

# LECTURE NOTES-UNIT II SOLVING PROBLEMS BY SEARCHING

ECS302  
ARTIFICIAL  
INTELLIGENCE  
3/4 B.Tech [B3, B6]

R. PAVANI  
DEPARTMENT OF  
CSE,GIT,GU

## Module II Lecture Notes [Problem solving]

### Syllabus

#### *Problem Solving by Search and Exploration: Solving Problems by Searching:*

*Problem solving agents, example problems, searching for solutions, Uninformed search strategies, avoiding repeated states, searching with partial information; Informed Search and Exploration: Informed (heuristic) search strategies, heuristic functions, local search algorithms and optimization problems, local search in continuous spaces.*

### Problem Solving by Search:

An important aspect of intelligence is goal-based problem solving.

The solution of many problems can be described by finding a sequence of actions that lead to a desirable goal. Each action changes the state and the aim is to find the sequence of actions and states that lead from the initial (start) state to a final (goal) state.

A well-defined problem can be described by:

- **Initial state**
- **Operator or successor function** - for any state  $x$  returns  $s(x)$ , the set of states reachable from  $x$  with one action
- **State space** - all states reachable from initial by any sequence of actions
- **Path** - sequence through state space
- **Path cost** - function that assigns a cost to a path. Cost of a path is the sum of costs of individual actions along the path
- **Goal test** - test to determine if at goal state

### What is Search?


- Search is one of the operational tasks that characterize AI programs best. Almost every AI program depends on a search procedure to perform its prescribed functions.
- Problems are typically defined in terms of state, and solution corresponds to goal states. Problem solving using search technique performs two sequence of steps:

(i) *Define the problem* - Given problem is identified with its required "initial and goal state".

(ii) *Analyze the problem* - The "best search technique" for the given problem is chosen from different AI search technique which derives one or more goal states in minimum number of states.

### Types of problems:

In general the problem can be classified under anyone of the following four types which depends on two important properties. They are


- (i) Amount of knowledge of the agent on the state and action description. 
- (ii) How the agent is connected to its environment through its percepts and actions?

The four different types of problems are:

- (i) Single state problem
- (ii) Multiple state problems
- (iii) Contingency problem
- (iv) Exploration problem

## 1. Problem Solving Agents:

Problem solving agent is one kind of goal based agent, where the agent decides what to do by finding sequence of actions that lead to desirable states. The complexity arises here is the knowledge about the formulation process, (from current state to outcome action) of the agent.

If the agent understood the definition of problem, it is relatively straight forward to construct a search process for finding solutions, which implies that problem solving agent should be an intelligent agent to maximize the performance measure. The sequence of steps done by the intelligent agent to maximize the performance measure: 

- i) *Goal formulation* - based on current situation is the first step in problem solving. Actions that result to a failure case can be rejected without further consideration.
- ii) *Problem formulation* - is the process of deciding what actions and states to consider and follows goal formulation.

- iii) *Search* - is the process of finding different possible sequence of actions that lead to state of known value, and choosing the best one from the states.
- iv) *Solution* - a search algorithm takes a problem as input and returns a solution in the form of action sequence.
- v) *Execution phase* - if the solution exists, the action it recommends can be carried out.

Thus, we have a simple "formulate, search, execute" design for the agent. This is shown in below figure 1.1.

```

function SIMPLE-PROBLEM-SOLVING-AGENT (Percept) Returns an action
inputs: percept, a percept
static: seq, an action sequence, initially empty
state, some description of the current world state
goal, a goal, initially null
problem, a problem formulation
state  $\leftarrow$  UPDATE-STATE(State, Percept),
if seq is empty then do
  goal  $\leftarrow$  FORMULATE-GOAL(State)
  problem  $\leftarrow$  FORMULATE-PROBLEM(State, Goal),
  seq  $\leftarrow$  SEARCH(problem)
  action  $\leftarrow$  FIRST(Seq)
  seq  $\leftarrow$  REST(seq)
return action
  
```

**Figure 1.1** A simple problem-solving agent. It first formulates a goal and a problem, searches for a sequence of actions that would solve the problem, and then executes the actions one at a time. When this is complete, it formulates another goal and starts over.

*Search* - choosing the best one from the sequence of actions

*Formulate-problem* - sequence of actions and states that lead to goal state.

*Update-state* - initial state is forced to next state to reach the goal state

### 1.1. Well-defined problems and solutions:

A problem can be defined formally by four components:

1. *Initial state*
2. *Successor function*
3. *Goal test*
4. *Path cost*

*Initial state* - is that the agent starts in.

*Successor function (S)* - Given a particular state  $x$ ,  $S(x)$  returns a set of ordered pair  $\langle \text{Action}, \text{Successor} \rangle$ . Where each action is one of the legal actions in state  $x$  and each successor is a state that can be reached from  $x$  by applying an action.

*Goal test* - This determines whether a given state is a goal state. Sometimes there is an explicit set of possible goal states, and the test simply checks whether the given state is one of them.

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.

[**Path (state space)** - The sequence of action leading from one state to another]

[**State space (or) state set space** - The set of all possible states reachable from the initial state by any sequence of actions.]

[**Step cost** - Taking action 'a' to go from state  $x$  to state  $y$  is denoted by  $c(x, a, y)$ ]

A *solution* to a problem is a path from the initial state to a goal state. The effectiveness of a search can be measured using three factors. They are:

- 1 Solution is identified or not?
2. Is it a good solution? If yes, then path cost to be minimum.
3. Search cost of the problem that is associated with time and memory required to find a solution.

*For Example:* Imagine an agent in the city of Arad, Romania, enjoying a touring holiday. Now, suppose the agent has a nonrefundable ticket 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. The agent's task is to find out which sequence of actions will get it to a goal state.

This process of looking for such a sequence is called "*search*".

A search algorithm takes a problem as input and returns a solution in the form of an action sequence. Once a solution is found, the actions it recommends can be carried out. This is

called the execution phase.

### 1.2 Formulating problems:

**Initial state:** the initial state for our agent in Romania might be described as  $In(Arad)$

**Successor function:** Given a particular state  $x$ ,  $SUCCESSOR-FN(x)$  returns a set of (action, successor) ordered pairs, where each action is one of the legal actions in state  $x$  and each successor is a state that can be reached from  $x$  by applying the action. For example, from the state  $In(Arad)$ , the successor function for the Romania problem would return

$\{(Go(Sibiu), In(Sibiu)), (Go(Timisoara), In(Timisoara)), (Go(Zerind), In(Zerind))\}$

**Goal test:** The agent's goal in Romania is the singleton set  $\{In(Bucharest)\}$ .

**Path cost:** The step cost of taking action ' $a$ ' to go from state  $x$  to state  $y$  is denoted by  $c(x, a, y)$ .

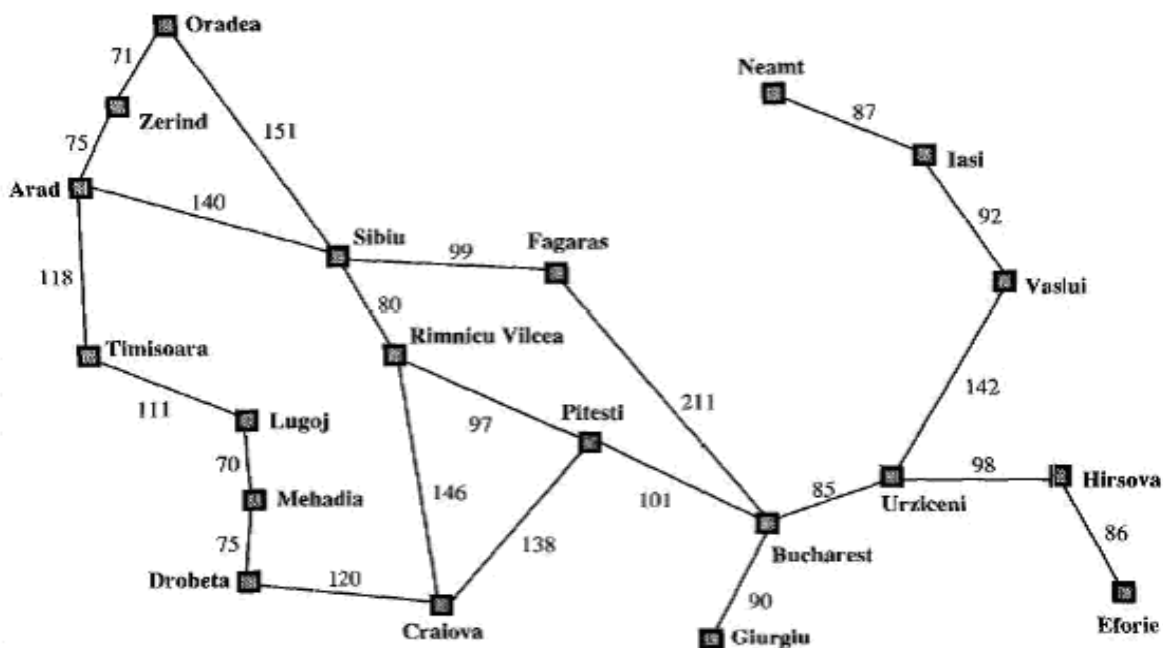


Figure 1.2: Simplified map of Romania

## 2. Example Problems:

The problem-solving approach has been applied to a vast array of task environments. A **toy problem** is intended to illustrate or exercise various problem-solving methods. It can be given a concise, exact description. It can be used easily by different researchers to compare the

performance of algorithms. A *real-world problem* is one whose solutions people actually care about. Some list of best known *toy* and real-world problems.

## 2.1 Toy Problems

i) **Vacuum world Problem States:** The agent is in one of two locations, each of which might or might not contain dirt. Thus there are  $2 * 2^2 = 8$  possible world states.

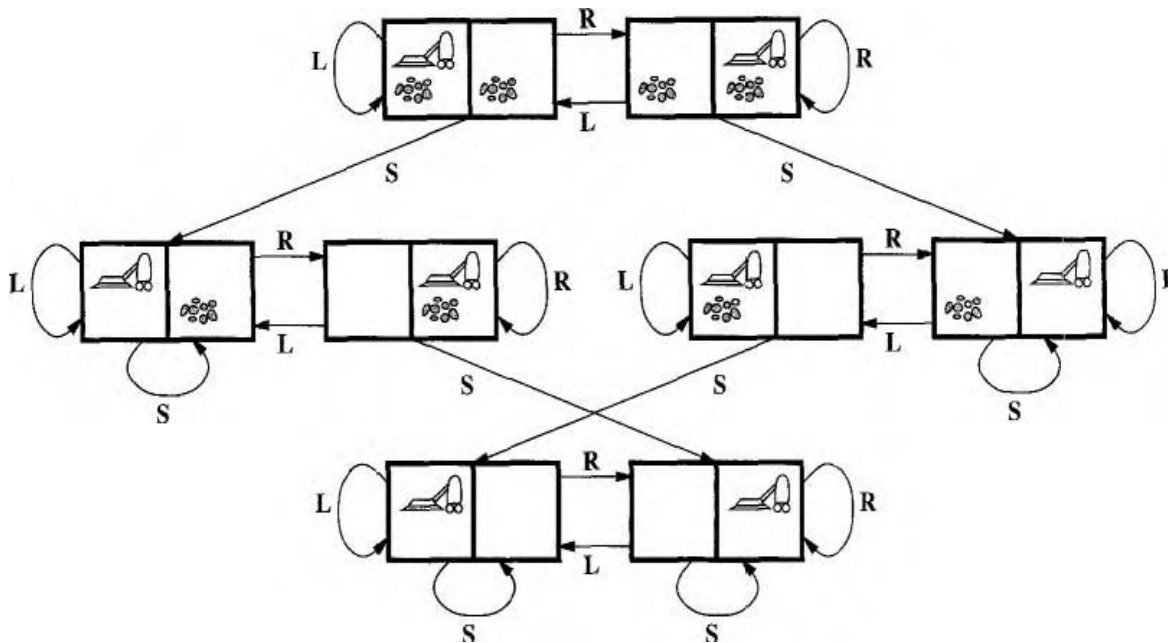
*Initial state:* Any state can be designated as the initial state.

*Successor function:* three actions (Left (L), Right (R), and Suck (S)).

*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.

This problem has discrete locations and the state is determined by both the location and the dirt cleaned. So, a larger environment with 'n' locations has  $n \times 2^n$  states.



**Figure: 2.1** complete state space for Vacuum World

ii) **8-puzzle Problem:** The 8-puzzle problem consists of a  $3 \times 3$  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

2	8	3
1	6	4
7		5

*Initial State*

	1	2
3	4	5
6	7	8

*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.

*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.

This 8- Puzzle belongs to the family of sliding block puzzles which are used as test problems for new search algorithms in AI.

**iii) 8-queens problem:** The goal of the 8-queens problem is to place eight queens on a chessboard such that no queen attacks any other.

(Note: A queen attacks any piece in the same row, column or diagonal)

*States:* Any arrangement of 0 to 8 queens on the board is a state.

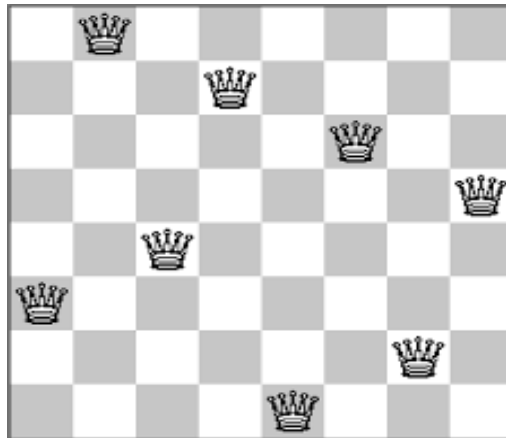
*Initial state:* No queens on the board.

*Successor function:* Add a queen to any empty square.

*Goal test:* 8 queens are on the board, none are attacked by others.

*Path cost:* Zero (search cost only exists)





*Figure 2.2: Solution to 8- queens*

### 2.2 Real-world problems:

**i) Route-finding problem:** This problem can be defined in terms of specified locations and transitions along links between them. Route-finding algorithms are used in a variety of applications, such as routing in computer networks, military operations planning, and airline travel planning systems.

Let us consider the Airline travel problem;

#### *Airline travel problem States:*

**States:** Each is represented by a location (e.g., an airport) and the current time.

**Initial state:** This is specified by the problem.

**Successor function:** This returns the states resulting from taking any scheduled flight (perhaps further specified by seat class and location), leaving later than the current time plus the within-airport transit time, from the current airport to another.

**Goal test:** Are we at the destination by some pre specified time?

**Path cost:** This depends on monetary cost, waiting time, flight time, customs and immigration procedures, seat quality, time of day, type of airplane, frequent-flyer mileage awards, and so on.

**ii) Touring Problems:** Which are closely related to Route-finding problems, but it has an

important difference. For example: Consider the problem “*Visit every city at least once*” as shown in Romania Map fig: 1.2

- In Route finding the actions correspond to trips between adjacent cities
- But state space is quite different i.e., each state must be included (Not only the current location but also set of cities that the agent has visited)
- For example: The initial state would be “Arad ” then visited{Arad}
- Suppose a typical intermediate state would be in “Lugoj” then visited{Arad, Timisoara, Lugoj}
- The goal test would check whether the agent is in Bucharest and all 20 cities have been visited.

### *iii) Travelling sales person Problem: [TSP]*

It is a touring problem in which each city must be visited exactly once. The aim is to find the *shortest* tour. These problems are known as NP-hard, but an enormous amount of effort has been expended to improve the capacity of TSP algorithms.

These algorithms are also used in tasks such as planning movements of Automatic circuit board drills and Stocking machines on shop floors.

### *iv) VLSI layout*

A VLSI layout problem requires positioning millions of components and connections on a chip to minimize area, minimize circuit delays, minimize stray capacitances, and maximize manufacturing yield. The layout problem comes after the logical design phase, and is usually split into two parts: *cell layout* and *channel routing*.

- In cell layout, the primitive components of the circuit are grouped into cells, each of which performs some recognized function. Each cell has a fixed footprint (size and shape) and requires a certain number of connections to each of the other cells. The aim is to place the cells on the chip so that they do not overlap and so that there is room for connecting wires to be placed between the cells.
- Channel routing finds a specific route for each wire through the gaps between the cells.

**v) Robot navigation**

Robot navigation is a generalization of the route-finding problem described earlier. Rather than a discrete set of routes, a robot can move in a continuous space with (in principle) an infinite set of possible actions and states.

- For a circular robot moving on a flat surface, the space is essentially two-dimensional.
- When the robot has arms and legs or wheels that must also be controlled, the search space becomes many-dimensional. Advanced techniques are required just to make the search space finite. In addition to the complexity of the problem, real robots must also deal with errors in their sensor readings and motor controls.

**vi) Automatic assembly sequencing**

Automatic assembly sequencing of complex objects by a robot was first demonstrated by FREDDY (Michie, 1972). In assembly problems, the aim is to find an order in which to assemble the parts of some object.

- If the wrong order is chosen, there will be no way to add some part later in the sequence without undoing some of the work already done. Checking a step in the sequence for feasibility is a difficult geometrical search problem closely related to robot navigation.
- Another important problem is protein design. The goal is to find a sequence of Amino acids that will be fold in to a three- dimensional protein with right properties to cure some disease.

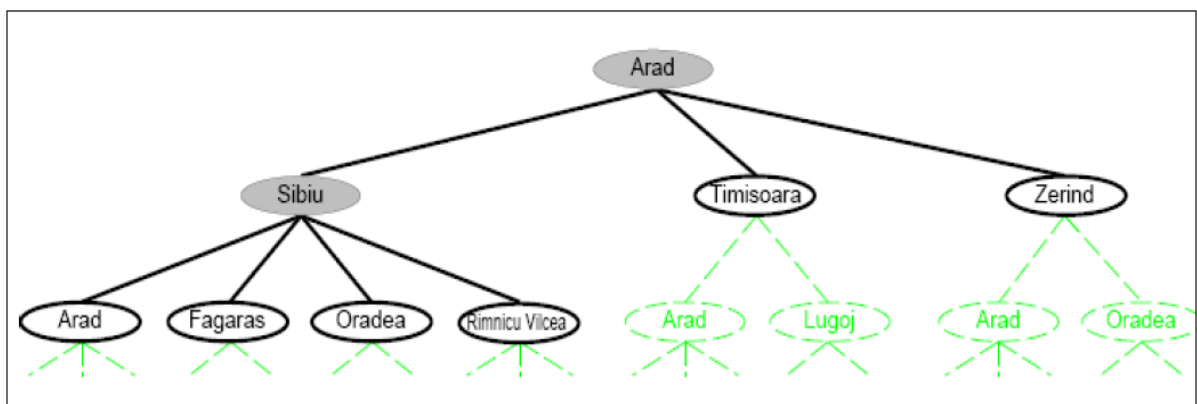
**vii) Internet Searching**

In recent years there is an increase of demand for *software robots* that performs Internet searching i.e., looking for answers to questions (or) for shopping deals. This is a good application for search technique i.e., internet is a graph of nodes connected by links.

### 3. Searching for Solutions

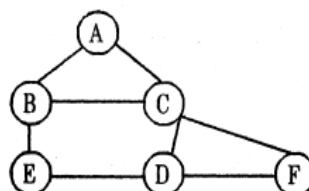
Search techniques use an explicit “*search tree*” that is generated by the initial state and the successor function that together define the state space. In general, we may have a “*search graph*” rather than a search tree, when the same state can be reached from multiple paths.

- Let us consider the example: Search tree for finding a route from “*Arad to Bucharest*” in Romania map.
- Here the Root of search tree is a “Search Node” corresponding to initial state  $In(Arad)$ .
- The first step is to test whether this is a goal state.
- Apply the successor function to the current state, and generate a new set of states. In this case, we get three new states:  $In(Sibiu)$ ,  $In(Timisoara)$ , and  $In(Zerind)$ . Now we must choose which of these three possibilities to consider further.
- Continue choosing, testing, and expanding until either a solution is found or there are no more states to be expanded.
- The choice of which state to expand is determined by the “*search strategy*”.



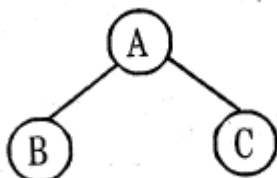
**Figure 3.1** Partial search trees for finding a route from Arad to Bucharest. Nodes that have been expanded are shaded. Nodes that have been generated but not yet expanded are outlined in bold. Nodes that have not yet been generated are shown in faint dashed line.

Consider an example for tree Search algorithm: The task is to find a path to reach F from A

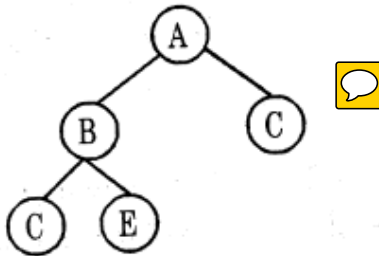


1. Start the sequence with the initial state and check whether it is a goal state or not. If it is a goal state return success. Otherwise perform the following sequence of steps from the initial state (current state) generate and expand the new set of states.
2. The collection of nodes that have been generated but not expanded is called as *fringe*.
3. Each element of the fringe is a *leaf node*, a node with no successors in the tree.
4. Sequence of steps to reach the goal state F from (A = A - C - F)

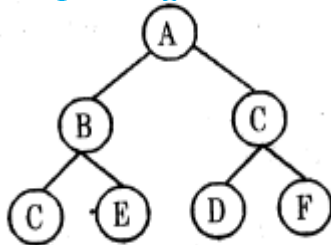
#### Expanding A



#### Expanding B



#### Expanding C



#### A General Tree-Search Algorithm:

**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

There are many ways to represent nodes, but we will assume that a node is a data structure with five components:

**State:** the state in the state space to which the node corresponds

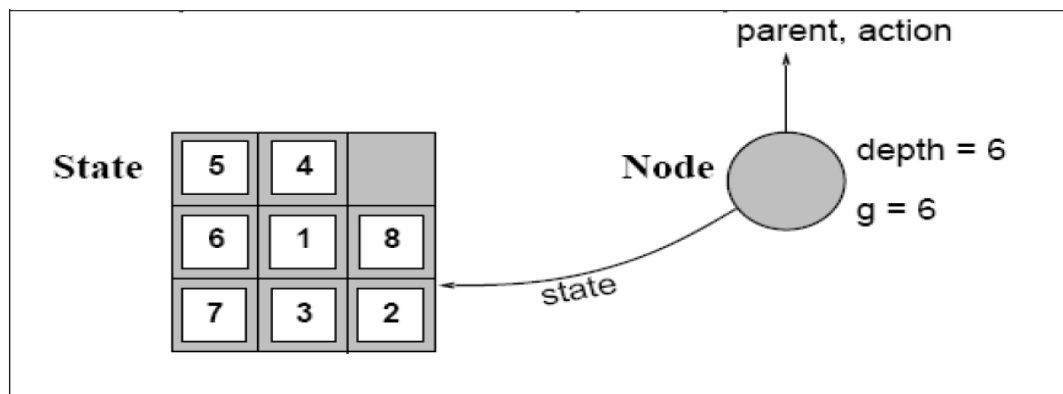
**Parent-node:** the node in the search tree that generated this node;

**Action (rule):** the action that was applied to the parent to generate the node;

**Path-cost:** the cost, traditionally denoted by  $g(n)$ , of the path from the initial state to the node as indicated by parent pointers.

**Depth:** the number of steps along the path from the initial state. The collection of nodes represented in the search tree is defined using set or queue representation.

A node is a book keeping data structure used to represent the search tree. A state corresponds to configuration of the world.



**Figure 3.2** Nodes are data structures from which the search tree is constructed. Each has a parent, a state, Arrows point from child to parent.

### **Fringe:**

Fringe is a collection of nodes that have been generated but not yet been expanded. Each element of the fringe is a leaf node, that is, a node with no successors in the tree. The fringe of each tree consists of those nodes with bold outlines.

The collection of these nodes is implemented as a "Queue".

The operations specified on a queue are as follows:

**MAKE-QUEUE (element...)** creates a queue with the given element(s).

**EMPTY? (Queue)** returns true only if there are no more elements in the queue.

**FIRST (queue)** returns FIRST (queue) and removes it from the queue.

**INSERT (element, queue)** inserts an element into the queue and returns the resulting queue.

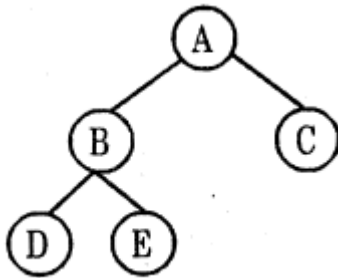
**INSERT-ALL (elements, queue)** inserts a set of elements into the queue and returns the resulting queue.

```
function TREE-SEARCH(problem, fringe) returns a solution, or failure
fringe <- INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
loop do
  if EMPTY?(fringe) then return failure
  node <- REMOVE-FIRST(fringe)
  if GOAL-TEST[problem] applied to STATE[node] succeeds then return SOLUTION(node)
  fringe <- INSERT-ALL(EXPAND(node, problem), fringe)
```

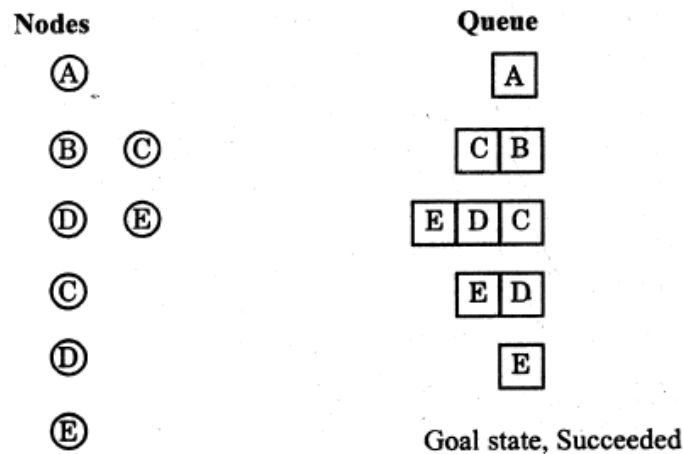
```
function EXPAND(node, problem) returns a set of nodes
successors <- the empty set
for each <action, result> in SUCCESSOR-FN [problem](STATE[node]) do
  S <- a new NODE
  STATE[s] <- result
  PARENT-NODE[s] <- node
  ACTION[s] <- action
  PATH-COST[s] <- PATH-COST[node] + STEP-COST(node, action, s)
  DEPTH[s] <- DEPTH[node] + 1
  add s to successors
return successors
```

**Figure: 3.3** General tree search algorithm with queue representation

Example: Route finding problem



Find a path to reach E using Queuing function in general tree search algorithm.



### **Measuring problem solving performance:**

The output of a problem-solving Algorithm is either failure or Solution. The search strategy algorithms are evaluated depends on four important criteria's. They are:

- (i) Completeness: The strategy guaranteed to find a solution when there is one.
- (ii) Time complexity: Time taken to run a solution.
- (iii) Space complexity: Memory needed to perform the search.
- (iv) Optimality: If more than one way exists to derive the solution then the best one is Selected

**Definition of branching factor (b):** The number of nodes which is connected to each of the node in the search tree. Branching factor is used to find space and time complexity of the search strategy. The total cost will be calculated as the combination of Search Cost and path cost to find a solution.



## 4. Solving Problems by Searching:

It divides the Algorithm in to two categories: 1) **Uninformed Search Algorithms [Blind search]**  
2) **Informed Search Algorithms [Heuristic search]**

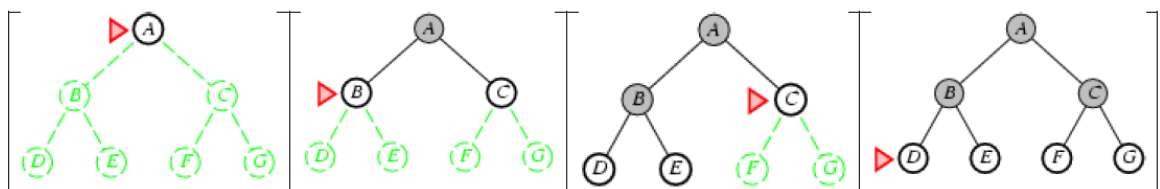
Algorithms Comes Under Uninformed Search;

- Breadth First Search [BFS]
- Uniform-cost search
- Depth-first search [DFS]
- Depth-limited search [DLS]
- Iterative deepening depth-first search [IDDS / IDS]
- Bidirectional Search

### i) *Breadth-first search*

It is a simple strategy in which the root node is expanded first, then all the successors of the root node are expanded next, then their successors, and so on. In general, all the nodes are expanded at a given depth in the search tree before any nodes at the next level are expanded.

Breadth-first search can be implemented by calling TREE-SEARCH with an empty fringe that is a first-in-first-out (FIFO) queue, assuring that the nodes that are visited first will be expanded first. In other words, calling **TREE-SEARCH (Problem, FIFO-QUEUE())** results in a breadth-first search. The FIFO queue puts all newly generated successors at the end of the queue, which means that shallow nodes are expanded before deeper nodes.



**Figure 4.1:** General expansion of Nodes

### Time complexity for BFS:

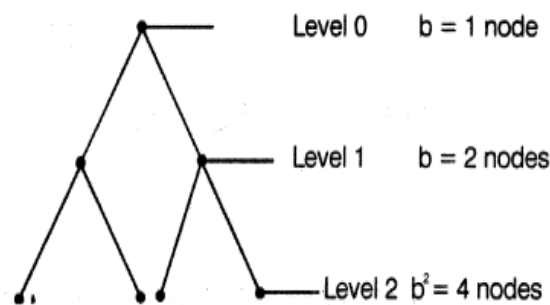
Assume that every state has  $b$  successors. The root of the search tree generates  $b$  nodes at the first level, each of which generates  $b$  more nodes, for a total of  $b^2$  at the second level. Each of these generates  $b$  more nodes, yielding  $b^3$  nodes at the third level, and so on.

Now suppose that the solution is at depth  $d$ . In the worst case, we would expand all but the last node at level  $d$ , generating  $b^{d+1} - b$  nodes at level  $d+1$ .

Then the total number of nodes generated is

$$b + b^2 + b^3 + \dots + b^d + (b^{d+1} - b) = O(b^{d+1}).$$

Every node that is generated must remain in memory, because it is either part of the fringe or is an ancestor of a fringe node. The space complexity is, therefore, the same as the time complexity.



### Advantages:

- BFS will **never get trapped** by **exploring the useless path** forever.
- **If there is a solution** then BFS will **definitely find it out**.
- If there is more than one solution then **BFS can find the minimal that requires less number of steps**.

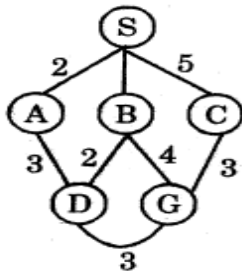
### Disadvantages:

- The main problem of **BFS is its memory requirement**. Since each level of tree must be saved in order to generate the next level, and the amount of memory is proportional to the number of nodes stored.
- Suitable for only smallest instances problem (i.e.) (number of levels to be minimum (or) branching factor to be minimum)

### Algorithm:

```
function BFS{(problem)} returns a solution or failure
return TREE-SEARCH (problem, FIFO-QUEUE())
```

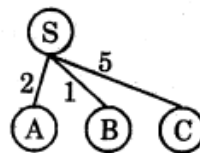
Example: Route finding problem [Find a , path from. S to G using BFS]



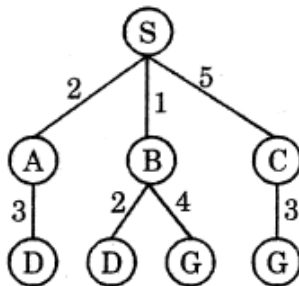
(i)



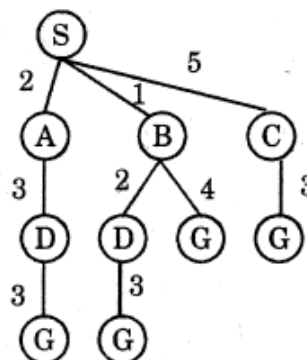
(ii)



(iii)



(iv)

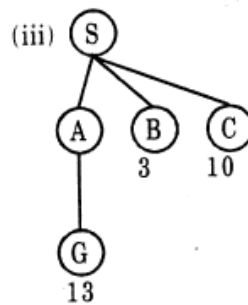
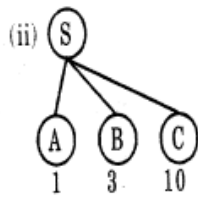
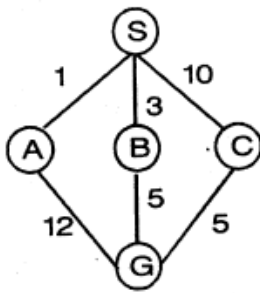


The path in the 2nd depth level is selected, (i.e.,) SBG {or} SCG.

## ii) Uniform-cost search

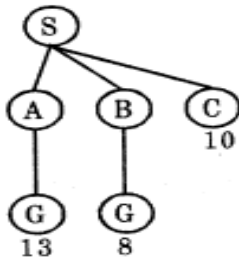
Breadth-first search is optimal when all step costs are equal, because it always expands the **shallowest** unexpanded node. By a simple extension, we can find an algorithm that is optimal with any step cost function. Instead of expanding the shallowest node, *uniform-cost search* **expands the node n with** the **lowest path cost**. Note that if all step costs are equal, this is identical to breadth-first search. Uniform-cost search **does not care about the number of steps** a path has, **but only about their total cost.**

Example: Find a minimum path cost from S to G



Since the value of A is less it is expanded first, but it is not optimal.

B to be expanded next;



**SBG** is the path with minimum path cost. No need to expand the next path SC, because its path cost is high to reach C from S, as well as goal state is reached in the previous path with minimum cost.

**Time and space complexity:** Time complexity is same as breadth first search because instead of depth level the minimum path cost is considered.

**Time complexity:**  $O(b^d)$  **Space complexity:**  $O(b^d)$  **Completeness:** Yes **Optimality:** Yes

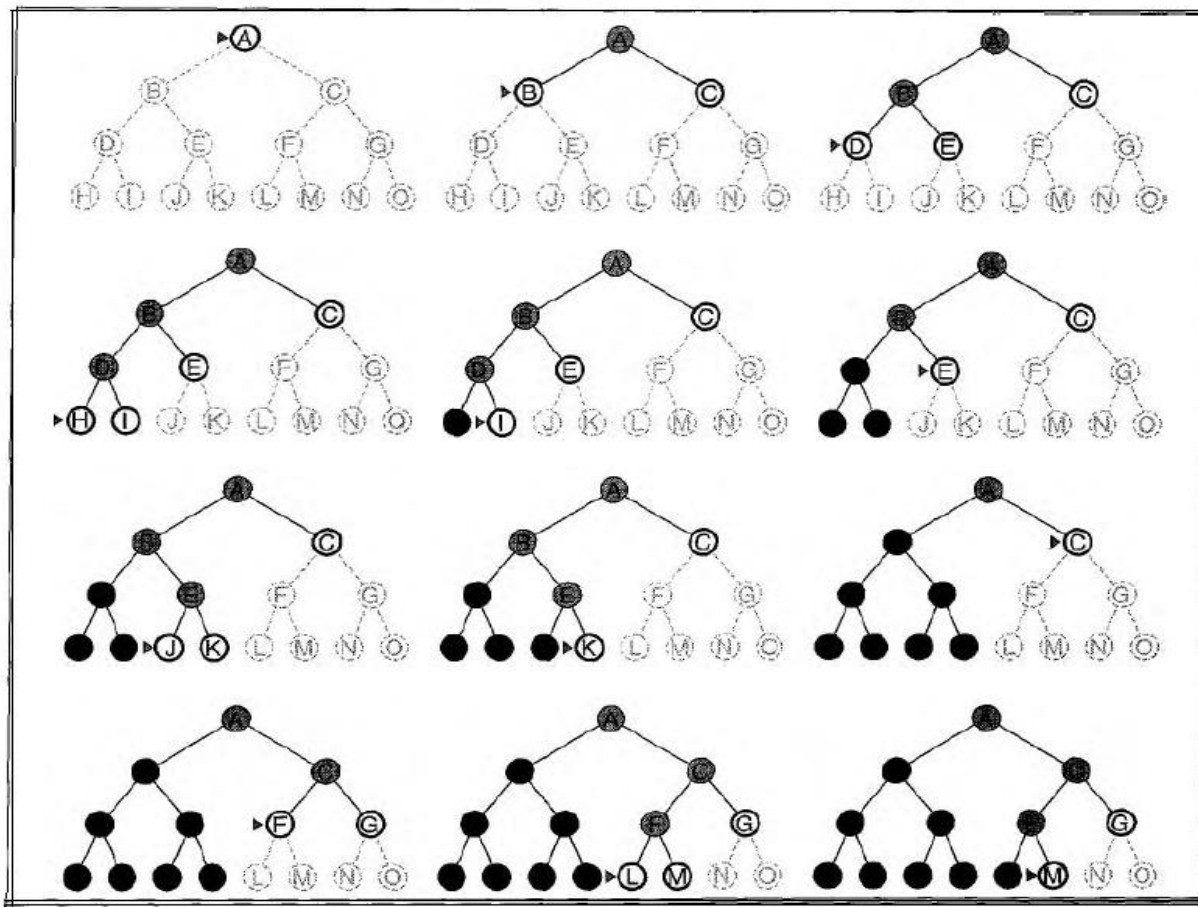
**Advantage:** Guaranteed to find the single solution at minimum path cost.

**Disadvantage:** Suitable for only smallest instances problem.

### iii) Depth- First Search:

Depth-first search always **expands the deepest node in the current fringe** of the search tree. Here the search proceeds immediately to the deepest level of the search tree, where the nodes have no successors.

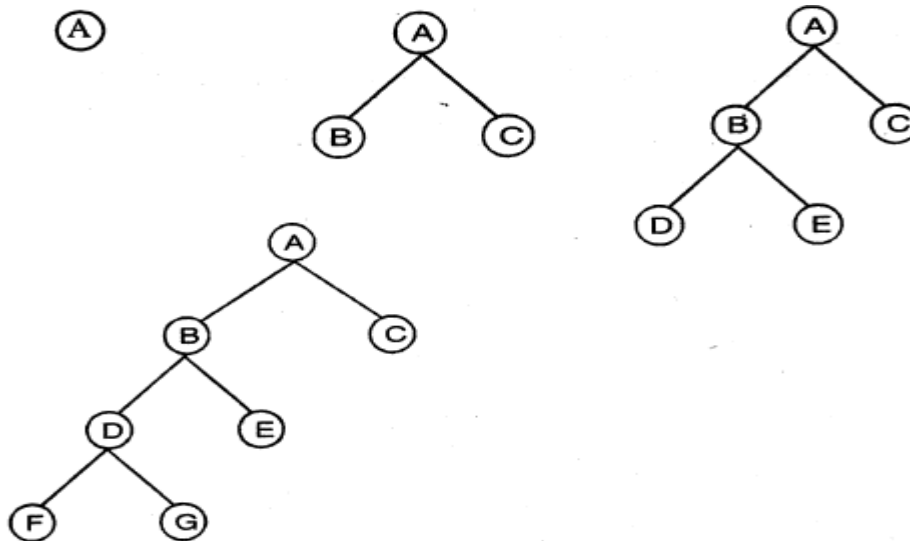
- As those nodes are expanded, they are dropped from the fringe, so then the search "backs up" to the next shallowest node that still has unexplored successors.
- This strategy can be implemented by TREE-SEARCH with a last-in-first-out (LIFO) queue, also known as a stack.



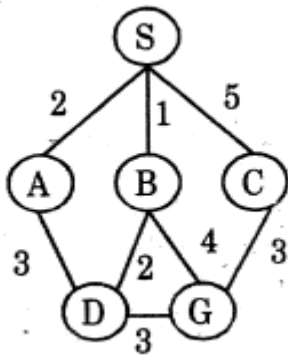
**Figure 4.2** Depth-first search on a binary tree. Nodes that have been expanded and have no descendants in the fringe can be removed from memory; these are shown in black. Nodes at depth 3 are assumed to have no successors and M is the only goal node.

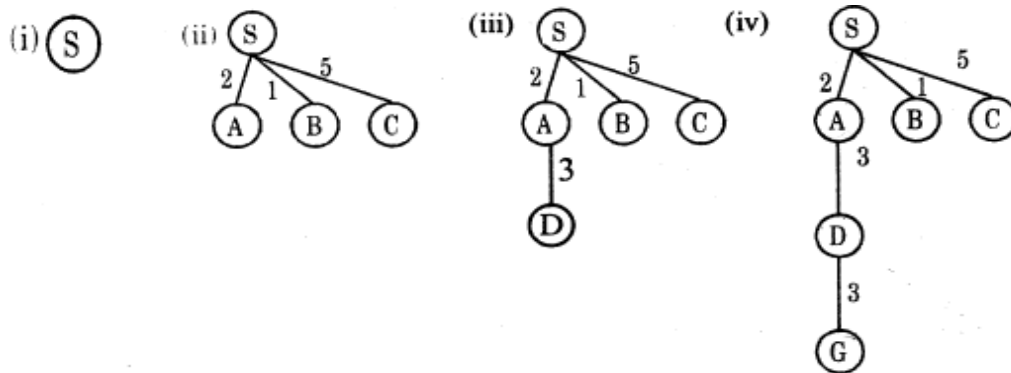
A variant of depth-first search called *backtracking search* uses still less memory. In backtracking, only one successor is generated at a time rather than all successors; each partially expanded node remembers which successor to generate next. In this way, only  $O(m)$  memory is needed rather than  $O(bm)$ . Backtracking search facilitates yet another memory saving (and time-saving) trick: the idea of generating a successor by *modifying* the current state description directly rather than copying it first. This reduces the memory requirements to just one state description and  $O(m)$  actions.

Depth first search tree with 3 level expansions.



Example: Find a path from S to G using DFS





The path in the 3rd depth level is selected. (i.e., S-A-D-G)

### Algorithm:

**function** *DFS*(*problem*) **return** a solution or failure

*TREE-SEARCH* (*problem*, LIFO-QUEUE())

### Time and space complexity:

For a state space with branching factor  $b$  and maximum depth  $m$ , depth-first search requires storage of only  $bm + 1$  nodes. In the worst case depth first search has to expand all the nodes

**Time complexity:**  $O(b^m)$ .

The nodes are expanded towards one particular direction requires memory for only that nodes.

**Space complexity:**  $O(bm)$

$b=2$   $m = 2 \therefore bm=4$

**Completeness:** No

**Optimality:** No

Advantage: If more than one solution exists (or) number of levels is high then DFS is best because exploration is done only in a small portion of the whole space.

Disadvantage: It can make a wrong choice and get stuck going down a very long (or even infinite) path when a different choice would lead to a solution near the root of the search tree. For example, in Figure 4.2, depth-first search will explore the entire left sub tree even if node C is a goal node. If node J were also a goal node, then depth-first search would return it as a solution; hence, depth-first search is not optimal.

#### iv) Depth-limited search

To overcome the problem of DFS here we supply depth-first search with a predetermined depth limit  $l$ . That is, nodes at depth  $l$  are treated as if they have no successors. This approach is called **depth-limited search**. The depth limit solves the infinite-path problem. Unfortunately, it also introduces an additional source of incompleteness if we choose  $l < d$ , that is, the shallowest goal is beyond the depth limit. (This is not unlikely when  $d$  is unknown.) Depth-limited search will also be non optimal if we choose  $l > d$ . Its *time complexity* is  $O(b^l)$  and its *space complexity* is  $O(bl)$ .

Depth-first search can be viewed as a special case of depth-limited search with  $l=10$ .

Sometimes, depth limits can be based on knowledge of the problem. For example, on the map of *Romania* there are 20 cities. Therefore, we know that if there is a solution, it must be of length 19 at the longest, so  $l = 19$  is a possible choice. But in fact if we studied the map carefully, we would discover that any city can be reached from any other city in at most 9 steps. This number, known as the *diameter* of the state space, gives us a better depth limit, which leads to a more efficient depth-limited search. For most problems, however, we will not know a good depth limit until we have solved the problem.

#### Algorithm:

```

function Depth-Limited-Search( problem, limit) returns a solution/fail/cutoff
return Recursive-DLS(Make-Node(Initial-State[problem]), problem, limit)
function Recursive-DLS(node, problem, limit) returns solution/fail/cutoff
Cutoff - occurred? false
if Goal-Test(problem, State[node]) then return Solution(node)
else if Depth[node] = limit then return cutoff
else for each successor in Expand(node, problem) do
  result Recursive-DLS(successor, problem, limit)
  if result = cutoff then cutoff _ occurred? true
  else if result not = failure then return result
if cutoff _ occurred? then return cutoff else return failure
  
```



Depth-limited search can terminate with two kinds of failure:

- 1) the standard failure value indicates no solution;
- 2) The *cutoff* value indicates no solution within the depth limit.

### Time and space complexity:

The worst case time complexity is equivalent to BFS and worst case DFS.

**Time complexity:**  $O(b^l)$  The nodes which is expanded in one particular direction above to be stored.

**Space complexity:**  $O(bl)$

**Optimality:** No, because not guaranteed to find the shortest solution first in the search technique.

**Completeness:** Yes, guaranteed to find the solution if it exists.

Advantage: Cut off level is introduced in the DFS technique

Disadvantage: Not guaranteed to find the optimal solution.

### **v) Iterative deepening search** (or iterative deepening depth-first search)

It is a general strategy, often used in combination with depth-first search that finds the best depth limit. It does this by **gradually increasing the limit**-first 0, then 1, then 2, and so on-until a goal is found. This will occur when the depth limit reaches  $d$ , the depth of the shallowest goal node. The algorithm is shown in Figure 3.14. **Iterative deepening combines the benefits of depth-first and breadth-first search**. Like depth-first search, its memory requirements are very modest:  $O(bd)$  to be precise. Like breadth-first search, it is complete when the branching factor is finite and optimal when the path cost is a non decreasing function of the depth of the node.

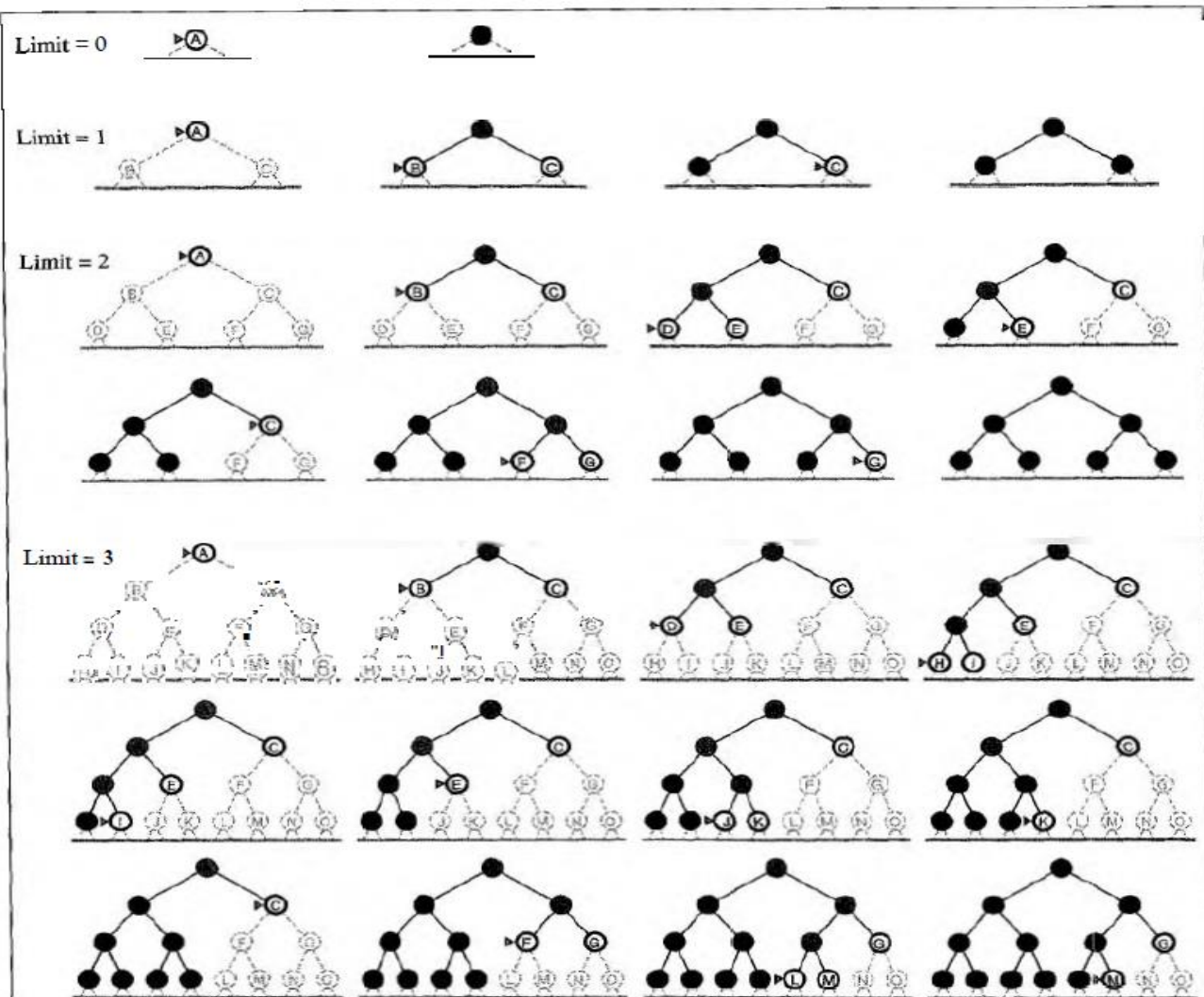
### Algorithm:

**function ITERATIVE-DEEPENING-SEARCH** (*problem*) **returns** a solution, or failure

**inputs :** *problem*

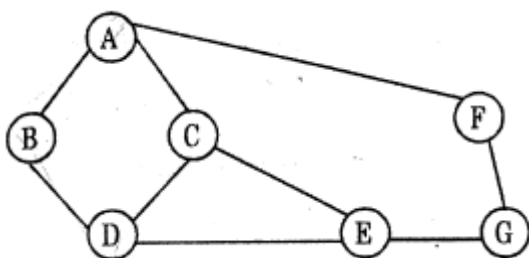
**for** *depth*  $\leftarrow$  0 **do** *result*  $\leftarrow$  **DEPTH-LIMITED-SEARCH**(*problem*, *depth*)

**if** *result* not equal to *cutoff* **then** return *result* .



*Figure 4.3 Four iterations of iterative deepening search on a binary tree.*

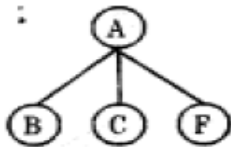
Example: Find a path from A to G



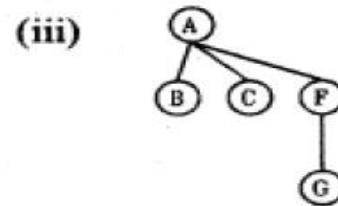
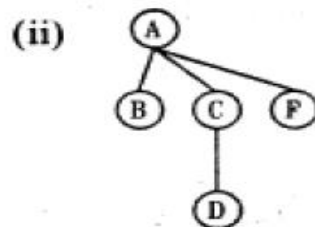
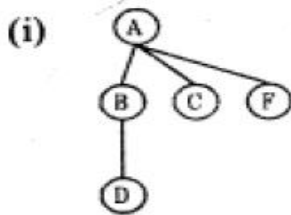
Limit = 0



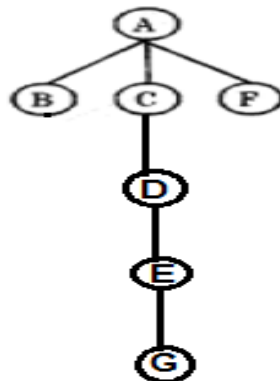
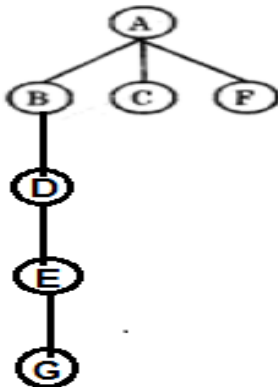
Limit = 1



Limit = 2



Limit = 4



Solution: The goal state G can be reached from A in four ways. They are:

1. A - B - D - E - G ----- Limit 4
2. A - C - D - E - G ----- Limit 4
3. A - C - E - G ----- Limit 3
4. A - F - G ----- Limit 2

Since it is an iterative deepening search it selects lowest depth limit (i.e.) **A-F-G** is selected as the solution path.

Time complexity:

- Iterative deepening search may seem wasteful, because states are generated multiple times. It turns out this is not very costly. The reason is that in a search tree with the same (or nearly the same) branching factor at each level, most of the nodes are in the bottom level, so it does not matter much that the upper levels are generated multiple times.
- In an iterative deepening search, the nodes on the bottom level (depth  $d$ ) are generated once, those on the next to bottom level are generated twice, and so on, up to the children of the root, which are generated  $d$  times. So the total number of nodes generated is

$$N(1DS) = (d)b + (d-1)b^2 + \dots + (1)b^d,$$

Which gives a time complexity of  $O(b^d)$ . We can compare this to the nodes generated by a breadth-first search:

$$N(BFS) = b + b^2 + \dots + b^d + (b^{d+1} - b).$$

For example, if  $b = 10$  and  $d = 5$ , the numbers are

$$N(1DS) = 50 + 400 + 3,000 + 20,000 + 100,000 = 123,450$$

$$N(BFS) = 10 + 100 + 1,000 + 10,000 + 100,000 + 999,990 = 1,111,100$$

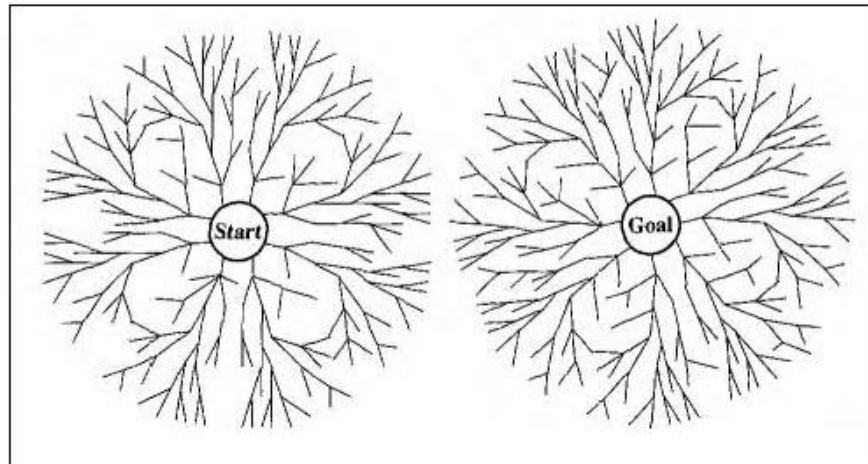
Advantage: This method is preferred for large state space and the depth of the search is not known.

Disadvantage: Many states can be expanded multiple times.

**vi) Bidirectional search:**

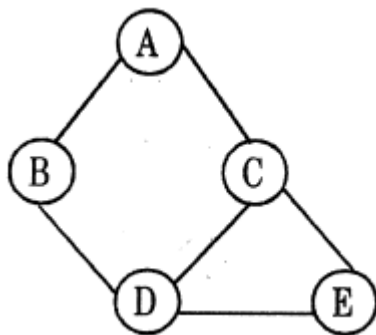
This is a search strategy that can simultaneously search both the directions (i.e.)

- Forward from the initial state and
- Backward from the goal, and stops when the two searches meet in the middle.

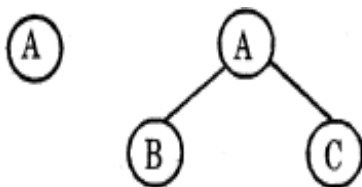


**Figure 4.4** A schematic view of a bidirectional search that is about to succeed, when a Branch from the Start node meets a Branch from the goal node.

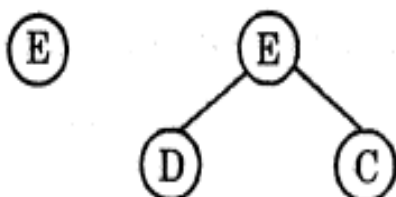
Example: Find a path from A to E.



Search from forward (A):



Search from backward (E):



Here backward and forward searches done at same time will lead to a solution i.e.,  $b^{d/2} + b^{d/2} = O(2b^{d/2})$ .

This space requirement is most significant weakness of bidirectional search.

- This algorithm is complete and optimal if both searches are breadth-first.
- Here the reduction in time complexity makes bidirectional search as an attractive, but if we search backwards? This is not an easy one.
- Consider suppose, Let the *predecessors* of a state  $x$ ,  $Pred(x)$ , be all those states that have  $x$  as a successor. Bidirectional search requires that  $Pred(x)$  be efficiently computable.
- The easiest case is when all the actions in the state space are reversible, so that  $Pred(x) = Succ(x)$ .

Consider the question of what we mean by "the goal" in searching "backward from the goal." For the 8-puzzle and for finding a route in Romania, there is just one goal state, so the backward search is very much like the forward search. If there are several *explicitly listed* goal states-for example, the two dirt-free goal states, then we can construct a new dummy goal state whose immediate predecessors are all the actual goal states. Alternatively, some redundant node generations can be avoided by viewing the set of goal states as a single state, each of whose predecessors is also a set of states-specifically, the set of states having a corresponding successor in the set of goal states.

### Advantage:

Time and space complexity is reduced

### Disadvantage:

If two searches doesn't meet at all complexity arises in search technique. In backward search calculating predecessor is difficult task. If more than one goal state exists then explicit multiple state searches is required.

→ More efficient search;

Example:  $b=10$ ,  $d=6$  then BFS will be examined as  $[b^d]$  i.e.,  $[10^6] = 1,000,000$  nodes

Where as Bidirectional search is  $[2b^{d/2}]$  i.e.,  $[2 \times 10^3] = 2,000$  nodes

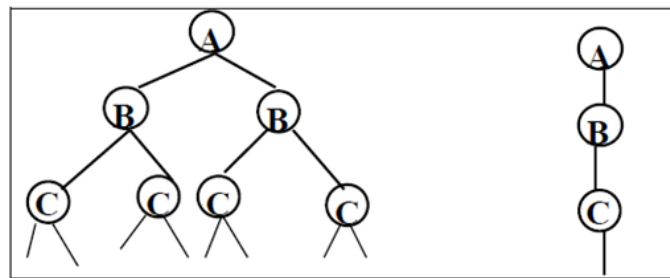
Comparing uninformed search strategies:

Criterion	Breadth First	Uniform Cost	Depth First	Depth Limited	Iterative Deepening	Bi direction
Complete	Yes	Yes	No	No	Yes	Yes
Time	$O(b^d)$	$O(b^d)$	$O(bm)$	$O(b^l)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^d)$	$O(bm)$	$O(bl)$	$O(bd)$	$O(b^{d/2})$
Optimal	Yes	Yes	No	No	Yes	Yes

## 5. Avoiding Repeated States

In searching, time is wasted by expanding states that have already been encountered and expanded before.

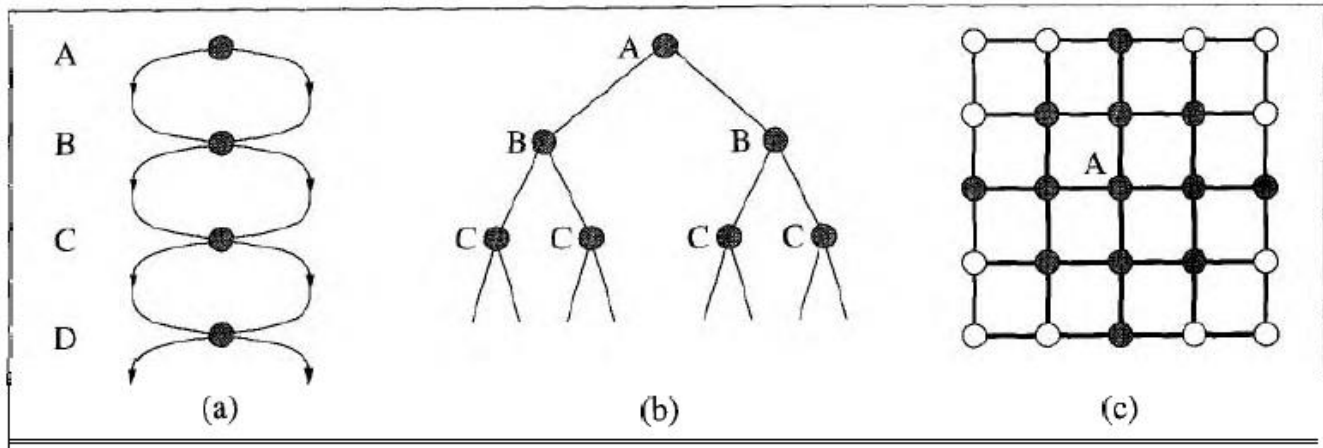
- For some problems repeated states are unavoidable. The search trees for these problems are infinite. If we prune some of the repeated states, we can cut the search tree down to finite size.
- Considering search tree up to a fixed depth, eliminating repeated states yields an exponential reduction in search cost. Repeated states can cause a solvable problem to become unsolvable if the algorithm does not detect them. Repeated states can be the source of great inefficiency: identical sub trees will be explored many times!



In the extreme case, a state space of size  $d + 1$  (Figure 5.1(a)) becomes a tree with  $2d$  leaves (Figure 5.1(b)). A more realistic example is the *rectangular grid* as illustrated in Figure 5.1(c). On a grid, each state has four successors, so the search tree including repeated states has  $4^d$  leaves; but there are only about  $2d^2$  distinct states within  $d$  steps of any given state. For  $d = 20$ , this means about a trillion nodes but only about 800 distinct states. Repeated states, then, can cause a solvable problem to become unsolvable if the algorithm does not detect them. Detection usually means comparing the node about to be expanded to those that have been expanded already; if a



match is found, then the algorithm has discovered two paths to the same state and can discard one of them.



**Figure 5.1** State spaces that generate an exponentially larger search tree. (a) A state space in which there are two possible actions leading from A to B, two from B to C, and so on. The state space contains  $d + 1$  states, where  $d$  is the maximum depth. (b) The corresponding search tree, which has  $2d$  branches corresponding to the  $2d$  paths through the space. (c) A rectangular grid space. States within 2 steps of the initial state (A) are shown in gray.

*The repeated states can be avoided using three different ways. They are:*

1. Do not return to the state you just came from (i.e) avoid any successor that is the same state as the node's parent.
2. Do not create path with cycles (i.e) avoid any successor of a node that is the same as any of the node's ancestors.
3. Do not generate any state that was ever generated before.

The general TREE-SEARCH algorithm is modified with additional data structure, such as:

**Closed list** - This stores every expanded node.

**Open list** - fringe of unexpanded nodes.

If the current node matches a node on the closed list, then it is discarded and it is not considered for expansion. This is done with **GRAPH-SEARCH** algorithm. This algorithm is efficient for problems with many repeated states.



Algorithm:

```

function GRAPH-SEARCH (problem, fringe) returns a solution, or failure
  closed <- an empty set
  fringe <- INSERT (MAKE-NODE(INITIAL-STATE[problem]), fringe)
  loop do
    if EMPTY?(fringe) then return failure
    node <- REMOVE-FIRST (fringe)
    if GOAL-TEST [problem] (STATE[node]) then return SOLUTION (node)
    if STATE [node] is not in closed then add STATE [node] to closed
    fringe <- INSERT-ALL(EXPAND(node, problem), fringe)
  end

```

The worst-case time and space requirements are proportional to the size of the state space, this may be much smaller than  $O(b^d)$ .

## 6. Searching with Partial Information

When the knowledge of the states or actions is incomplete about the environment, then only partial information is known to the agent. This incompleteness lead to three distinct problem types. They are:

- (i) **Sensor less problems (conformant problems)**: If the agent has no sensors at all, then it could be in one of several possible initial states, and each action might therefore lead to one of possible successor states.
- (ii) **Contingency problems**: If the environment is partially observable or if actions are uncertain, then the agent's percepts provide new information after each action. A problem is called adversarial if the uncertainty is caused by the actions of another agent. To handle the situation of unknown circumstances the agent needs a contingency plan.

(iii) *Exploration problem*: It is an extreme case of contingency problems, where the states and actions of the environment are unknown and the agent must act to discover them.

Consider an example: "Vacuum World Agent"

Case 1: Where the agent know the eight possible states of vacuum world

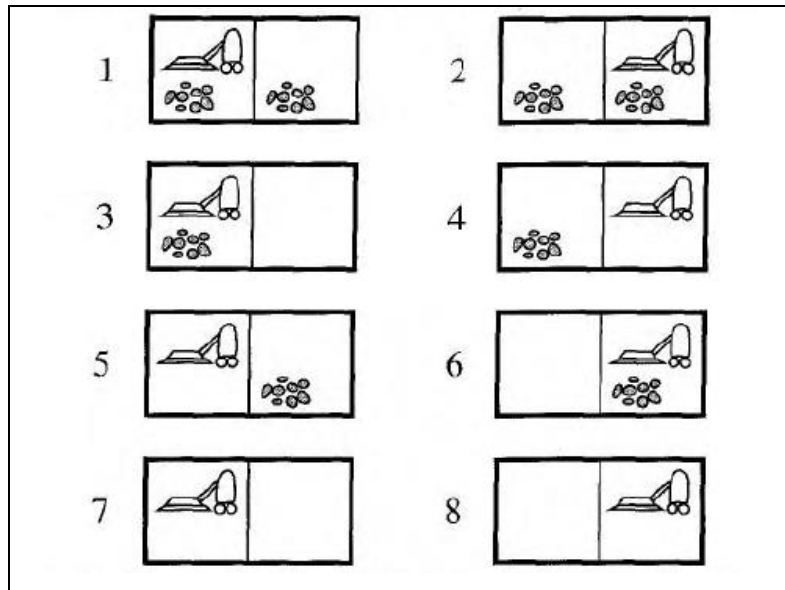


Figure 6.1 Eight possible states of vacuum world

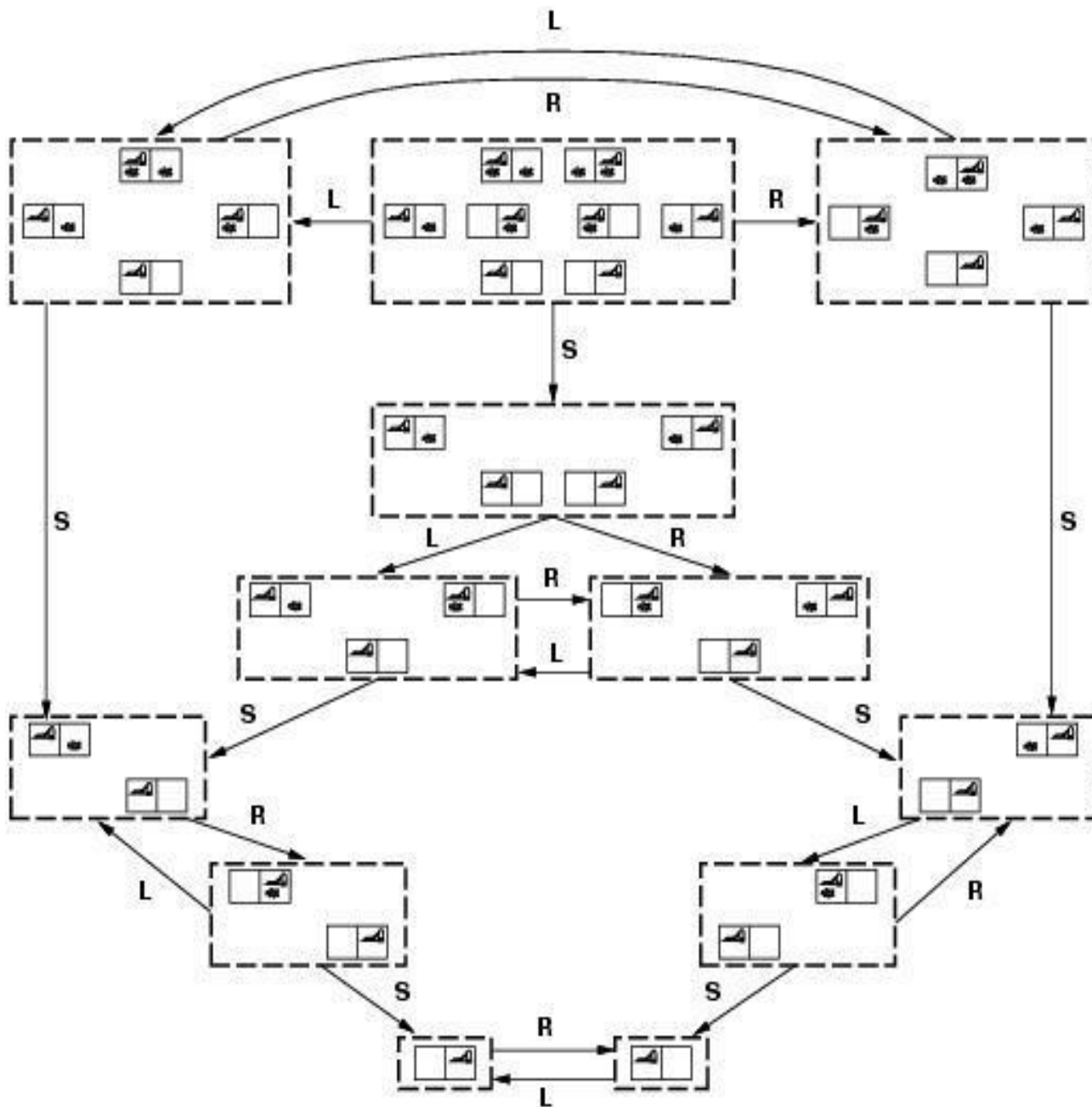
There are three actions-Left, Right, and Suck-and the goal is to clean up all the dirt (states 7 and 8). If the environment is observable, deterministic, and completely known, then the problem is trivially solvable by any of the algorithms we have described. For example, if the initial state is 5, then the action sequence [Right, Suck] will reach a goal state, 8.

Case 2: This section deals with the sensor less and contingency versions of the problem.

Suppose that the vacuum agent knows all the effects of its actions, but has no sensors. Then it knows only that its initial state is one of the set {1,2,3,4,5,6,7,8}. One might suppose that the agent's predicament is hopeless, but in fact it can do quite well. Because it knows what its actions do, it can, for example, calculate that the action Right will cause it to be in one of the states {2,4,6,8}, and the action sequence [Right, Suck] will always end up in one of the states {4,8}. Finally, the sequence [Right, Suck, Left, Suck] is guaranteed to reach the goal state 7 no matter

what the start state. We say that the agent can coerce the world into state 7, even when it doesn't know where it started.

**Belief State:** when the world is not fully observable, the agent must reason about sets of states that it might get to, rather than single states. We call each such set of states a belief state, representing the agent's current belief about the possible physical states it might be;



**Figure 6.2:** Sensor less vacuum world each box corresponds to single belief state

Contingency, start in {1,3}.

**Murphy 's Law**, Suck *can* dirty a clean carpet.

Local sensing: dirt, location only.

- Percept = [L, Dirty] = {1,3}
- [Suck] = {5,7}
- [Right] = {6,8}
- [Suck] in {6}={8} (Success)
- BUT [Suck] in {8} = failure

**Solution?**

- Belief-state: no fixed action sequence guarantees solution

**Relax requirement:**

- [Suck, Right, if [R, dirty] then Suck]
- Select actions based on contingencies arising during execution.

Time and space complexity are always considered with respect to some measure of the problem difficulty. In theoretical computer science, the typical measure is the size of the state space.