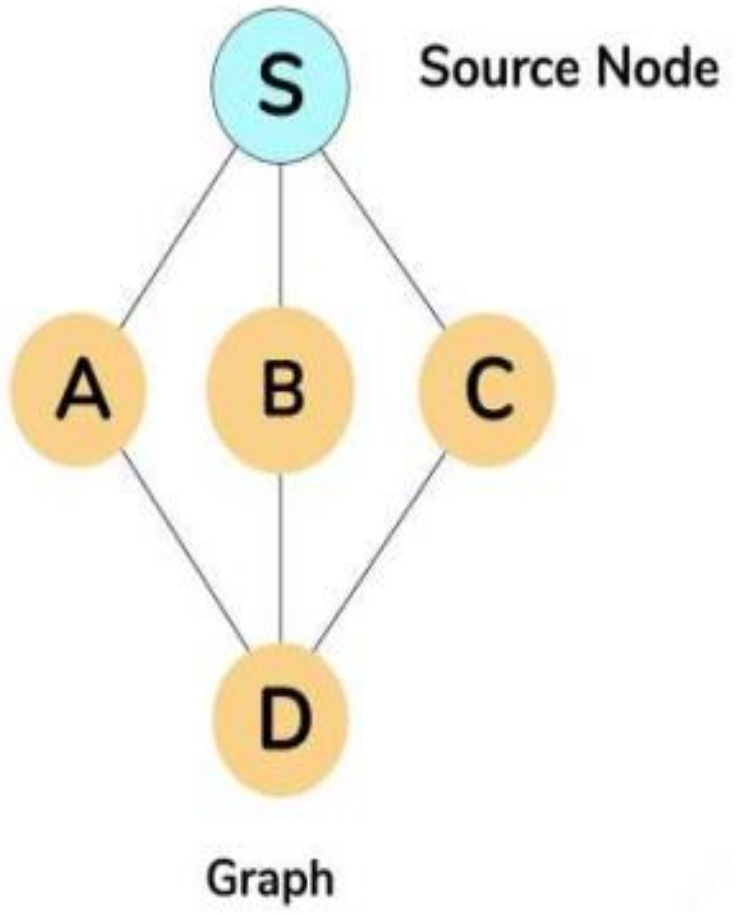
- BFS is a traversing algorithm where we start traversing from a selected source node **layerwise** by exploring the neighboring nodes.
- The data structure used in BFS is a **queue** [queue characterized by **FIFO**] and a graph[**Tree**].
- The algorithm makes sure that **every node is visited not more than once.**
- BFS follows the following 4 steps:
  1. Begin the search algorithm, by knowing the key which is to be searched. Once the key/element to be searched is decided the searching begins with the root (source) first.
  2. Visit the contiguous unvisited vertex. Mark it as visited. Display it (if needed). If this is the required key, stop. Else, add it in a queue.
  3. On the off chance that no neighboring vertex is discovered, expand the first vertex from the Queue.
  4. Repeat step 2 and 3 until the queue is empty.

Uninformed search algorithms
Breadth First Search Algorithm (BFS)

- The above algorithm is a search algorithm that identifies whether a node exists in the graph. We can convert the algorithm to traversal algorithm to find all the reachable nodes from a given node.
- **Note:**
  - If the nodes are not marked as visited, then we might visit the same node more than once and we will possibly end up in an infinite loop.

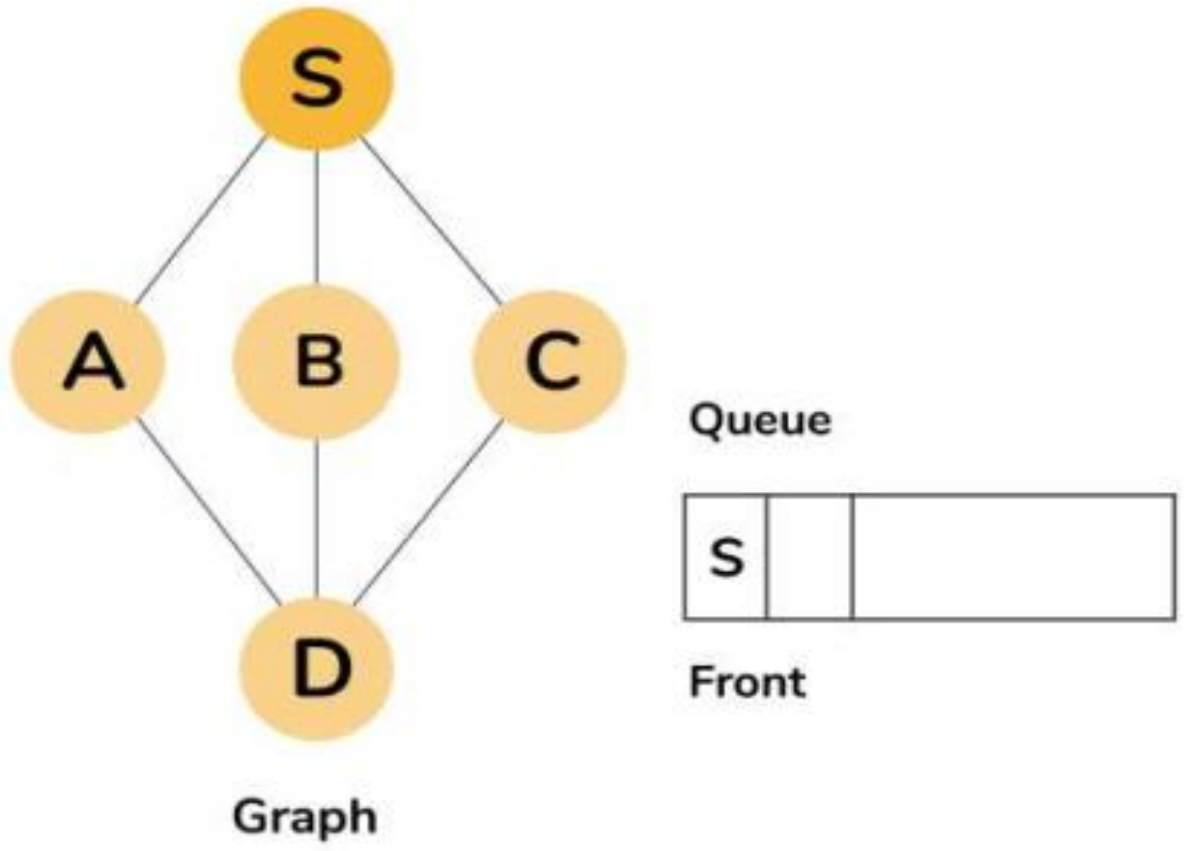
BFS Algorithm Example1

- Consider the following graph structure where S is the Source node to begin BFS with :
- **The goal here is to find whether the node E is present in the graph.**
- Just by seeing the graph, we can say that node E is not present.
  Lets see how BFS works to identify this.



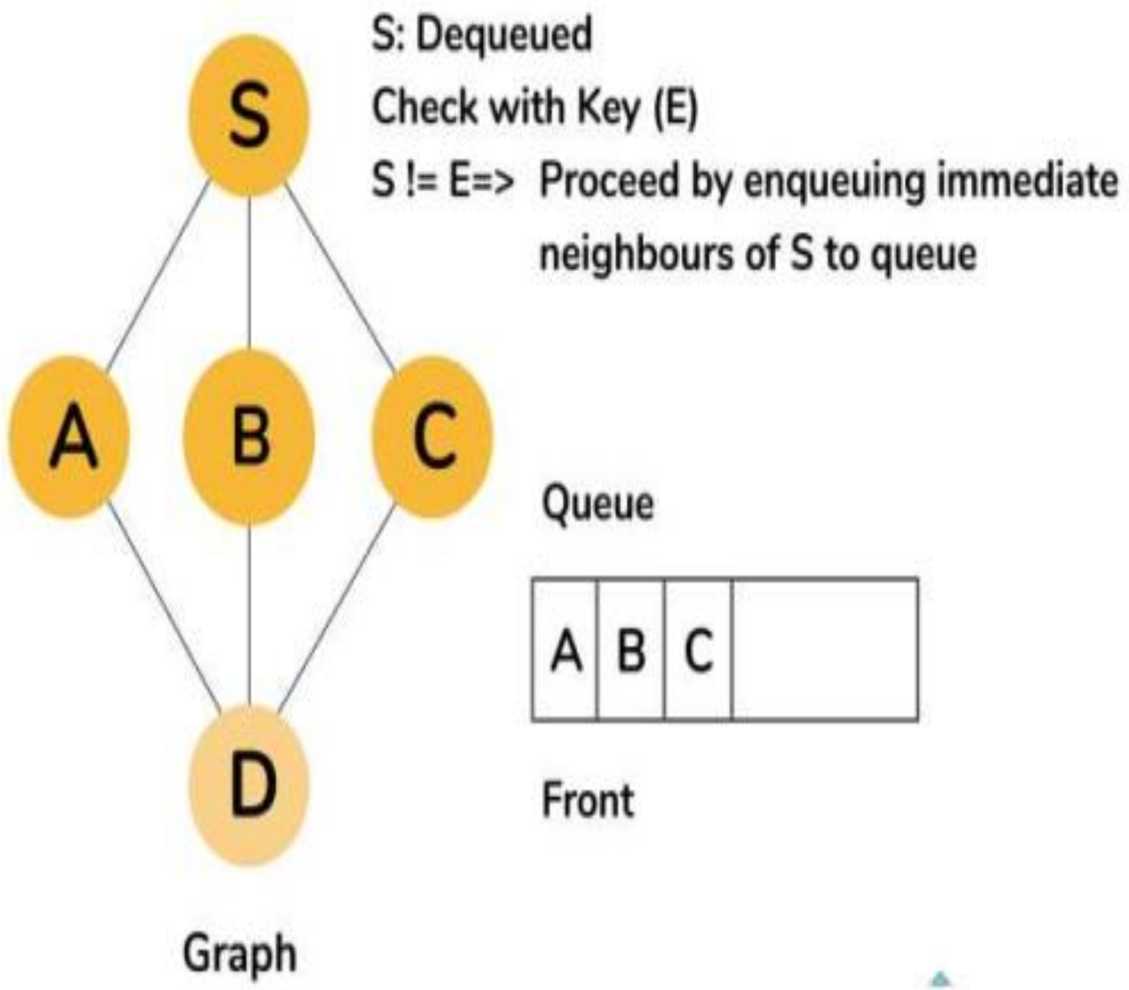
BFS Algorithm Example1

- ***Step 1:*** We enqueue S to the QUEUE.
  Step 2: Mark S as Visited.



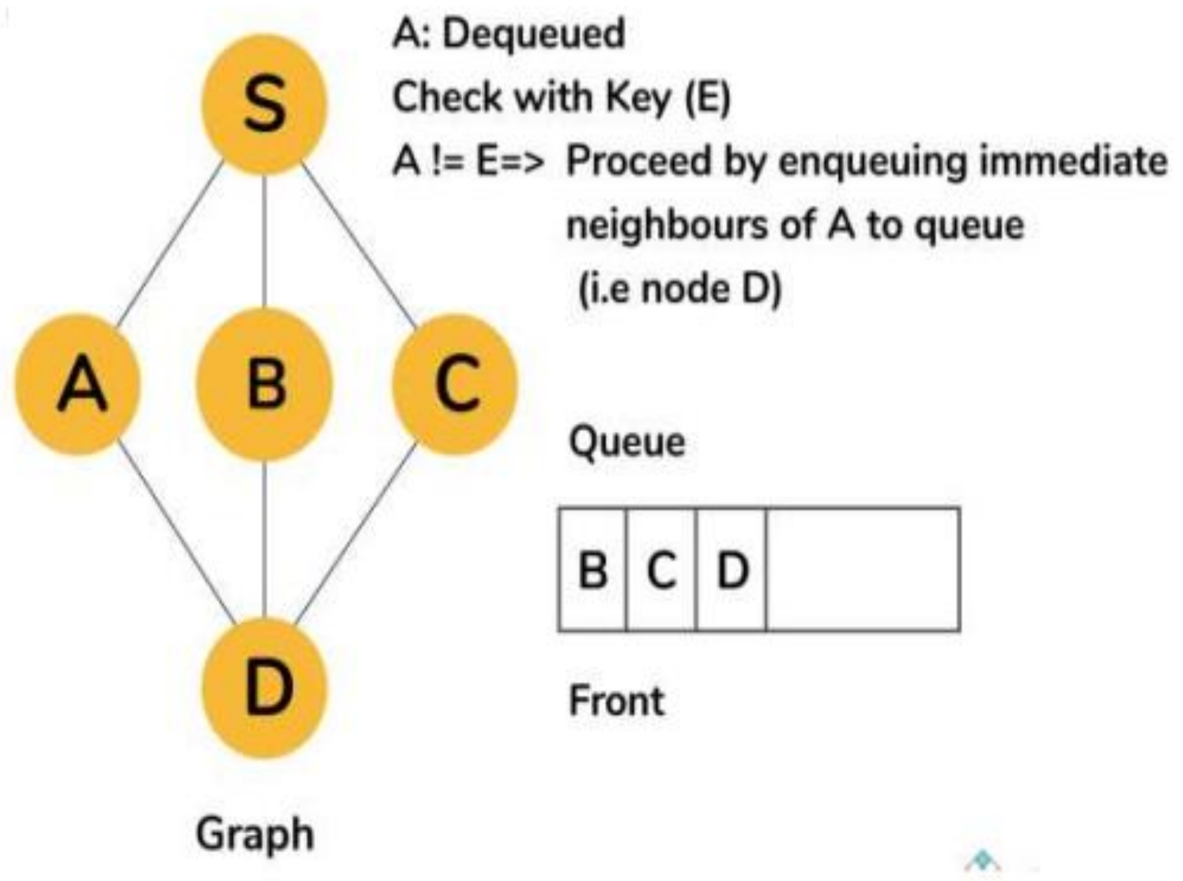
BFS Algorithm Example1

- ***Step 3:*** Now, call the BFS function with S in the queue
- Dequeue S from queue and we compare dequeued node with key E. It doesn’t match.
- Hence, proceed by looking for the unexplored nodes from S. There exist three namely, A, B, and C.
- We start traversing from A. Mark it as visited and enqueue.
- After this, there are two neighboring nodes from A, i.e., B and C. We next visit B.
- And insert it into the queue and mark as visited.
- The similar procedure begins with node C, and we insert it into the queue.


S: Dequeued
Check with Key (E)
S != E=> Proceed by enqueuing immediate neighbours of S to queue
S
A
B
C
D
Queue
A
B
C
Front
Graph

BFS Algorithm Example1

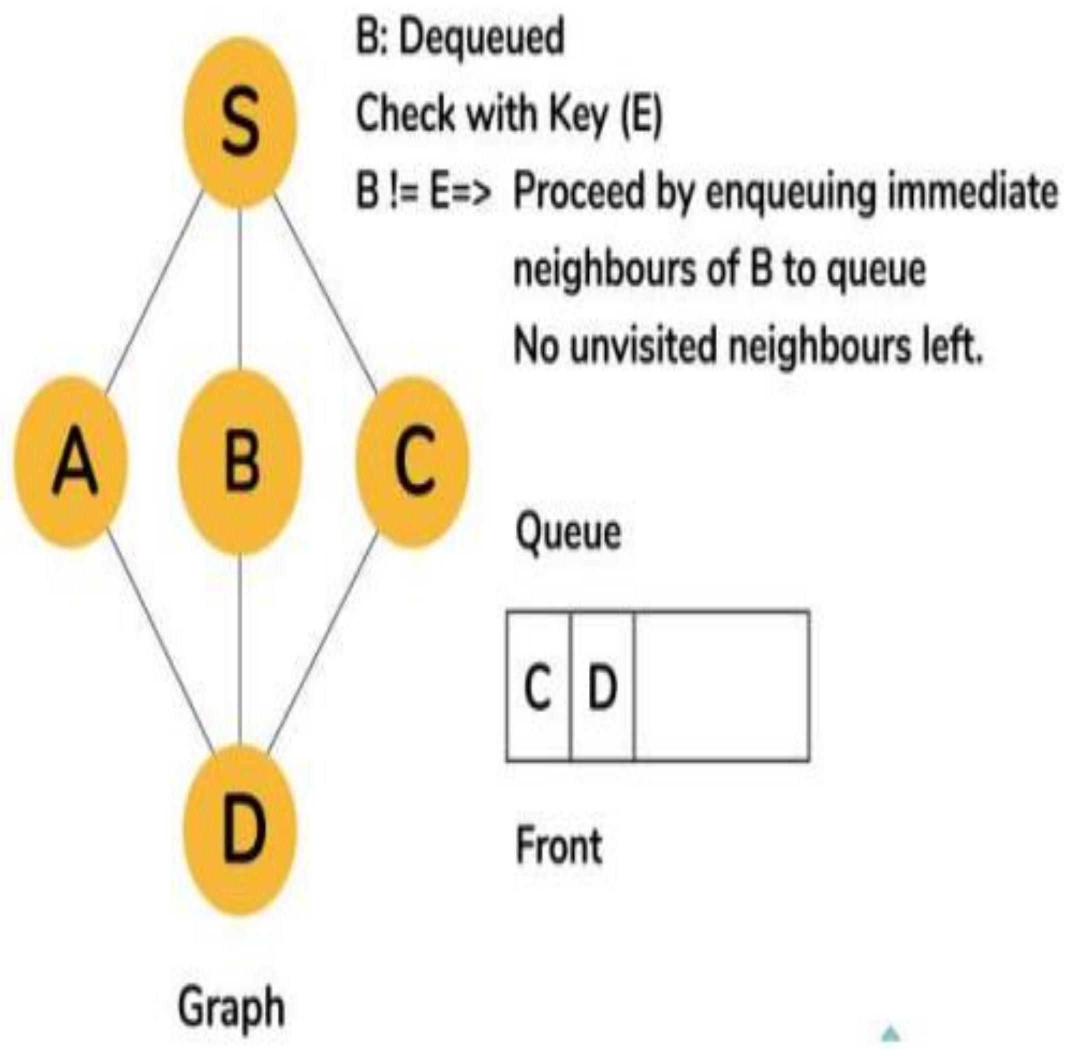
- ***Step 4:*** Dequeue A and check whether A matches the key.
- It doesnt match, hence proceed by enqueueing all unvisited neighbours of A (Here, D is the unvisited neighbor to A) to the queue.



S
A
B
C
D
Graph
A: Dequeued
Check with Key (E)
A != E=> Proceed by enqueuing immediate neighbours of A to queue (i.e node D)
Queue
B
C
D
Front

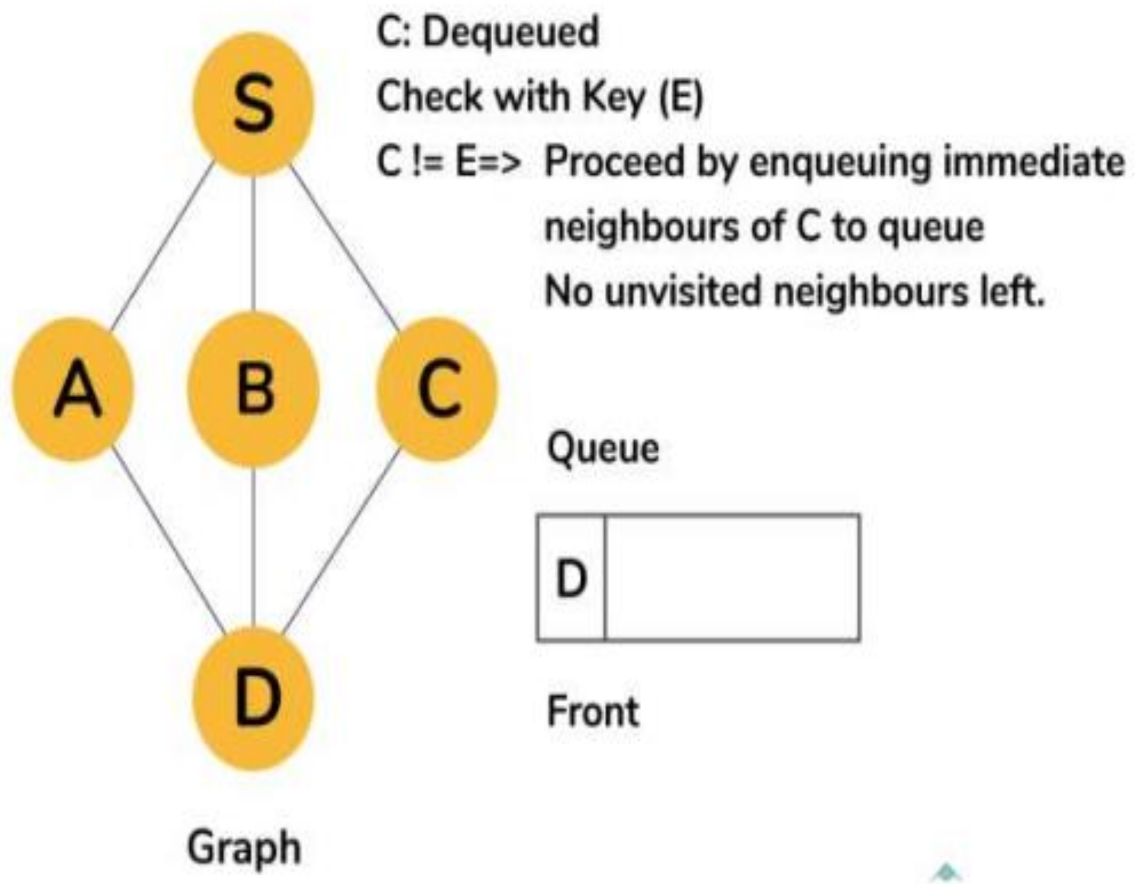
BFS Algorithm Example1

- ***Step 5:*** Dequeue B and check whether B matches the key E. It doesn't match.
- So, proceed by enqueueing all unvisited neighbors of B to queue. Here all neighboring nodes to B has been marked visited. Hence, no nodes are enqueued


B: Dequeued
Check with Key (E)
B != E=> Proceed by enqueuing immediate neighbours of B to queue
No unvisited neighbours left.
S
A
B
C
D
Queue
C
D
Front
Graph

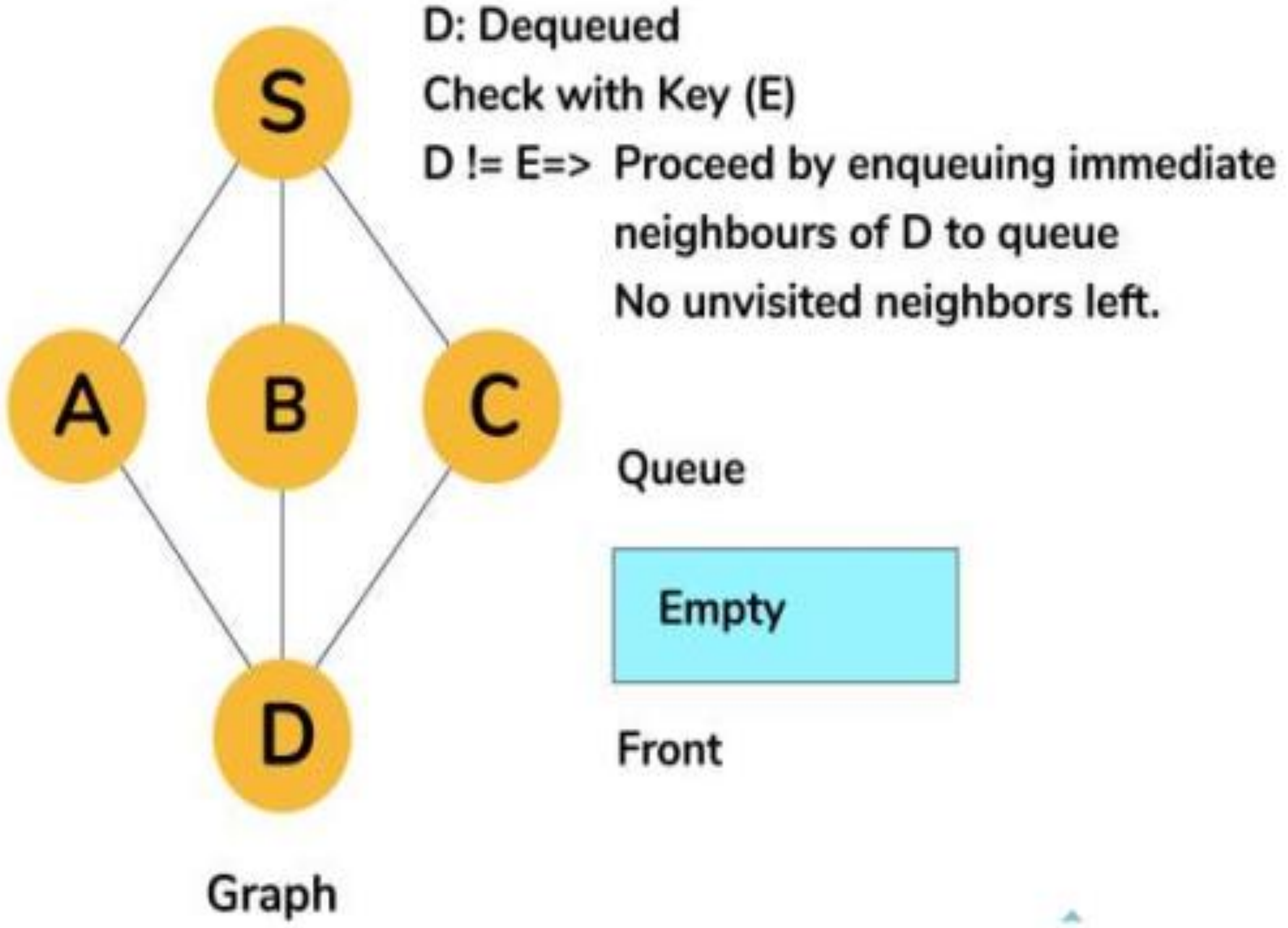
BFS Algorithm Example1

- ***Step 6:*** Dequeue C and check whether C matches the key E. It doesn’t match.
- Enqueue all unvisited neighbors of C to queue.
- Here again all neighboring nodes to C has been marked visited. Hence, no nodes are enqueued.


C: Dequeued
Check with Key (E)
C != E=> Proceed by enqueuing immediate neighbours of C to queue
No unvisited neighbours left.
S
A
B
C
D
Queue
D
Front
Graph

BFS Algorithm Example1

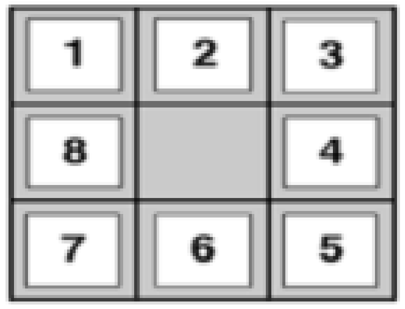
- ***Step 7:*** Dequeue D and check whether D matches the key E. It doesn't match.
- So, enqueue all unvisited neighbors of D to queue.
- Again all neighboring nodes to D
  has been marked visited.
  Hence, no nodes are enqueued.


S
A
B
C
D
Graph
D: Dequeued
Check with Key (E)
D != E=> Proceed by enqueuing immediate neighbours of D to queue
No unvisited neighbors left.
Queue
Empty
Front

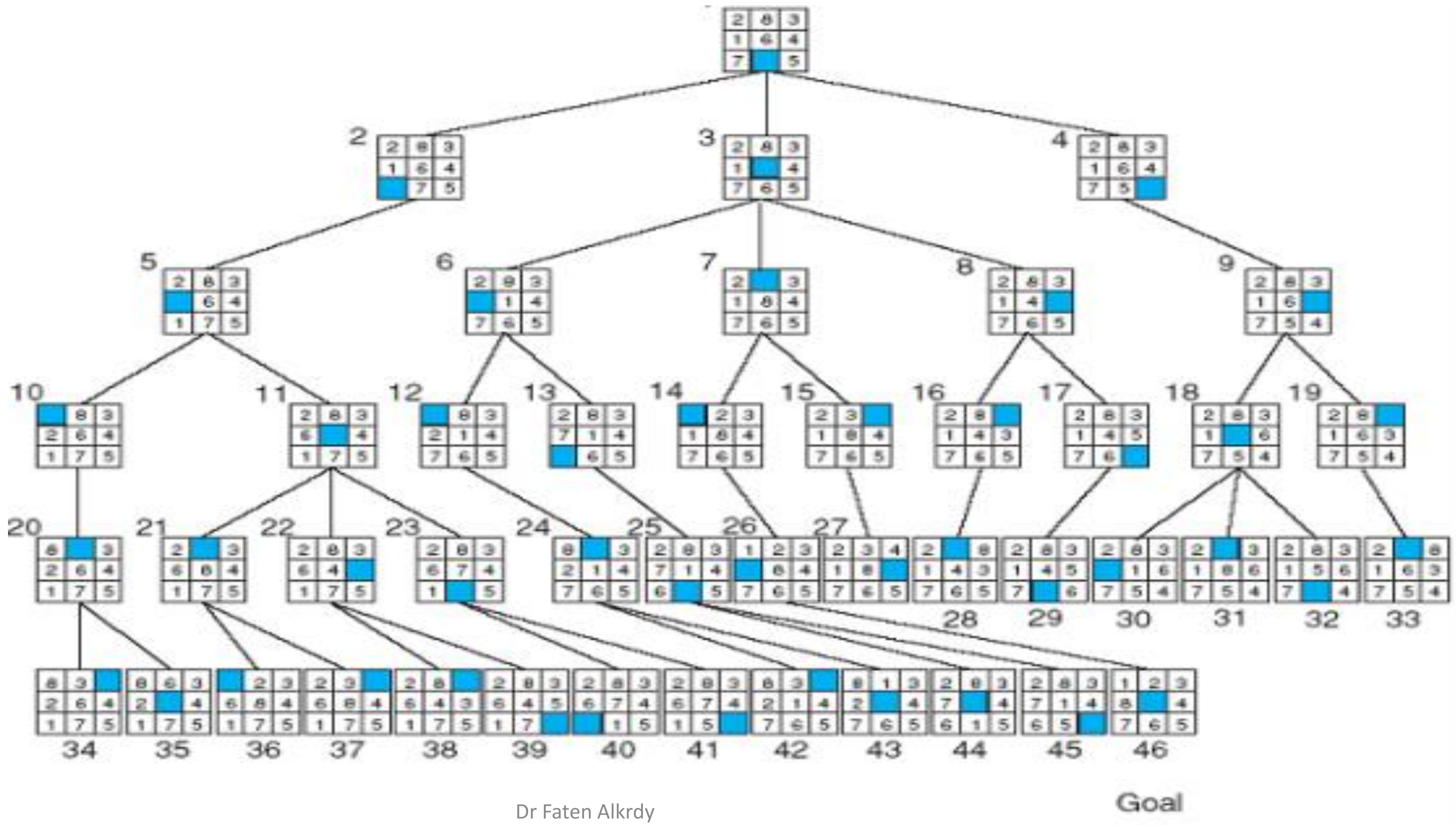
BFS Algorithm Example1

- ***Step 8:*** As we can see that the queue is empty and there are no unvisited nodes left, we can safely say that the search key is not present in the graph. Hence we return false or “Not Found” accordingly.
- This is how a breadth-first search works, **by traversing the nodes levelwise.**
- We **stop** BFS and return Found when we find the required node (key). We return Not Found when we have not found the key despite of exploring all the nodes.

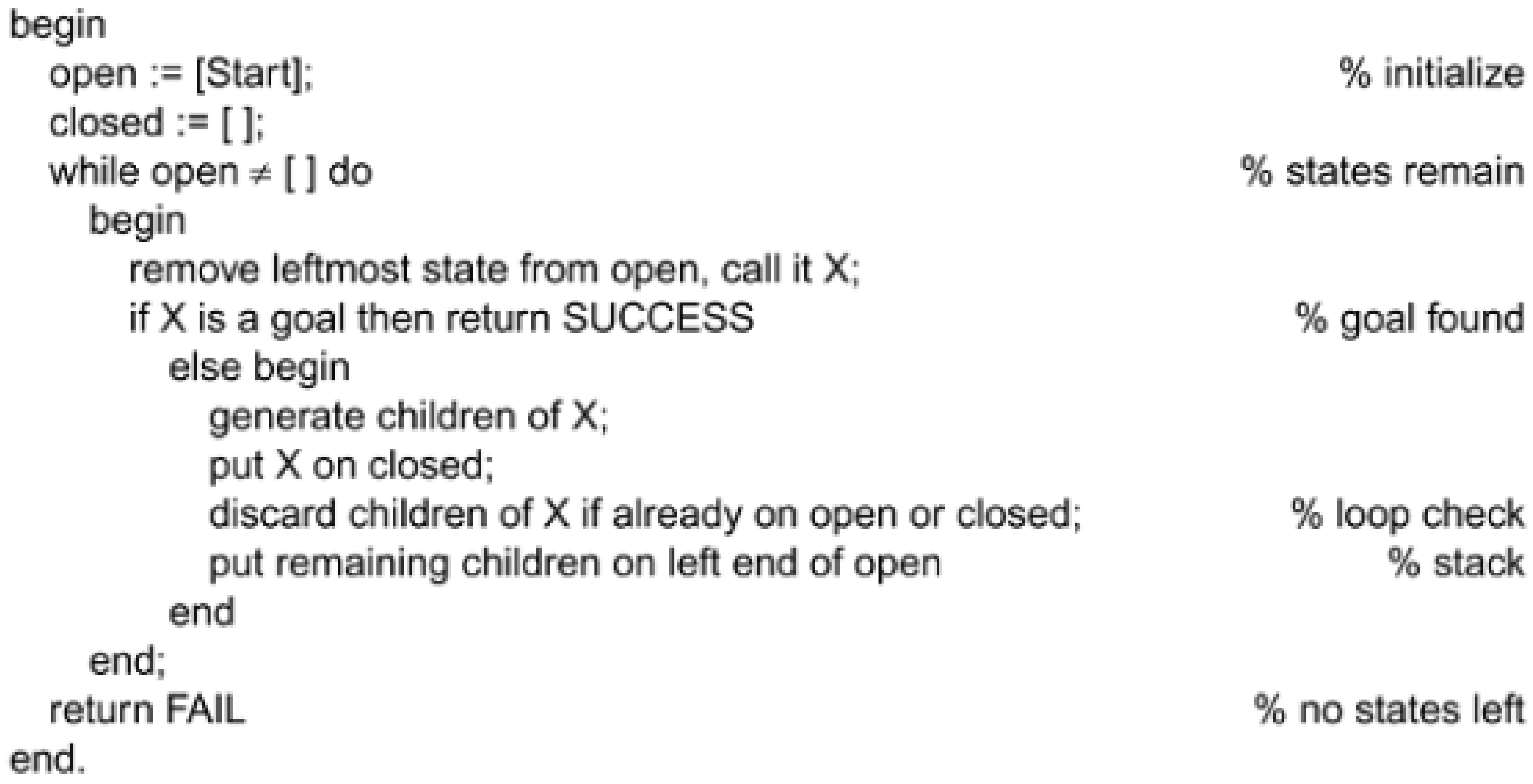
BFS Algorithm Example3



Goal State



BFS code



```
begin
    open := [Start];                                              % initialize
    closed := [ ];
    while open ≠ [ ] do                                           % states remain
        begin
            remove leftmost state from open, call it X;
            if X is a goal then return SUCCESS                    % goal found
                else begin
                    generate children of X;
                    put X on closed;
                    discard children of X if already on open or closed;   % loop check
                    put remaining children on left end of open            % stack
                end
        end;
    return FAIL                                                   % no states left
end.
```

BFS Algorithm Example2

النقاط المستكشفة وفق البحث العرضي BFS
A B C D E F G H I J K L M N O P Q

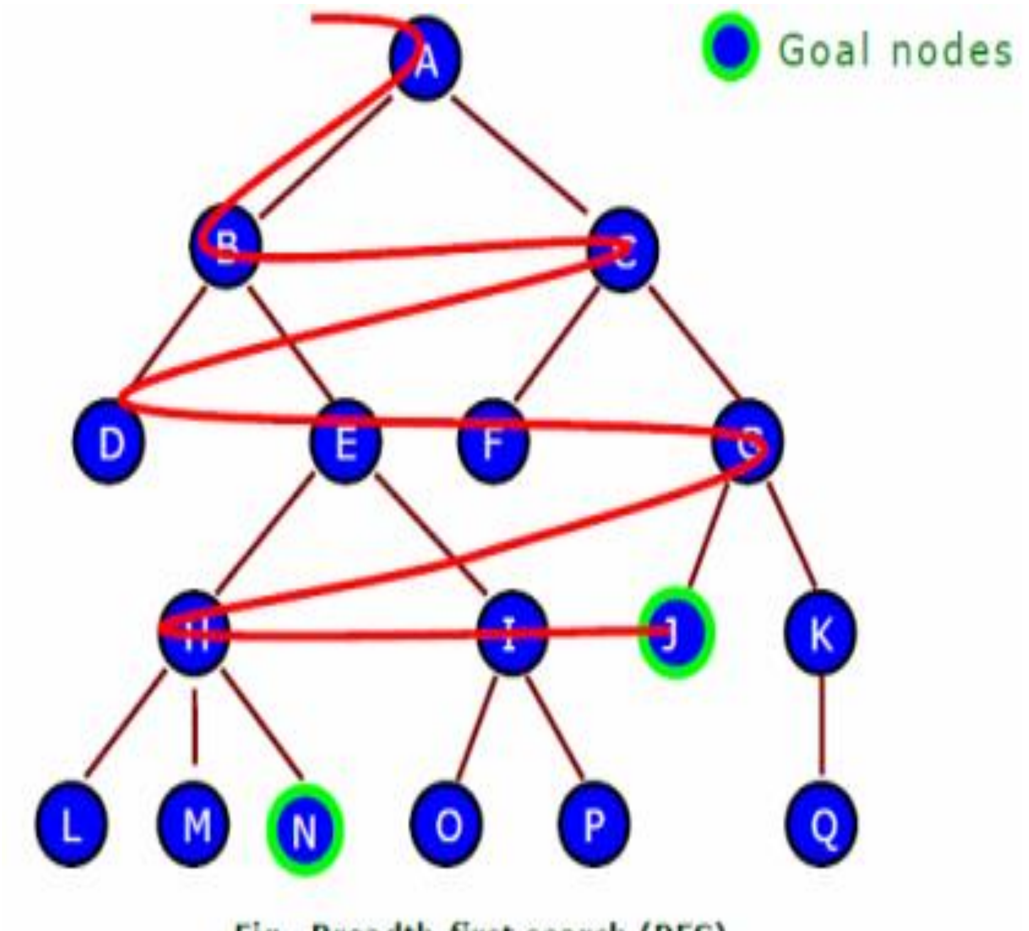
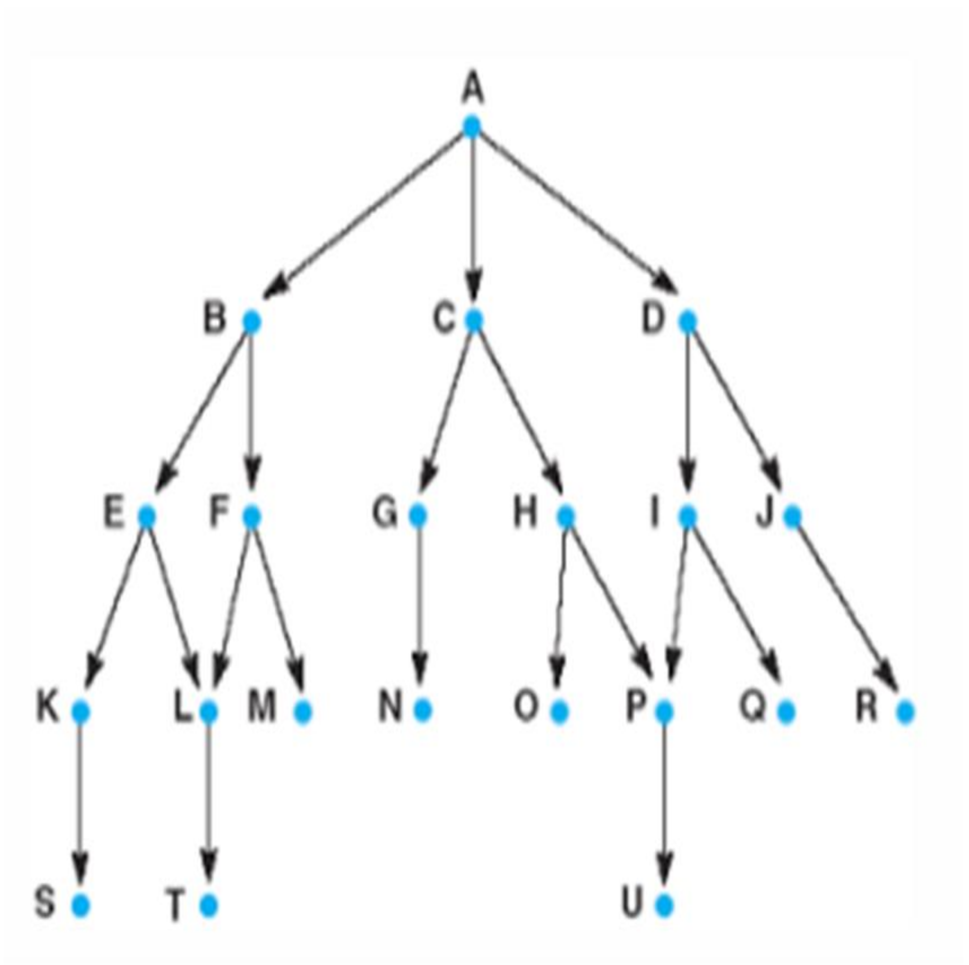


Fig. Breadth-first search (BFS)

BFS Algorithm Example2

1. **open = [A]; closed = [ ]**
2. **open = [B,C,D]; closed = [A]**
3. **open = [C,D,E,F]; closed = [B,A]**
4. **open = [D,E,F,G,H]; closed = [C,B,A]**
5. **open = [E,F,G,H,I,J]; closed = [D,C,B,A]**
6. **open = [F,G,H,I,J,K,L]; closed = [E,D,C,B,A]**
7. **open = [G,H,I,J,K,L,M]** (as L is already on open); **closed = [F,E,D,C,B,A]**
8. **open = [H,I,J,K,L,M,N]; closed = [G,F,E,D,C,B,A]**
9. and so on until either U is found or **open** = [ ]


A
B
C
D
E
F
G
H
I
J
K
L
M
N
O
P
Q
R
S
T
U