PROBLEM SOLVING & SEARCH

18BTCS504

PROF. SURESH KAPARE

PROBLEM SOLVING AGENT (GOAL BASED)

- decide what to do by finding sequences of actions that lead to desirable states.
- Problem?
- Solution?
- General Purpose Search Algorithm to solve problem-
- Analysis of algorithms
- Goal formulation, based on the current situation and the agent's performance measure, is the first step in problem solving- Tour Agent Example.

- The agent's task is to find out which sequence of actions will get it to a goal state.
- Problem formulation is the process of deciding what actions and states to consider, given a goal. (To avoid many small states)

- An agent with several immediate options of unknown value can decide what to do by just examining different possible sequences of actions that lead to states of known value, and then choosing the best sequence.
    - 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. (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.**

"formulate, search, execute"

```
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 ← UPDATE-STATE(state, percept)
  if seq is empty then do
      goal ← FORMULATE-GOAL(state)
      problem ← FORMULATE-PROBLEM(state, goal)
      seq ← SEARCH(problem)
  action ← FIRST(seq)
  seq ← REST(seq)
  return action
```

Figure 3.1 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. Note that when it is executing the sequence it ignores its percepts: it assumes that the solution it has found will always work.

Environment is Static, observable, deterministic. Initial state known

WELL-DEFINED PROBLEMS AND SOLUTIONS

- **Initial state**
- A description of the possible **actions** available to the agent**.** The most common formulation uses a **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.
- Ex. (Go(Mumbai), In (Airport))
- **state space of the problem = Initial State + Successor function (set of all states)**
- A **path** in the state space is a sequence of states connected by a sequence of actions.
- The **goal test,** which determines whether a given state is a goal state.
- A **path cost** function that assigns a numeric cost to each path. reflects its own performance measure. The **step cost** of taking action *a* to go from state x to state y is denoted by ***c(x, a, y).***

Above elements define a problem, gathered into single data structure given as input to a problem-solving algorithm.

A solution to a problem is a path from the initial state to a goal state. Optimal Solution has low path cost.

TOY PROBLEMS

- **The vacuum world**
- States: The agent is in one of two locations, each of which might or might not contain dirt. Thus there are 2 x 2^2 = 8 possible world states.
- Initial state: Any state can be designated as the initial state.
- Successor function: This generates the legal states that result from trying the three actions (Left, Right, and Suck). 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.

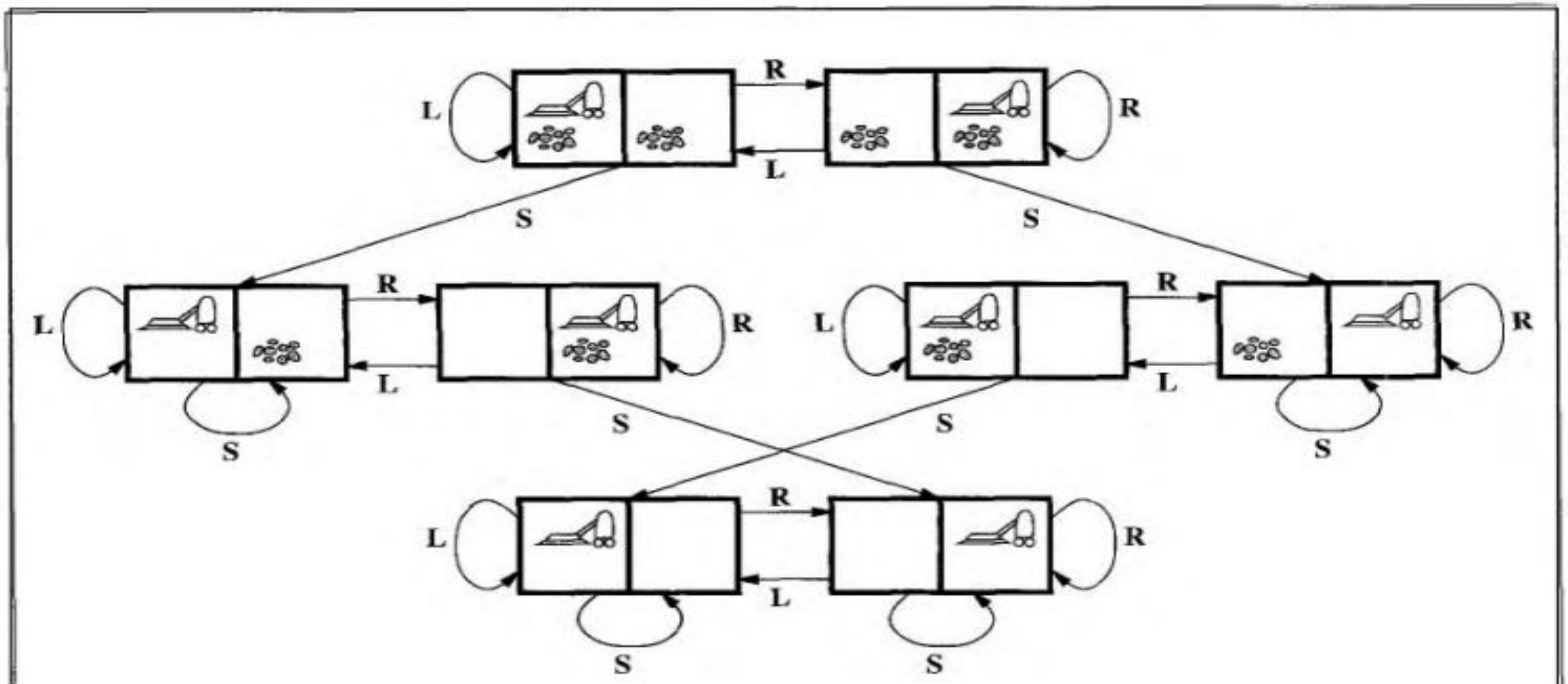
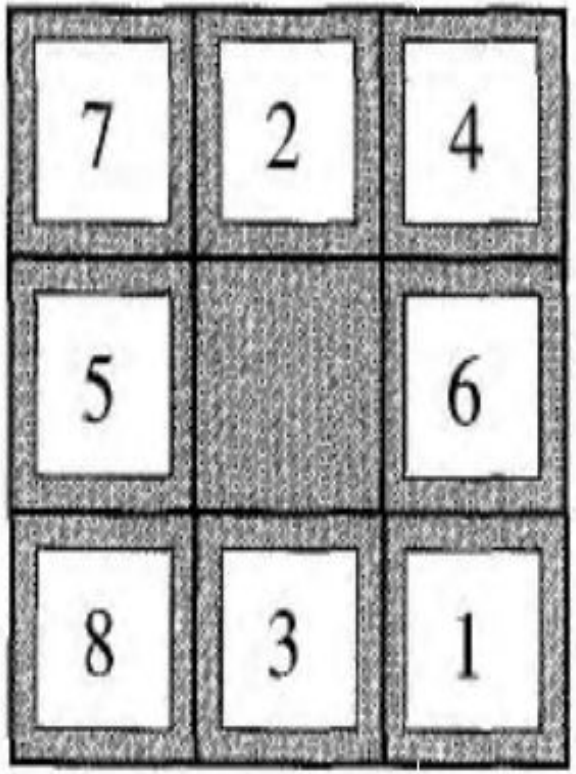


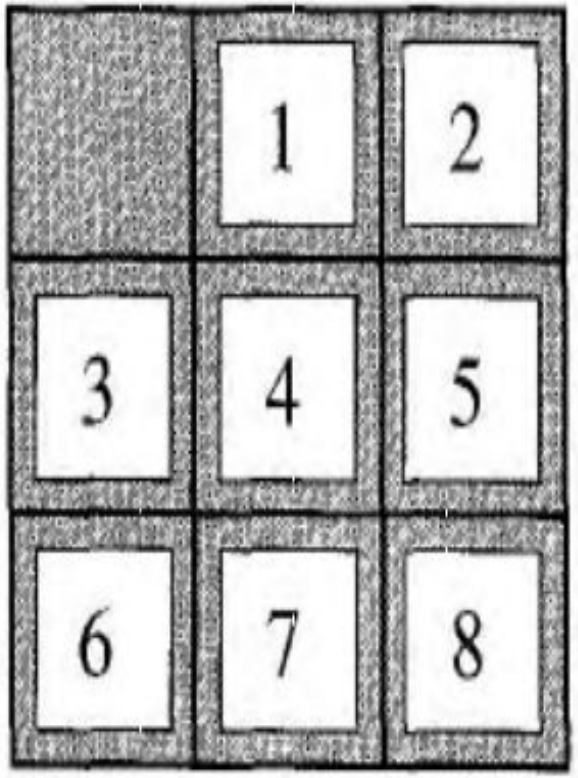
Figure 3.3 The state space for the vacuum world. Arcs denote actions: L = *Left*, R = *Right*, S = Suck.

THE 8-PUZZLE

- consists of a 3 x 3 board with eight numbered tiles and a blank space
- A tile adjacent to the blank space can slide into the space
- object is to reach a specified goal state, such as the one shown on the right of the figure
- **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
- **Successor function**: This generates the legal states that result from trying the four actions (blank moves Left, Right, Up, or Down).
- **Path cost**: Each step costs 1, so the path cost is the number of steps in the path.
- **Goal test:** This checks whether the state matches the goal configuration shown in Fig



Start State



Goal State

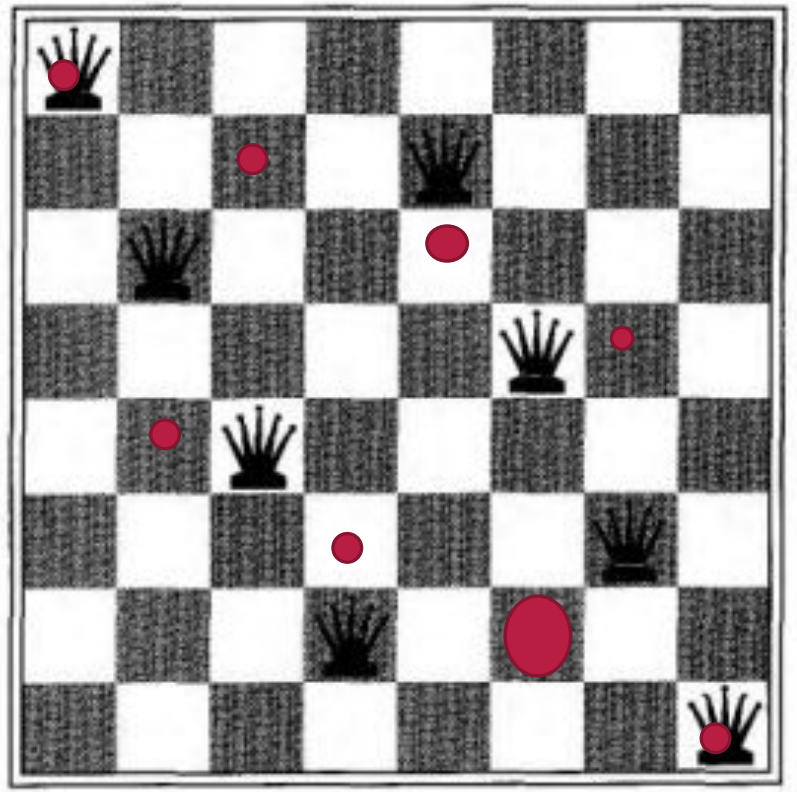
Figure 3.4 A typical instance of the 8-puzzle.

ANALYSIS

- belongs to the family of sliding-block puzzles, which are often used as test problems for new search algorithms in AI.
- This general class is known to be NP-complete, so one does not expect to find methods significantly better in the worst case than the search.
- The 8-puzzle has 9!/2 = 181,440 reachable states and is easily solved.
- The 15-puzzle (on a 4 x 4 board) has around 1.3 trillion states, and random instances can be solved optimally in a few milliseconds by the best search algorithms.
- The 24-puzzle (on a 5 x 5 board) has around loz5 states, and random instances are still quite difficult to solve optimally with current machines and algorithms.

THE 8-QUEENS PROBLEM

- to place eight queens on a chessboard such that no queen attacks any other.
- 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.



- **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 attacked.

- In this formulation, we have 64 . 63 . . .57 =1.8 x 10^14 possible sequences to investigate. A better formulation would prohibit placing a queen in any square that is already attacked:
- **States:** Arrangements of n queens (0 <= n <= 8), one per column in the leftmost n columns, with no queen attacking another are states.
- **Improved Successor function:** Add a queen to any square in the leftmost empty column such that it is not attacked by any other queen.
- This formulation reduces the 8-queens state space from 3 x 10^14 to just 2,057, and solutions are easy to find.
- On the other hand, for 100 queens the initial formulation has roughly 10^400 states whereas the improved formulation has about 10 ^52 states.

REAL-WORLD PROBLEMS

- Route-finding algorithms are used in a variety of applications, such as routing in computer networks, military operations planning, and airline travel planning systems
- Consider a simplified example of an airline travel problem specified as follows:
  - **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 prespecified 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.

- **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. In addition to planning trips for traveling salespersons, these algorithms have been used for tasks such as planning movements of automatic circuit-board drills and of stocking machines on shop floors

- 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 the connecting wires to be placed between the cells.
- 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.

UNIFORMED SEARCH STRATEGIES

- This section covers five search strategies that come under the heading of uninformed search (also called blind search).
- The term means that they 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 nongoal state. Strategies that know whether one non goal state is "more promising" than another are called informed search or heuristic search strategies.
- All search strategies are distinguished by the order in which nodes are expanded

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.
- 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.

case, we would expand all but the last node at level d (since the goal itself is not expanded), generating $b^{d+1} - b$ nodes at level $d+1$. Then the total number of nodes generated is

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

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 (plus one node for the root).

Those who do complexity analysis are worried (or excited, if they like a challenge) by exponential complexity bounds such as $O(b^{d+1})$. Figure 3.11 shows why. It lists the time and memory required for a breadth-first search with branching factor $b = 10$, for various values of the solution depth d. The table assumes that 10,000 nodes can be generated per second and that a node requires 1000 bytes of storage. Many search problems fit roughly within these assumptions (give or take a factor of 100) when run on a modern personal computer.

There are two lessons to be learned from Figure 3.11. First, *the memory requirements are a bigger problem for breadth-first search than is the execution time.* 31 hours would not be too long to wait for the solution to an important problem of depth 8, but few computers have the terabyte of main memory it would take. Fortunately, there are other search strategies that require less memory.

The second lesson is that the time requirements are still a major factor. If your problem has a solution at depth 12, then (given our assumptions) it will take 35 years for breadth-first search (or indeed any uninformed search) to find it. In general, *exponential-complexity search problems cannot be solved by uninformed methods for any but the smallest instances.*

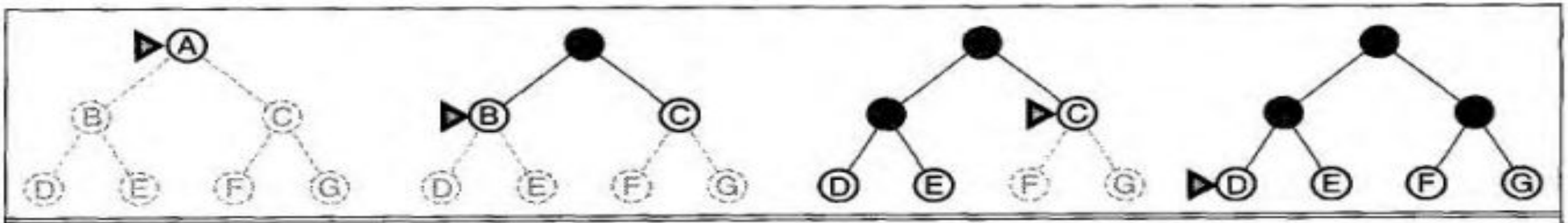


Figure 3.10 Breadth-first search on a simple binary tree. At each stage, the node to be expanded next is indicated by a marker.

Depth	Nodes	Time	Memory
2	1100	.11 seconds	1 megabyte
4	111,100	11 seconds	106 megabytes
6	10^7	19 minutes	10 gigabytes
8	10^9	31 hours	1 terabytes
10	10^{11}	129 days	101 terabytes
12	10^{13}	35 years	10 petabytes
14	10^{15}	3,523 years	1 exabyte

Figure 3.11 Time and memory requirements for breadth-first search. The numbers shown assume branching factor b = 10; 10,000 nodes/second; 1000 bytes/node.