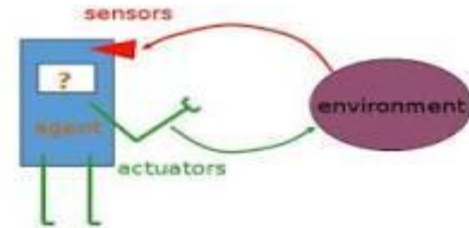


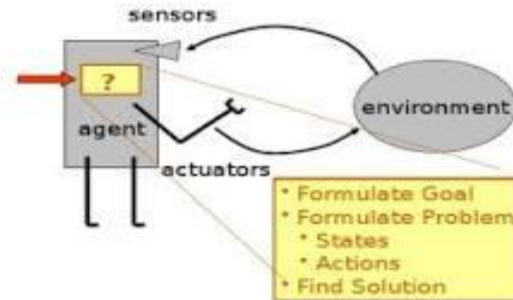
MODULE -2

Subject:ARTIFICIAL INTELLIGENCE
Subject Code: BCS515B

Problem-Solving Agent



Problem-Solving Agent



Example Problem



PROBLEM SOLVING → PROBLEM-SOLVING AGENTS

- Intelligent agents are supposed to maximize their performance measure.
- Achieving this is sometimes simplified if the agent can adopt a **goal** and aim at satisfying it.
- Goals help organize behaviour by limiting the objectives that the agent is trying to achieve and hence the actions it needs to consider.
- **Goal formulation**, based on the current situation and the agent's performance measure, is the first step in problem solving.
- **Problem formulation** is the process of deciding what actions and states to consider, given a goal.
- The process of looking for a sequence of actions that reaches the goal 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.

PROBLEM SOLVING → PROBLEM-SOLVING AGENTS

- Thus, we have a simple “**formulate, search, execute**” design for the agent, as shown in Figure 1.
- After formulating a goal and a problem to solve, the agent calls a search procedure to solve it.
- It then uses the solution to guide its actions, doing whatever the solution recommends as the next thing to do—typically, the first action of the sequence—and then removing that step from the sequence.
- Once the solution has been executed, the agent will formulate a new goal.

function SIMPLE-PROBLEM-SOLVING-AGENT(*percept*) **returns** an action

persistent: *seq*, an action sequence, initially empty
state, some description of the current world state
goal, a goal, initially null
problem, a problem formulation

state ← UPDATE-STATE(*state*, *percept*)
if *seq* is empty **then**
 goal ← FORMULATE-GOAL(*state*)
 problem ← FORMULATE-PROBLEM(*state*, *goal*)
 seq ← SEARCH(*problem*)
 if *seq* = *failure* **then return** a null action
action ← FIRST(*seq*)
seq ← REST(*seq*)
return *action*

Figure 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.



PROBLEM SOLVING → PROBLEM-SOLVING AGENTS → Well-defined Problems and Solutions

A **problem** can be defined formally by five components:

1. The **initial state** that the agent starts in. For example, the initial state for our agent in Romania might be described as $In(Arad)$.
2. A description of the possible **actions** available to the agent. Given a particular state s , $ACTIONS(s)$ returns the set of actions that can be executed in s . We say that each of these actions is **applicable** in s . For example, from the state $In(Arad)$, the applicable actions are $\{Go(Sibiu), Go(Timisoara), Go(Zerind)\}$.
3. A description of what each action does; the formal name for this is the **transition model**, specified by a function $RESULT(s, a)$ that returns the state that results from doing action a in state s . We also use the term **successor** to refer to any state reachable from a given state by a single action. For example, we have

$$RESULT(In(Arad), Go(Zerind)) = In(Zerind)$$

Together, the initial state, actions, and transition model implicitly define the **state space** of the problem—the set of all states reachable from the initial state by any sequence of actions. The state space forms a directed network or **graph** in which the nodes are states and the links between nodes are actions. (*The map of Romania shown in Figure 2 can be interpreted as a state-space graph if we view each road as standing for two driving actions, one in each direction*) A **path** in the state space is a sequence of states connected by a sequence of actions.

PROBLEM SOLVING → PROBLEM-SOLVING AGENTS → Well-defined Problems and Solutions

4. The **goal test**, which 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. The agent's goal in Romania is the singleton set $\{In(Bucharest)\}$.
5. A **path cost** function that assigns a numeric cost to each path. The problem-solving agent chooses a cost function that reflects its own performance measure. For the agent trying to get to *Bucharest*, time is of the essence, so the cost of a path might be its length in kilometers. The **step cost** of taking action a in state s to reach state s' is denoted by $c(s, a, s')$. The step costs for Romania are shown in Figure 2 as route distances. We assume that step costs are nonnegative.

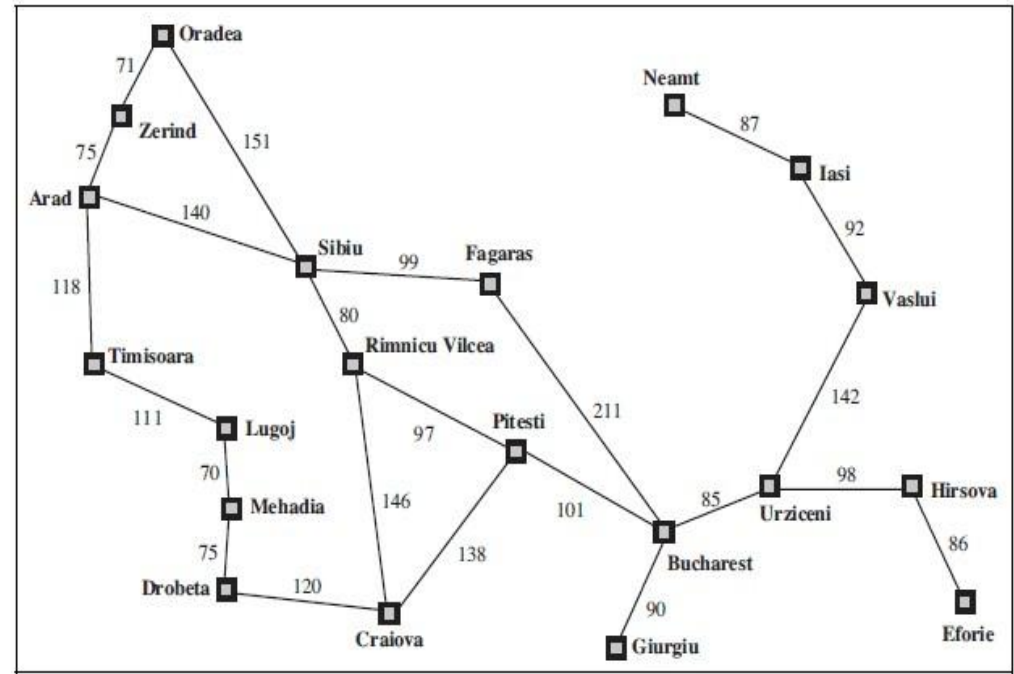


Figure 1.2 A simplified road map of part of Romania.

PROBLEM SOLVING → EXAMPLE PROBLEMS

- The problem-solving approach has been applied to a vast array of task environments.
- We list some of the best known here, distinguishing between **toy** and **real-world** problems.
- A **toy problem** is intended to illustrate or exercise various problem-solving methods. It can be given a concise, exact description and hence is usable by different researchers to compare the performance of algorithms.
- A **real-world problem** is one whose solutions people actually care about.

PROBLEM SOLVING → EXAMPLE PROBLEMS → Toy Problem

The first example we examine is the **vacuum world**. (See Figure 3) This can be formulated as a problem as follows:

- **States:** The state is determined by both the agent location and the dirt locations. The agent is in one of two locations, each of which might or might not contain dirt. Thus, there are $2 \times 2^2 = 8$ possible world states. A larger environment with n locations has $n * 2^n$ 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.
- **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.

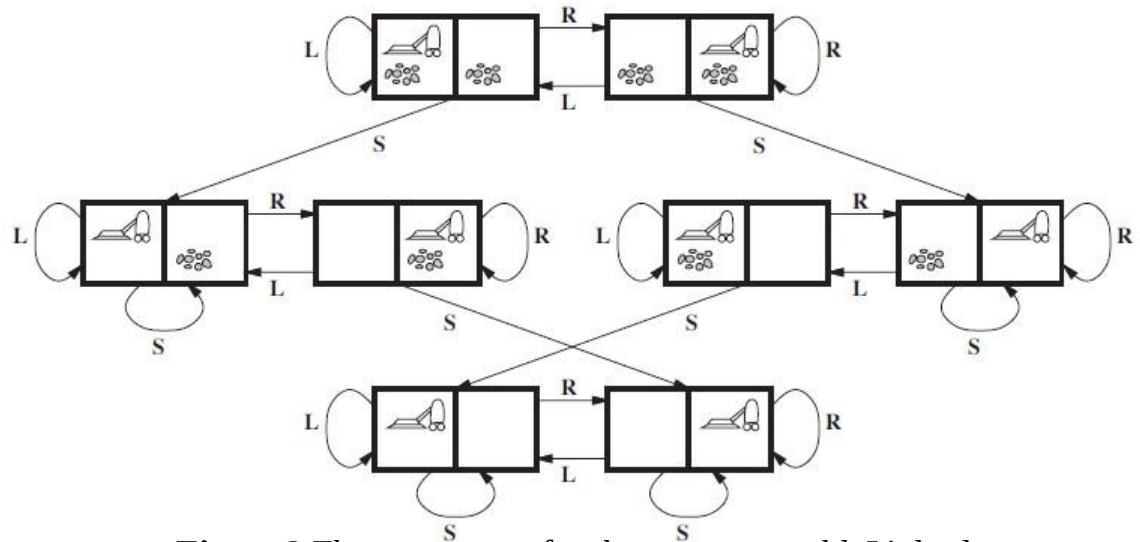


Figure 3 The state space for the vacuum world. Links denote actions: L = Left, R = Right, S = Suck.

PROBLEM SOLVING → EXAMPLE PROBLEMS → Toy Problem

The **8-puzzle**, an instance of which is shown in Figure 4, 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. The standard formulation is as follows:

- **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. Note that any given goal can be reached from exactly half of the possible initial states.
- **Actions:** The simplest formulation defines the actions as movements of the blank space *Left*, *Right*, *Up*, or *Down*.
- **Transition model:** Given a state and action, this returns the resulting state; for example, if we apply *Left* to the start state, the resulting state has the 5 and the blank switched.
- **Goal test:** This checks whether the state matches the goal state.
- **Path cost:** Each step costs 1, so the path cost is the number of steps in the path.

7	2	4
5		6
8	3	1

Start State

	1	2
3	4	5
6	7	8

Goal State

Figure 4 A typical instance of the 8-puzzle.

PROBLEM SOLVING → EXAMPLE PROBLEMS → Toy Problem

The goal of the **8-queens problem** is to place eight queens on a chessboard such that no queen attacks any other. (A queen attacks any piece in the same row, column or diagonal.) Figure 5 shows an attempted solution that fails: the queen in the rightmost column is attacked by the queen at the top left.

There are two main kinds of formulation:

- ❑ An **incremental formulation** involves operators that *augment* the state description, starting with an empty state; for the 8-queens problem, this means that each action adds a queen to the state.
- ❑ A **complete-state formulation** starts with all 8 queens on the board and moves them around. In either case, the path cost is of no interest because only the final state counts. The first incremental formulation one might try is the following:
 - **States:** Any arrangement of 0 to 8 queens on the board is a state.
 - **Initial state:** No queens on the board.
 - **Actions:** Add a queen to any empty square.
 - **Transition model:** Returns the board with a queen added to the specified square.
 - **Goal test:** 8 queens are on the board, none attacked.

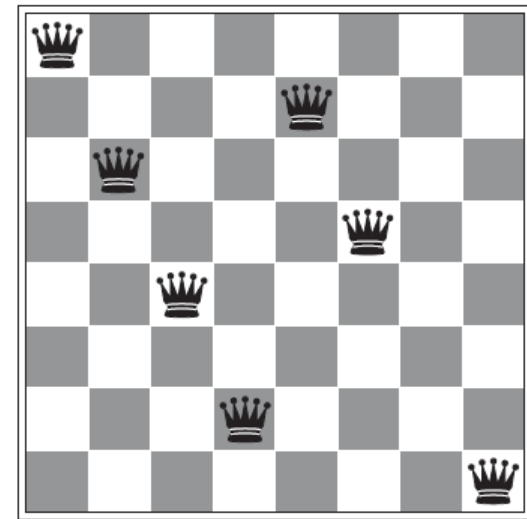


Figure 5 Almost a solution to the 8-queens problem.

PROBLEM SOLVING → EXAMPLE PROBLEMS → Real-world Problems

- **Route-finding problem** is defined in terms of specified locations and transitions along links between them. Route-finding algorithms are used in a variety of applications. Some, such as Web sites and in-car systems that provide driving directions, are relatively straightforward extensions of the Romania example.
- **Touring problems** are closely related to route-finding problems, but with an important difference. As with route finding, the actions correspond to trips between adjacent cities. The state space, however, is quite different. Each state must include not just the current location but also the *set of cities the agent has visited*.
- The **traveling salesperson problem (TSP)** is a touring problem in which each city must be visited exactly once. The aim is to find the *shortest* tour. The problem is known to be NP-hard, but an enormous amount of effort has been expended to improve the capabilities of TSP algorithms.
- 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. Channel routing finds a specific route for each wire through the gaps between the cells. These search problems are extremely complex, but definitely worth solving.
- **Robot navigation** is a generalization of the route-finding problem described earlier. Rather than following a discrete set of routes, a robot can move in a continuous space with an infinite set of possible actions and states. 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.

PROBLEM SOLVING → SEARCHING FOR SOLUTIONS

- Having formulated some problems, we now need to solve them. A solution is an action sequence, so search algorithms work by considering various possible action sequences. The possible action sequences starting at the initial state form a **search tree** with the initial state at the root; the branches are actions and the **nodes** correspond to states in the state space of the problem.
- Figure 6 shows the first few steps in growing the search tree for finding a route from *Arad* to *Bucharest*. The root node of the tree corresponds to the initial state, *In(Arad)*.
- The first step is to test whether this is a goal state, if not, then we need to consider taking various actions. We do this by **expanding** the current state; that is, applying each legal action to the current state, thereby **generating** a new set of states.
- In this case, we add three branches from the **parent node** *In(Arad)* leading to three new **child nodes**: *In(Sibiu)*, *In(Timisoara)*, and *In(Zerind)*.
- Now we must choose which of these three possibilities to consider further.
- Each of these six nodes is a **leaf node**, that is, a node with no children in the tree.
- The set of all leaf nodes available for expansion at any given point is called the **frontier/open list**.

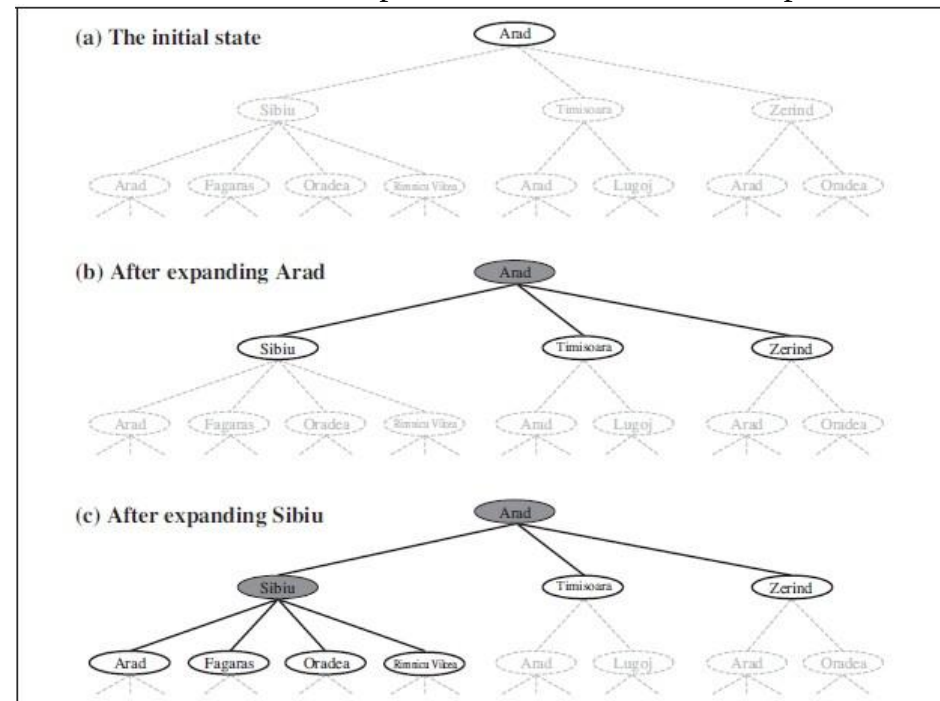


Figure 6 Partial search trees for finding a route from Arad to Bucharest.

PROBLEM SOLVING → SEARCHING FOR SOLUTIONS

- The process of expanding nodes on the frontier continues until either a solution is found or there are no more states to expand. The general TREE-SEARCH algorithm is shown informally in the Figure 7.
- Search algorithms all share this basic structure; they vary primarily according to how they choose which state to expand next—the so-called **search strategy**.
- Can be noticed one peculiar thing about the search tree shown in Figure 6, it includes the path from *Arad* to *Sibiu* and back to *Arad* again! We say that *In(Arad)* is a **repeated state** in the search tree, generated in this case by a **loopy path**.
- Loopy paths are a special case of the more general concept of **redundant paths**, which exist whenever there is more than one way to get from one state to another.
 - Consider the paths Arad–Sibiu (140 km long) and Arad–Zerind–Oradea–Sibiu (297 km long). Obviously, the second path is redundant—it's just a worse way to get to the same state.
- The way to avoid exploring redundant paths is to remember where one has been. To do this, we augment the TREE-SEARCH algorithm with a data structure called the **explored set** (also known as the **closed list**), which remembers every expanded node. Newly generated nodes that match previously generated nodes—ones in the explored set or the frontier—can be discarded instead of being added to the frontier.

```
function TREE-SEARCH(problem) returns a solution, or failure
    initialize the frontier using the initial state of problem
    loop do
        if the frontier is empty then return failure
        choose a leaf node and remove it from the frontier
        if the node contains a goal state then return the corresponding solution
        expand the chosen node, adding the resulting nodes to the frontier
```

```
function GRAPH-SEARCH(problem) returns a solution, or failure
    initialize the frontier using the initial state of problem
    initialize the explored set to be empty
    loop do
        if the frontier is empty then return failure
        choose a leaf node and remove it from the frontier
        if the node contains a goal state then return the corresponding solution
        add the node to the explored set
        expand the chosen node, adding the resulting nodes to the frontier
        only if not in the frontier or explored set
```

Figure 7 An informal description of the general tree-search and graph-search algorithms. The parts of **GRAPH-SEARCH** marked in bold italic are the additions needed to handle repeated states.

PROBLEM SOLVING → SEARCHING FOR SOLUTIONS

- The search tree constructed by the GRAPH-SEARCH algorithm contains at most one copy of each state, so we can think of it as growing a tree directly on the state-space graph, as shown in Figure 8.
- The algorithm has another nice property: the frontier **separates** the state-space graph into the explored region and the unexplored region, so that every path from the initial state to an unexplored state has to pass through a state in the frontier.
- This property is illustrated in Figure 9. As every step moves a state from the frontier into the explored region while moving some states from the unexplored region into the frontier, we see that the algorithm is *systematically* examining the states in the state space, one by one, until it finds a solution.

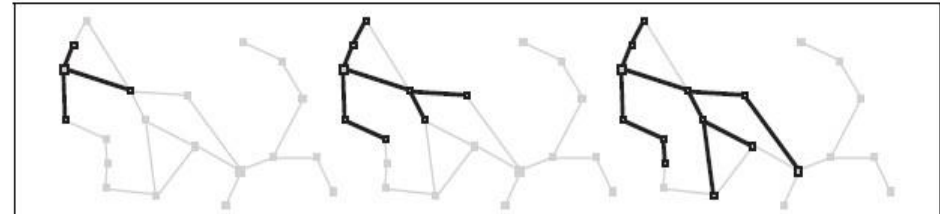


Figure 3.8 A sequence of search trees generated by a graph search on the Romania problem of Figure 3.2. At each stage, we have extended each path by one step. Notice that at the third stage, the northernmost city (Oradea) has become a dead end: both of its successors are already explored via other paths.

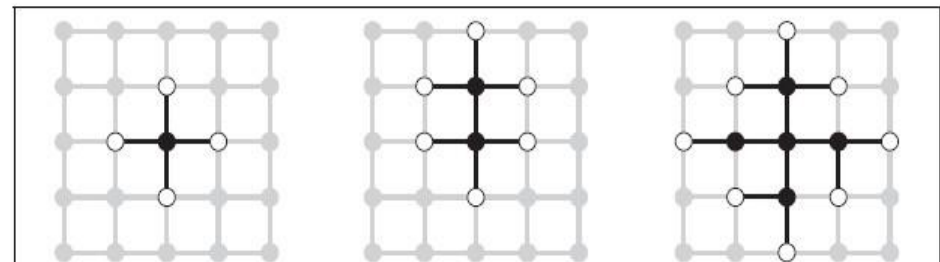


Figure 3.9 The separation property of GRAPH-SEARCH, illustrated on a rectangular-grid problem. The frontier (white nodes) always separates the explored region of the state space (black nodes) from the unexplored region (gray nodes). In (a), just the root has been expanded. In (b), one leaf node has been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

PROBLEM SOLVING → SEARCHING FOR SOLUTIONS → Infrastructure for Search Algorithms

- Search algorithms require a data structure to keep track of the search tree that is being constructed. For each node n of the tree, we have a structure that contains four components:
 - n .STATE: the state in the state space to which the node corresponds;
 - n .PARENT: the node in the search tree that generated this node;
 - n .ACTION: the action that was applied to the parent to generate the node;
 - n .PATH-COST: the cost, traditionally denoted by $g(n)$, of the path from the initial state to the node, as indicated by the parent pointers.
- Given the components for a parent node, it is easy to see how to compute the necessary components for a child node. The function CHILD-NODE takes a parent node and an action and returns the resulting child node:

```
function CHILD-NODE(problem, parent, action) returns a node
  return a node with
    STATE = problem.RESULT(parent.STATE, action),
    PARENT = parent, ACTION = action,
    PATH-COST = parent.PATH-COST + problem.STEP-COST(parent.STATE, action)
```

- The node data structure is depicted in Figure 10.

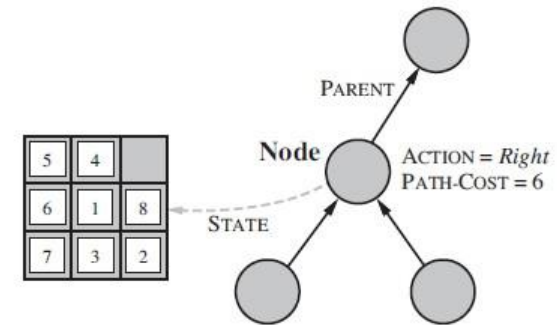


Figure 10 Nodes are the data structures from which the search tree is constructed. Each has a parent, a state, and various bookkeeping fields. Arrows point from child to parent.

PROBLEM SOLVING → SEARCHING FOR SOLUTIONS → Infrastructure for Search Algorithms

- Now that we have nodes, we need somewhere to put them.
- The frontier needs to be stored in such a way that the search algorithm can easily choose the next node to expand according to its preferred strategy.
- The appropriate data structure for this is a **queue**.
- The operations on a queue are as follows:
 - EMPTY?(queue) returns true only if there are no more elements in the queue.
 - POP(queue) removes the first element of the queue and returns it.
 - INSERT(element, queue) inserts an element and returns the resulting queue.
- Three common variants are the first-in, first-out FIFO QUEUE or **FIFO queue**, which pops the *oldest* element of the queue; the last-in, first-out or **LIFO queue** (also known as a **stack**), which pops the *newest* element of the queue; and the **priority queue**, which pops the element of the queue with the highest priority according to some ordering function.

PROBLEM SOLVING → SEARCHING FOR SOLUTIONS → Measuring Problem-solving Performance

We can evaluate an algorithm's performance in four ways:

1. **Completeness:** Is the algorithm guaranteed to find a solution when there is one?
2. **Optimality:** Does the strategy find the optimal solution?
3. **Time complexity:** How long does it take to find a solution?
4. **Space complexity:** How much memory is needed to perform the search?

In AI, the graph is often represented *implicitly* by the initial state, actions, and transition model and is frequently infinite. For these reasons, complexity is expressed in terms of three quantities:

1. b , the **branching factor** or maximum number of successors of any node;
2. d , the **depth** of the shallowest goal node (i.e., the number of steps along the path from the root); and
3. m , the **maximum length** of any path in the state space.

PROBLEM SOLVING → UNINFORMED SEARCH STRATEGIES

This section covers several search strategies that come under the heading of **uninformed search** (also called **blind search**).

- The term means that the strategies have no additional information about states beyond that provided in the problem definition. All they can do is generate successors and distinguish a goal state from a non-goal state.

Breadth-first Search:

- **Breadth-first search** 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 is an instance of the general graph-search algorithm in which the *shallowest* unexpanded node is chosen for expansion.
- This is achieved very simply by using a FIFO queue for the frontier. Thus, new nodes (which are always deeper than their parents) go to the back of the queue, and old nodes, which are shallower than the new nodes, get expanded first.
- Pseudocode is given in Figure 11 and Figure 12 shows the progress of the search on a simple binary tree.
- The time complexity of BFS is $O(b^d)$, and space complexity is also $O(b^d)$ where b is a branching factor and d is a depth.

PROBLEM SOLVING → UNINFORMED SEARCH STRATEGIES

```

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.

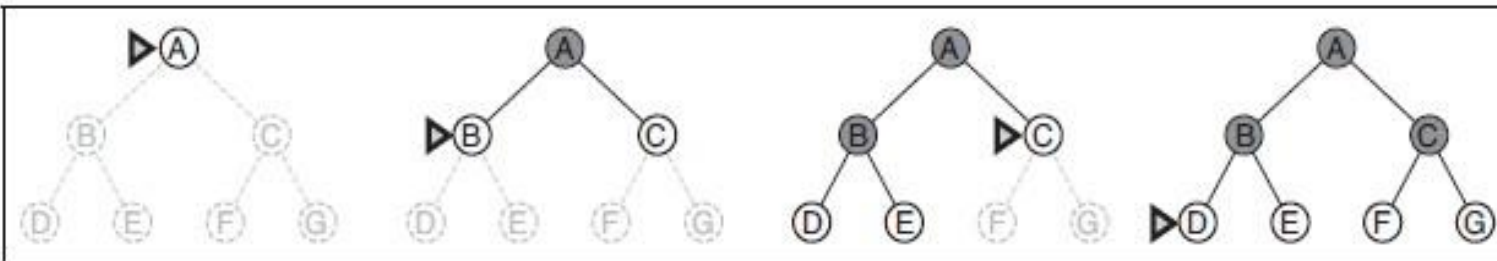


Figure 11 Breadth-first search on a graph.

Figure 12 Breadth-first search on a simple binary tree. At each stage, the node to be expanded next is indicated by a marker.

Uniform Cost Search:

- When all step costs are equal, breadth-first search is optimal 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 $g(n)$. This is done by storing the frontier as a priority queue ordered by g .

```

function UNIFORM-COST-SEARCH(problem) returns a solution, or failure
    node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
    frontier ← a priority queue ordered by PATH-COST, with node as the only element
    explored ← an empty set
    loop do
        if EMPTY?(frontier) then return failure
        node ← POP(frontier) /* chooses the lowest-cost node in frontier */
        if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
        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
                frontier ← INSERT(child, frontier)
            else if child.STATE is in frontier with higher PATH-COST then
                replace that frontier node with child
    
```

Figure 3.14 Uniform-cost search on a graph. The algorithm is identical to the general graph search algorithm in Figure 3.7, except for the use of a priority queue and the addition of an extra check in case a shorter path to a frontier state is discovered. The data structure for *frontier* needs to support efficient membership testing, so it should combine the capabilities of a priority queue and a hash table.

PROBLEM SOLVING → UNINFORMED SEARCH STRATEGIES

Depth-first search:

- **Depth-first search** always expands the *deepest* node in the current frontier of the search tree.
- The progress of the search is illustrated in Figure 13.
- 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 frontier, so then the search “backs up” to the next deepest node that still has unexplored successors.
- The depth-first search algorithm is an instance of the graph-search algorithm; whereas breadth-first-search uses a FIFO queue, depth-first search uses a LIFO queue.
 - A LIFO queue means that the most recently generated node is chosen for expansion.
 - This must be the deepest unexpanded node because it is one deeper than its parent.
- The time complexity of DFS is $O(b^m)$, and space complexity is $O(bm)$, where b is a branching factor and m is a maximum depth.

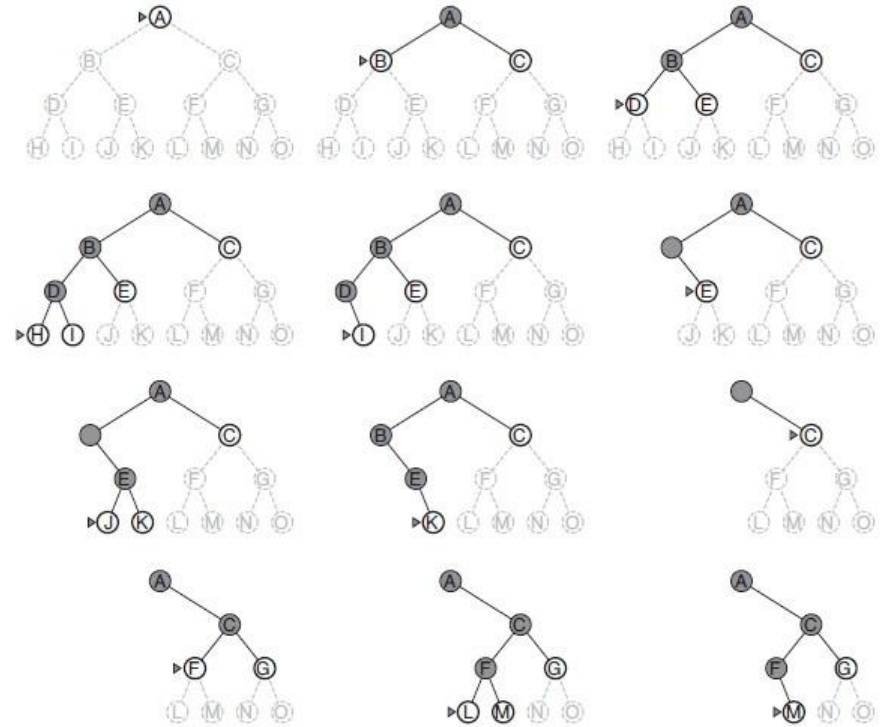


Figure 13 Depth-first search on a binary tree. The unexplored region is shown in light gray. Explored nodes with no descendants in the frontier are removed from memory. Nodes at depth 3 have no successors and *M* is the only goal node.

Breadth First search

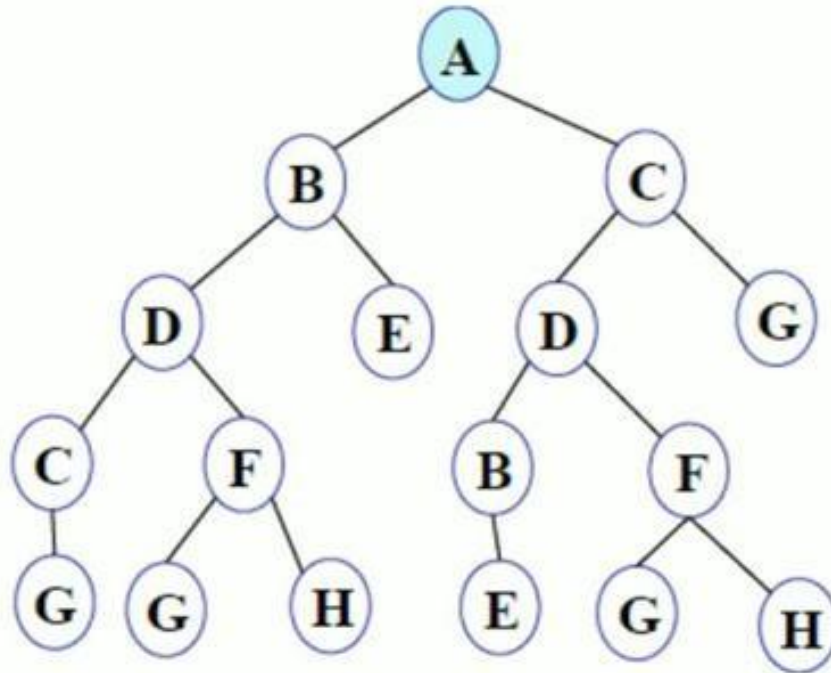
Breadth-First Search – Algorithm

1. Create a variable called NODE-LIST and set it to the initial state.
2. Until a goal state is found, or NODE-LIST is empty:
 - a) Remove the first element from NODE-LIST and call it E. If NODE-LIST was empty, then quit.
 - b) For element E do the following:
 - i. Apply the rule to generate a new state,
 - ii. If the new state is a goal state. quit and return this state.
 - iii. Otherwise, add the new state to the end of NODE-LIST

Breadth-First Search - Example

- Step 1:** Initially NODE-LIST contains only one node corresponding to the source state A.

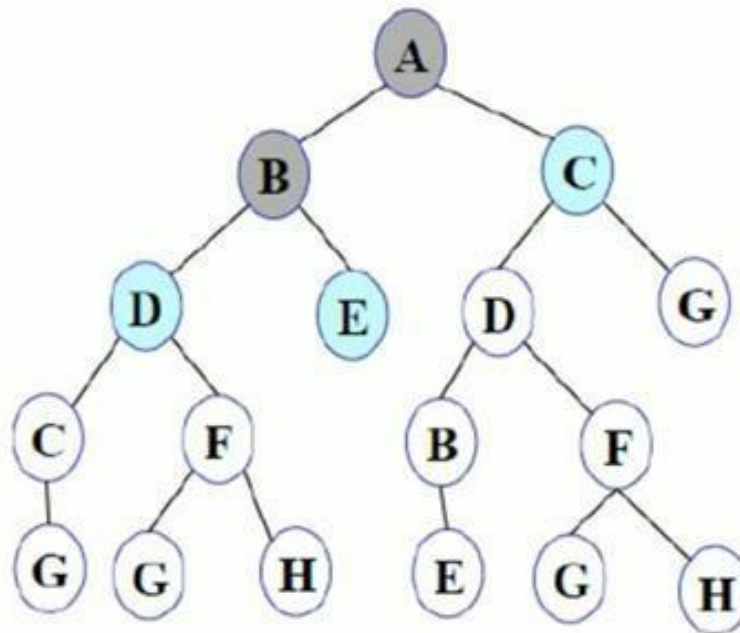
- NODE-LIST: A**



Breadth-First Search - Example

- Step 3:** Node B is removed from NODE-LIST and is expanded. Its children D, E are generated and put at the back of NODE-LIST.

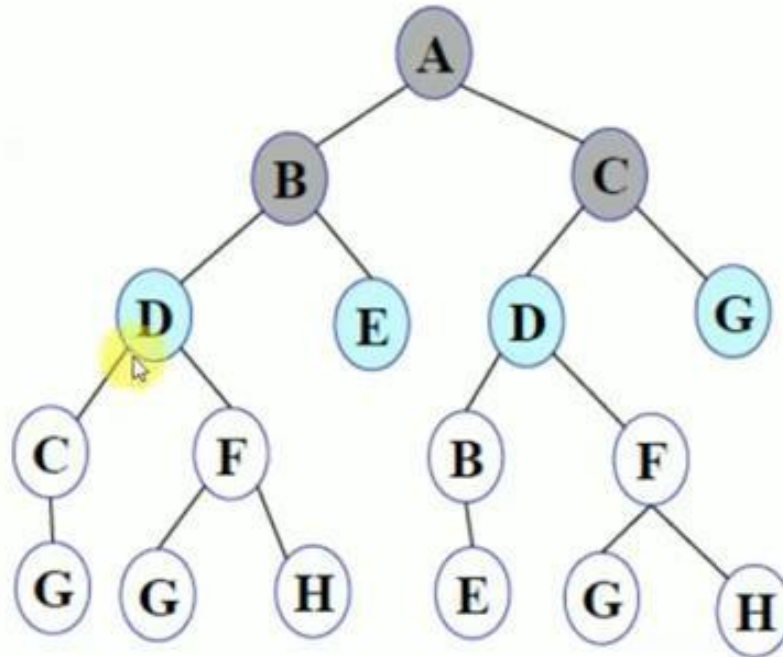
NODE-LIST: C D E



Breadth-First Search - Example

- Step 4:** Node C is removed from NODE-LIST and is expanded. Its children D and G are added to the back of NODE-LIST.

- NODE-LIST: D E D G**

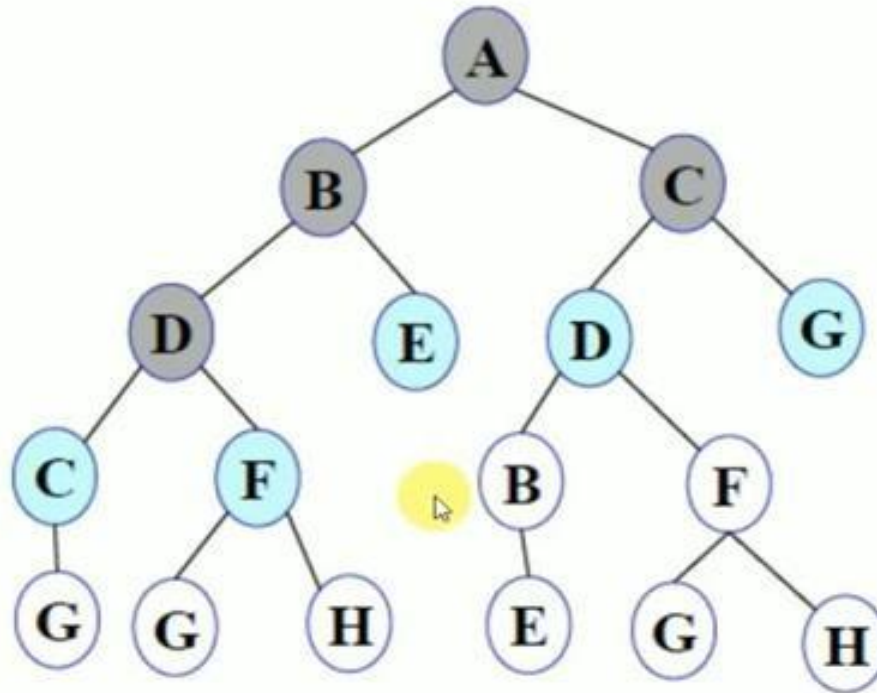


Breadth-First Search - Example

Step 5: Node D is removed from NODE-LIST. Its children C and F are generated and added to the back of NODE-LIST.

NODE-LIST:

E D G C F

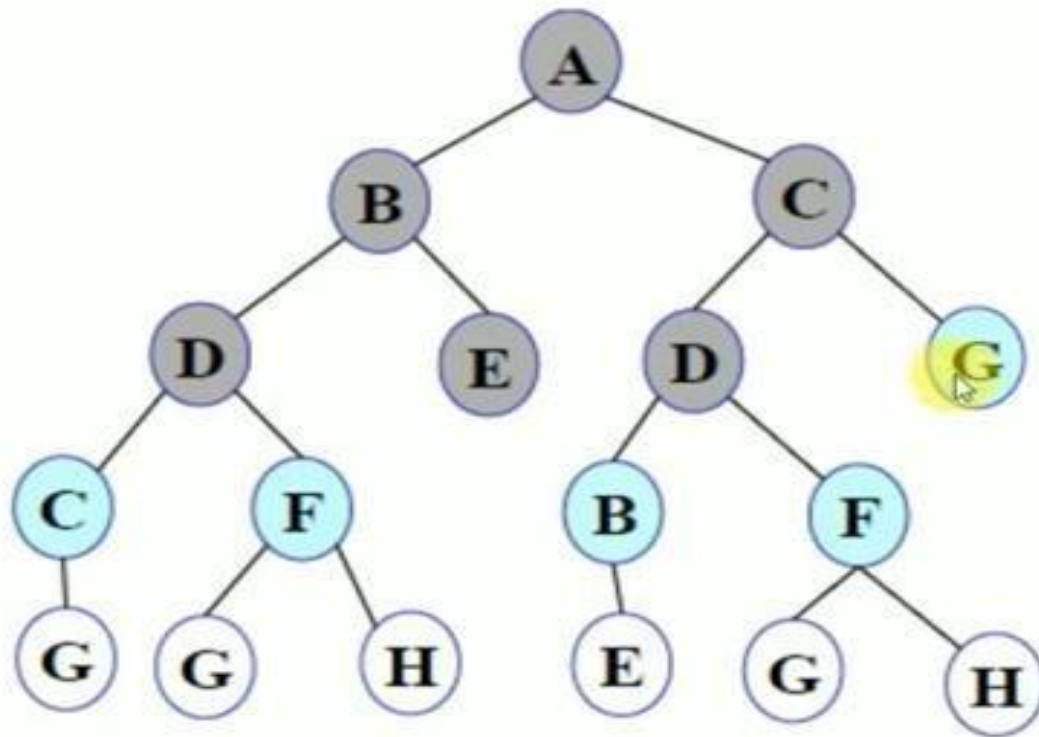


Breadth-First Search - Example

Step 7: D is expanded; B and F are put in OPEN.

NODE-LIST:

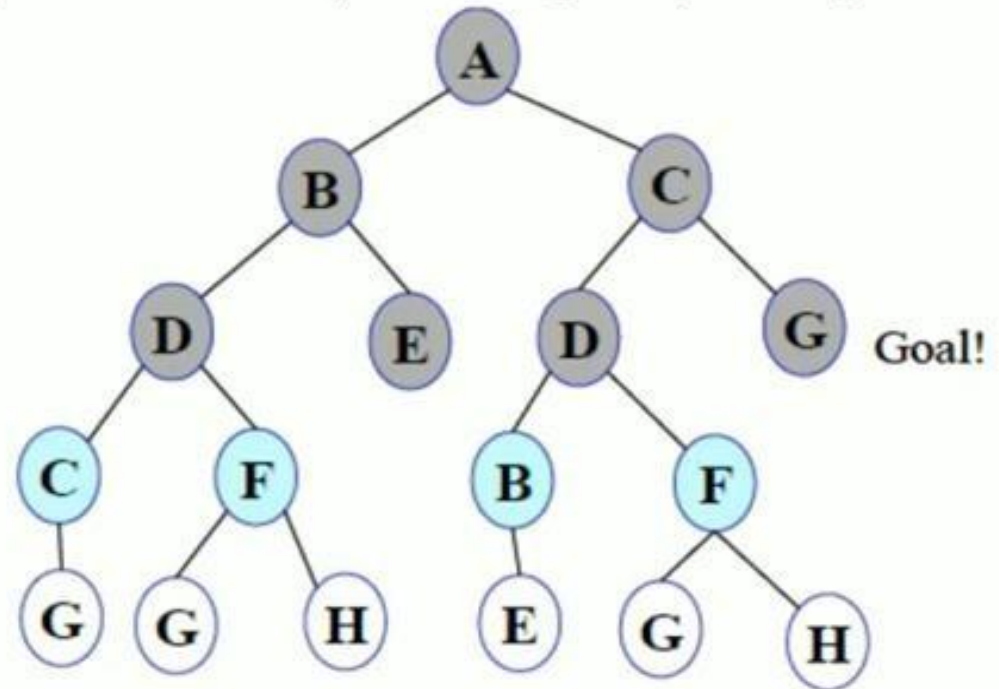
G C F B F



Breadth-First Search - Example

Step 8: G is selected for expansion. It is found to be a goal node.

Hence the algorithm returns the path A - C - G by following the parent pointers of the node corresponding to G.

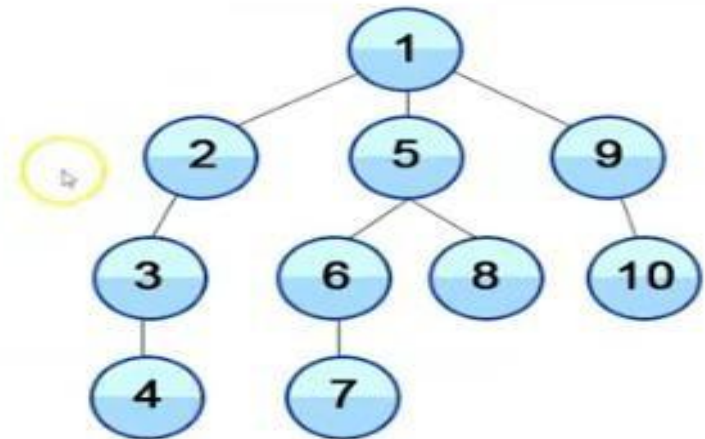


Depth First search

Depth-first search is an algorithm for traversing or searching tree or graph data structures. The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking.

Introduction to Depth First Search

- **Depth First Search or DFS** starts with the initial node of the graph, then goes deeper until it finds the goal nodes or nodes having no children.
- In DFS, then backtracks from the dead-end towards the most recent node that is yet to be completely **unexplored**.
- **Stack** data structure being used in 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]

The step by step process to the DFS traversal is given as follows -

First, create a stack with the total number of vertices in the graph.

Now, choose any vertex as the starting point of traversal, and push that vertex into the stack.

After that, push a non-visited vertex (adjacent to the vertex on the top of the stack) to the top of the stack.

Now, repeat steps 3 and 4 until no vertices are left to visit from the vertex on the stack's top.

If no vertex is left, go back and pop a vertex from the stack. Repeat steps 2, 3, and 4 until the stack is empty.

Depth-Limited search:

- The failure of depth-first search in infinite state spaces can be alleviated by supplying 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.

```
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.

PROBLEM SOLVING → UNINFORMED SEARCH STRATEGIES

Iterative deepening depth-first search:

- Iterative deepening search (or iterative deepening depth-first search) is a general strategy, often used in combination with depth-first tree 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

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution, or failure
  for depth = 0 to  $\infty$  do
    result  $\leftarrow$  DEPTH-LIMITED-SEARCH(problem, depth)
    if result  $\neq$  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.

PROBLEM SOLVING → UNINFORMED SEARCH STRATEGIES

Iterative deepening depth-first search:

- Iterative deepening search (or iterative deepening depth-first search) is a general strategy, often used in combination with depth-first tree 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

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution, or failure
  for depth = 0 to  $\infty$  do
    result  $\leftarrow$  DEPTH-LIMITED-SEARCH(problem, depth)
    if result  $\neq$  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.



PROBLEM SOLVING → UNINFORMED SEARCH STRATEGIES

Bidirectional search:

- Bidirectional search is implemented by having one or both of the searches check each node before it is expanded to see if it is in the fringe of the other search tree; if so, a solution has been found.
- • For example, if a problem has solution depth $d = 6$, and each direction runs breadth-first search one node at a time, then in the worst case the two searches meet when each has expanded all but one of the nodes at depth 3. For $b = 10$, this means a total of 22,200 node generations, compared with 11,111,100 for a standard breadth-first search.
- • Checking a node for membership in the other search tree can be done in constant time with a hash table, so the time complexity of bidirectional search is $O(b^{d/2})$.
- • At least one of the search trees must be kept in memory so that the membership check can be done, hence the space complexity is also $O(b^{d/2})$. This space requirement is the most significant weakness of bidirectional search.
- • The algorithm is complete and optimal (for uniform step costs) if both searches are breadth-first; other combinations may sacrifice completeness, optimality, or both.
- • The reduction in time complexity makes bidirectional search attractive, but how do we search backwards? This is not as easy as it sounds.
- • Let the predecessors of a state x , $\text{Pred}(x)$, be all those states that have x as a successor. Bidirectional search requires that $\text{Pred}(x)$ be efficiently computable.
- • The easiest case is when all the actions in the state space are reversible, so that $\text{Pred}(x) = \text{Succ}(x)$.
- • difficult case for bidirectional search is when the goal test gives only an implicit description of some possibly large set of goal states.
- • A backward search would need to construct compact descriptions of "all states that lead to checkmate by move m_1 and so on; and those descriptions would have to be tested against the states generated by the forward search. There is no general way to do this efficiently

PROBLEM SOLVING → UNINFORMED SEARCH STRATEGIES

Bidirectional search:

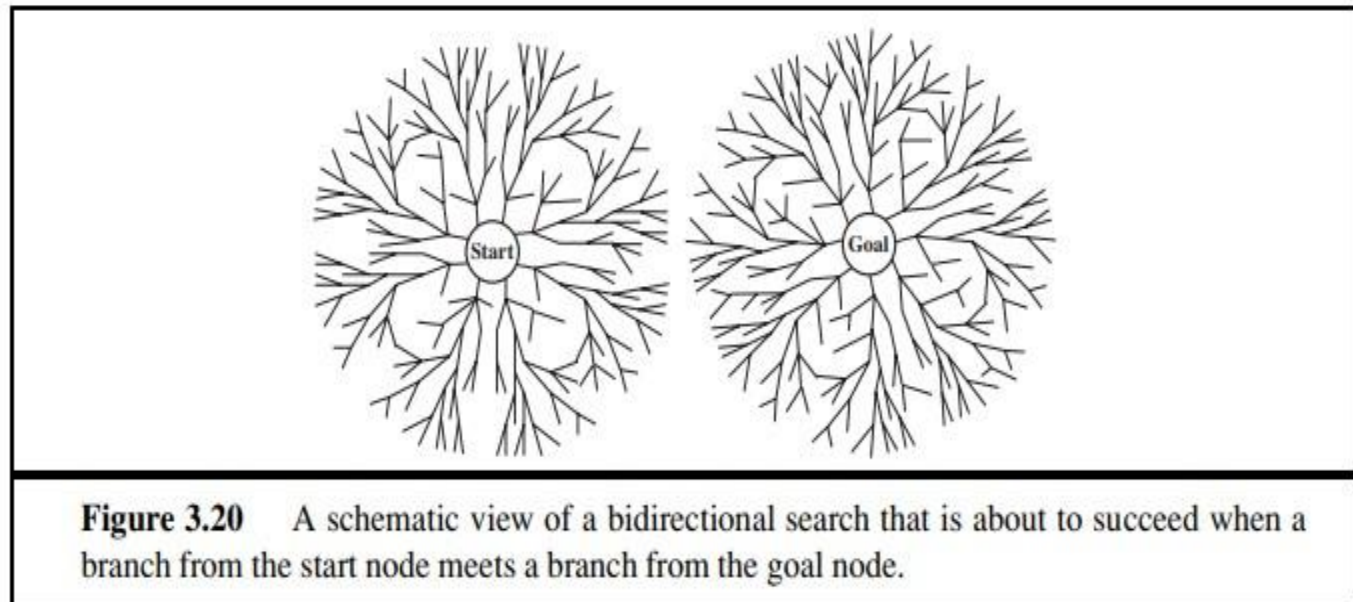


Figure 3.20 A schematic view of a bidirectional search that is about to succeed when a branch from the start node meets a branch from the goal node.

Comparing uninformed search strategies

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes ^a	Yes ^{a,b}	No	No	Yes ^a	Yes ^{a,d}
Time	$O(b^d)$	$O(b^{1+\lceil C^*/\epsilon \rceil})$	$O(b^m)$	$O(b^l)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+\lceil C^*/\epsilon \rceil})$	$O(bm)$	$O(bl)$	$O(bd)$	$O(b^{d/2})$
Optimal?	Yes ^c	Yes	No	No	Yes ^c	Yes ^{c,d}

Figure 3.21 Evaluation of tree-search strategies. b is the branching factor; d is the depth of the shallowest solution; m is the maximum depth of the search tree; l is the depth limit. Superscript caveats are as follows: ^a complete if b is finite; ^b complete if step costs $\geq \epsilon$ for positive ϵ ; ^c optimal if step costs are all identical; ^d if both directions use breadth-first search.



A T M E
College of Engineering

GOLD

QS I-GAUGE
INDIAN COLLEGE RATINGS



THANK YOU