CS 362: Intelligent Systems

UNIT 3

STRUCTURES & STRATEGIES FOR STATE SPACE SEARCH

Introduction

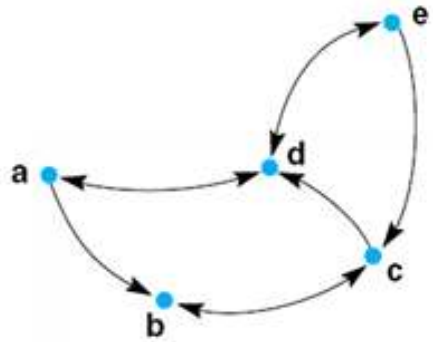
- Well-formed predicate calculus expressions provide a means of describing objects and relations in a problem domain, and inference rules allow us to infer new knowledge from these descriptions.

- These inferences define a space that is searched to find a problem solution.

- The theory of state space search is our primary tool for finding the solution by:
  - Representing the problem as a state space graph
  - Using graph theory to analyze the structure and complexity of both the problem and the search procedures that we employ to solve it.

Graph terminology (i)

- The graph consists of:
  - A set of nodes N1, N2, N3, ..., Nn, ..., which need not be finite.
  - A set of arcs that connect pairs of nodes.
- Arcs are ordered pairs of nodes; i.e., the arc (N3, N4) connects node N3 to node N4. This indicates a direct connection from node N3 to N4 but not from N4 to N3, unless (N4, N3) is also an arc, and then the arc joining N3 and N4 is undirected.
- A labeled graph has one or more descriptors (labels) attached to each node that distinguish that node from any other node in the graph.
- A graph is directed if arcs have an associated directionality.
- The arcs in a directed graph are usually drawn as arrows or have an arrow attached to indicate direction
  - (E.g.: in the shown graph arc (a, b) may only be crossed from node a to node b, but arc (b, c) is crossable in either direction.)



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

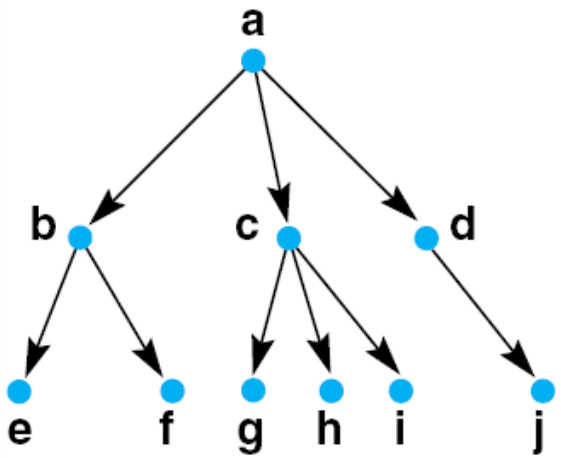
Arcs = {(a,b),(a,d),(b,c),(c,b),(c,d),(d,a),(d,e),(e,c),(e,d)}

Graph terminology (ii)

- If a directed arc connects Nj and Nk, then:
  - Nj is called the parent of Nk and Nk, the child of Nj.
  - If the graph also contains an arc (Nj, Nl), then Nk and Nl are called siblings.
- A rooted graph has a unique node NS from which all paths in the graph originate. That is, the root has no parent in the graph.
- A tip or leaf node is a node that has no children.
- An ordered sequence of nodes [N1, N2, N3, ..., Nn], where each pair Ni, Ni+1 in the sequence represents an arc, i.e., (Ni, Ni+1), is called a path of length n - 1.
  - On a path in a rooted graph, a node is said to be an ancestor of all nodes positioned after it (to its right) as well as a descendant of all nodes before it.
  - A path that contains any node more than once (some Nj in the definition of path above is repeated) is said to contain a cycle or loop.

Graph terminology (iii)

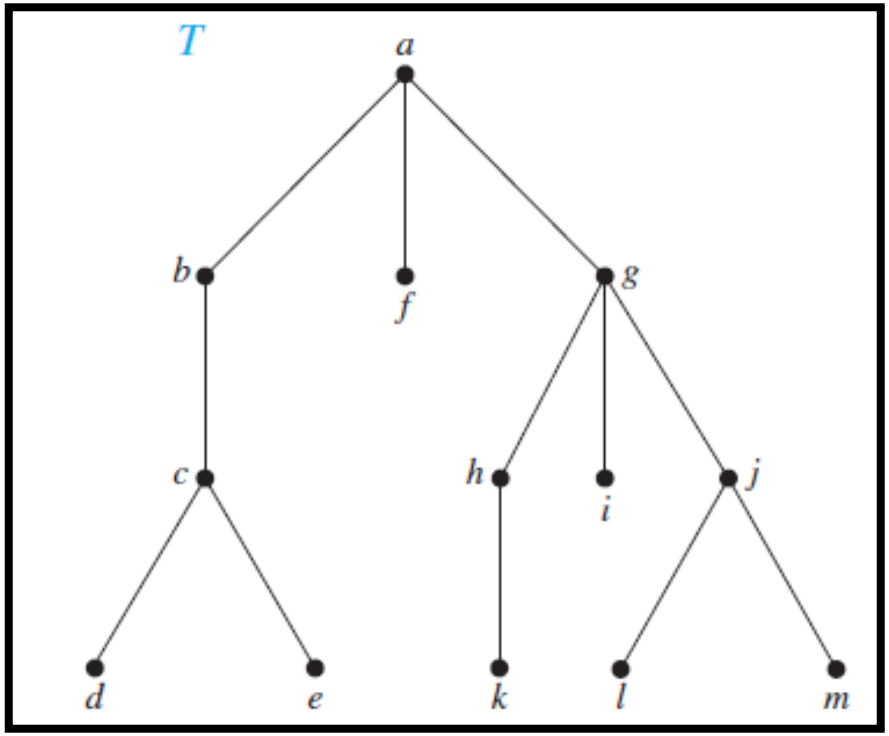
- A tree is a graph in which there is a unique path between every pair of nodes.
  - The paths in a tree, therefore, contain no cycles.
  - The arcs in a rooted tree are directed away from the root.
  - Each node in a rooted tree has a unique parent.
  - Two nodes are said to be connected if a path exists that includes them both.



Example

In the shown tree find:

- The root
  a
- The parent of c is
  b
- The children of g are:
  h, i, and j
- The siblings of h are:
  i and j
- The ancestors of e are:
  c, b, and a.
- The descendants of b are:
  c, d, and e.
- The tips (leaves) are:
  d, e, f , i, k, l, and m.



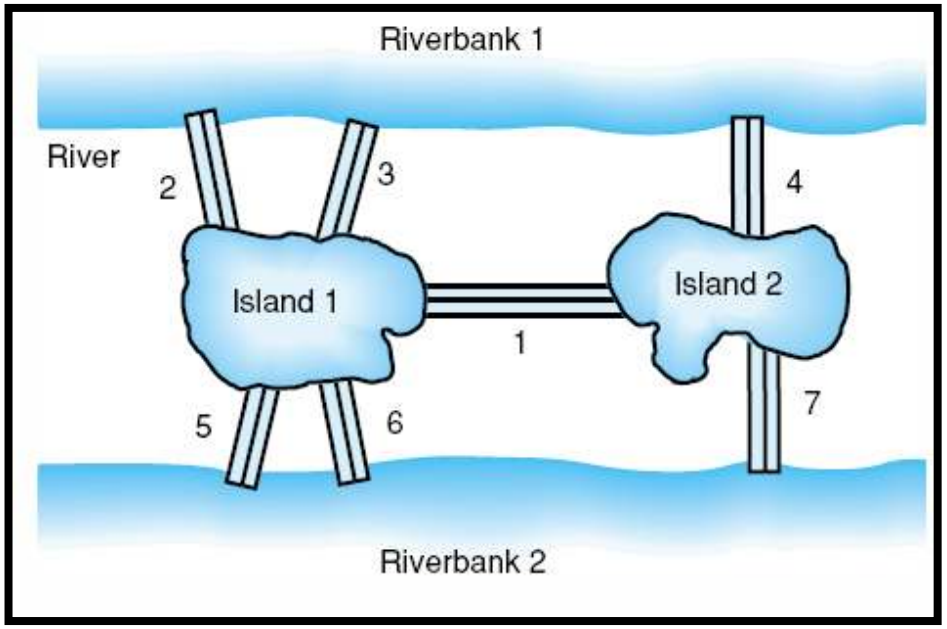
Use of graph theory in state space search

- In the state space model of problem solving:
  - The nodes of a graph are taken to represent discrete states in a problem-solving process.
  - The arcs of the graph represent transitions between states.
- Graph theory is our best tool for reasoning about the structure of objects and relations.
- The Swiss mathematician Leonhard Euler invented graph theory to solve the “bridges of Königsberg problem.”

The Problem of Königsberg Bridges (i)

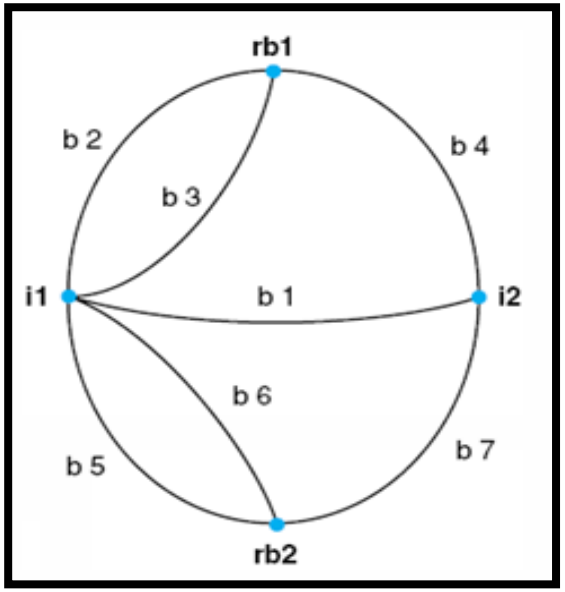
- The city of Königsberg occupied both banks and two islands of a river. The islands and the riverbanks were connected by seven bridges.

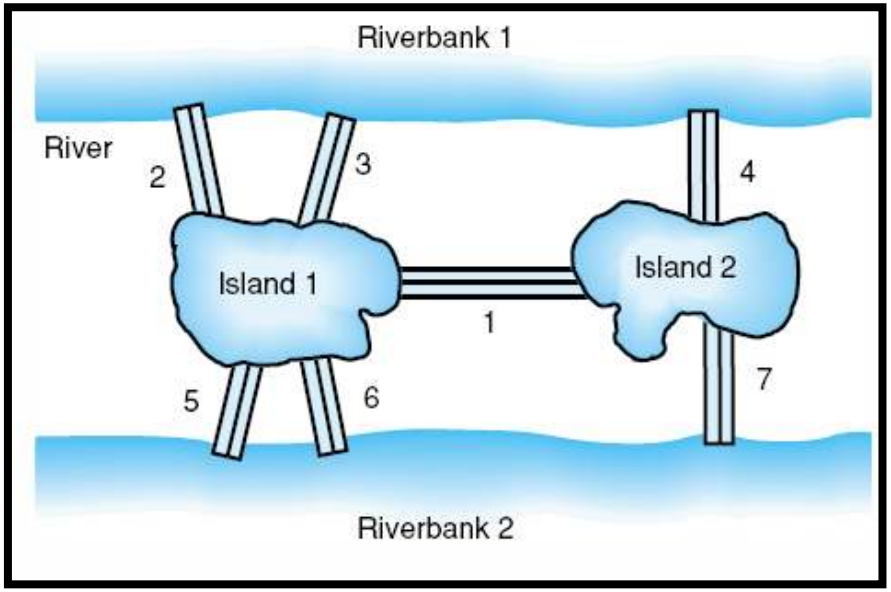
- The bridges of Königsberg problem asks if there is a walk around the city that crosses each bridge exactly once.



The Problem of Königsberg Bridges (ii)

- The residents had failed to find such a walk and doubted that it was possible, no one had proved its impossibility.
- Devising a form of graph theory, Euler created an alternative representation for the map.



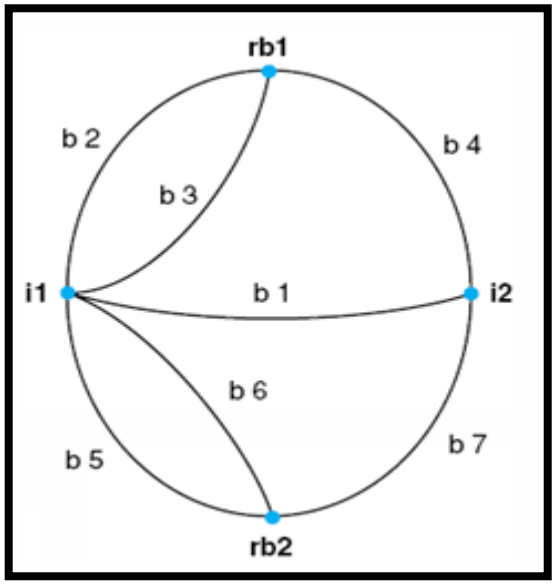


The riverbanks are (rb1and rb2) ,
islands are (i1and i2) and
the bridges are (b1, b2,…., b7).

The Problem of Königsberg Bridges (iii)

- The riverbanks (rb1and rb2) and islands (i1and i2) are described by the nodes of a graph ; the bridges are represented by labeled arcs between nodes (b1, b2,… , b7).

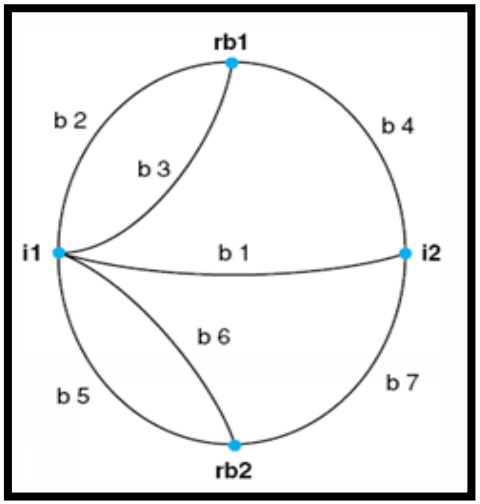
- The graph representation preserves the essential structure of the bridge system, while ignoring extraneous features such as bridge lengths, distances, and order of bridges in the walk.



The Problem of Königsberg Bridges (iv)

- We may represent the Königsberg bridge system using predicate calculus. The connect predicate corresponds to an arc of the graph, asserting that two land masses are connected by a particular bridge.
- Each bridge requires two connect predicates, one for each direction in which the bridge may be crossed.
- A predicate expression, connect(X, Y, Z) = connect (Y, X, Z), indicating that any bridge can be crossed in either direction, would allow removal of half the following connect facts:

connect(i1, i2, b1)	connect(i2, i1, b1)
connect(rb1, i1, b2)	connect(i1, rb1, b2)
connect(rb1, i1, b3)	connect(i1, rb1, b3)
connect(rb1, i2, b4)	connect(i2, rb1, b4)
connect(rb2, i1, b5)	connect(i1, rb2, b5)
connect(rb2, i1, b6)	connect(i1, rb2, b6)
connect(rb2, i2, b7)	connect(i2, rb2, b7)



The Problem of Königsberg Bridges (v)

- Euler focused on the degree of the nodes of the graph
  - With the exception of its beginning and ending nodes, the desired walk would have to leave each node exactly as often as it entered it.
  - Nodes of odd degree could be used only as the beginning or ending of the walk, because such nodes could be crossed only a certain number of times before they proved to be a dead end. The traveler could not exit the node without using a previously traveled arc.
  - Euler noted that unless a graph contained either exactly zero or two nodes of odd degree, the walk was impossible.
  - If there were two odd-degree nodes, the walk could start at the first and end at the second.
  - If there were no nodes of odd degree, the walk could begin and end at the same node.

The Finite State Machine (FSM)

- A finite state machine (FSM) is a finite, directed, connected graph, having a:
  - Finite set of states (S)
  - Finite set of input values (I), and
  - State transition function(F) that describes the effect that the elements of the input stream have on the states of the graph.
    - Thus ∀ i ∈ I, Fi : (S → S). If the machine is in state sj and input i occurs, the next state of the machine will be Fi (sj).

- The stream of input values produces a path within the graph of the states of this finite machine.

- The primary use for such a machine is to recognize components of a formal language. These components are often strings of characters ("words" made from characters of an "alphabet").

FSM Representation

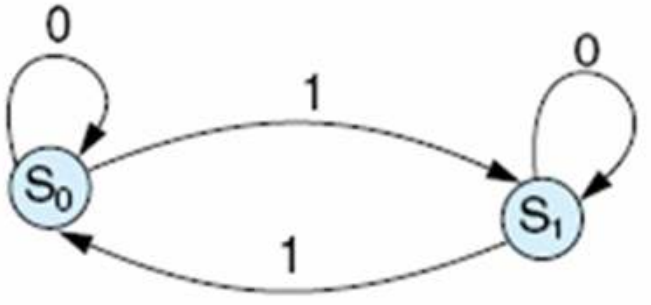
- FSM may be represented in two equivalent ways using:
  - A transition matrix, where:
    - input values are listed along the top row
    - the states are in the leftmost column
    - the output (next state) for an input applied to a state is at the intersection point.
  - A finite graph
    - that has directed arcs, labeled with the input values.

Example

For a finite state machine that has S = {s0, s1}, I = {0,1}, f0(s0) = s0, f0(s1) = (s1), f1(s0) = s1, and f1(s1) = s0, draw:

1. The finite state graph

2. The transition matrix



STATES	Input values: 0	1
S_0	S_0	S_1
S_1	S_1	S_0

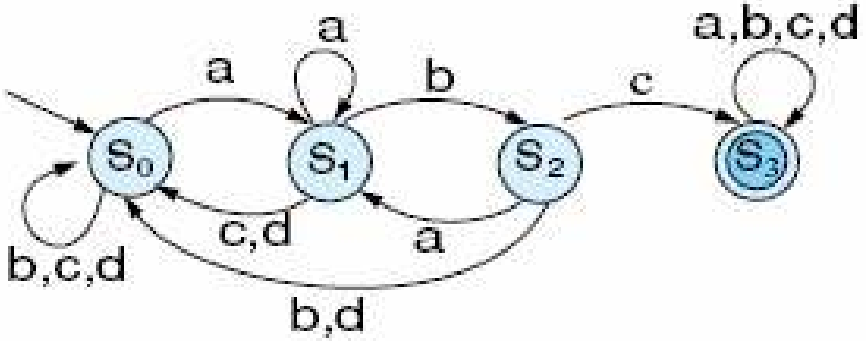
The Finite State Acceptor (MOORE Machine)

- The finite state acceptor is a finite state machine (S, I, F), where:
  - ∃ s0 ∈ S: the starting state such that the input stream starts at s0
    - We use the convention of placing an arrow from no state that terminates in the starting state of the Moore machine
  - ∃ sn ∈ S: called the accepting state.
    - The input stream is accepted if it terminates in that state.
    - In fact, there may be a set of accept states.
    - The accepting state (or states) is represented using a doubled circle
- The finite state acceptor is represented as (S, s0, {sn}, I, F)

Example of the Finite State Acceptor (MOORE Machine): string recognition

- With two assumptions, this machine could be seen as a recognizer of all strings of characters from the alphabet {a, b, c, d} that contain the exact sequence "a b c"

- The two assumptions are:
  - State s0 has a special role as the starting state
  - State s3 is the accepting state.
  - Thus, the input stream will present its first element to state s0.
  - If the stream later terminates with the machine in state s3, it will have recognized that there is the sequence "a b c" within that input stream.



	a	b	c	d
▶ S_0	S_1	S_0	S_0	S_0
S_1	S_1	S_2	S_0	S_0
S_2	S_1	S_0	S_3	S_0
S_3	S_3	S_3	S_3	S_3

Example

- For the shown FSM which of these inputs are valid (will be accepted by the shown FSM):
  - aaacdb

CORRECT

  - ababacdaaac

CORRECT

  - abcdb

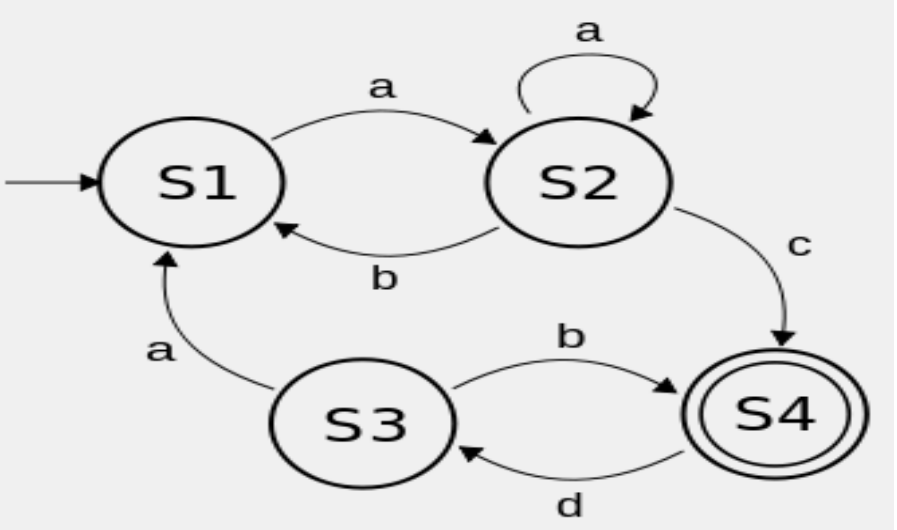
(ERROR; there is no input that accepts b then c)

  - acda

(ERROR S1 is not a accepting state)

  - acdbdb

CORRECT



State Space Representation of Problems (i)

- As mentioned earlier, the theory of state space search is our primary tool for finding the solution by representing the problem as a state space graph in which:
  - The nodes of a graph correspond to partial problem solution states
  - The arcs correspond to steps in a problem-solving (solution) process.
  - One or more start states (corresponding to the given information in a problem instance) form the root of the graph.
  - The goal state(s) of the problem. These states are described using either:
    - A measurable property of the states encountered in the search.
    - A measurable property of the path developed in the search, for example, the sum of the transition costs for the arcs of the path.
- State space search characterizes problem solving as the process of finding a solution path from the start state to a goal.

State Space Representation of Problems (ii)

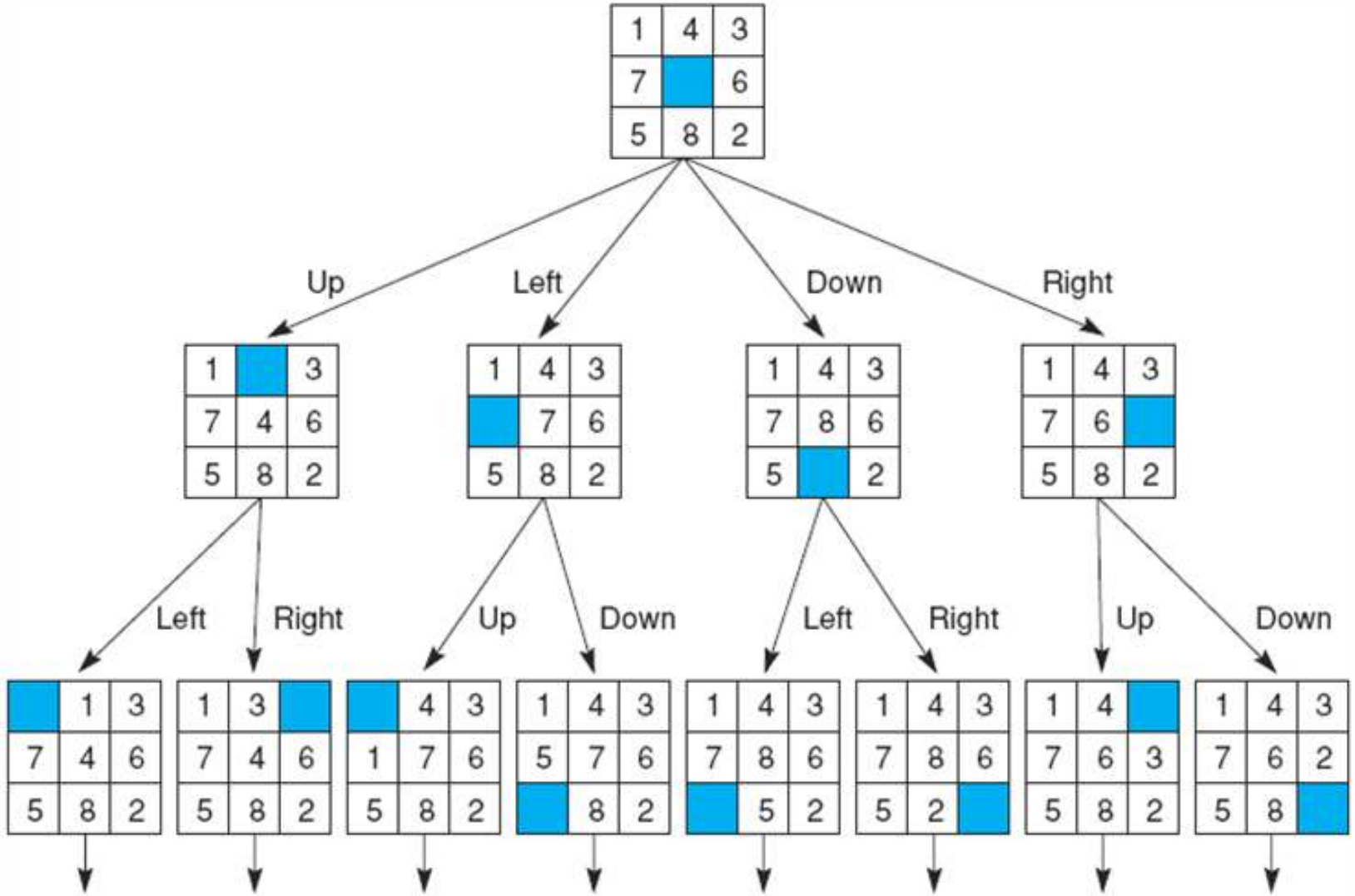
- **Paths** through the space represent solutions in various stages of completion.
  - Paths are searched, beginning at the start state and continuing through the graph, until either the goal description is satisfied or they are abandoned.
  - The actual generation of new states along the path is done by applying operators, such as "legal moves" in a game or inference rules in a logic problem or expert system, to existing states on a path.
  - **A solution path** is a path through the graph from a start nodes to goal nodes
  - A **path cost** function: assigns a numeric cost to each path = performance measure, denoted by ***g***, to distinguish the best path from others. Usually the path cost is the sum of the **step costs** of the individual actions (in the action list).
  - **Optimal solution** : the solution with lowest path cost among all solutions.
- The task of a search algorithm is to find a solution path through such a problem space.
  - Search algorithms must keep track of the paths from a start to a goal node, because these paths contain the series of operations that lead to the problem solution.

State Space Representation of Problems
Example (i): the 8-puzzle

1	4	3
7		6
5	8	2

- Eight tiles can be moved around in nine spaces.
  - Although physical puzzle moves are made by moving tiles, it's simpler to think in terms of "moving the blank space". This simplifies the definition of move rules because there are eight tiles but only a single blank.
- States: state description specifies the location of each of the eight tiles and the blank in one of the nine squares.
- Starting state: any state can be designated as the starting state
- Goal description of the state space is a particular state or board configuration.
  - When this state is found on a path, the search terminates.
  - The path from start to goal is the desired series of moves.

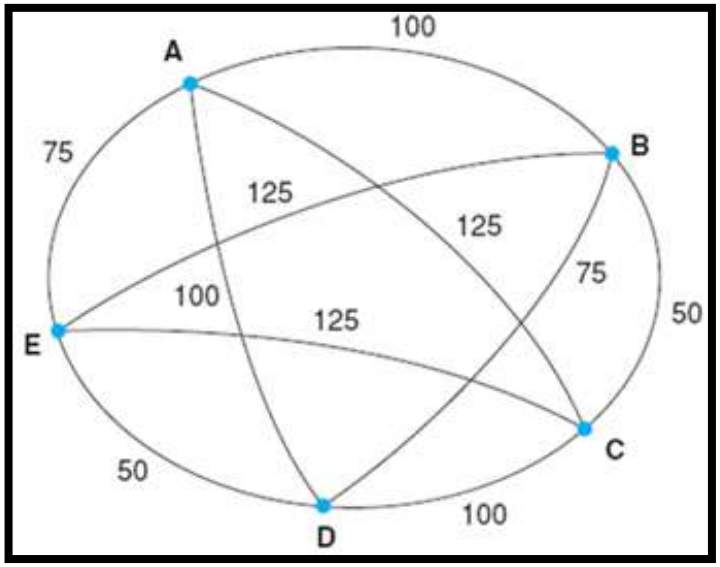
State space of the 8-puzzle: generated by "move blank" operations



State Space Representation of Problems

Example (ii): the travelling salesperson_1

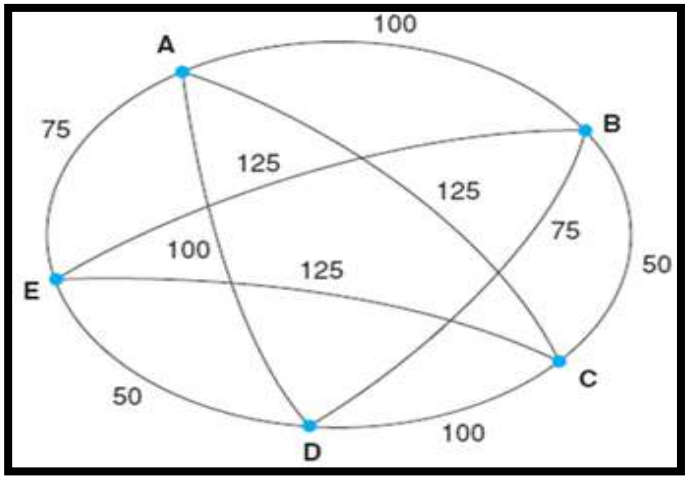
- A salesperson has five cities to visit (cities A, B, C, D and E) and then must return home.
- The goal of the problem is to find the shortest path for the salesperson to travel, visiting each city (starting from city A, where he lives), and then returning to the starting city (city A).



State Space Representation of Problems

Example (ii): the travelling salesperson_2

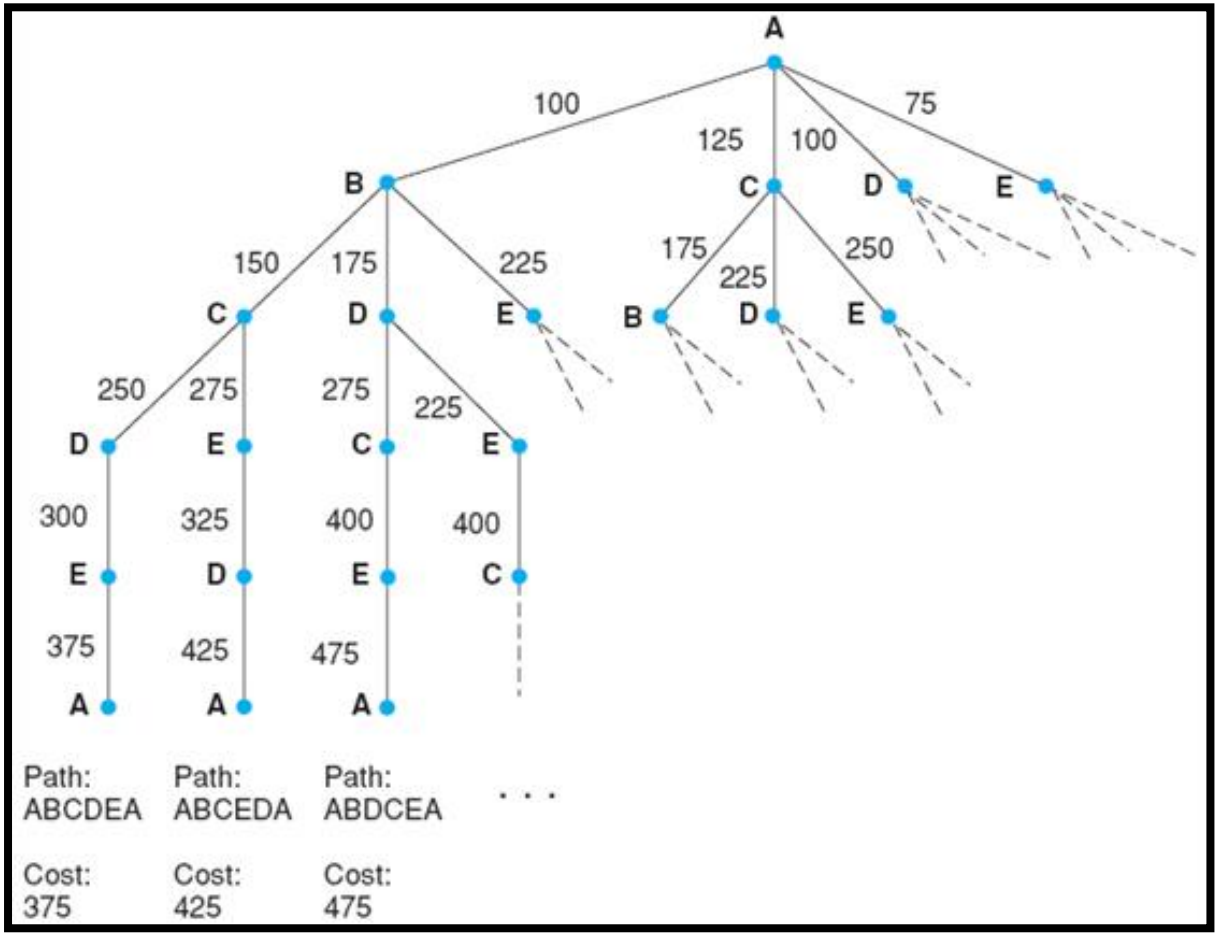
- The nodes of the graph represent cities
- Each arc is labeled with a weight indicating the cost of traveling that arc. This cost might be a representation of the miles necessary in car travel or cost of an air flight between the two cities.
- The goal description requires a complete circuit with minimum cost (the goal description here is a property of the entire path, rather than of a single state)



State Space Representation of Problems

Example (ii): the travelling salesperson_3

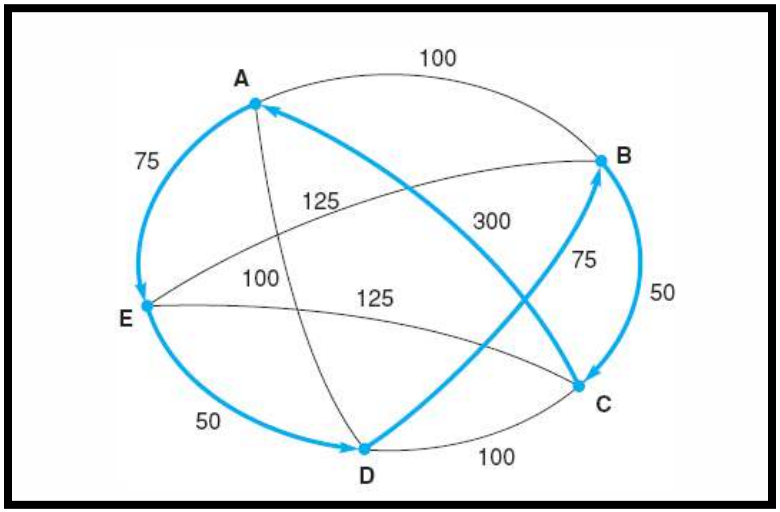
- The figure shows one way in which possible solution paths may be generated and compared.
- Beginning with node A, possible next states are added until all cities are included and the path returns home.
- The goal is the lowest-cost path.



State Space Representation of Problems

Example (ii): the travelling salesperson_4

- An instance of the travelling salesperson problem with the nearest neighbor path in bold is shown in the graph below.
  - This path (A, E, D, B, C, A), at a cost of 550, is not the shortest path.
  - The comparatively high cost of arc (C, A) defeated the heuristic.
- However, this method is highly efficient, as there is only one path to be tried. The nearest neighbor heuristic is fallible, as graphs exist for which it does not find the shortest path, but it is a possible compromise when the time required makes exhaustive search impractical.

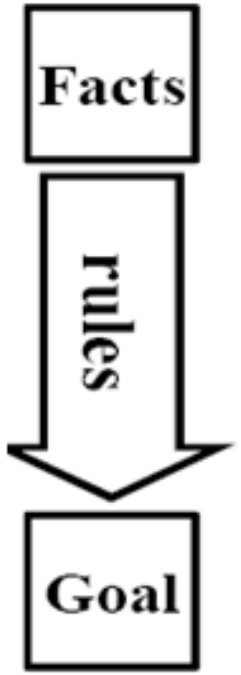


Strategies for Space State Search

- A state space may be searched in two directions:
  - From the given data of a problem instance toward a goal using data driven search "forward chaining"
  - From a goal back to the data using goal driven search "backward chaining".

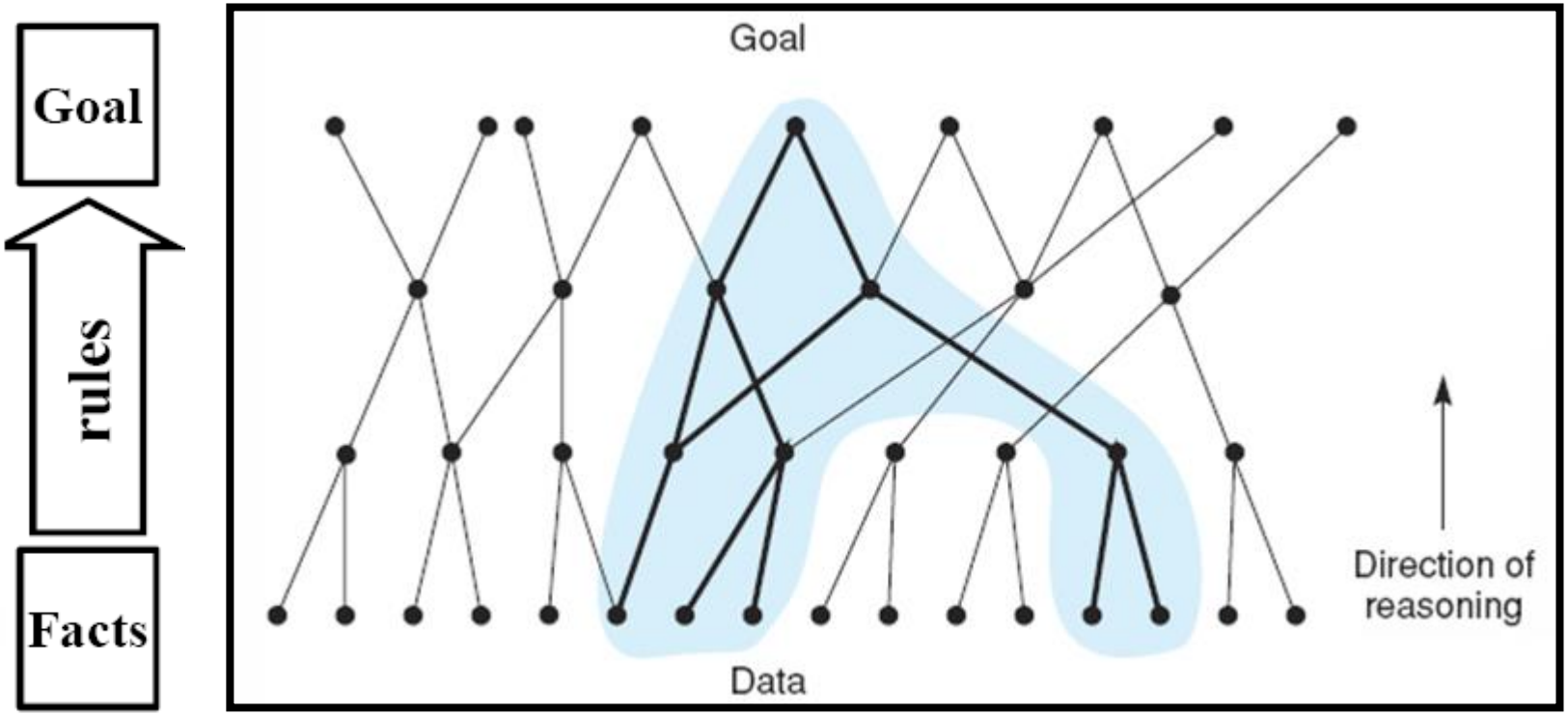
Data Driven search "Forward chaining" (i)

- In data-driven search, sometimes called forward chaining, the problem solver begins with the given facts of the problem and a set of legal moves or rules for changing state.
- Search proceeds by applying rules to facts to produce new facts, which are in turn used by the rules to generate more new facts. This process continues until (hopefully !) it generates a path that satisfies the goal condition.
- To summarize: data-driven reasoning takes the facts of the problem and applies the rules or legal moves to produce new facts that lead to a goal.



Data Driven search "Forward chaining" (ii)

- The figure shows the state space in which data-directed search prunes irrelevant data and their consequents and determines one of a number of possible goals.

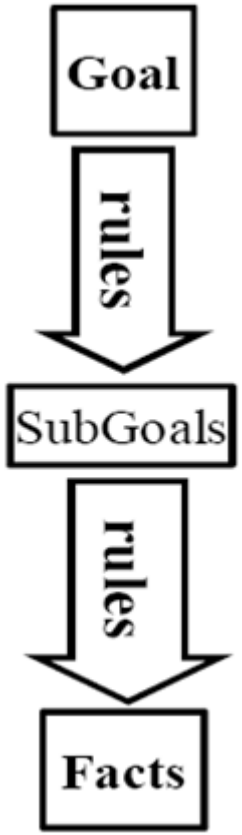


When to use the data driven search

- Data-driven search is appropriate for problems in which:
  - All or most of the data are given in the initial problem statement. Interpretation problems often fit this form by presenting a collection of data and asking the system to provide a high-level interpretation.
  - There are a large number of potential goals, but there are only a few ways to use the facts and given information of a particular problem instance.
  - It is difficult to form a goal or hypothesis.

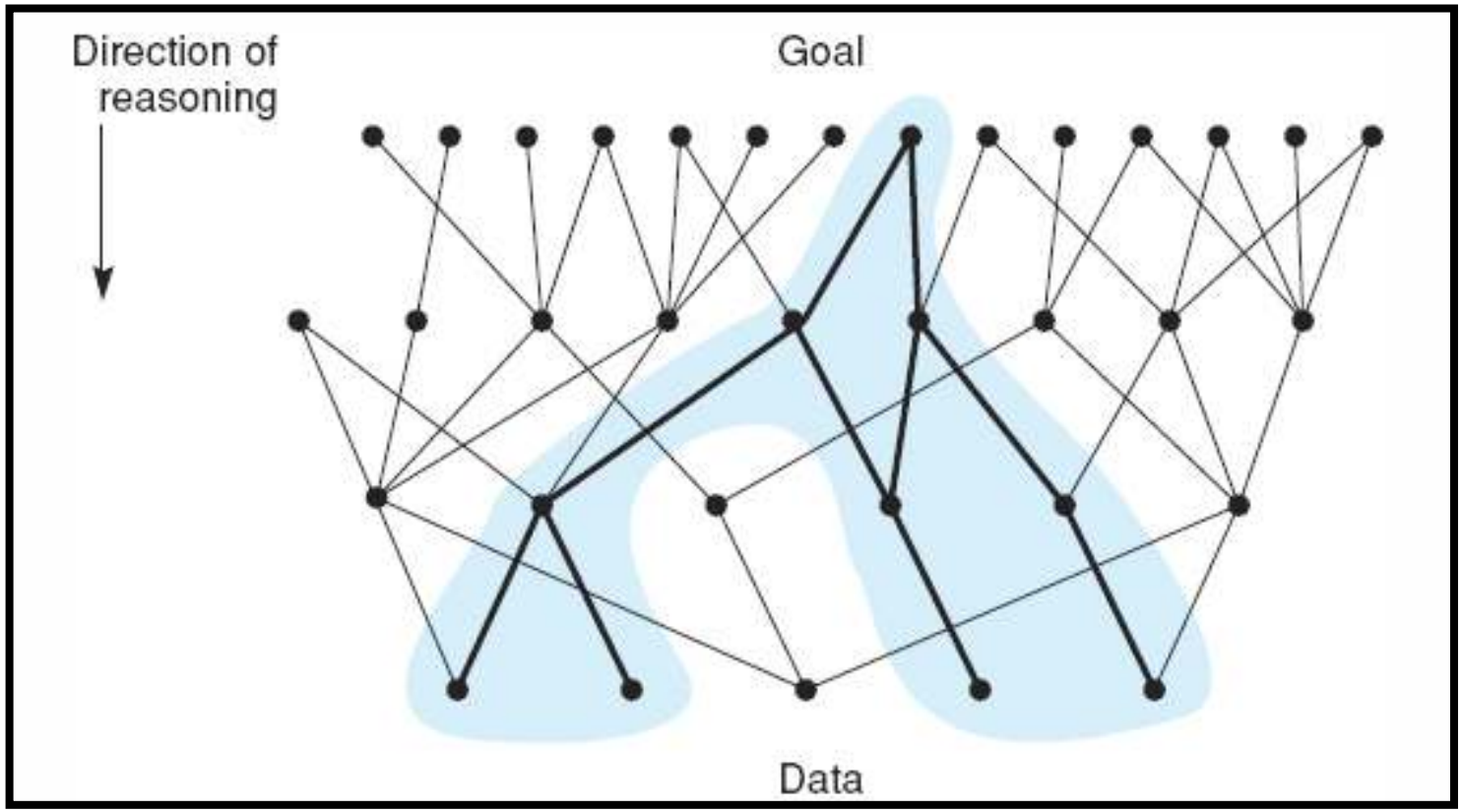
Goal Driven search "Backward chaining" (i)

- In goal driven search we:
  - Take the goal to solve.
  - See what rules or legal moves could be used to generate this goal and determine what conditions must be true to use them.
  - These conditions become the new goals, or subgoals, for the search.
  - Search continues, working backward through successive subgoals until (hopefully !) it works back to the facts of the problem.
- To summarize: goal-driven reasoning focuses on the goal, finds the rules that could produce the goal, and chains backward through successive rules and subgoals to the given facts of the problem.



Goal Driven search "Backward chaining" (ii)

- State space in which goal-directed search effectively prunes extraneous search paths.



When to use the goal driven search

- Goal-driven search is suggested if:
  - A goal or hypothesis is given in the problem statement.
  - There are a large number of rules that match the facts of the problem and thus produce an increasing number of conclusions or goals.
  - Problem data are not given but must be acquired by the problem solver. In this case, goal-driven search can help guide data acquisition.

- Goal-driven search thus uses knowledge of the desired goal to guide the search through relevant rules and eliminate branches of the space.

Which one to choose? Data or goal-driven search

- Both data-driven and goal-driven problem solvers search the same state space graph; however, the order and actual number of states searched can differ.
- The strategy is determined by the properties of the problem itself. These include:
  - The complexity of the rules
  - The "shape" of the state space
  - The nature and availability of the problem data.
- The decision to choose between data- and goal-driven search is based on the structure of the problem to be solved.
- All of these vary for different problems.

Strategies for Space State Search

- A strategy is defined by picking the order of node expansion.

- Strategies are evaluated along the following dimensions:
  - Completeness: does it always find a solution if one exists?
  - Optimality: does it always find a least-cost solution?
  - Time complexity: how many steps does it take to solve a problem?
  - Space complexity: how much memory does it take to solve a problem?

- Time and space complexities are measured in terms of:
  - b: maximum branching factor (the number of children for each node) of the search tree.
  - d: depth of the least-cost solution.
  - m: maximum depth of the state space (may be infinite).

Search Strategies (i)

- **Uninformed search**
  - No information about the number of steps
  - No information on the path cost from the current state to the goal
  - Search the state space blindly

- **Informed search, or heuristic search**
  - A cleverer strategy that searches toward the goal, based on the information from the current state so far

- Uninformed (blind) search strategies use only the information available in the problem definition.

- Informed search techniques might have additional information (e.g. a compass) in solving the problem.

Search Strategies (ii)

- Uninformed search
  - Easy
  - Very inefficient in most cases
  - Have huge search tree

- Informed search
  - Uses problem-specific information
  - Reduces the search tree into a small one
  - Resolves time and memory complexities

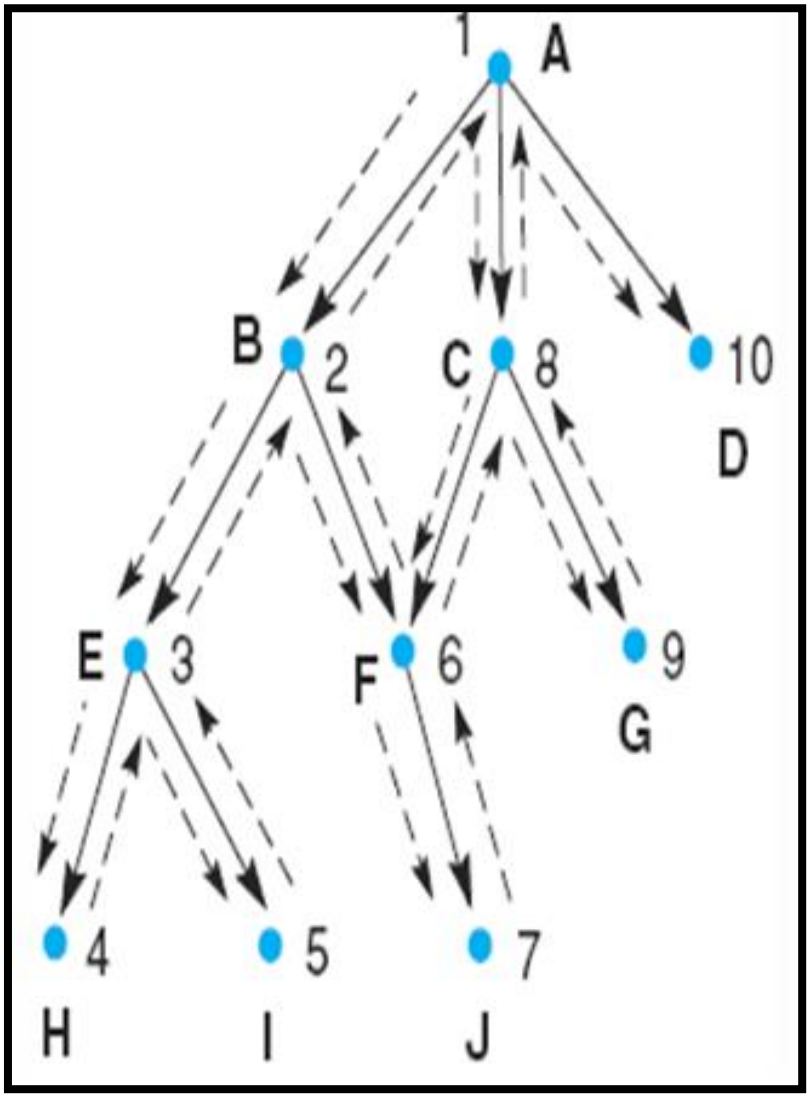
Implementing Graph Search

- In solving a problem using either goal- or data-driven search, the problem solver must find a path from a start state to a goal through the state space graph.

- The sequence of arcs in this path corresponds to the ordered steps of the solution

Backtracking

- **Backtracking** is a technique for systematically trying all paths through a state space.
- We will present a simpler version of the backtrack algorithm with **depth-first search (uninformed search)**
- Backtracking search begins at the start state and pursues a path until it reaches either a **goal** or a "**dead end**."
- If it finds a goal, it quits and returns the solution path.
- If it reaches a dead end, it "backtracks" to the most recent node on the path having unexamined siblings and continues down one of these branches



Backtracking Algorithm (i)

- A backtracking algorithm can be defined using three lists to keep track of nodes in the state space:
  - **State list (SL):** it lists the states in the current path being tried. If a goal is found, SL contains the ordered list of states on the solution path.
  - **New state list (NSL):** it contains nodes awaiting evaluation, i.e., nodes whose children have not yet been generated and searched.
  - **Dead ends list (DE):** it lists states whose children have failed to contain a goal. If these states are encountered again, they will be detected as elements of DE and eliminated from consideration immediately.

Backtracking Algorithm (ii)

- In backtrack, the state currently under consideration is called current state (CS).
  - CS is always equal to the state most recently added to SL and represents the "frontier" of the solution path currently being explored.
  - The result is an ordered set of new states, the children of CS.
  - The first of these children is made the new current state and the rest are placed in order on NSL for future examination.
- The new current state is added to SL and search continues.
- If CS has no children, it is removed from SL (this is where the algorithm "backtracks") and any remaining children of its predecessor on SL are examined.

Backtracking Algorithm (iii)

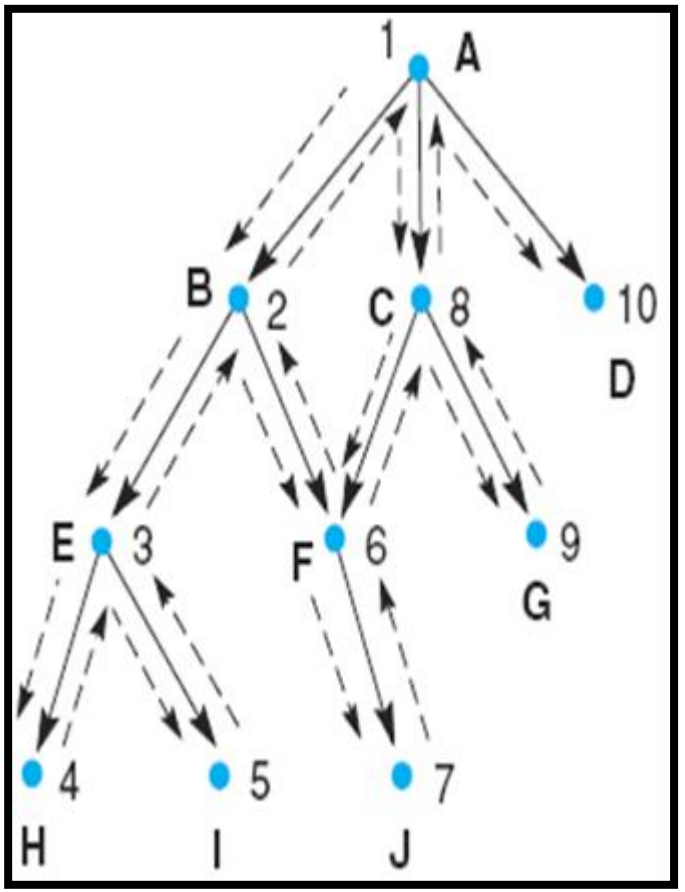

```
function backtrack;

begin
  SL := [Start];  NSL := [Start];  DE := [ ];  CS := Start;                    % initialize:
  while NSL ≠ [ ] do                                  % while there are states to be tried
    begin
      if CS = goal (or meets goal description)
        then return SL;                    % on success, return list of states in path.
      if CS has no children (excluding nodes already on DE, SL, and NSL)
        then begin
          while SL is not empty and CS = the first element of SL do
            begin
              add CS to DE;                                % record state as dead end
              remove first element from SL;                                %backtrack
              remove first element from NSL;
              CS := first element of NSL;
            end
          add CS to SL;
        end
        else begin
          place children of CS (except nodes already on DE, SL, or NSL) on NSL;
          CS := first element of NSL;
          add CS to SL
        end
    end;
    return FAIL;
end.
```

Backtracking Algorithm (iii)

Initialize: SL = [A]; NSL = [A]; DE = [ ]; CS = A;

AFTER ITERATION	CS	SL	NSL	DE
0	A	[A]	[A]	[ ]
1	B	[B A]	[B C D A]	[ ]
2	E	[E B A]	[E F B C D A]	[ ]
3	H	[H E B A]	[H I E F B C D A]	[ ]
4	I	[I E B A]	[I E F B C D A]	[H]
5	F	[F B A]	[F B C D A]	[E I H]
6	J	[J F B A]	[J F B C D A]	[E I H]
7	C	[C A]	[C D A]	[B F J E I H]
8	G	[G C A]	[G C D A]	[B F J E I H]



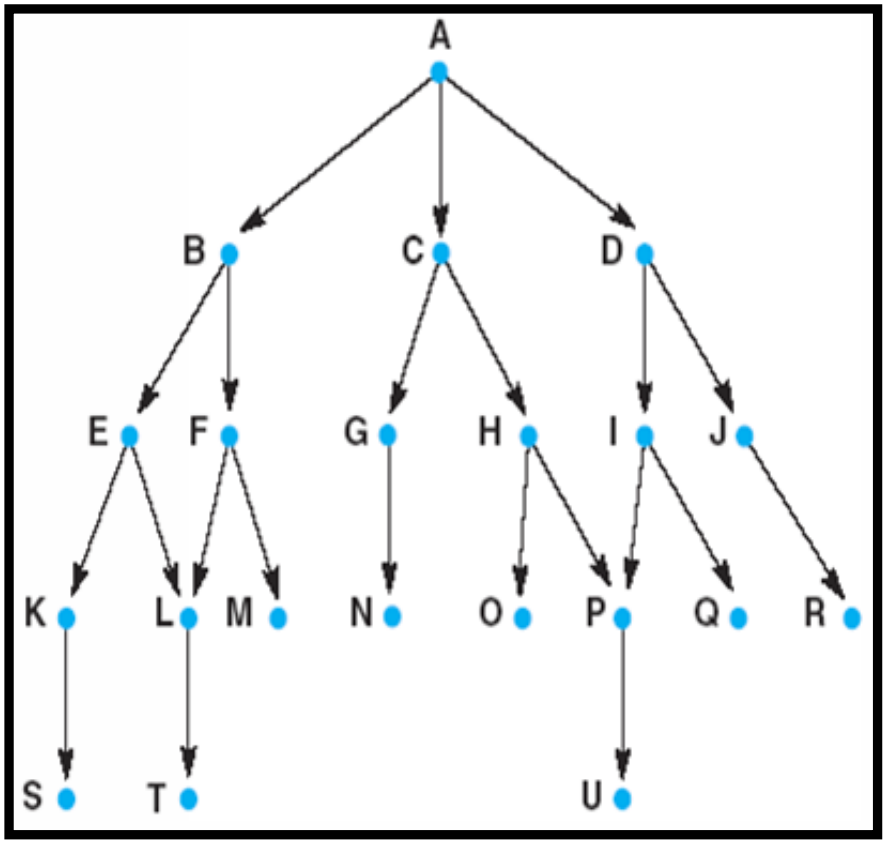
Depth-First and Breadth-First Search

- In addition to specifying a search direction (data-driven or goal-driven), a search algorithm must determine the order in which states are examined in the tree or the graph.

- We here considers two possibilities for the order in which the nodes of the graph are considered:
  - Depth-first search
  - Breadth-first search.

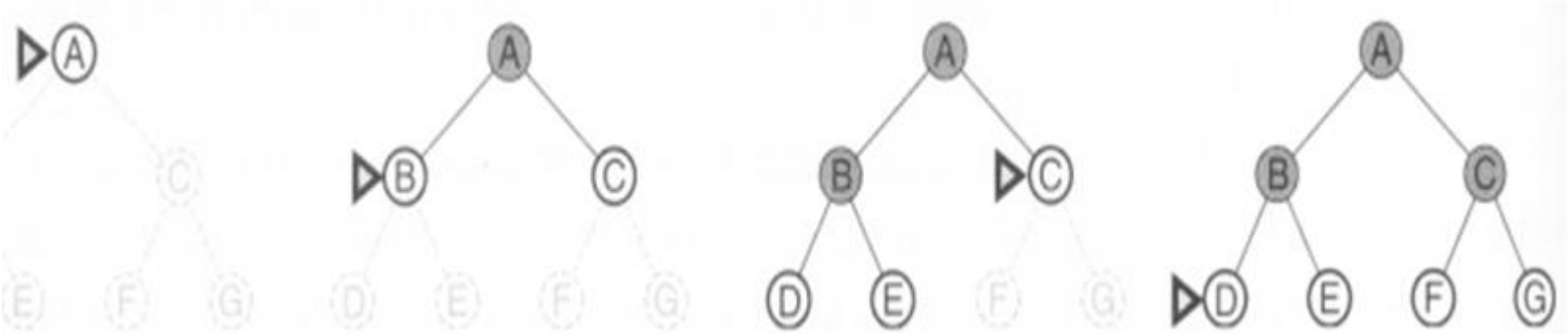
Breadth-First Search (BFS) (i)

- Breadth-first search, explores the space in a level-by-level fashion.
- Only when there are no more states to be explored at a given level, the algorithm moves onto the next deeper level.
- E.g.: a breadth-first search of the shown graph considers the states in the order A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U.



Breadth-First Search (ii)

- The root node is expanded first (FIFO)
- All the nodes generated by the root node are then expanded
- And then their successors and so on



Implementing Breadth-First Search

- Breadth-first search is implemented using two lists to keep track of progress through the state space.
  - Open (like NSL in backtrack) lists states that have been generated but whose children have not been examined. The order in which states are removed from open determines the order of the search.
  - Closed records states already examined. Closed is the union of the DE and SL lists of the backtrack algorithm.
- Child states are generated by inference rules, legal moves of a game, or other state transition operators.
- Each iteration produces all children of the state X an adds them to open.
- Note that open is maintained as a queue, or first-in-first out (FIFO) data structure.
- States are added to the right of the list and removed from the left.

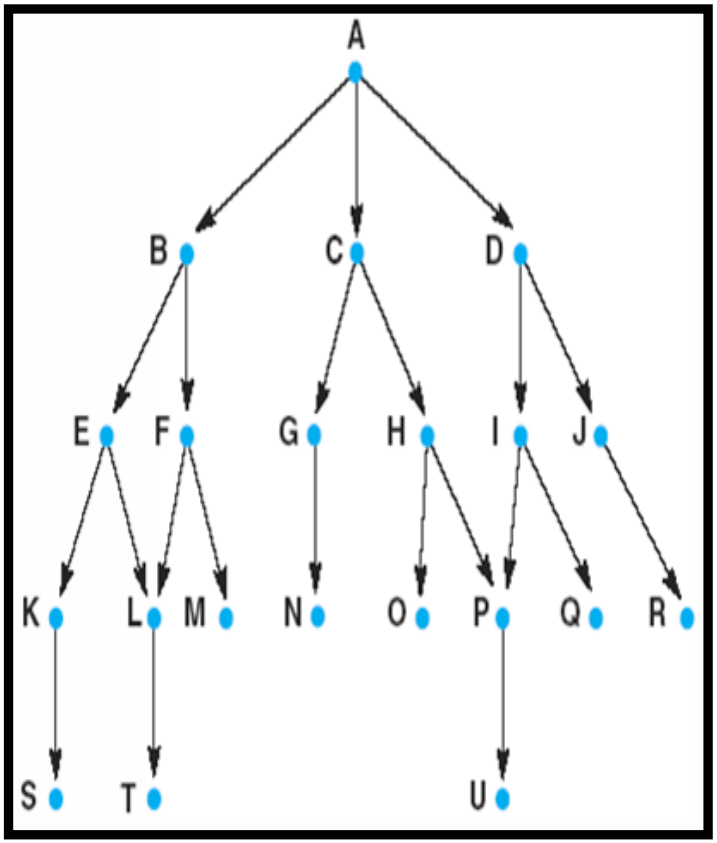
Breadth-First Search Algorithm


```
function breadth_first_search;

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

A trace of Breadth-first search

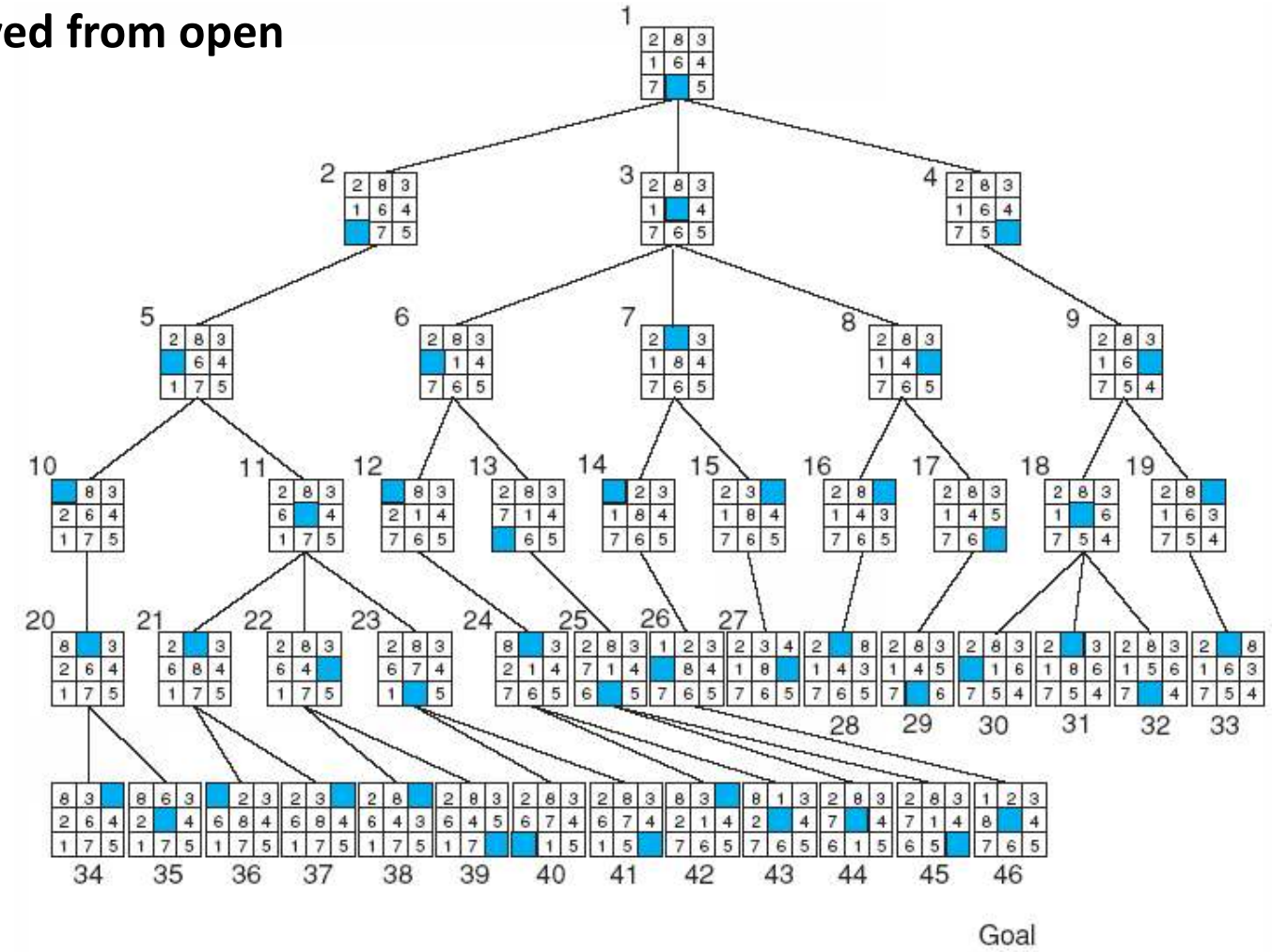
- Each successive number, 2,3,4, . . . , represents an iteration of the "while" loop.
- U is the goal state.



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

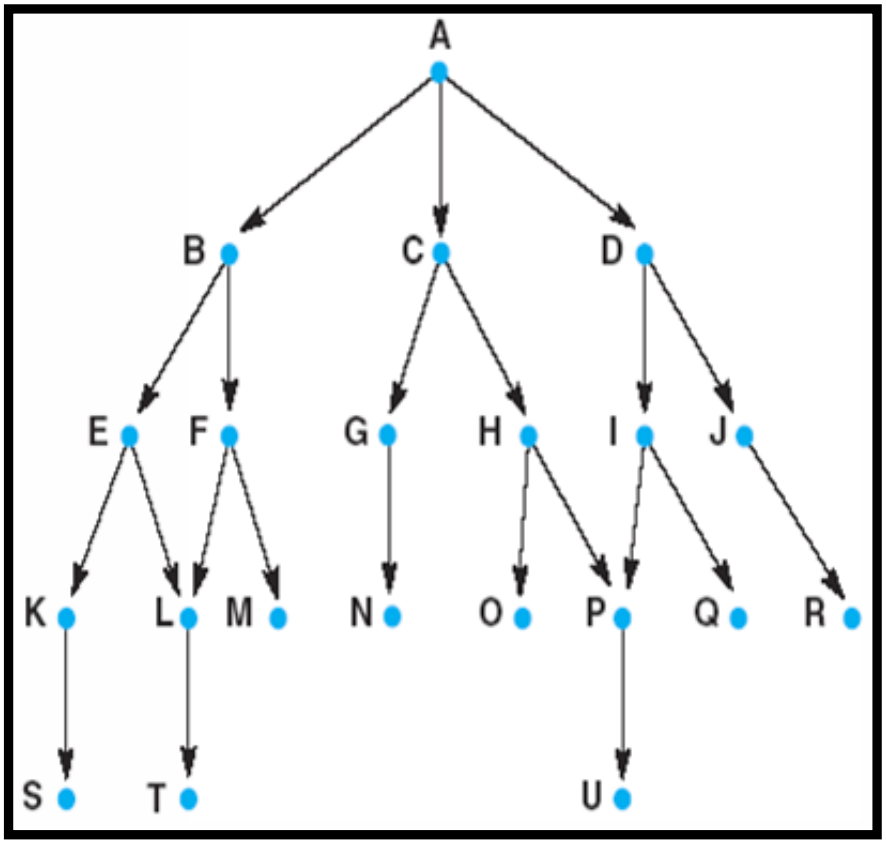
Breadth-first search of the 8-puzzle

- Breadth-first search of the 8-puzzle, showing order in which states were removed from open



Depth-First Search (DFS)

- In depth-first search, when a state is examined, all of its children and their descendants are examined before any of its siblings.
- Depth-first search goes deeper into the search space whenever this is possible.
- Only when no further descendants of a state can be found, its siblings are considered.
- E.g.: depth-first search examines the states in the shown graph in the order A, B, E, K,S, L, T, F, M, C, G, N, H, O, P, U, D, I, Q, J, R.



Implementing Depth-First Search

- In a depth-first search algorithm, the descendant states are added and removed from the left end of open

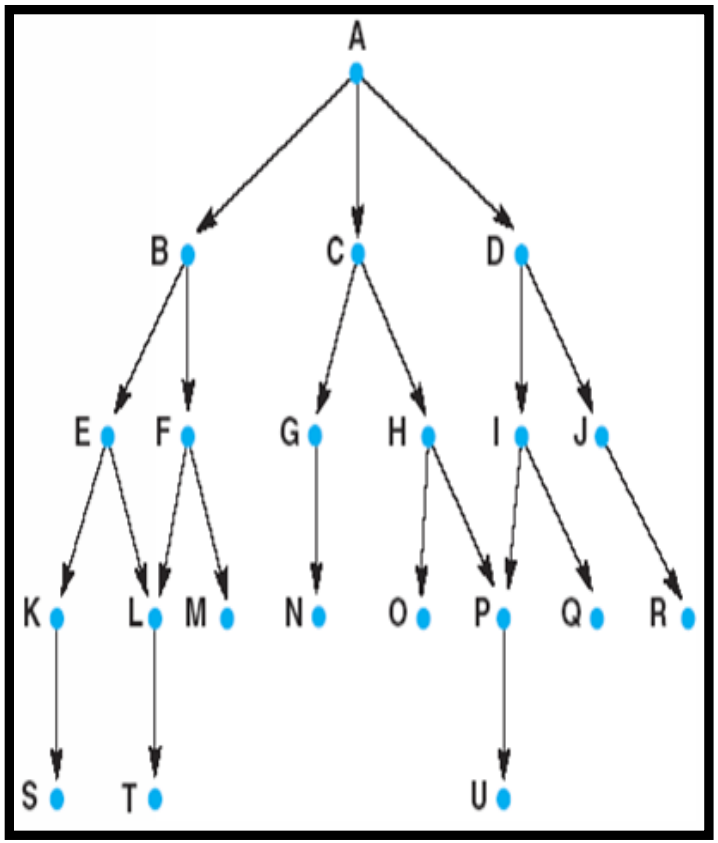
- Open is maintained as a stack, or last-in first-out(LIFO) structure. The organization of open as a stack directs search toward the most recently generated states, producing a depth-first search order.

Depth_first_search algorithm


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

A trace of depth-first search

- Each successive iteration of the “while” loop is indicated by a single line (2, 3, 4, . . .).
- The initial states of open and closed are given on line 1.
- U is the goal state.



1. open = [A]; closed = [ ]
2. open = [B,C,D]; closed = [A]
3. open = [E,F,C,D]; closed = [B,A]
4. open = [K,L,F,C,D]; closed = [E,B,A]
5. open = [S,L,F,C,D]; closed = [K,E,B,A]
6. open = [L,F,C,D]; closed = [S,K,E,B,A]
7. open = [T,F,C,D]; closed = [L,S,K,E,B,A]
8. open = [F,C,D]; closed = [T,L,S,K,E,B,A]
9. open = [M,C,D], as L is already on closed; closed = [F,T,L,S,K,E,B,A]
10. open = [C,D]; closed = [M,F,T,L,S,K,E,B,A]
11. open = [G,H,D]; closed = [C,M,F,T,L,S,K,E,B,A]

Depth-first search of the 8-puzzle with a depth bound of 5