Artificial Intelligence-BSCE-306L

Module 2:

Problem Solving based on Searching

Dr . Saurabh Agrawal

Faculty Id: 20165

School of Computer Science and Engineering

VIT, Vellore-632014

Tamil Nadu, India

Outline

- Introduction to Problem Solving by searching Methods-State Space search
- Uninformed Search Methods
- Uniform Cost Search
- Breadth First Search
- Depth First Search
- Depth Limited Search
- Iterative Deepening Depth-First-Search
- Informed Search Methods
- Best First Search
- A* Search

Introduction to Problem Solving by Searching Methods-State Space Search

❑Problem:

- Problems are can resolve by logical algorithms, differential equations, utilizing effective polynomial and executing all of them with modelling patterns.
- One problem can have multiple solutions by using different problem-solving techniques.
- In parallel, some problems have unique solutions and can be only resolved by one method.
- It depends on the nature of the problems that how many solutions are possible.

Introduction to Problem Solving by Searching Methods-State Space Search

❑Problem Solving:

- The reflex agent of AI directly maps states into action.
- Whenever these agents fail to operate in an environment where the state of mapping is too large and not easily performed by the agent, then the stated problem dissolves and sent to a problem-solving domain which breaks the large stored problem into the smaller storage area and resolves one by one.
- The final integrated action will be the desired outcomes.

❑Steps Problem Solving:

→These are the following steps which require to solve a problem :

•**Goal Formulation:** This one is the first and simple step in problem-solving. It organizes finite steps to formulate a target/goals which require some action to achieve the goal. Today the formulation of the goal is based on AI agents.

•**Problem formulation:** It is one of the core steps of problem-solving which decides what action should be taken to achieve the formulated goal. In AI this core part is dependent upon software agent which consisted of the following components to formulate the associated problem.

Introduction to Problem Solving by Searching Methods-State Space Search

❑Steps Problem Solving:

→Components to formulate the associated problem:

- **Initial State:** This state requires an initial state for the problem which starts the AI agent towards a specified goal. In this state new methods also initialize problem domain solving by a specific class.
- **Action:** This stage of problem formulation works with function with a specific class taken from the initial state and all possible actions done in this stage.
- **Transition:** This stage of problem formulation integrates the actual action done by the previous action stage and collects the final stage to forward it to their next stage.
- **Goal test:** This stage determines that the specified goal achieved by the integrated transition model or not, whenever the goal achieves stop the action and forward into the next stage to determines the cost to achieve the goal.
- **Path costing:** This component of problem-solving numerical assigned what will be the cost to achieve the goal. It requires all hardware software and human working cost.

Introduction to Problem Solving by Searching Methods-State Space Search

❑Problem Solving Techniques in AI:

→Search Algorithms

•Search algorithms are one of the common methods to solve problems in AI.

•The problem solving or rational agents with search algorithms are used to find the best optimal solution.

•This type of problem-solving technique is often called **goal-based.**

❑Time Complexity

❑Space Complexity

❑Completeness

❑Optimality

Introduction to Problem Solving by Searching Methods-State Space Search

❑Problem Solving Techniques in AI:

→Search Algorithms

- A search problem consists of:
  - ➢**A State Space.** Set of all possible states where you can be.
  - ➢**A Start State.** The state from where the search begins.
  - ➢**A Goal Test.** A function that looks at the current state returns whether or not it is the goal state.
- The **Solution** to a search problem is a sequence of actions, called the **plan** that transforms the start state to the goal state.
- This plan is achieved through search algorithms.

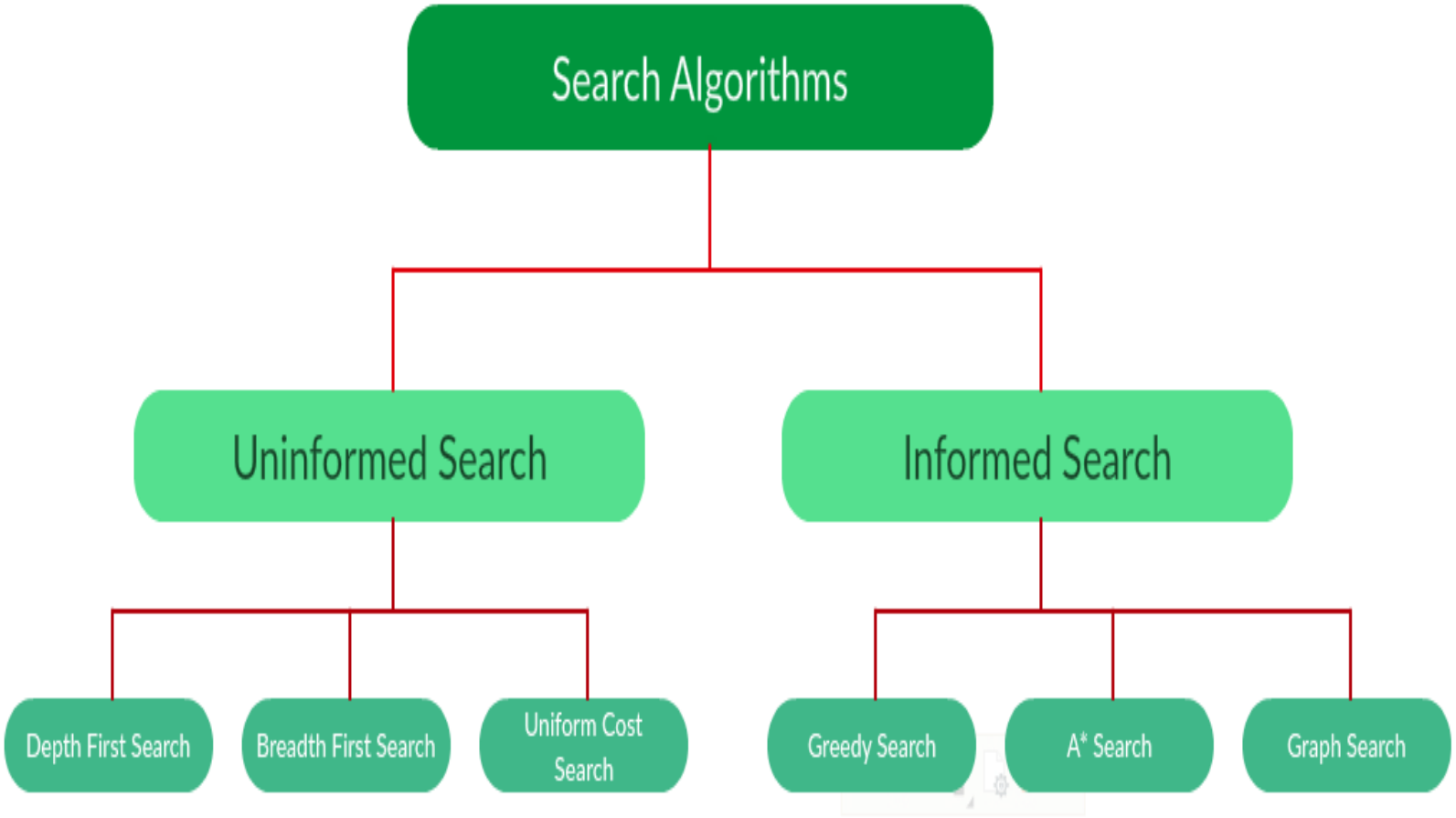
Introduction to Problem Solving by Searching Methods-State Space Search

❑What choices are we searching through?

- Problem solving : Action combinations (move 1, then move 3, then move 2...)
- Natural language : Ways to map words to parts of speech
- Computer vision : Ways to map features to object model
- Machine learning : Possible concepts that fit examples seen so far
- Motion planning : Sequence of moves to reach goal destination

Introduction to Problem Solving by Searching Methods-State Space Search

❑Types of search algorithms



❑Types of search algorithms

→Uninformed Search Algorithms:

•The search algorithms in this section 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/or length of actions.

•Uninformed search is also called ***Blind search***.

•These algorithms can only generate the successors and differentiate between the goal state and non goal state.

→Informed Search Algorithms:

• The algorithms have information on the goal state, which helps in more efficient searching.

•This information is obtained by something called a ***heuristic Search.***

Introduction to Problem Solving by Searching Methods-State Space Search

❑Steps for State Space Search

- Define state space that contains all possible configurations of relevant objects, without enumerating.
- Define some initial states and some goal states.
- Specify rules as possible actions.
- Good control strategy.

S: (S,A,Action(S),Result(S,A),Cost(S,A))

❑Example Problems

8 puzzle

- The **8-puzzle**, an instance consists of a 3×3 board with eight numbered tiles and a blank space. A tile adjacent to the blank space can slide into the space. The object is to reach a specified goal state, such as the one shown on the right of the figure.

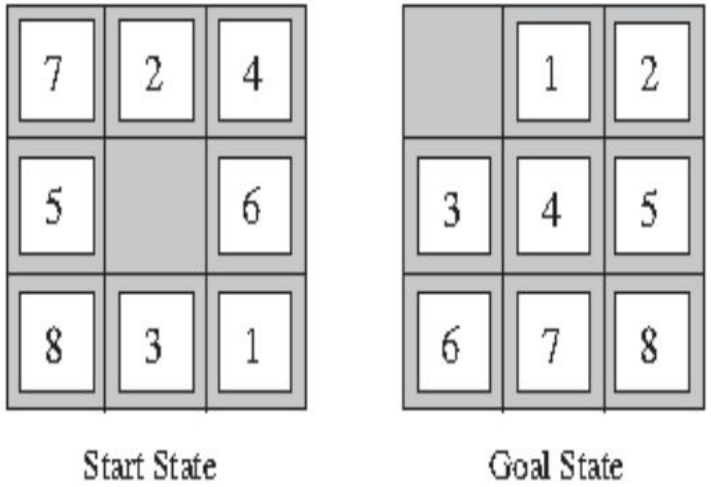


Figure 3.4 A typical instance of the 8-puzzle

❑Example Problems

8 puzzle

- **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.
- **Actions**: The simplest formulation defines the actions as movements of the blank space *Left*, *Right*, *Up*, or *Down*. Different subsets of these are possible depending on where the blank is.
- **Transition model**: Given a state and action, this returns the resulting state; for example, if we apply *Left* to the start state in Figure 3.4, the resulting state has the 5 and the blank switched.
- **Goal test**: This checks whether the state matches the goal configuration shown in Figure 3.4. (Other goal configurations are possible.)
- **Path cost**: Each step costs 1, so the path cost is the number of steps in the path.

❑Example Problems

Vacum World

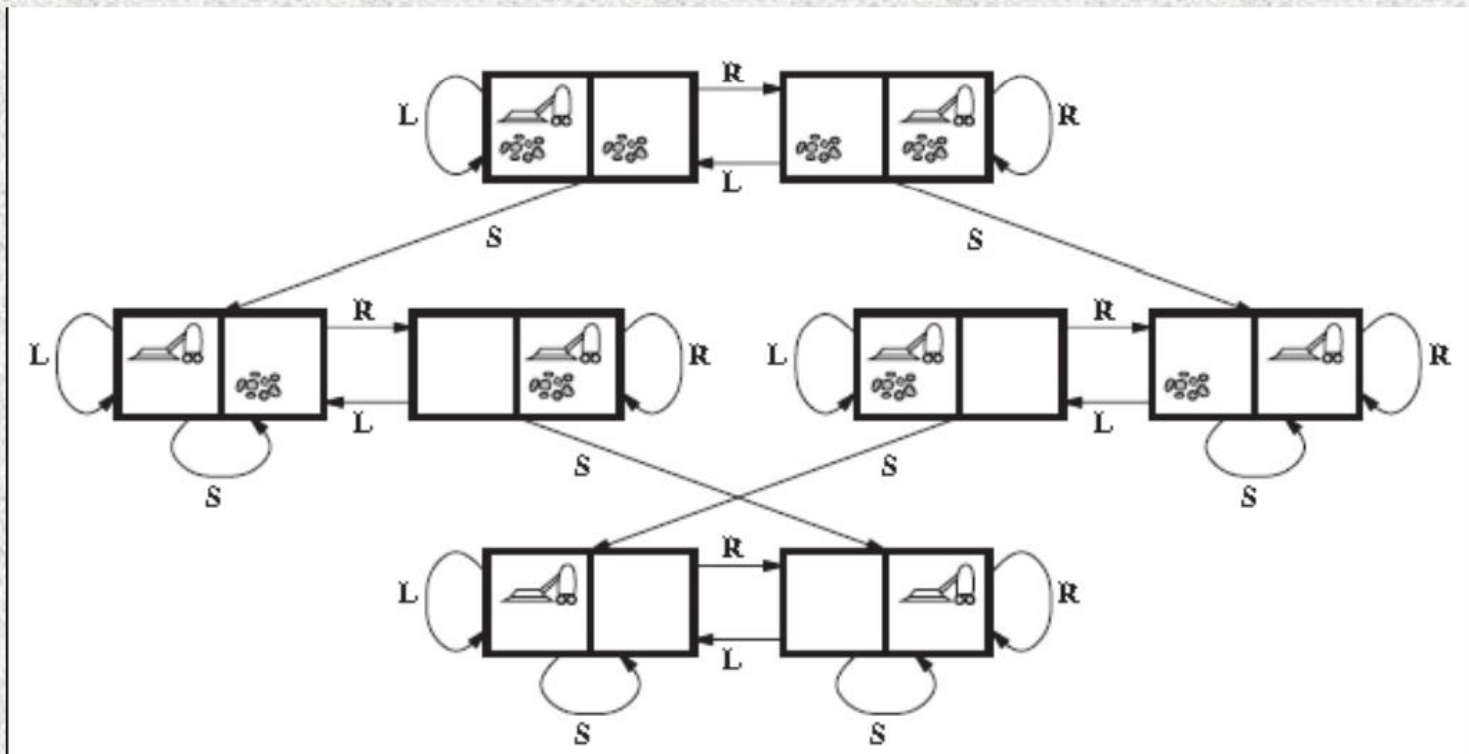


Figure 3.3 The state space for the vacuum world. Links denote actions: L = *Left*, R = *Right*, S = *Suck*.

❑Example Problems

Vacum World

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.

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*. Larger environments might also include *Up* and *Down*.

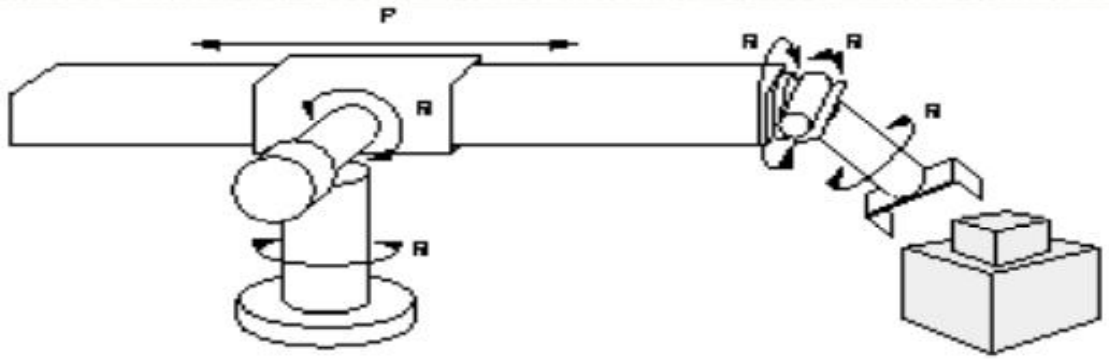
Transition model: The actions have their expected effects, except that moving *Left* in the leftmost square, moving *Right* in the rightmost square, and *Sucking* in a clean square have no effect. The complete state space is shown in Figure 3.3.

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 Problems

Example Problems – Robot Assembly



States: real-valued coordinates of
- robot joint angles
- parts of the object to be assembled

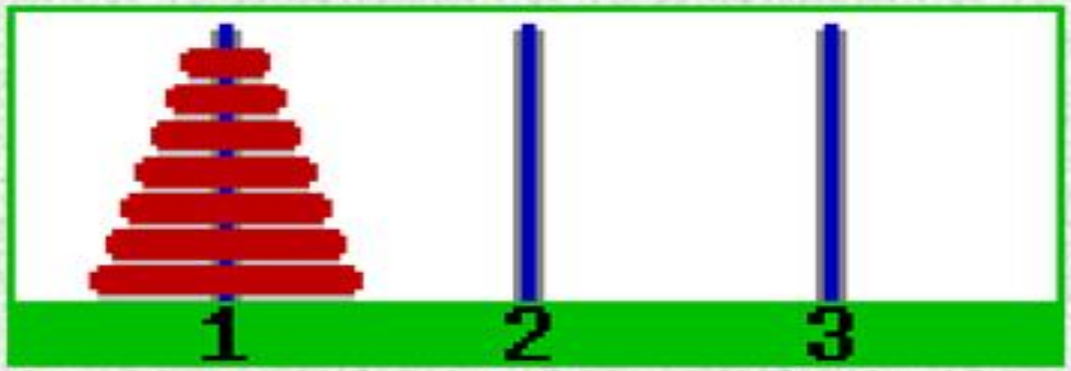
Operators: rotation of joint angles

Goal test: complete assembly

Path cost: time to complete assembly

❑Example Problems

Example Problems – Towers of Hanoi



States: combinations of poles and disks

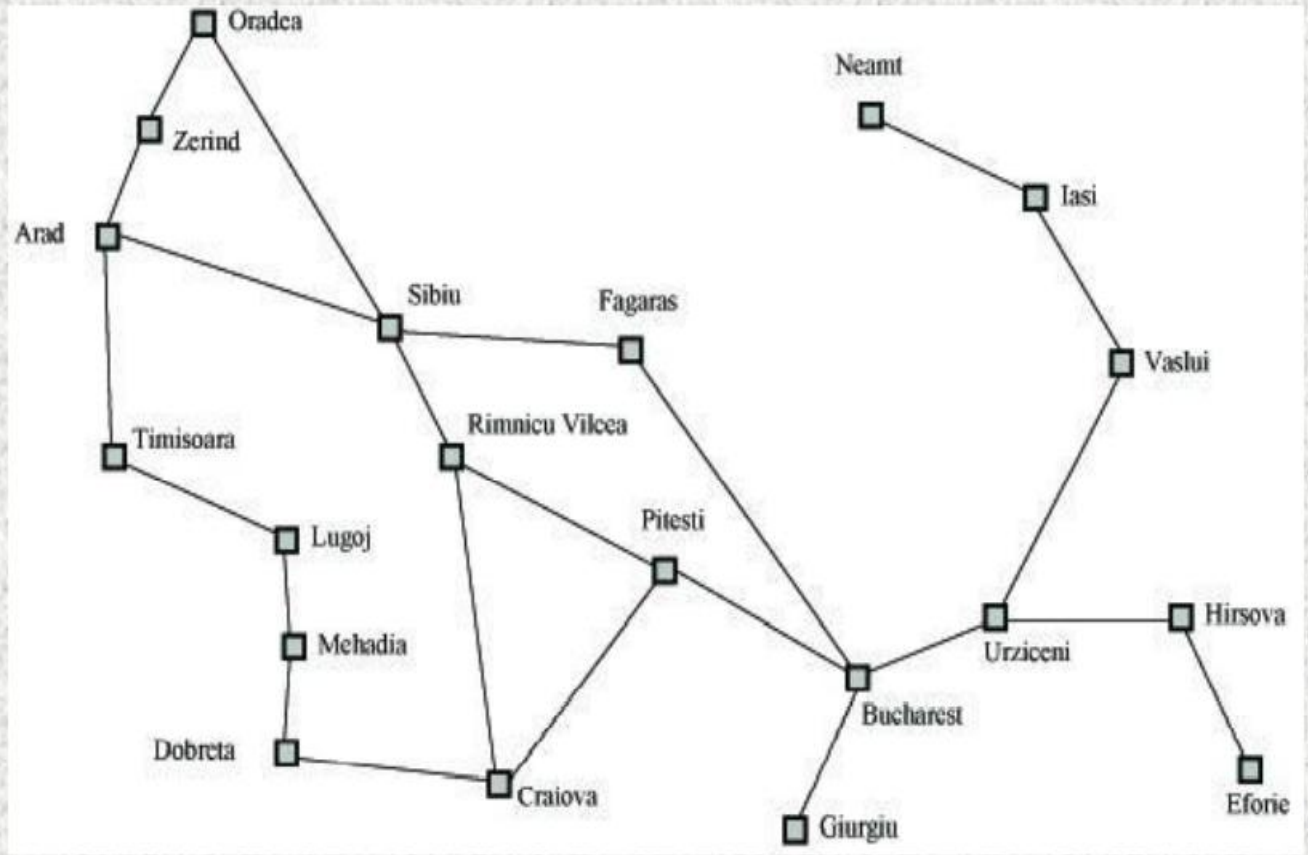
Operators: move disk x from pole y to pole z subject to constraints
- cannot move disk on top of smaller disk
- cannot move disk if other disks on top

Goal test: disks from largest (at bottom) to smallest on goal pole

Path cost: 1 per move

❑Example Problems

Search Example



Formulate goal: Be in Bucharest.

Formulate problem: states are cities, operators drive between pairs of cities

Find solution: Find a sequence of cities (e.g., Arad, Sibiu, Fagaras, Bucharest) that leads from the current state to a state meeting the goal condition

❑Example Problems

SEARCHING FOR SOLUTIONS

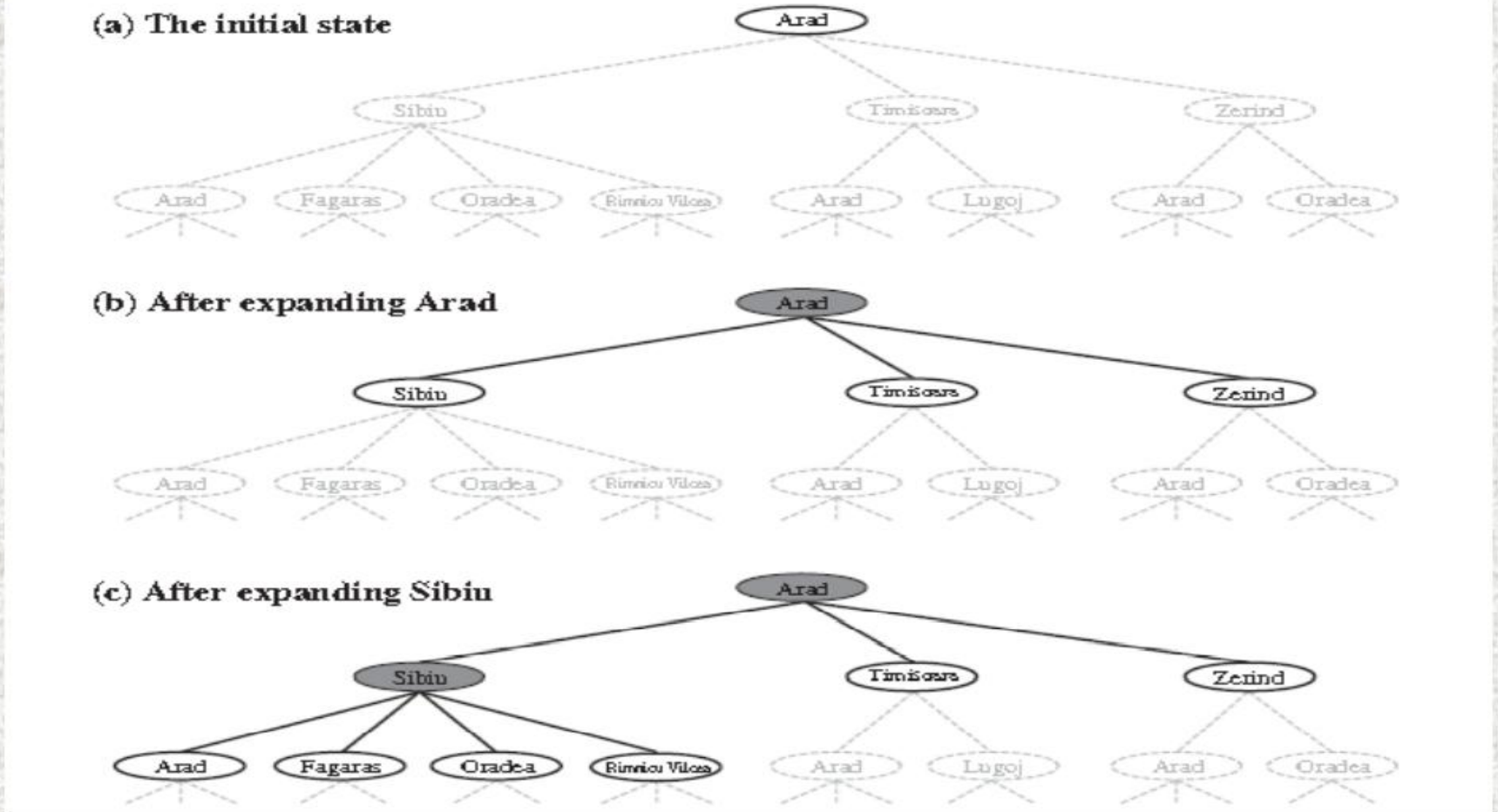


Figure 3.6 Partial search trees for finding a route from Arad to Bucharest. Nodes that have been expanded are shaded; nodes that have been generated but not yet expanded are outlined in bold: nodes that have not yet been generated are shown in faint dashed lines.

❑Example Problems

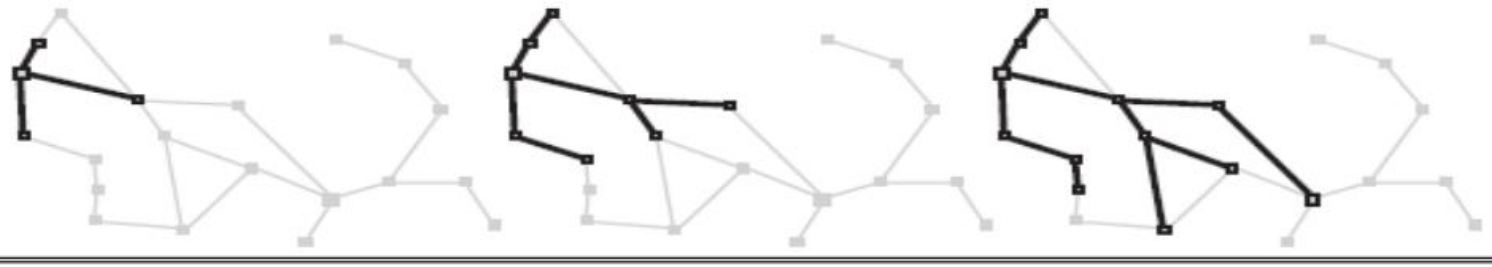


Figure 3.8 A sequence of search trees generated by a graph search on the Romania problem of Figure 3.2. At each stage, we have extended each path by one step. Notice that at the third stage, the northernmost city (Oradea) has become a dead end: both of its successors are already explored via other paths.

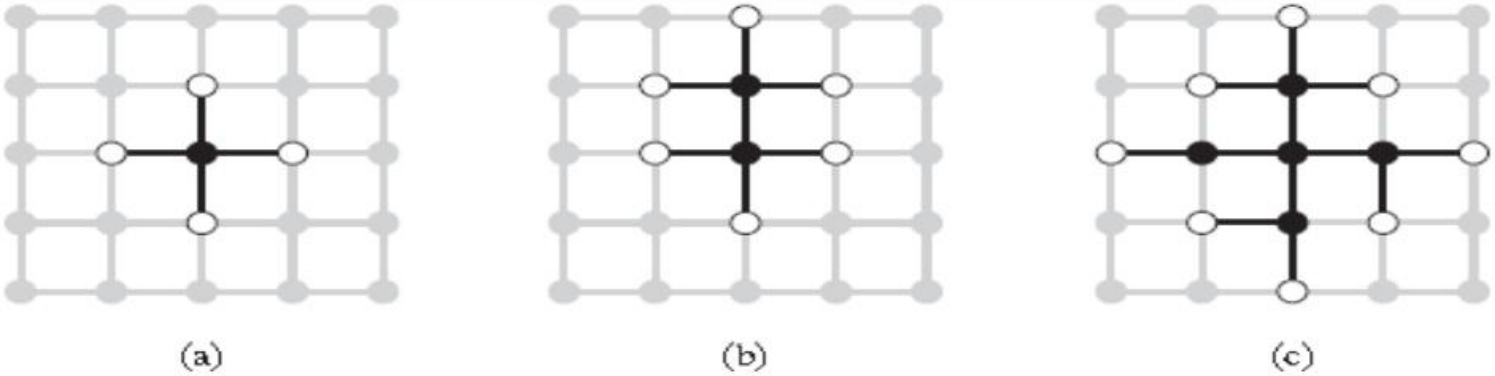


Figure 3.9 The separation property of GRAPH-SEARCH, illustrated on a rectangular-grid problem. The frontier (white nodes) always separates the explored region of the state space (black nodes) from the unexplored region (gray nodes). In (a), just the root has been expanded. In (b), one leaf node has been expanded. In (c), the remaining successors of the root have been expanded in clockwise order.

❑Example Problems

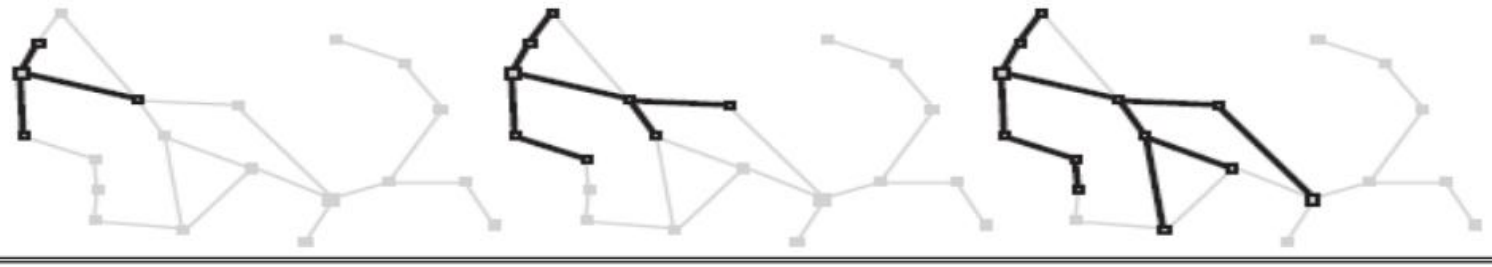


Figure 3.8 A sequence of search trees generated by a graph search on the Romania problem of Figure 3.2. At each stage, we have extended each path by one step. Notice that at the third stage, the northernmost city (Oradea) has become a dead end: both of its successors are already explored via other paths.

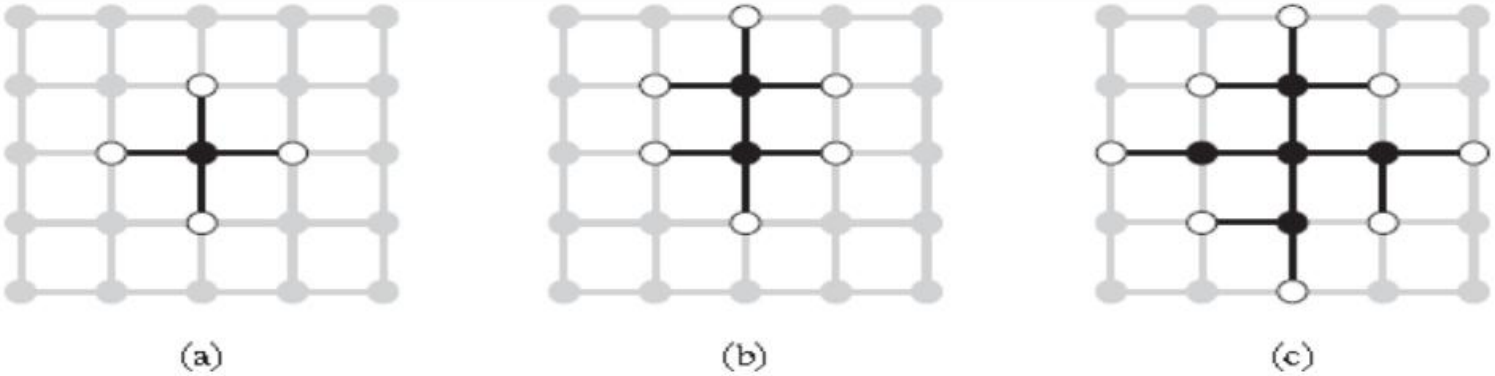


Figure 3.9 The separation property of GRAPH-SEARCH, illustrated on a rectangular-grid problem. The frontier (white nodes) always separates the explored region of the state space (black nodes) from the unexplored region (gray nodes). In (a), just the root has been expanded. In (b), one leaf node has been expanded. In (c), the remaining successors of the root have been expanded in clockwise order.

❑Sample Search Problems

- Graph coloring
- Protein folding
- Game playing
- Airline travel
- Proving algebraic equalities
- Robot motion planning

Uniform Cost Search

- Uniform Cost Search is an algorithm used to move around a directed weighted search space to go from a start node to one of the ending nodes with a minimum cumulative cost.
- This search is an uninformed search algorithm since it operates in a brute-force manner, i.e. it does not take the state of the node or search space into consideration.
- It is used to find the path with the lowest cumulative cost in a weighted graph where nodes are expanded according to their cost of traversal from the root node.
- This is implemented using a priority queue where lower the cost higher is its priority.

Uniform Cost Search

❑Algorithm

- Insert RootNode into the queue.
- Repeat till queue is not empty:
  - ➢Remove the next element with the highest priority from the queue.
  - ➢If the node is a destination node, then print the cost and the path and exit
  - ➢else insert all the children of removed elements into the queue with their cumulative cost as their priorities.

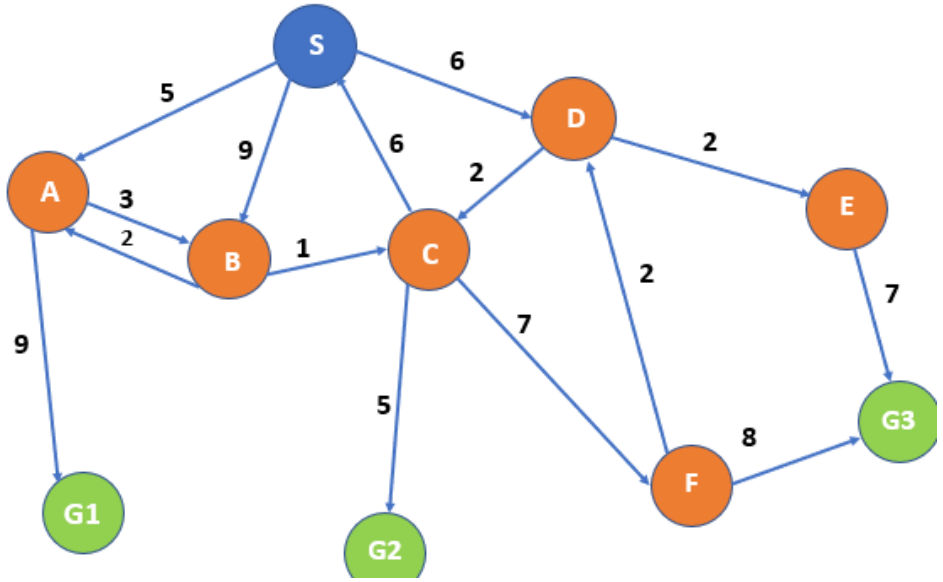
→NOTE: Here rootNode is the starting node for the path, and a priority queue is being maintained to maintain the path with the least cost to be chosen for the next traversal. In case 2 paths have the same cost of traversal, nodes are considered alphabetically.

Uniform Cost Search

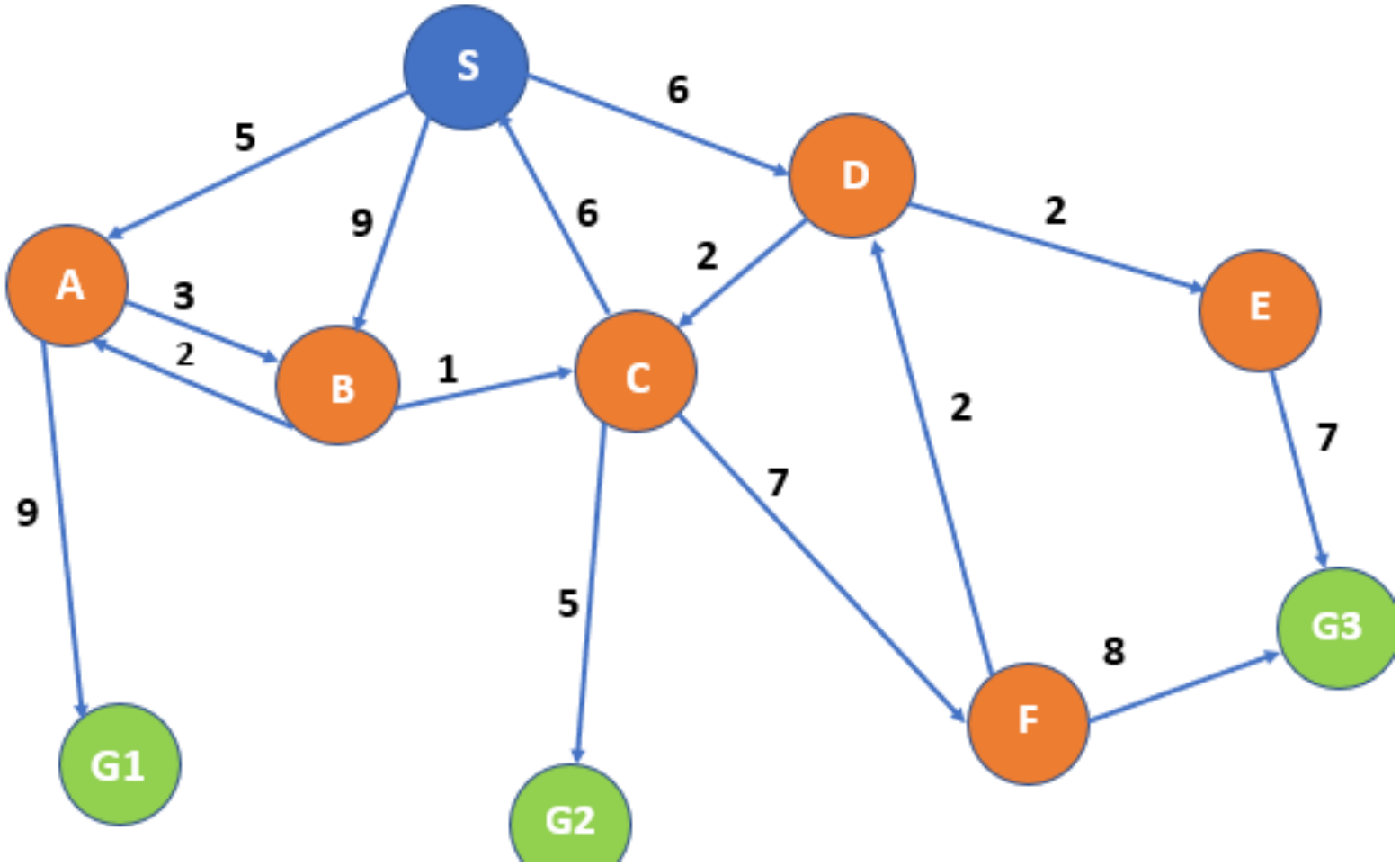
❑Example of Uniform Cost Search

▪Consider the below example, where we need to reach any one of the destination node{G1, G2, G3} starting from node S. Node{A, B, C, D, E and F} are the intermediate nodes. Our motive is to find the path from S to any of the destination state with the least cumulative cost. Each directed edge represents the direction of movement allowed through that path, and its labelling represents the cost is one travels through that path. Thus Overall cost of the path is a sum of all the paths.

▪For e.g. – a path from S to G1- {S->A -> G1} whose cost is SA +AG1 = 5 + 9 = 14.



Uniform Cost Search



Uniform Cost Search

❑We are going to use a tree to show all the paths possible and also maintain a visited list to keep track of all the visited nodes as we need not visit any node twice.

Explanation	Visited
Step 1-We will start with start node and check if we have reached any of the destination nodes, i.e. No thus continue.	
Step 2– We reach all the nodes that can be reached from S I .eA, B, D. And Since node S has been visited thus added to the visited List. Now we select the cheapest path first for further expansion, i.e. A	S
Step 3 – Node B and G1 can be reached from A and since node A is visited thus move to the visited list. Since G1 is reached but for the optimal solution, we need to consider every possible case; thus, we will expand the next cheapest path, i.e. S->D.	S, A
Step 4– Now node D has been visited thus it goes to visited list and now since we have three paths with the same cost, we will choose alphabetically thus will expand node B	S,A,D

Uniform Cost Search

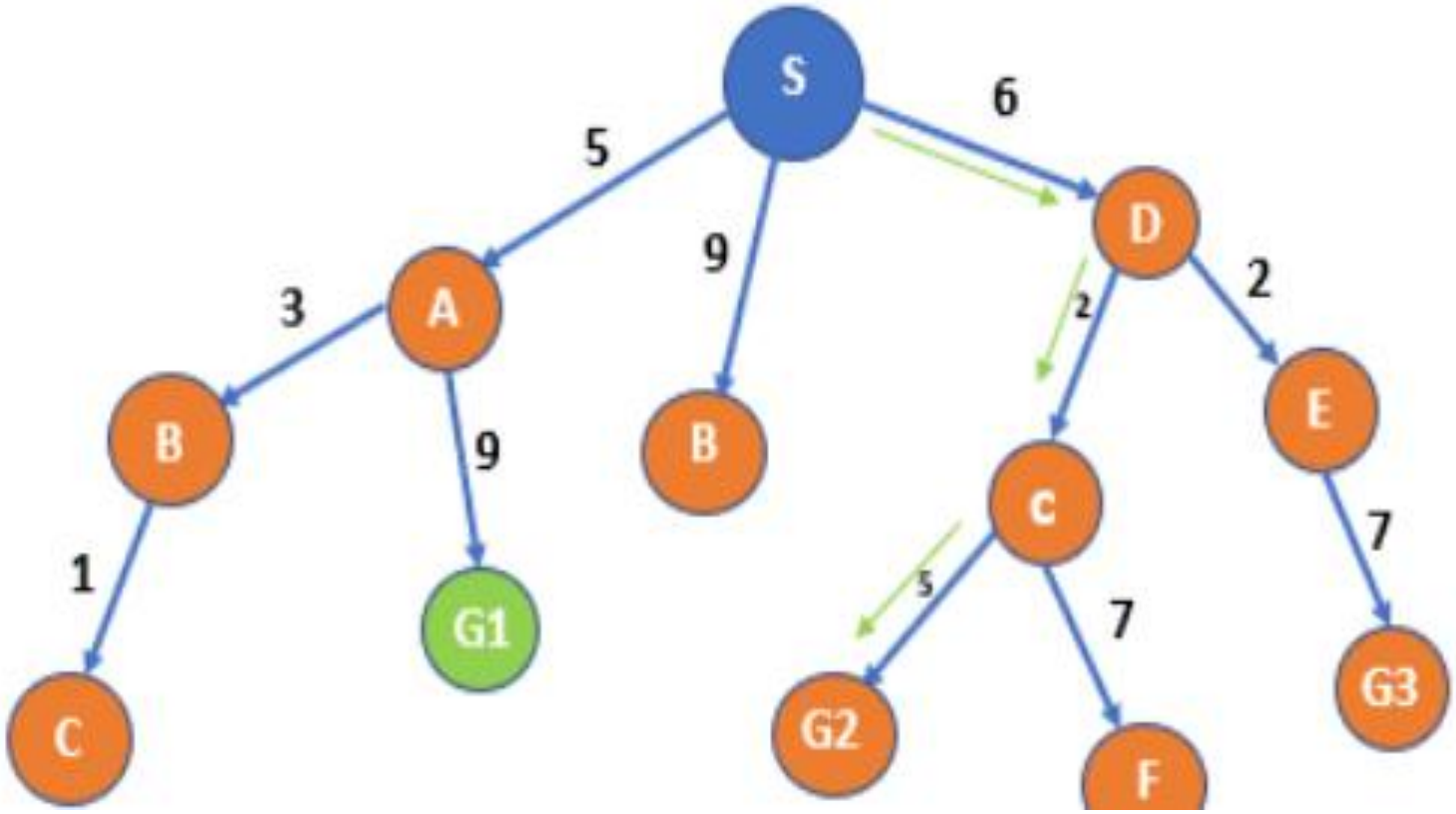
Step 5-: From B, we can only reach node C. Now the path with minimum weight is S->D->C, i.e. 8. Thus expand C. And B has now visited node.	S,A,D,B
Step 6:- From C we can reach G2 and F node with 5 and 7 weights respectively. Since S is present in the visited list thus, we are not considering the C->S path. Now C will enter the visited list. Now the next node with the minimum total path is S->D->E, i.e. 8. Thus we will expand E.	S,A,D,B
Step 7:- From E we can reach only G3. E will move to the visited list.	S,A,D,B

Uniform Cost Search

Step 8 — In the last, we have 6 active paths . S->B — B is in the visited list; thus will be marked as a dead end. b.Same for S->A->B->C — C has already been visited thus is considered a dead end. Out of the remaining S->A->G1 S->D->C->G2 S->D->C->F S->D->E->G3 Minimum is S->D->C->G2 And also, G2 is one of the destination nodes. Thus, we found our path.	S,A,D,B

Uniform Cost Search

❑In this way we can find the path with the minimum cumulative cost from a start node to ending node – S->D->C->G2 with cost total cost as 13(marked with green colour).



Uniform Cost Search

→Advantages

•It helps to find the path with the lowest cumulative cost inside a weighted graph having a different cost associated with each of its edge from the root node to the destination node.

•It is considered to be an optimal solution since, at each state, the least path is considered to be followed.

→Disadvantage

•The open list is required to be kept sorted as priorities in priority queue needs to be maintained.

•The storage required is exponentially large.

•The algorithm may be stuck in an infinite loop as it considers every possible path going from the root node to the destination node.

Breadth First Search

- Breadth-first search (BFS) is an algorithm for traversing or searching tree or graph data structures.
- It starts at the tree root (or some arbitrary node of a graph, sometimes referred to as a ‘search key’), and explores all of the neighbor nodes at the present depth prior to moving on to the nodes at the next depth level.
- It is implemented using a queue(FIFO).
- BFS traverses the tree “shallowest node first”, it would always pick the shallower branch until it reaches the solution (or it runs out of nodes, and goes to the next branch).

Breadth First Search

❑Algorithm Breadth-first search (BFS)

- Declare a queue and insert the starting vertex.
- Initialize a **visited** array and mark the starting vertex as visited.
- Follow the below process till the queue becomes empty:
  - ➢Remove the first vertex of the queue.
  - ➢Mark that vertex as visited.
  - ➢Insert all the unvisited neighbors of the vertex into the queue.

Breadth First Search

❑Algorithm Breadth-first search (BFS)

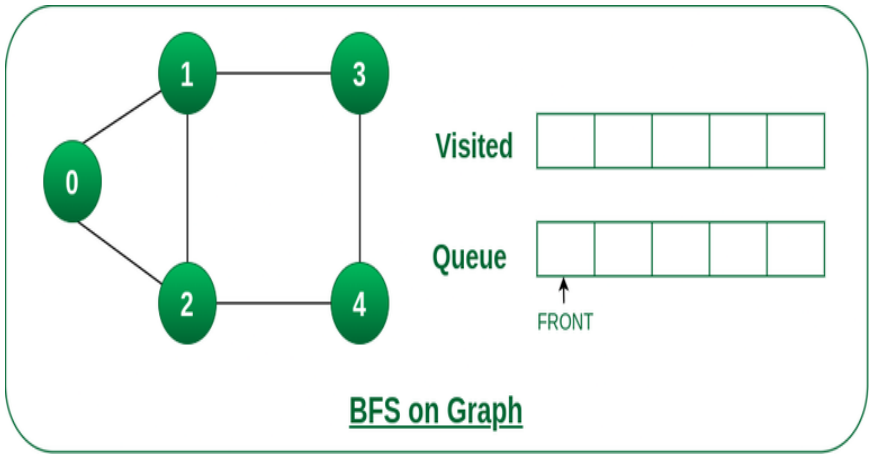
```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure

    node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    frontier ← a FIFO queue with node as the only element
    explored ← an empty set
    loop do
        if EMPTY?(frontier) then return failure
        node ← POP(frontier)  /* chooses the shallowest node in frontier */
        add node.STATE to explored
        for each action in problem.ACTIONS(node.STATE) do
            child ← CHILD-NODE(problem, node, action)
            if child.STATE is not in explored or frontier then
                if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
                frontier ← INSERT(child, frontier)
```

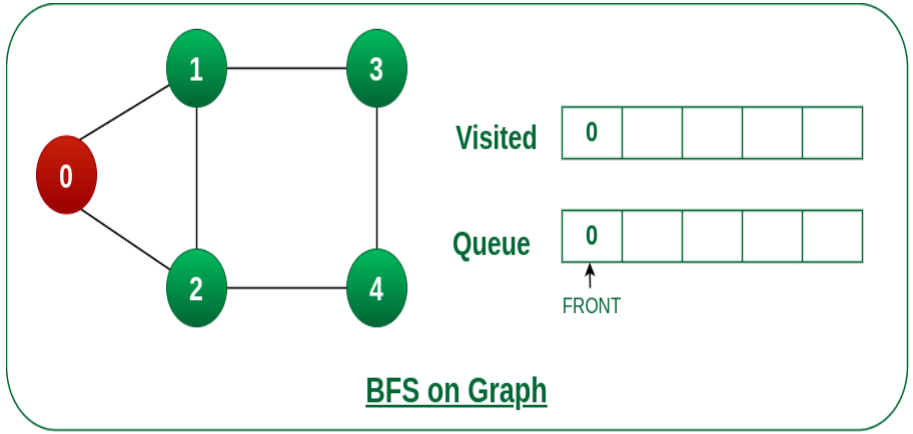
Figure 3.11 Breadth-first search on a graph.

Breadth First Search

❑**Step1:** Initially queue and visited arrays are empty.

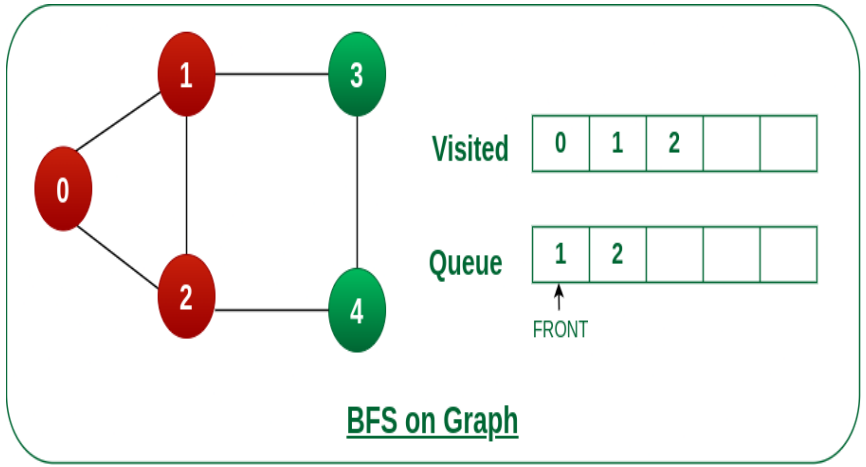


❑**Step2**: Push node 0 into queue and mark it visited.

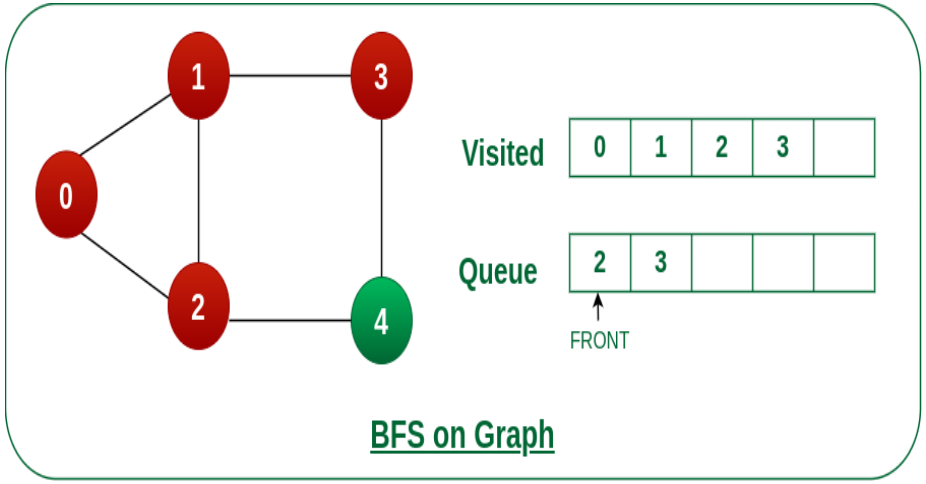


Breadth First Search

❑**Step 3**: Remove node 0 from the front of queue and visit the unvisited neighbours and push them into queue.

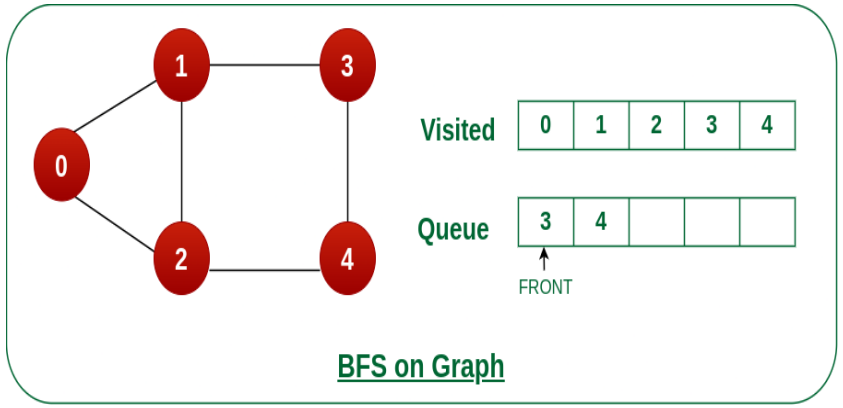


❑**Step 4**: Remove node 1 from the front of queue and visit the unvisited neighbours and push them into queue.



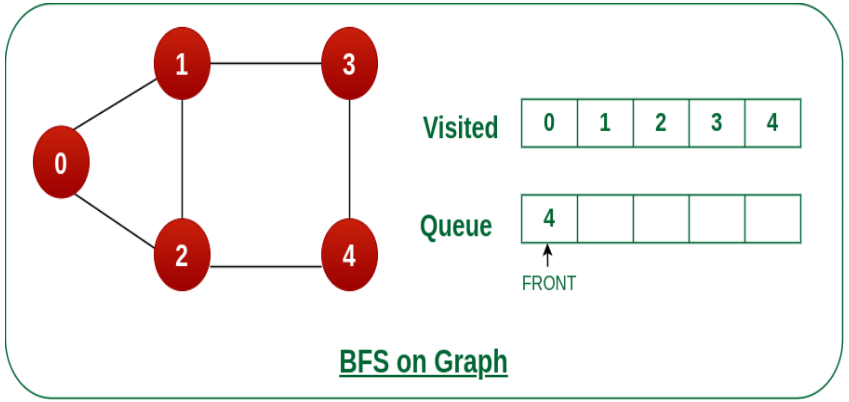
Breadth First Search

❑***Step 5:*** *Remove node 2 from the front of queue and visit the unvisited neighbours and push them into queue.*



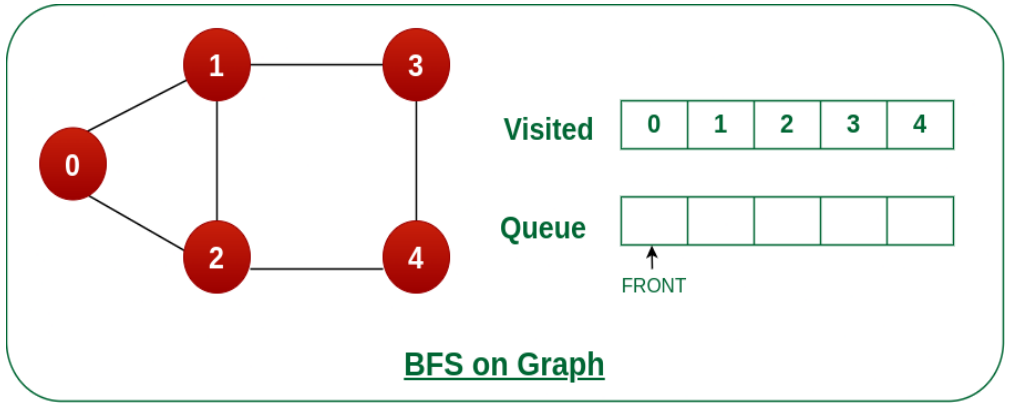
❑**Step 6**: Remove node 3 from the front of queue and visit the unvisited neighbours and push them into queue.

As we can see that every neighbours of node 3 is visited, so move to the next node that are in the front of the queue.



Breadth First Search

❑**Steps 7:** Remove node 4 from the front of queue and and visit the unvisited neighbours and push it hem into queue. As we can see that every neighbours of node 4 are visited, so move to the next node that is in the front of the queue.



❑Now, Queue becomes empty, So, terminate these process of iteration.

Breadth First Search

❑Advantages

- BFS will *provide a solution* if any solution exists.
- If there are more than one solution for a given problem, then BFS will provide *minimum solution* which requires the least number of steps.

❑Disadvantage

- It requires *lot of memory* since each level of the tree must be saved into memory to expand the next level.
- BFS needs *lot of time* if the solution is far a way from the root.

Breadth First Search

❑**Time complexity**: Equivalent to the number of nodes traversed in BFS until the shallowest solution. T(n) = 1 + n^2 + n^3 + ... + n^s = O(n^s)

- s = the depth of the shallowest solution.
- n^i = number of nodes in level i.

❑**Space complexity**: Equivalent to how large can the fringe get. S(n) = O(n^s)

Breadth First Search

- **Completeness:** BFS is complete, meaning for a given search tree, BFS will come up with a solution if it exists.
- **Optimality:** BFS is optimal as long as the costs of all edges are equal.

Depth First Search

- Depth-first search (DFS) is a recursive algorithm to search all the vertices of a tree data structure or a graph.
- The DFS algorithm starts with the initial node of graph G and goes deeper until we find the goal node or the node with no children.
- Because of the recursive nature, stack data structure can be used to implement the DFS algorithm.
- The process of implementing the DFS is similar to the BFS algorithm.

Depth First Search

❑The step by step process to implement the DFS traversal is given as follows:

1. First, create a stack with the total number of vertices in the graph.
2. Now, choose any vertex as the starting point of traversal, and push that vertex into the stack.
3. After that, push a non-visited vertex (adjacent to the vertex on the top of the stack) to the top of the stack.
4. Now, repeat steps 3 and 4 until no vertices are left to visit from the vertex on the stack's top.
5. If no vertex is left, go back and pop a vertex from the stack.
6. Repeat steps 2, 3, and 4 until the stack is empty.

Depth First Search

- The **applications** of using the DFS algorithm are given as follows:
  - DFS algorithm can be used to implement the topological sorting.
  - It can be used to find the paths between two vertices.
  - It can also be used to detect cycles in the graph.
  - DFS algorithm is also used for one solution puzzles.
  - DFS is used to determine if a graph is bipartite or not.

Depth First Search

❑Algorithm:

Step 1: SET STATUS = 1 (ready state) for each node in G

Step 2: Push the starting node A on the stack and set its STATUS = 2 (waiting state)

Step 3: Repeat Steps 4 and 5 until STACK is empty

Step 4: Pop the top node N. Process it and set its STATUS = 3 (processed state)

Step 5: Push on the stack all the neighbors of N that are in the ready state (whose STATUS = 1) and set their STATUS = 2 (waiting state)

[END OF LOOP]

Depth First Search

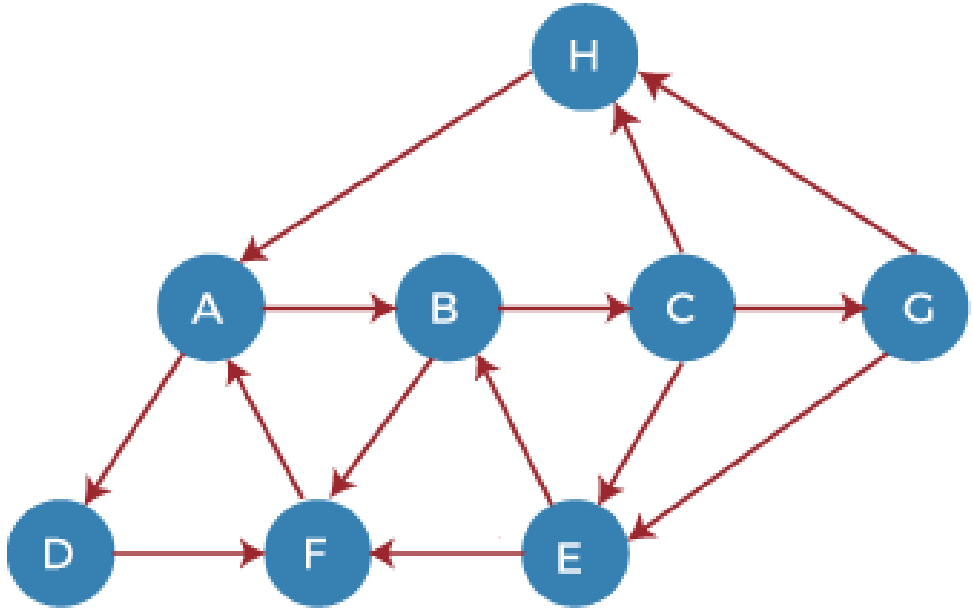
❑Pseudocode:

```
DFS(G,v)   ( v is the vertex where the search starts )
      Stack S := {};   ( start with an empty stack )
      for each vertex u, set visited[u] := false;
      push S, v;
      while (S is not empty) do
        u := pop S;
        if (not visited[u]) then
          visited[u] := true;
          for each unvisited neighbour w of uu
            push S, w;
        end if
      end while
    END DFS()
```

Depth First Search

❑Example of DFS algorithm

❑Now, let's understand the working of the DFS algorithm by using an example. In the example given below, there is a directed graph having 8 vertices.



Adjacency Lists

A : B, D
B : C, F
C : E, G, H
G : E, H
E : B, F
F : A
D : F
H : A

Depth First Search

❑**Step 1** - First, push H onto the stack.

STACK: H

❑**Step 2** - POP the top element from the stack, i.e., H, and print it. Now, PUSH all the neighbors of H onto the stack that are in ready state.

Print: H]STACK: A

❑**Step 3** - POP the top element from the stack, i.e., A, and print it. Now, PUSH all the neighbors of A onto the stack that are in ready state.

Print: A

STACK: B, D

❑**Step 4** - POP the top element from the stack, i.e., D, and print it. Now, PUSH all the neighbors of D onto the stack that are in ready state.

Print: D

STACK: B, F

Depth First Search

❑**Step 5** - POP the top element from the stack, i.e., F, and print it. Now, PUSH all the neighbors of F onto the stack that are in ready state.

Print: F

STACK: B

❑**Step 6** - POP the top element from the stack, i.e., B, and print it. Now, PUSH all the neighbors of B onto the stack that are in ready state.

Print: B

STACK: C

❑**Step 7** - POP the top element from the stack, i.e., C, and print it. Now, PUSH all the neighbors of C onto the stack that are in ready state.

Print: C

STACK: E, G

Depth First Search

❑**Step 8** - POP the top element from the stack, i.e., G and PUSH all the neighbors of G onto the stack that are in ready state.

Print: G

STACK: E

❑**Step 9** - POP the top element from the stack, i.e., E and PUSH all the neighbors of E onto the stack that are in ready state.

Print: E

STACK:

❑**Now, all the graph nodes have been traversed, and the stack is empty.**

Depth First Search

❑Complexity of Depth-first search algorithm

- The time complexity of the DFS algorithm is **O(V+E)**, where V is the number of vertices and E is the number of edges in the graph.
- The space complexity of the DFS algorithm is O(V).

Depth Limited Search

- Depth limited search is the new search algorithm for uninformed search.
- The unbounded tree problem happens to appear in the depth-first search algorithm, and it can be fixed by imposing a boundary or a limit to the depth of the search domain.
- We denote this limit by l, and thus this provides the solution to the infinite path problem that originated earlier in the DFS algorithm.
- Thus, Depth limited search can be called an extended and refined version of the DFS algorithm.
- In a nutshell, we can say that to avoid the infinite loop status while executing the codes, and depth limited search algorithm is being executed into a finite set of depth called depth limit.

Depth Limited Search

❑Algorithm

1. The start node or node 1 is added to the beginning of the stack.
2. Then it is marked as visited, and if node 1 is not the goal node in the search, then we push second node 2 on top of the stack.
3. Next, we mark it as visited and check if node 2 is the goal node or not.
4. If node 2 is not found to be the goal node, then we push node 4 on top of the stack.
5. Now we search in the same depth limit and move along depth-wise to check for the goal nodes.
6. If Node 4 is also not found to be the goal node and depth limit is found to be reached, then we retrace back to nearest nodes that remain unvisited or unexplored.
7. Then we push them into the stack and mark them visited.
8. We continue to perform these steps in iterative ways unless the goal node is reached or until all nodes within depth limit have been explored for the goal.

Depth Limited Search

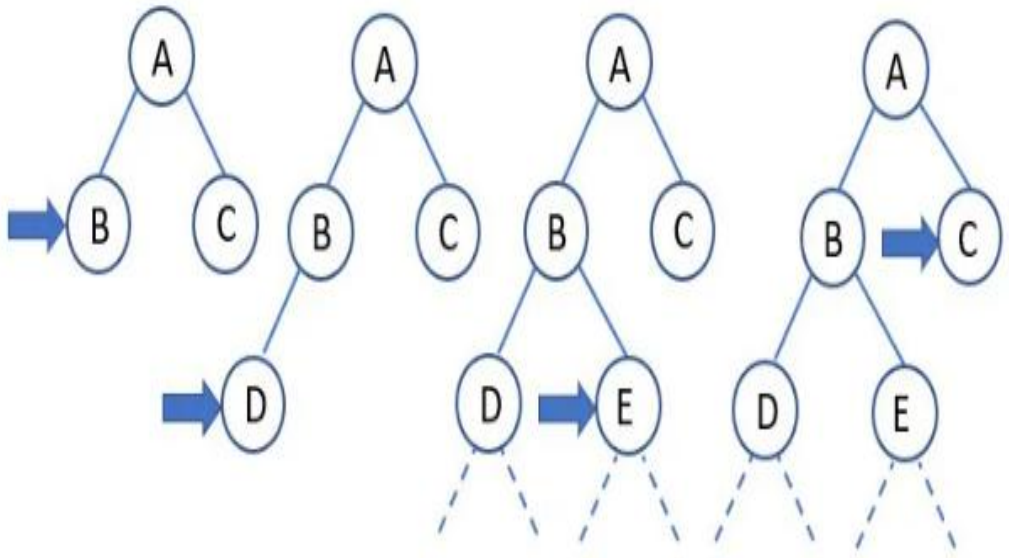
```
function DEPTH-LIMITED-SEARCH(problem, limit) returns a solution, or failure/cutoff
  return RECURSIVE-DLS(MAKE-NODE(problem.INITIAL-STATE), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns a solution, or failure/cutoff
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  else if limit = 0 then return cutoff
  else
      cutoff_occurred? ← false
      for each action in problem.ACTIONS(node.STATE) do
          child ← CHILD-NODE(problem, node, action)
          result ← RECURSIVE-DLS(child, problem, limit − 1)
          if result = cutoff then cutoff_occurred? ← true
          else if result ≠ failure then return result
      if cutoff_occurred? then return cutoff else return failure
```

Figure 3.17 A recursive implementation of depth-limited tree search.

Example of DLS Process

If we fix the depth limit to 2, DLS can be carried out similarly to the DFS until the goal node is found to exist in the tree's search domain.



Depth Limited Search

❑Advantages

- Depth limited search is better than DFS and requires less time and memory space.
- DFS assures that the solution will be found if it exists infinite time.
- There are applications of DLS in graph theory particularly similar to the DFS.
- To combat the disadvantages of DFS, we add a limit to the depth, and our search strategy performs recursively down the search tree.

❑Disadvantage

- The depth limit is compulsory for this algorithm to execute.
- The goal node may not exist in the depth limit set earlier, which will push the user to iterate further adding execution time.
- The goal node will not be found if it does not exist in the desired limit.

Depth Limited Search

❑Performance Measures

- **Completeness:** The DLS is a *incompleteness algorithm* in general except the case when the goal node is the shallowest node, and it is beyond the depth limit, i.e. l < d, and in this case, we never reach the goal node.
- **Optimality:** The DLS is a *non-optimal* algorithm since the depth that is chosen can be greater than d (l>d). Thus DLS is not optimal if l > d
- **Time complexity is expressed as**: It is similar to the DFS, i.e. $O(b^l)$, where l is the set depth limit.
- **Space Complexity is expressed as:** It is similar to DFS*e.* $O(b^l)$, where l is specified depth limit.

Iterative Deepening Depth-First-Search

- In the uninformed searching strategy, the BFS and DFS have not been so ideal in searching the element in optimum time and space.
- The algorithms only guarantee that the path will be found in exponential time and space.
- So we found a method where we can use the amalgamation of space competence of DFS and optimum solution approach of BFS methods, and there we develop a new method called iterative deepening using the two of them.
- The main idea here lies in utilizing the re-computation of entities of the boundary instead of stocking them up.
- Every re-computation is made up of DFS and thus it uses less space.

Iterative Deepening Depth-First-Search

❑**Algorithm**: Now let us also consider using BFS in iterative deepening search.

1. Consider making a breadth-first search into an iterative deepening search.
2. We can do this by having aside a DFS which will search up to a limit. It first does searching to a pre-defined limit depth to depth and then generates a route length1.
3. This is done by creating routes of length 1 in the DFS way. Next, it makes way for routes of depth limit 2, 3 and onwards.
4. It even can delete all the preceding calculation all-time at the beginning of the loop and iterate.
5. Hence at some depth eventually the solution will be found if there is any in the tree because the enumeration takes place in order.

Iterative Deepening Depth-First-Search

❑**Algorithm**:

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution, or failure
    for depth = 0 to ∞ do
        result ← DEPTH-LIMITED-SEARCH(problem, depth)
        if result ≠ cutoff then return result
```

Figure 3.18 The iterative deepening search algorithm, which repeatedly applies depth-limited search with increasing limits. It terminates when a solution is found or if the depth-limited search returns *failure*, meaning that no solution exists.

Iterative Deepening Depth-First-Search

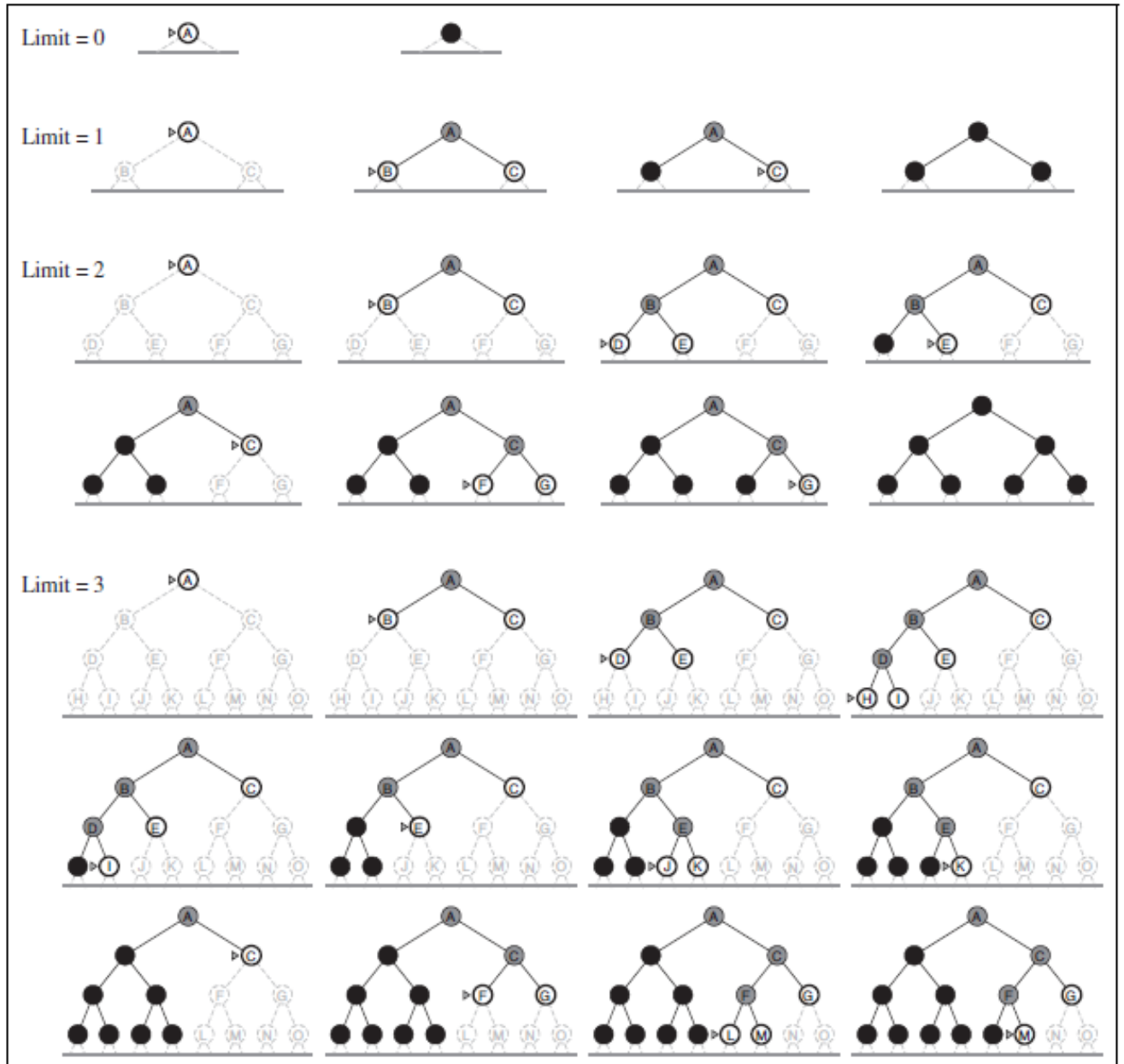
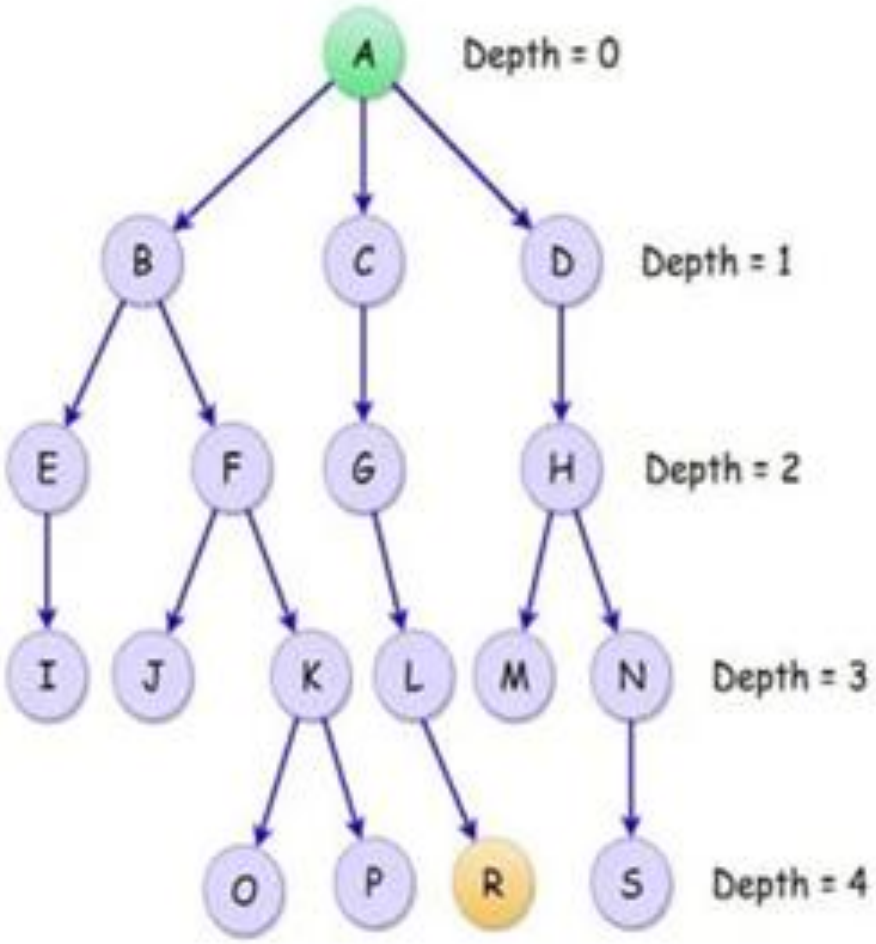


Figure 3.19 Four iterations of iterative deepening search on a binary tree.

Iterative Deepening Depth-First-Search

❑**Example:**



DEPTH = {0, 1, 2, 3, 4}

DEPTH LIMITS	IDDFS
0	A
1	ABCD
2	ABEFCGDH
3	ABEIFJKCGLDHMN
4	ABEIFJKOPCGLRDHMNS

Iterative Deepening Depth-First-Search

❑**Advantages**

- IDDFS gives us the hope to find the solution if it exists in the tree.
- When the solutions are found at the lower depths say n, then the algorithm proves to be efficient and in time.
- The great advantage of IDDFS is found in-game tree searching where the IDDFS search operation tries to improve the depth definition, heuristics, and scores of searching nodes so as to enable efficiency in the search algorithm.
- Another major advantage of the IDDFS algorithm is its quick responsiveness. The early results indications are a plus point in this algorithm. This followed up with multiple refinements after the individual iteration is completed.
- Though the work is done here is more yet the performance of IDDFS is better than single BFS and DFS operating exclusively.

❑**Disadvantage**

- The time taken is exponential to reach the goal node.
- The main problem with IDDFS is the time and wasted calculations that take place at each depth.
- The situation is not as bad as we may think of especially when the branching factor is found to be high.

Iterative Deepening Depth-First-Search

❑**Performance Measures:**The IDDFS is a *completeness and optimal algorithm.*

▪**Time complexity is expressed as**:

$O(b^d)$and here d is defined as goal depth.

▪**Space Complexity is expressed as:**

$O(b)$ and here b is defined as total nodes.

Informed (Heuristic) Search Methods

- The algorithms have information on the goal state, which helps in more efficient searching.
- This information is obtained by something called a ***heuristic Search.***
- 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 help 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 (Heuristic) Search Methods

❑Heuristics Function:

- 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 reasonable time.
- Heuristic function estimates how close a state is to the goal.
- 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

Informed (Heuristic) Search Methods

❑Pure Heuristic Search

- Pure heuristic search is the simplest form of heuristic search algorithms. It expands nodes based on their heuristic value h(n). It maintains two lists, OPEN and CLOSED list. In the CLOSED list, it places those nodes which have already expanded and in the OPEN list, it places nodes which have yet not been expanded.
- On each iteration, each node n with the lowest heuristic value is expanded and generates all its successors and n is placed to the closed list. The algorithm continues unit a goal state is found.
- In the informed search we will discuss two main algorithms which are given below:
- **Best First Search Algorithm(Greedy search)**
- **A* Search Algorithm**

Best First Search

- Greedy best-first search algorithm always selects the path which appears best at that moment.
- It is the combination of depth-first search and breadth-first search algorithms.
- It uses the heuristic function and search.
- Best-first search allows us to take the advantages of both algorithms.
- With the help of best-first search, at each step, we can choose the most promising node.
- 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.

$$f(n)= g(n)$$

- Were, h(n)= estimated cost from node n to the goal.
- The greedy best first algorithm is implemented by the priority queue.

Best First Search

❑Algorithm:

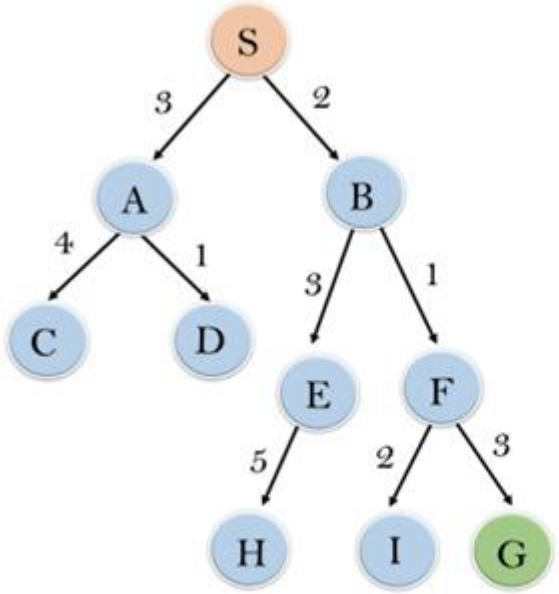
1. *Step 1:* Place the starting node into the OPEN list.
2. *Step 2:* If the OPEN list is empty, Stop and return failure.
3. *Step 3:* Remove the node n, from the OPEN list which has the lowest value of h(n), and places it in the CLOSED list.
4. *Step 4*: Expand the node n, and generate the successors of node n.
5. *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.
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.
7. *Step 7*: Return to Step 2.

Best First Search

❑**Example:**

❑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

Best First Search

❑**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]

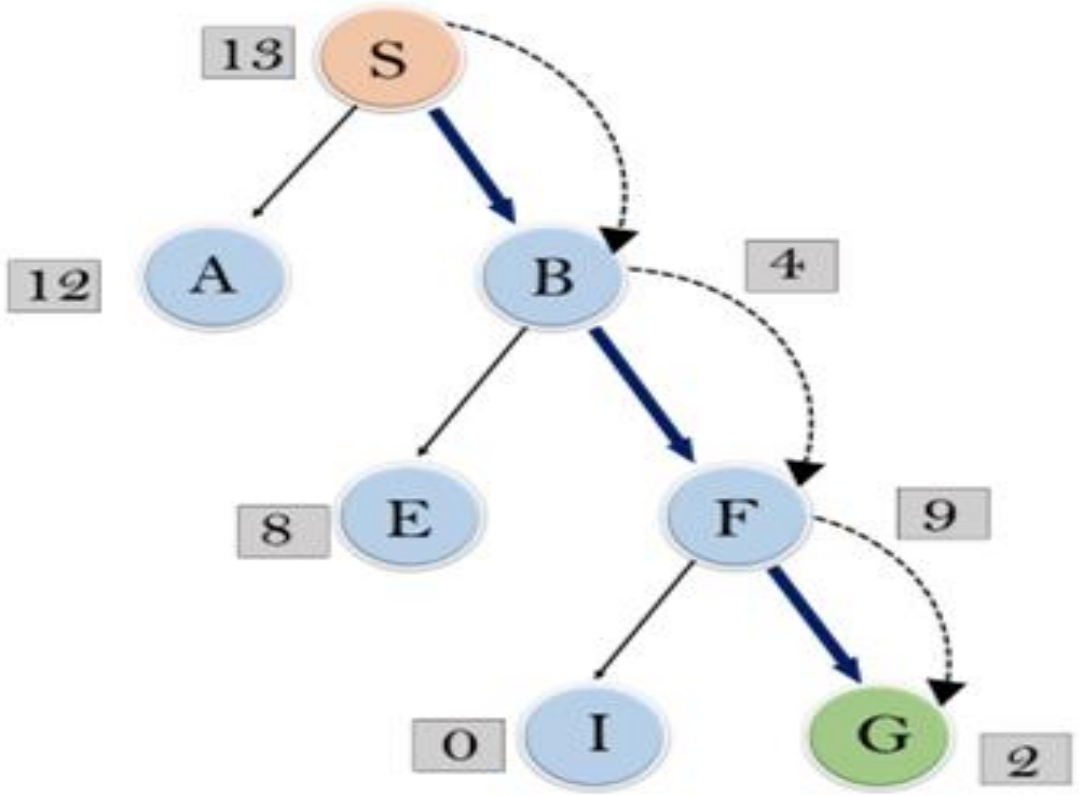
•**Iteration 3:** Open [I, G, E, A], Closed [S, B, F]

: Open [I, E, A], Closed [S, B, F, G]

•Hence the final solution path will be:

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

Best First Search



Best First Search

❑**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.

Best First Search

❑Performance Measure

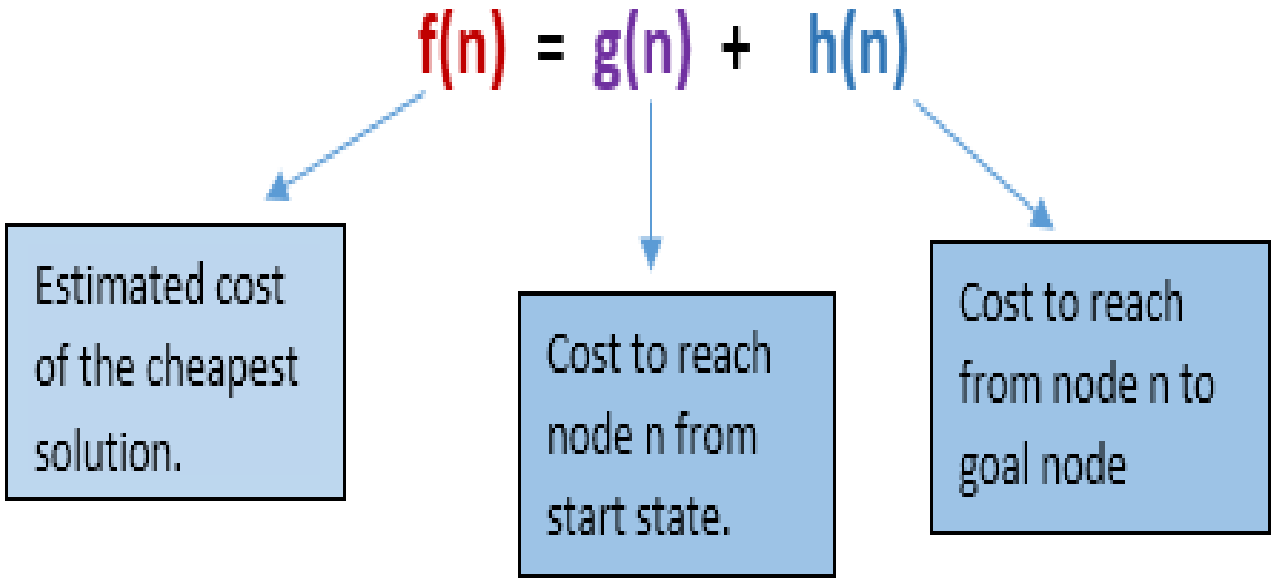
- **Time Complexity:** The worst case time complexity of Greedy best first search is **$O(b^m)$.**
- **Space Complexity:** The worst case space complexity of Greedy best first search is **$O(b^m)$**. Where, m is the maximum depth of the search space.
- **Complete:** Greedy best-first search is also *incomplete*, even if the given state space is finite.
- **Optimal:** Greedy best first search algorithm is *not optimal.*

A* Search

- 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).

A* Search

❑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.**



A* Search

❑A* Search Algorithm:

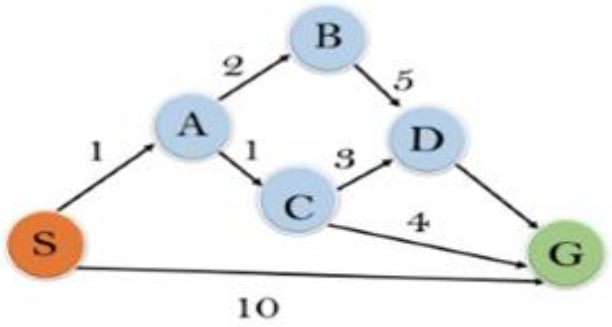
1. **Step1:** Place the starting node in the OPEN list.
2. **Step 2:** Check if the OPEN list is empty or not, if the list is empty then return failure and stops.
3. **Step 3:** Select the node from the OPEN list which has the smallest value of evaluation function (g+h), if node n is goal node then return success and stop, otherwise
4. **Step 4:** Expand node n and generate all of its successors, and put n into the closed list. For each successor n', check whether n' is already in the OPEN or CLOSED list, if not then compute evaluation function for n' and place into Open list.
5. **Step 5:** Else if node n' is already in OPEN and CLOSED, then it should be attached to the back pointer which reflects the lowest g(n') value.
6. **Step 6:** Return to **Step 2**.

A* Search

❑Example

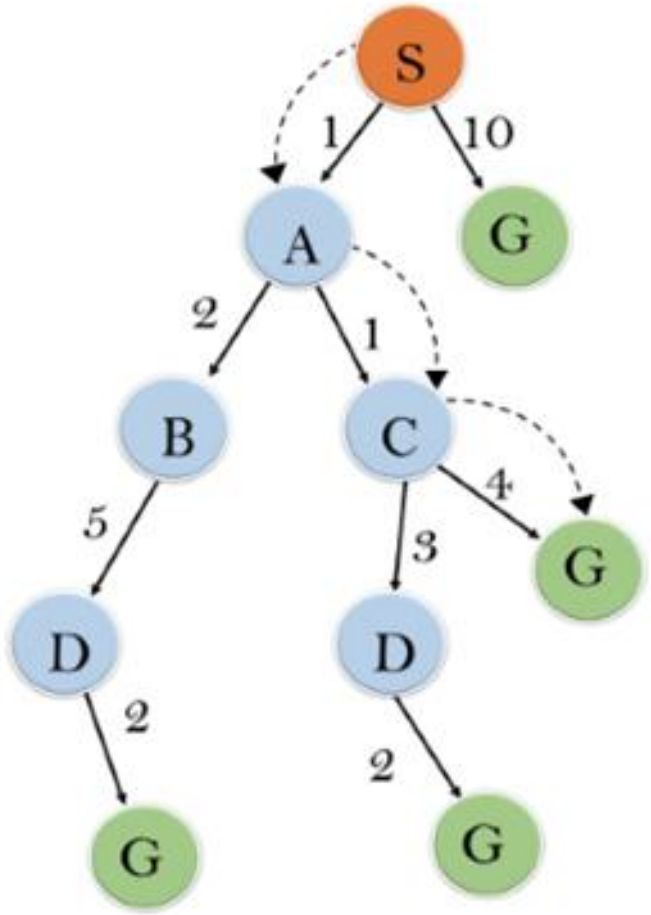
❑In this example, we will traverse the given graph using the A* algorithm.

❑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.



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

A* Search



A* Search

- **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, 7), (S-->G, 10)}
- **Iteration 4** will give the final result, as

S--->A--->C--->G

it provides the optimal path with cost 6.

A* Search

- **Initialization:** {(S, 5)}
- **Iteration1:** {(S--> A, 3), (S-->G, 10)}
- **Iteration2:** {(S--> A-->C, 5), (S--> A-->B, 7), (S-->G, 10)}
- **Iteration3:** {(S--> A-->C--->G, 5), (S--> A-->C--->D, 11), (S--> A-->B, 7), (S-->G, 10)}
- **Iteration 4** will give the final result, as

S--->A--->C--->G

it provides the optimal path with cost 5.

A* Search

❑**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 main drawback of A* is memory requirement as it keeps all generated nodes in the memory, so it is not practical for various large-scale problems.

A* Search

❑Complexity

- **Time Complexity**: The time complexity of A* search algorithm depends on heuristic function, and the number of nodes expanded is exponential to the depth of solution d. So the time complexity is **O(b^d)**, where b is the branching factor.
- **Space Complexity**: The space complexity of A* search algorithm is **O(b^d)**

A* Search

❑Performance Measure

•**Complete:** A* algorithm is complete as long as:

❑Branching factor is finite.

❑Cost at every action is fixed.

•**Optimal:** A* search algorithm is optimal if it follows below two conditions:

❑**Admissible:** the first condition requires for optimality is that h(n) should be an admissible heuristic for A* tree search. An admissible heuristic is optimistic in nature.

❑**Consistency:** Second required condition is consistency for only A* graph-search.

Note for Students

- This power point presentation is for lecture, therefore it is suggested that also utilize the text books and lecture notes.