3

Structures and Strategies For Space State Search

Figure 3.1: The city of Königsberg.

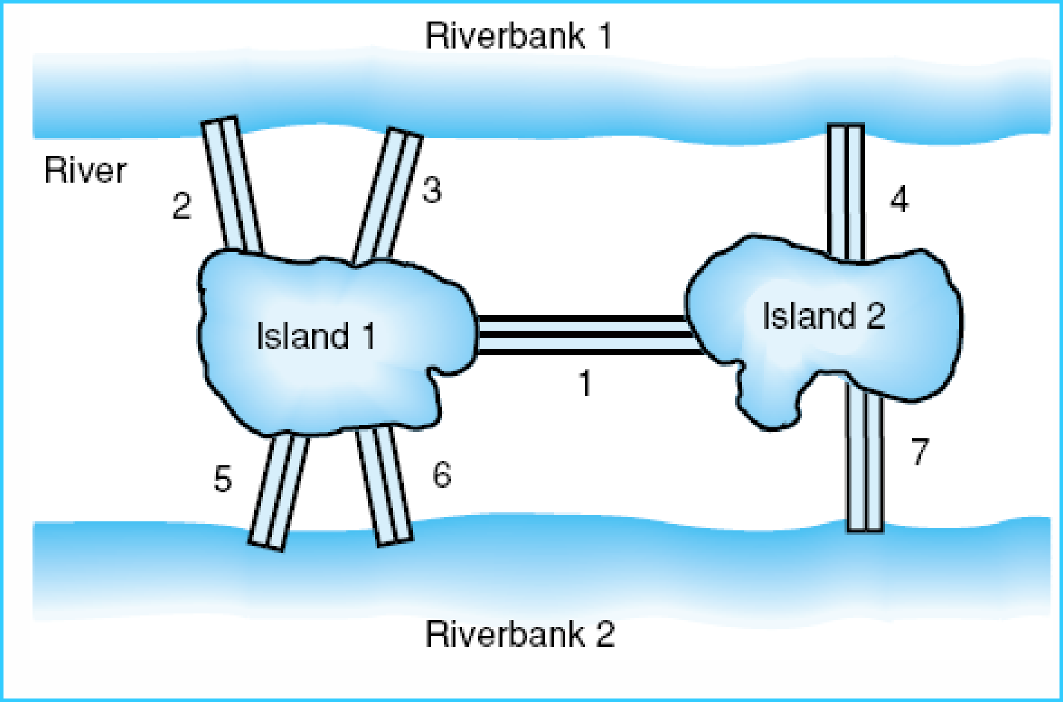


Figure 3.2: Graph of the Königsberg bridge system.

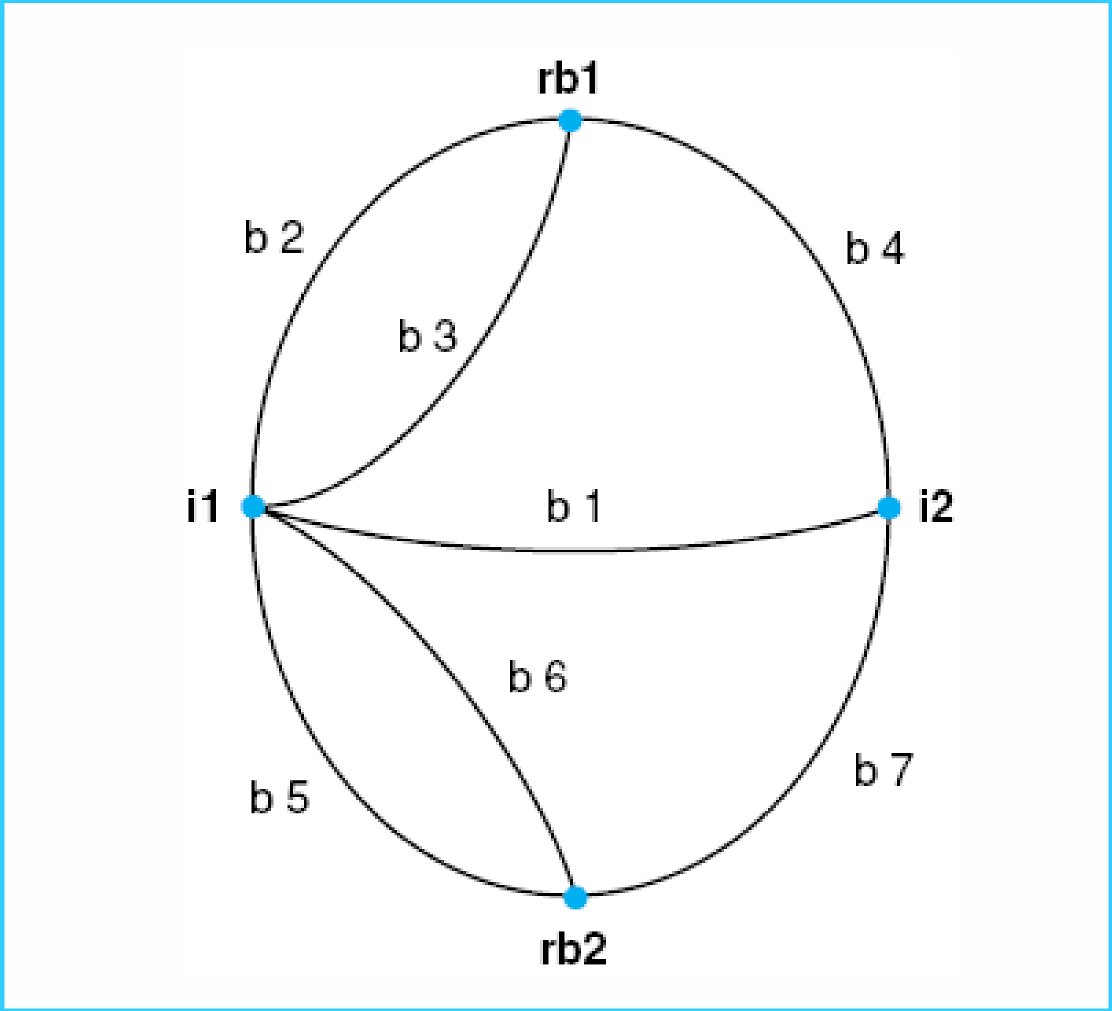
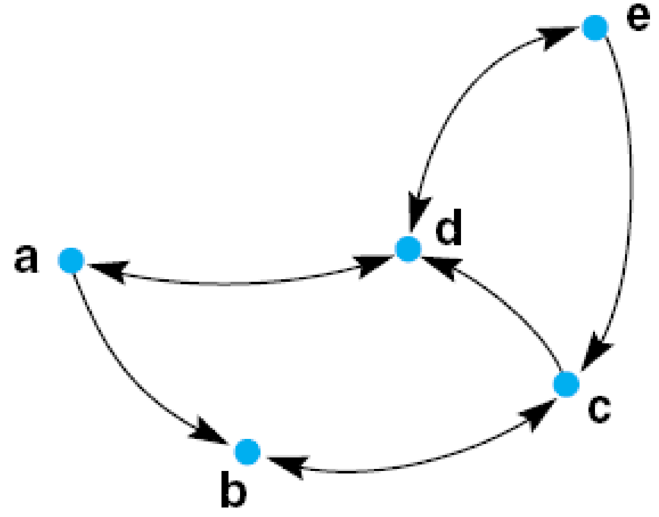


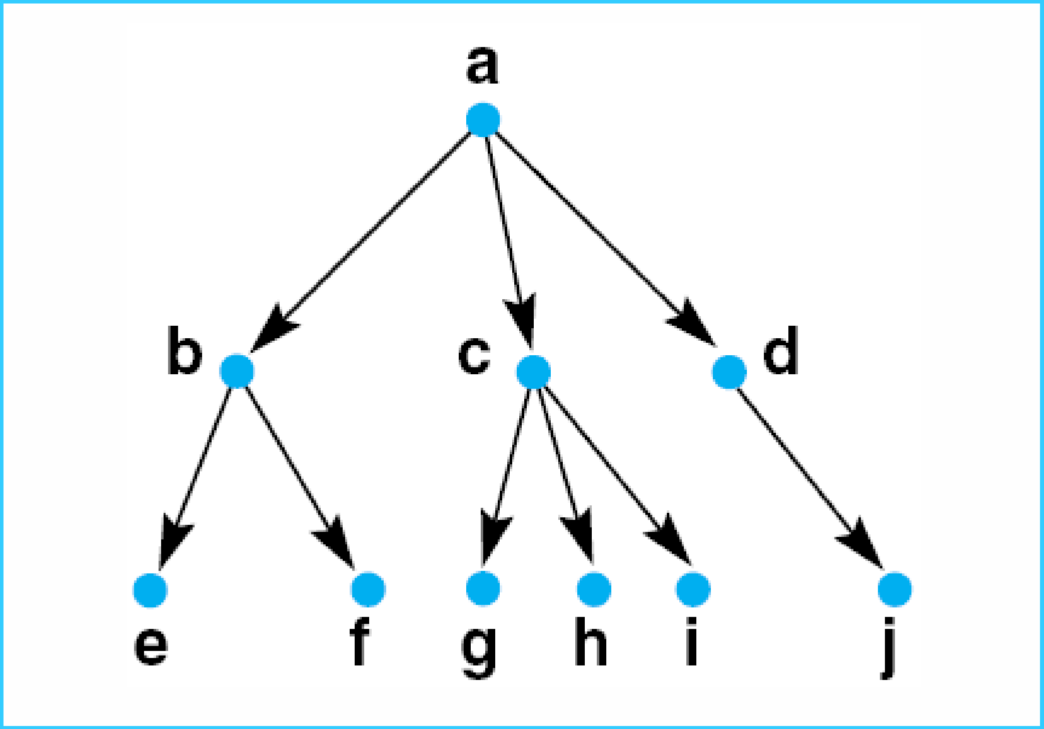
Figure 3.3: A labeled directed graph.



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)}

Figure 3.4: A rooted tree, exemplifying family relationships.



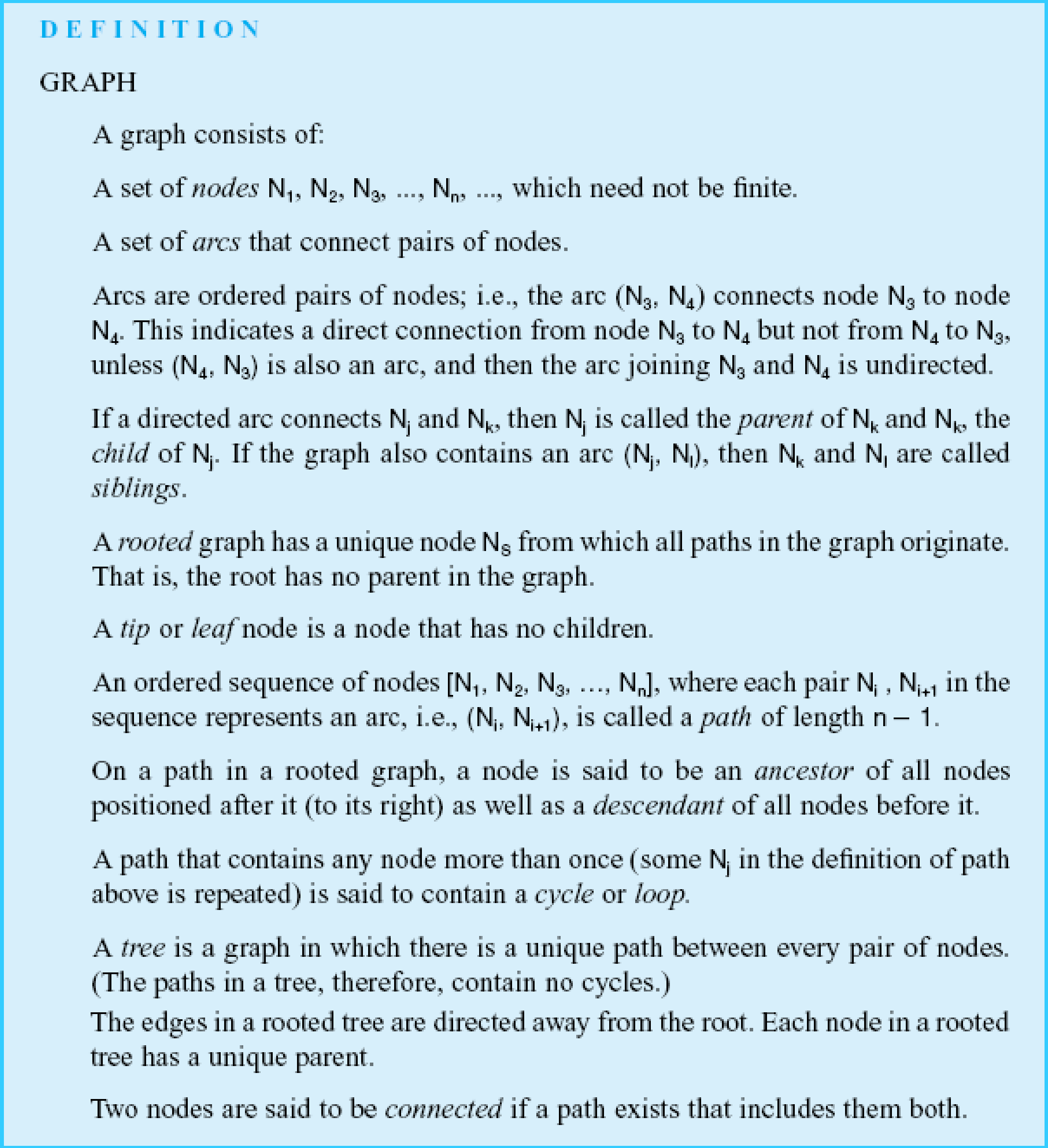
In the state space representation

- nodes -- states representing partial problem solutions
- arcs -- steps in a problem-solving process
- solution -- a goal state or a path from the start state to a goal state

By representing a problem as a state space graph, we can use graph theory to analyze the structure and complexity of the problem and the search algorithms.

The goal states are described by

- a measurable property of the states, or
- a property of the path developed in the search.



DEFINITION

GRAPH

A graph consists of:

A set of *nodes* $N_1, N_2, N_3, \ldots, N_n, \ldots$, which need not be finite.

A set of *arcs* that connect pairs of nodes.

Arcs are ordered pairs of nodes; i.e., the arc (N_3, N_4) connects node N_3 to node N_4. This indicates a direct connection from node N_3 to N_4 but not from N_4 to N_3, unless (N_4, N_3) is also an arc, and then the arc joining N_3 and N_4 is undirected.

If a directed arc connects N_j and N_k, then N_j is called the *parent* of N_k and N_k, the *child* of N_j. If the graph also contains an arc (N_j, N_l), then N_k and N_l are called *siblings*.

A *rooted* graph has a unique node N_S 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 $[N_1, N_2, N_3, \ldots, N_n]$, where each pair N_i , N_{i+1} in the sequence represents an arc, i.e., (N_i, N_{i+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 N_j in the definition of path above is repeated) is said to contain a *cycle* or *loop*.

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 edges 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.

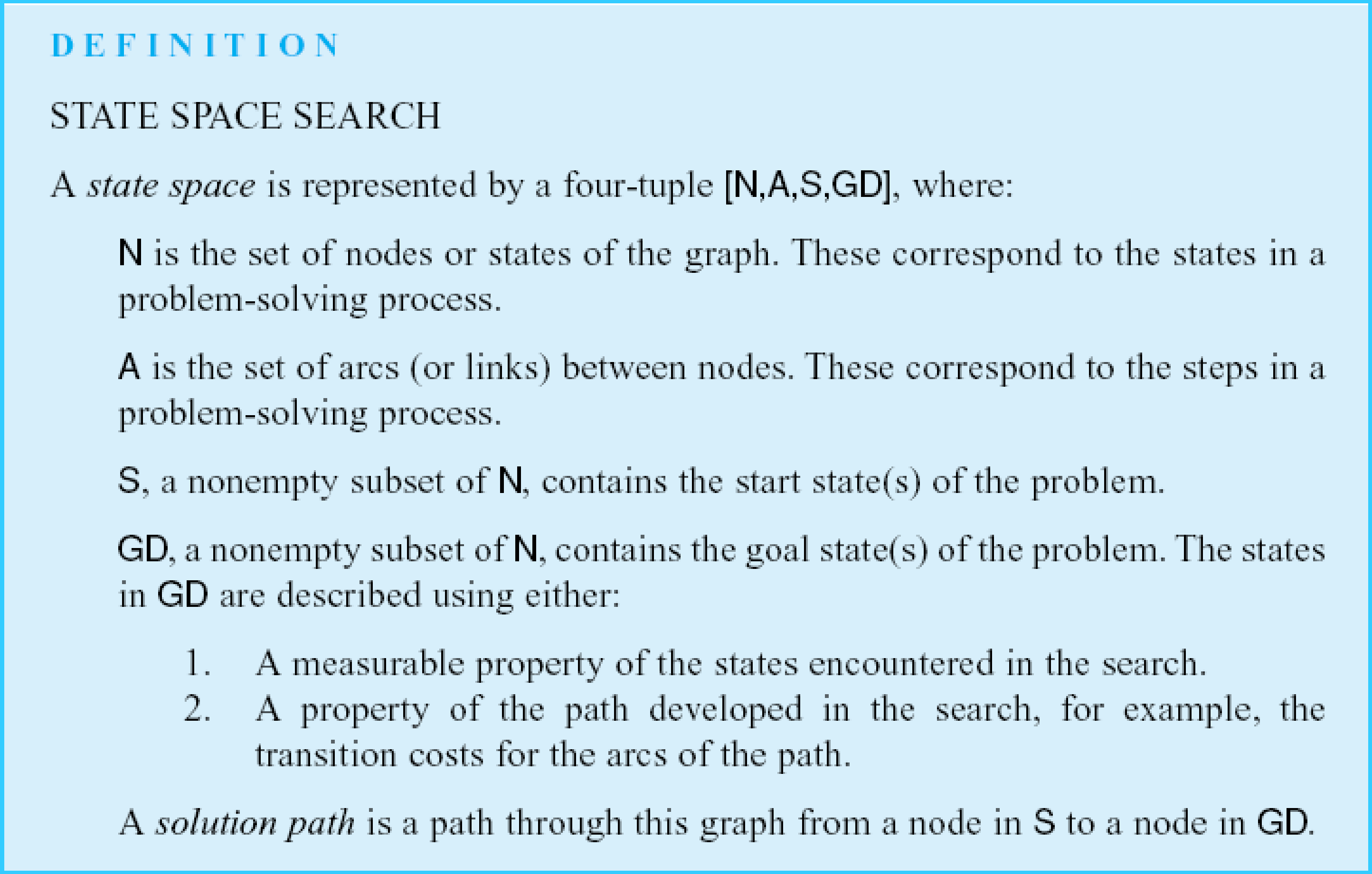
To design a search algorithm, we must consider

- Is it guaranteed to find a solution?
- Is it guaranteed to find an optimal solution?
- What is its complexity?
- Whether the complexity can be reduced?

Search algorithms must detect and eliminate loops from potential solution paths.

Strategies for state space search

- By the search directions:
  - data-driven search (forward chaining)
  - goal-driven search (backward chaining)
  - bidirectional search
- By whether using domain-specific information:
  - Blind search methods
  - Heuristic search



DEFINITION

STATE SPACE SEARCH

A *state space* is represented by a four-tuple [N,A,S,GD], where:

N is the set of nodes or states of the graph. These correspond to the states in a problem-solving process.

A is the set of arcs (or links) between nodes. These correspond to the steps in a problem-solving process.

S, a nonempty subset of N, contains the start state(s) of the problem.

GD, a nonempty subset of N, contains the goal state(s) of the problem. The states in GD are described using either:

1. A measurable property of the states encountered in the search.
2. A property of the path developed in the search, for example, the transition costs for the arcs of the path.

A *solution path* is a path through this graph from a node in S to a node in GD.

Fig 3.8 State space of the 8-puzzle generated by "move blank" operations

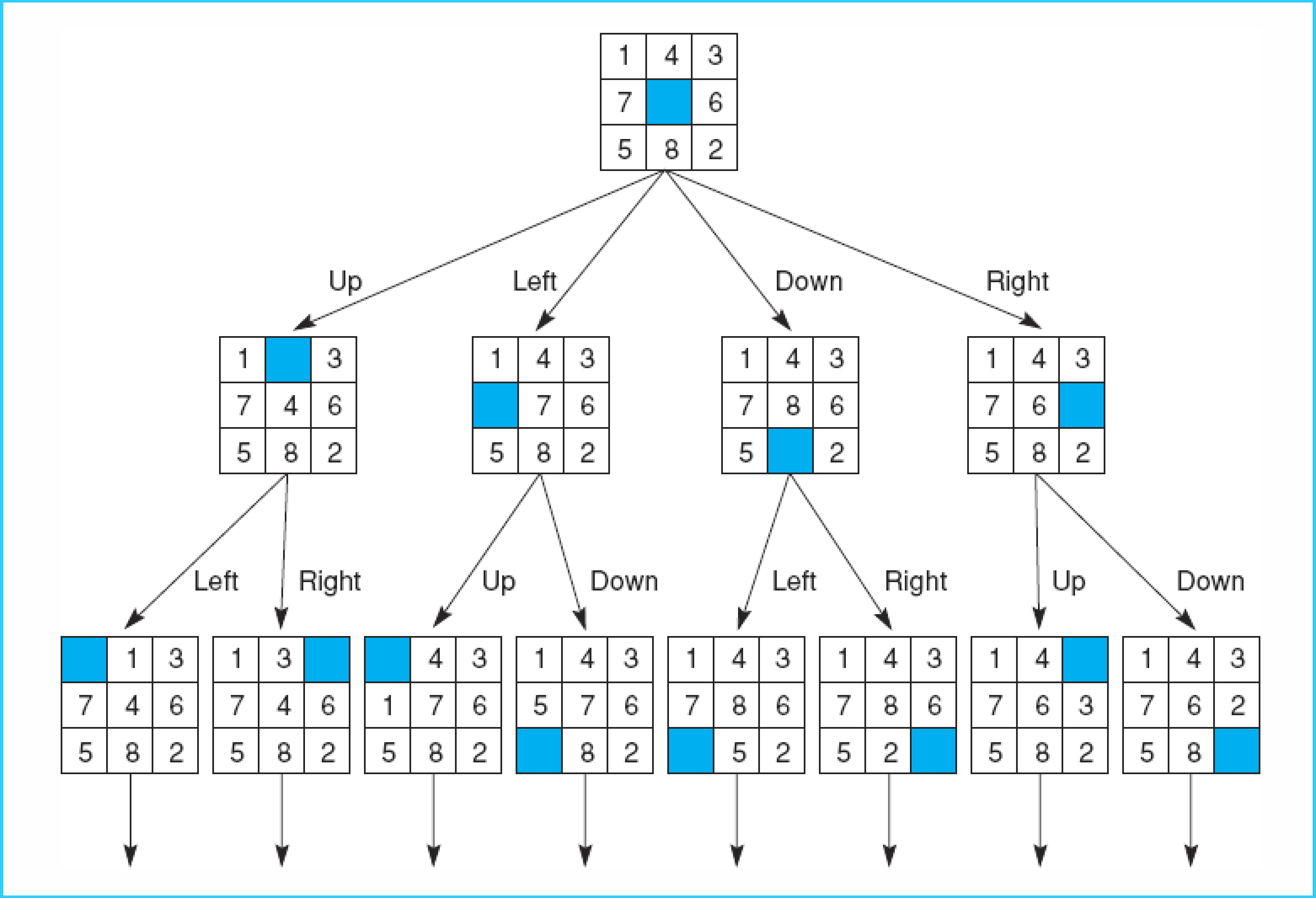


Fig 3.9 An instance of the travelling salesperson problem

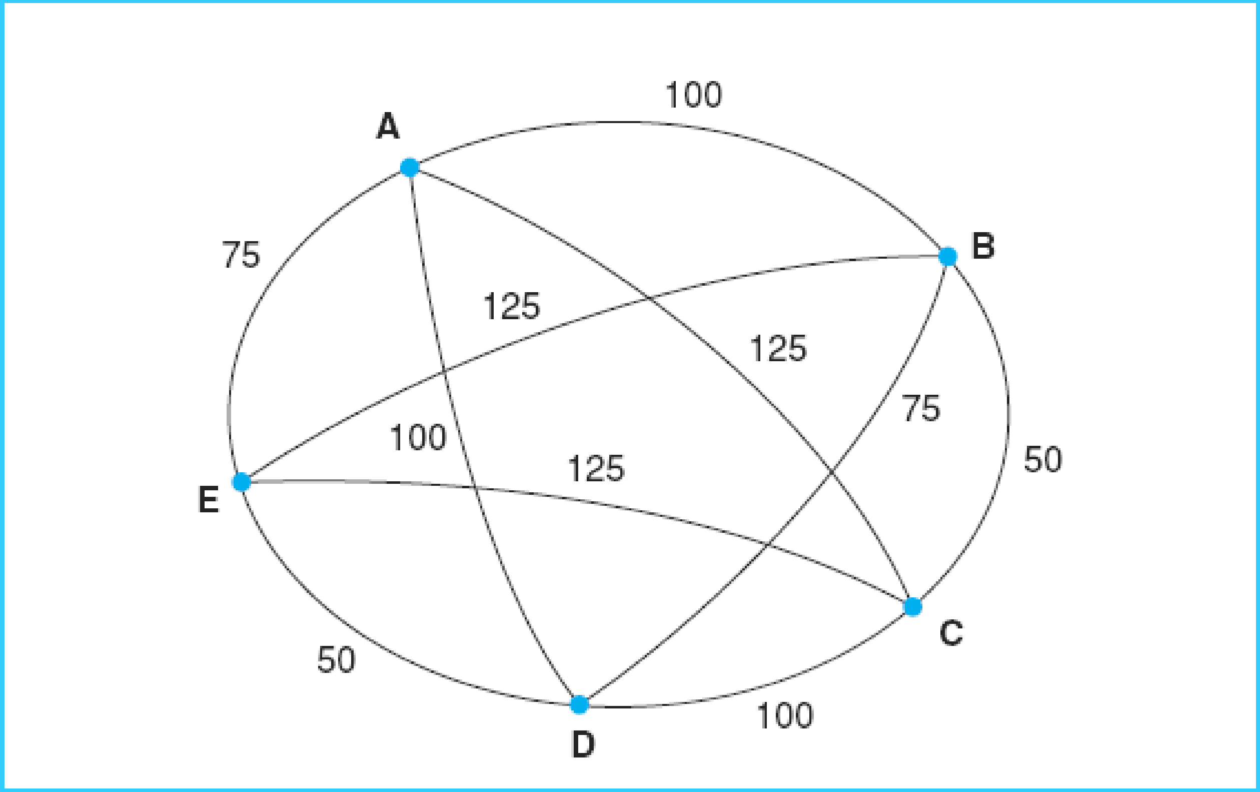


Fig 3.10 Search for the travelling salesperson problem. Each arc is marked with the total weight of all paths from the start node (A) to its endpoint.

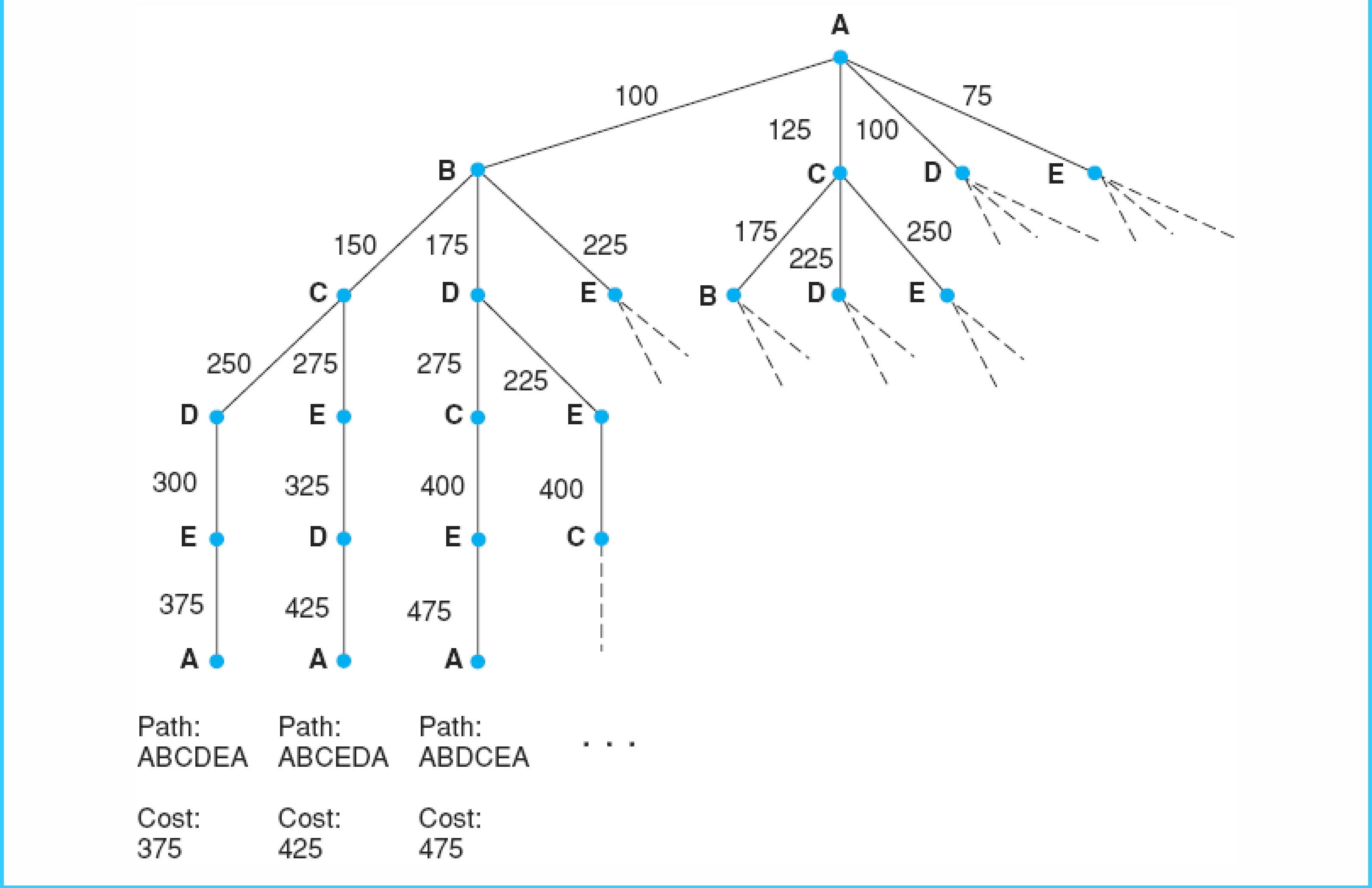


Fig 3.11 An instance of the travelling salesperson problem with the nearest neighbour path in bold. Note this path (A, E, D, B, C, A), at a cost of 550, is not the shortest path. The comparatively high cost of arc (C, A) defeated the heuristic.

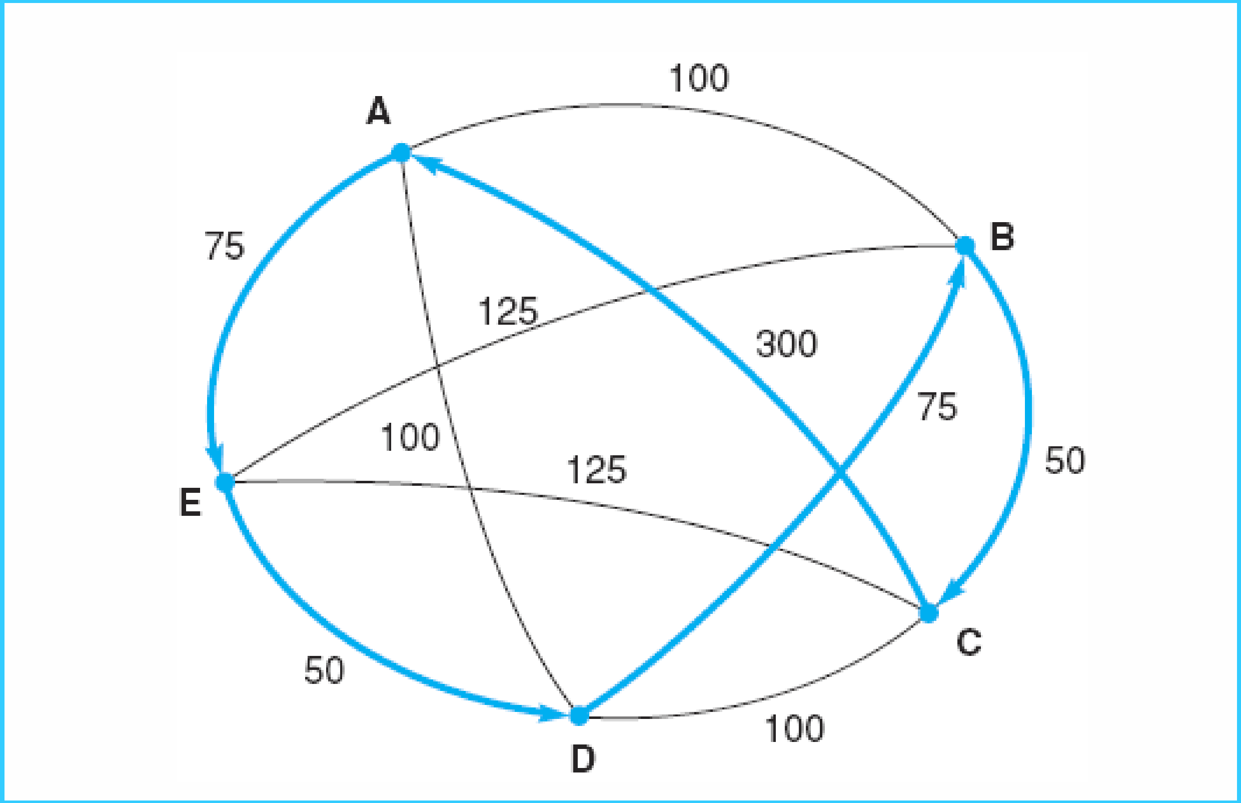


Fig 3.12 State space in which goal-directed search effectively prunes extraneous search paths.

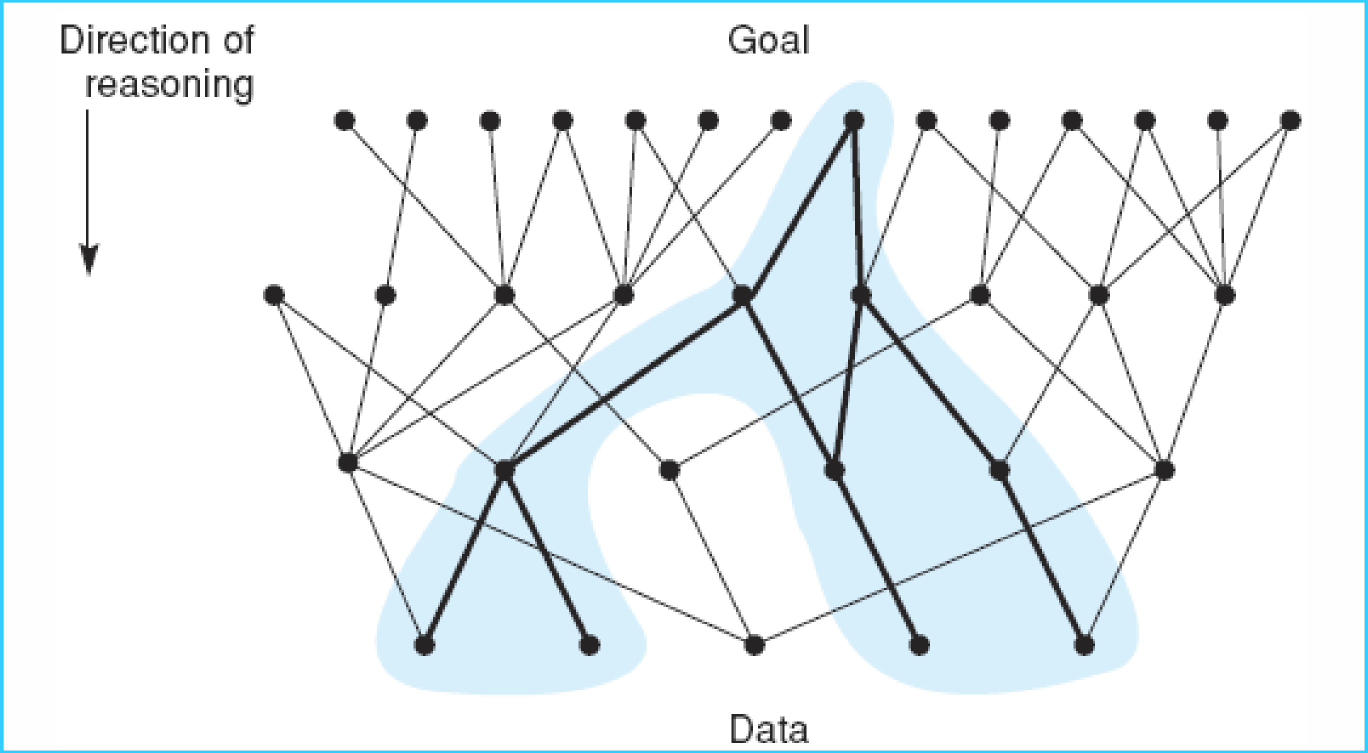
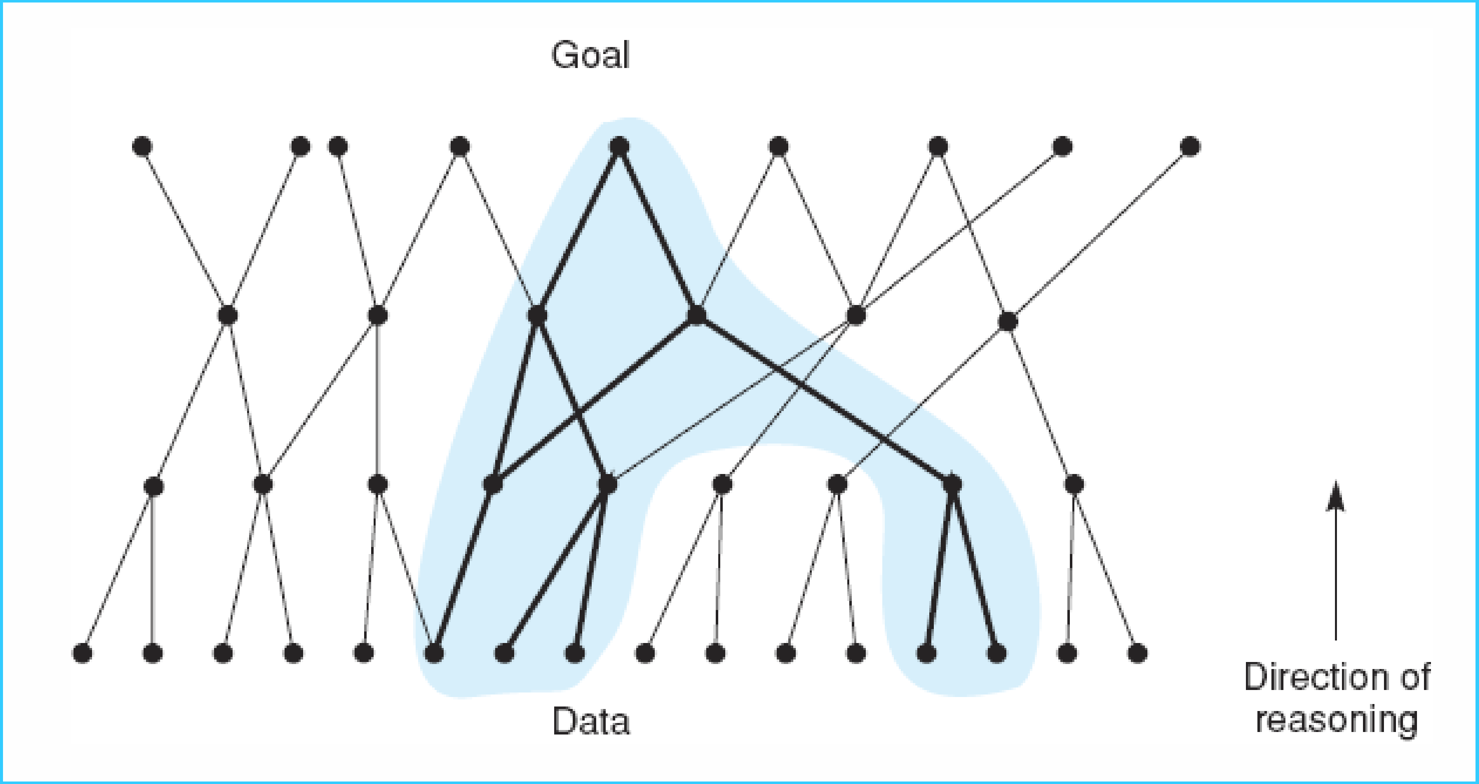


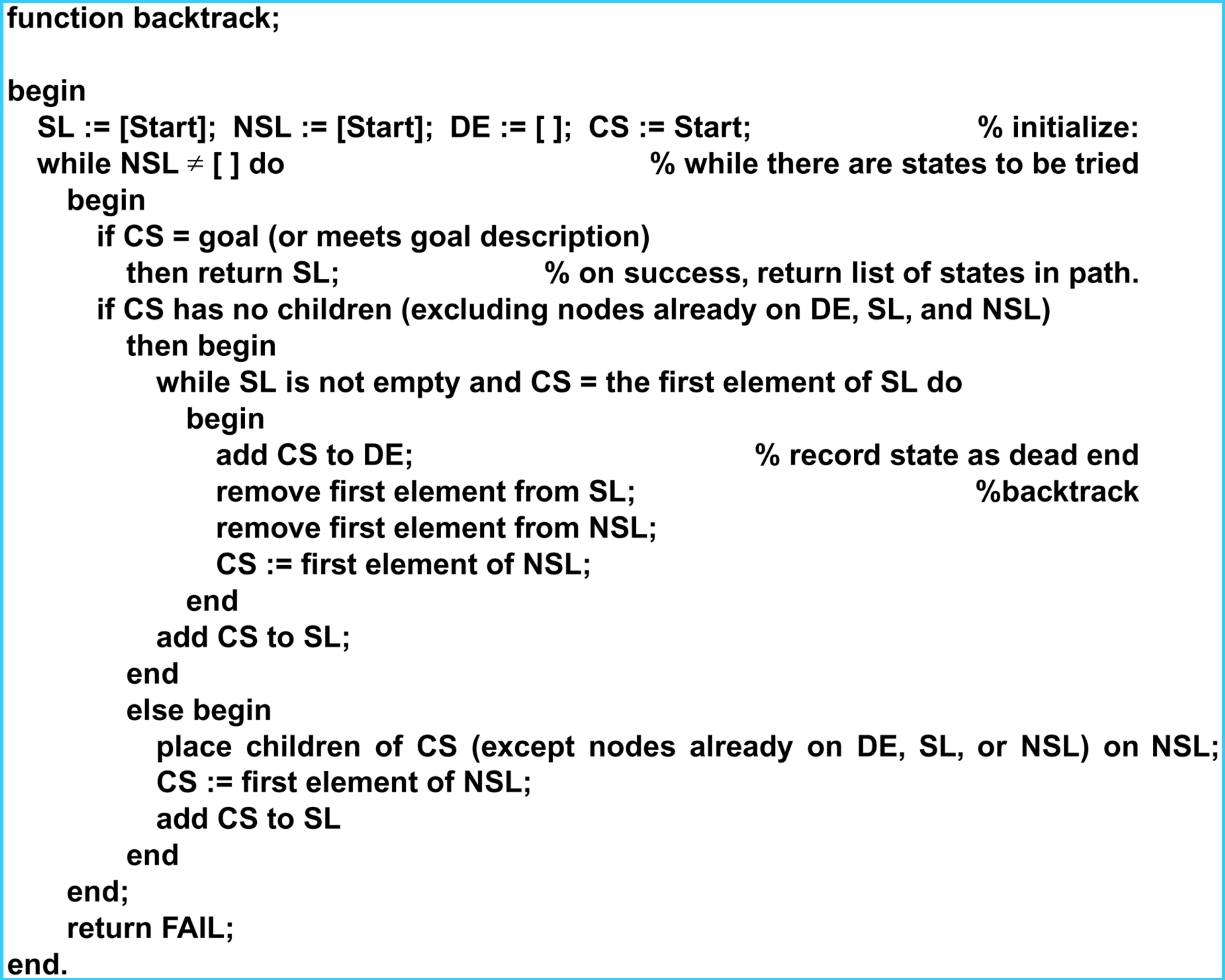
Fig 3.13 State space in which data-directed search prunes irrelevant data and their consequents and determines one of a number of possible goals.



- Data-driven and goal-driven searches search the same state space graph; but, the order and actual number of state searched can be differ.
- The preferred strategy is determined by the properties of the problem itself:
  - the complexity of the rules used for changing states
  - the shape of the state space
  - the nature and availability of the data

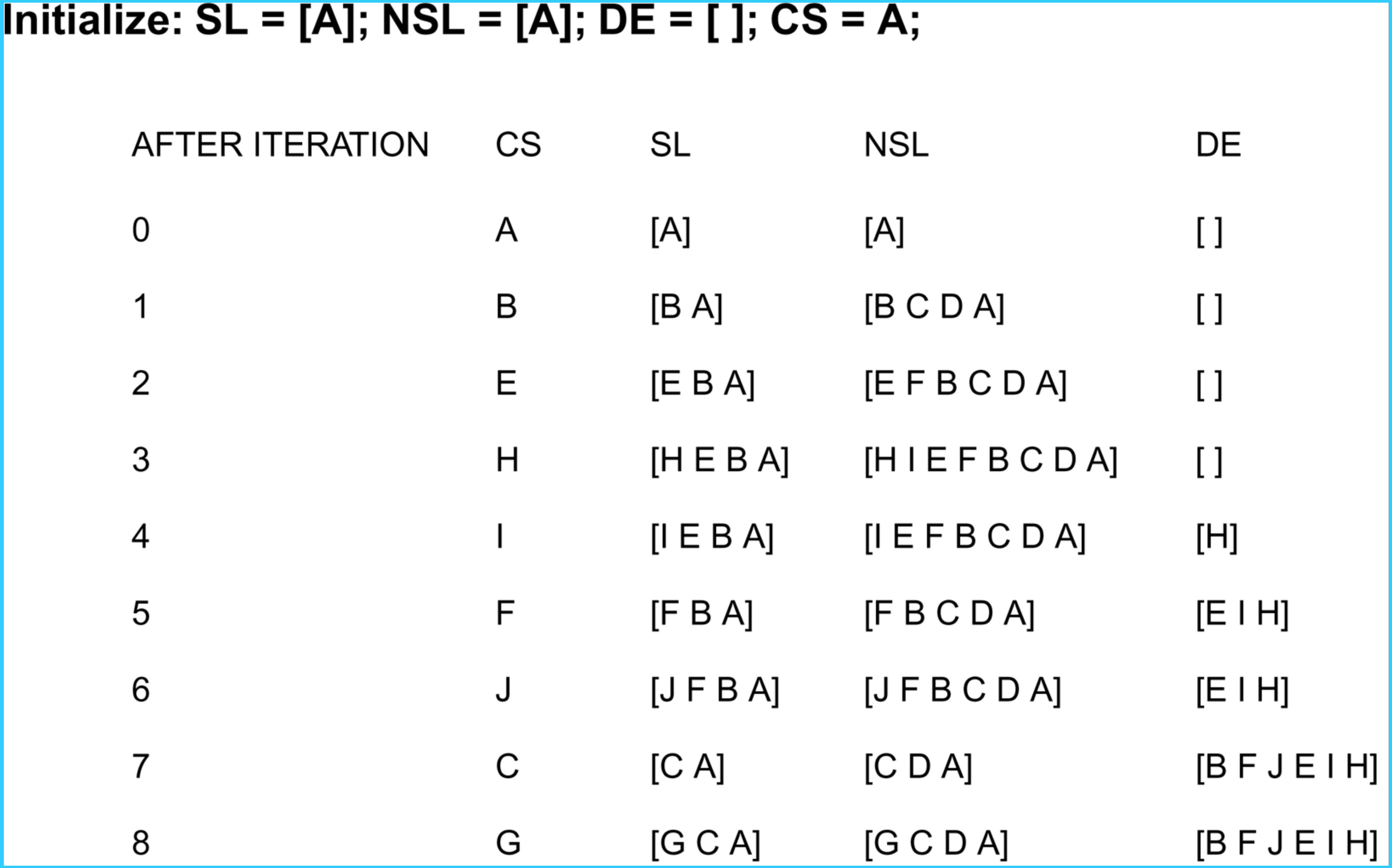
- Backtracking is a technique for systematically trying all different paths through a state space
  - For a data-driven search, it begins at the start state and pursues a path until it reaches either a goal or a "dead end".
    - If it finds a goal, it quits and returns the solution path.
    - Otherwise, it "backtracks" to the most recent node on the path having unexamined siblings and continues down one of these branches.
  - For a goal-driven search, it takes a goal be the root and evaluates descendants back in an attempt to find a start state.
  - The depth-first and breadth-first searches exploit the ideas used in backtrack.

Function backtrack algorithm



```
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.
```

A trace of backtrack on the graph of figure 3.12



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]

Fig 3.14 Backtracking search of a hypothetical state space space.

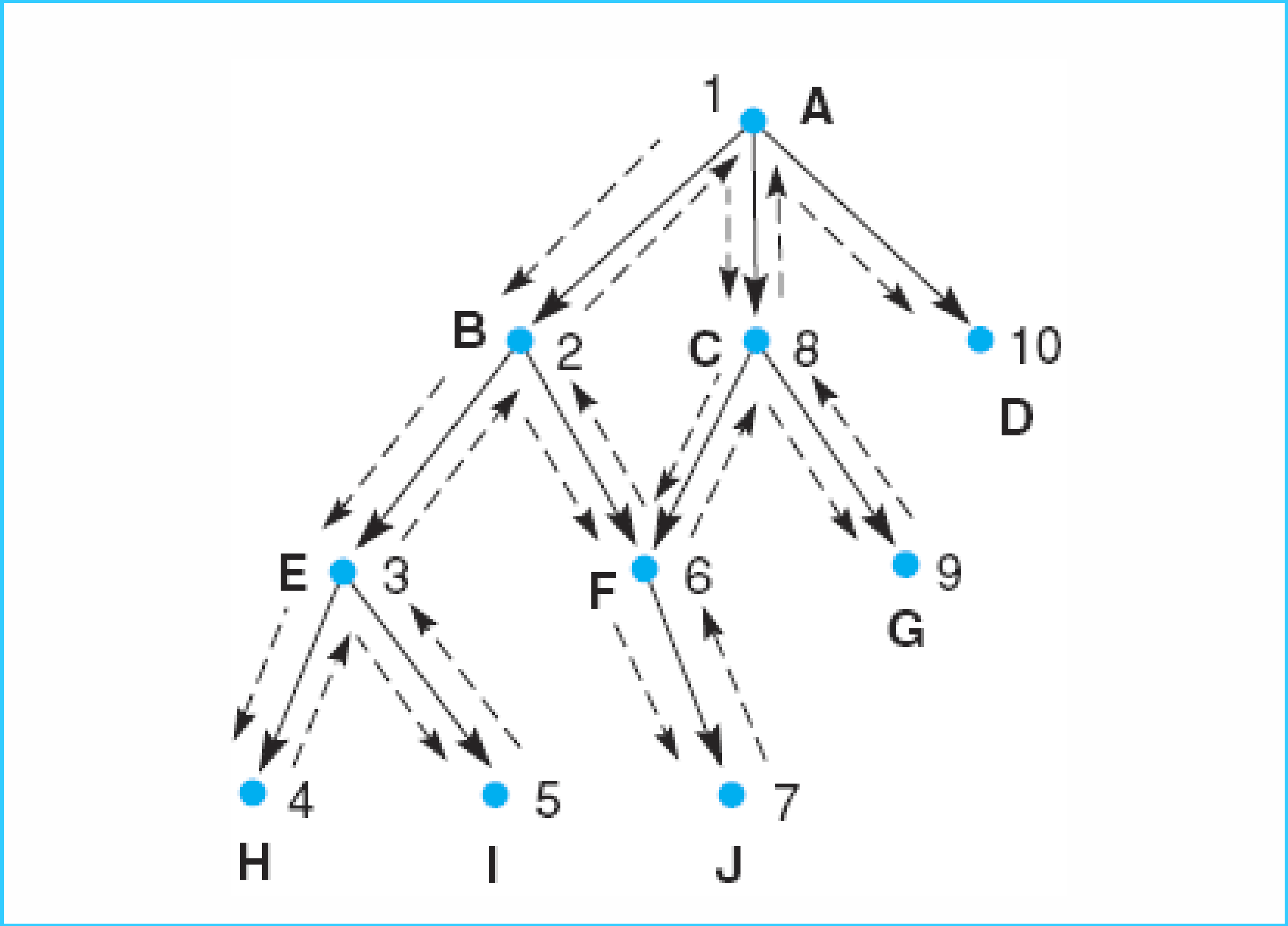
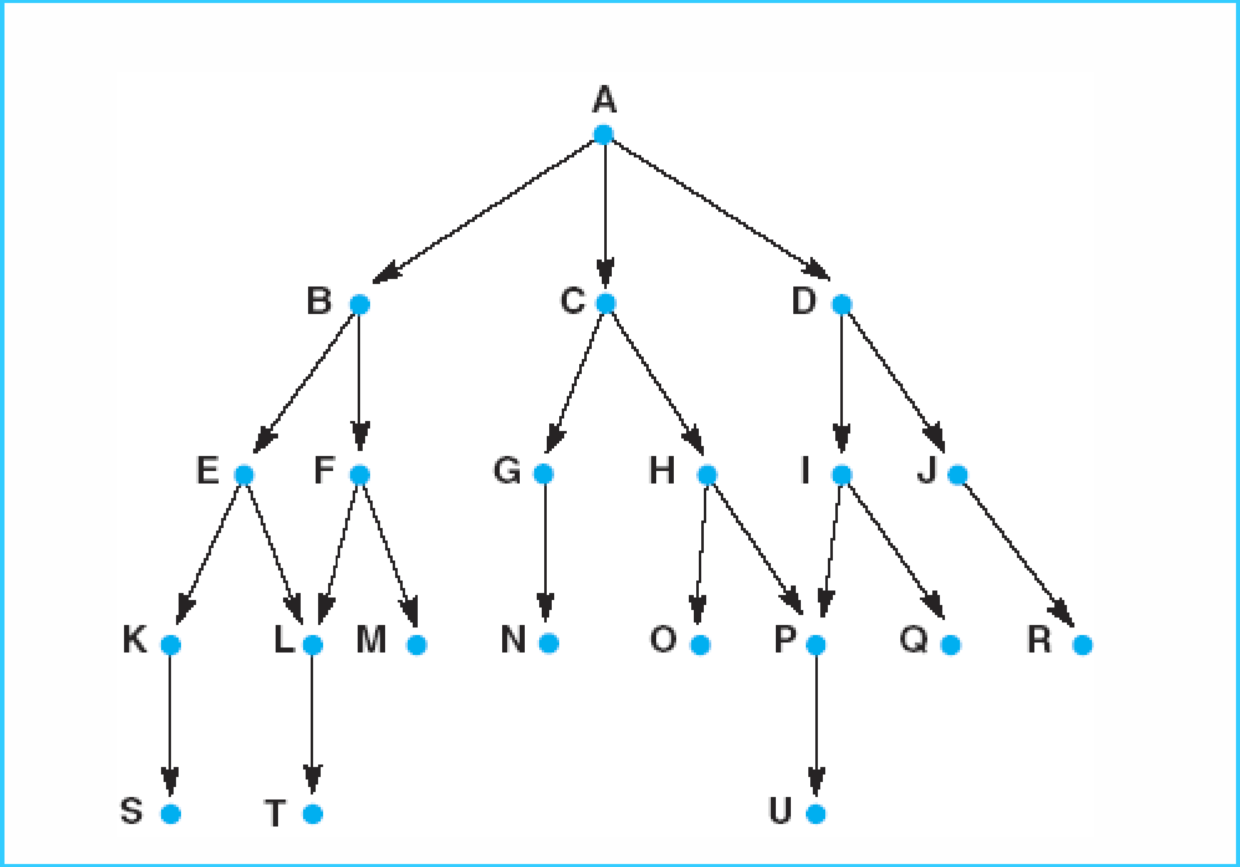
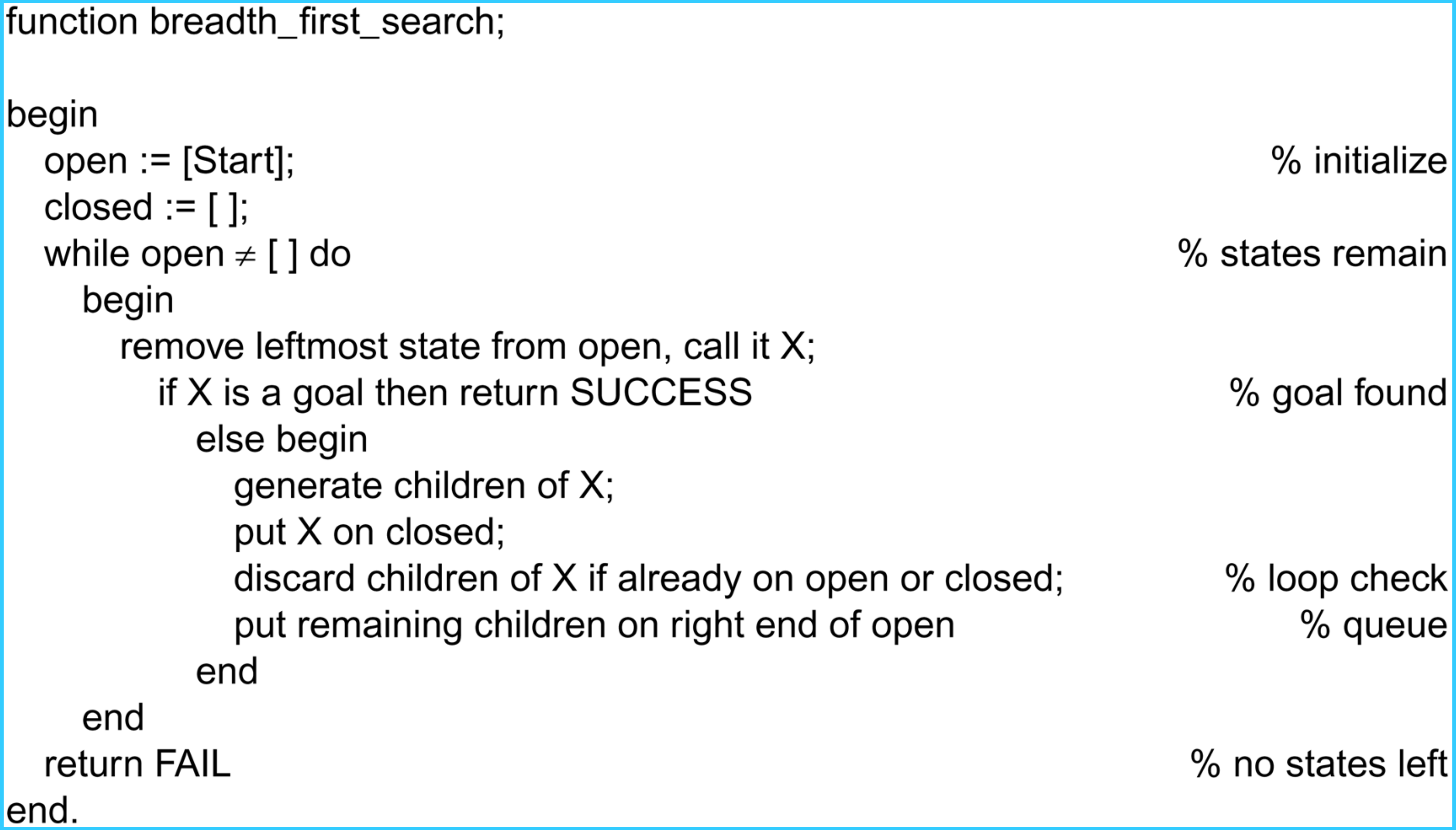


Fig 3.15 Graph for breadth - and depth - first search examples.

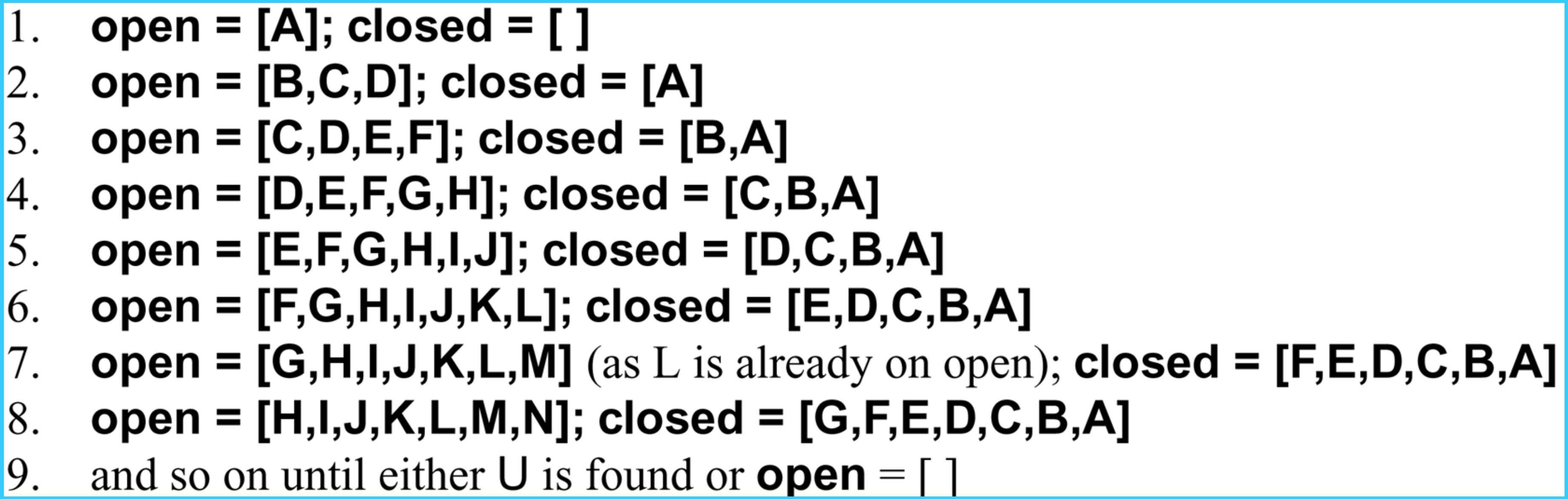


Function breadth_first search algorithm



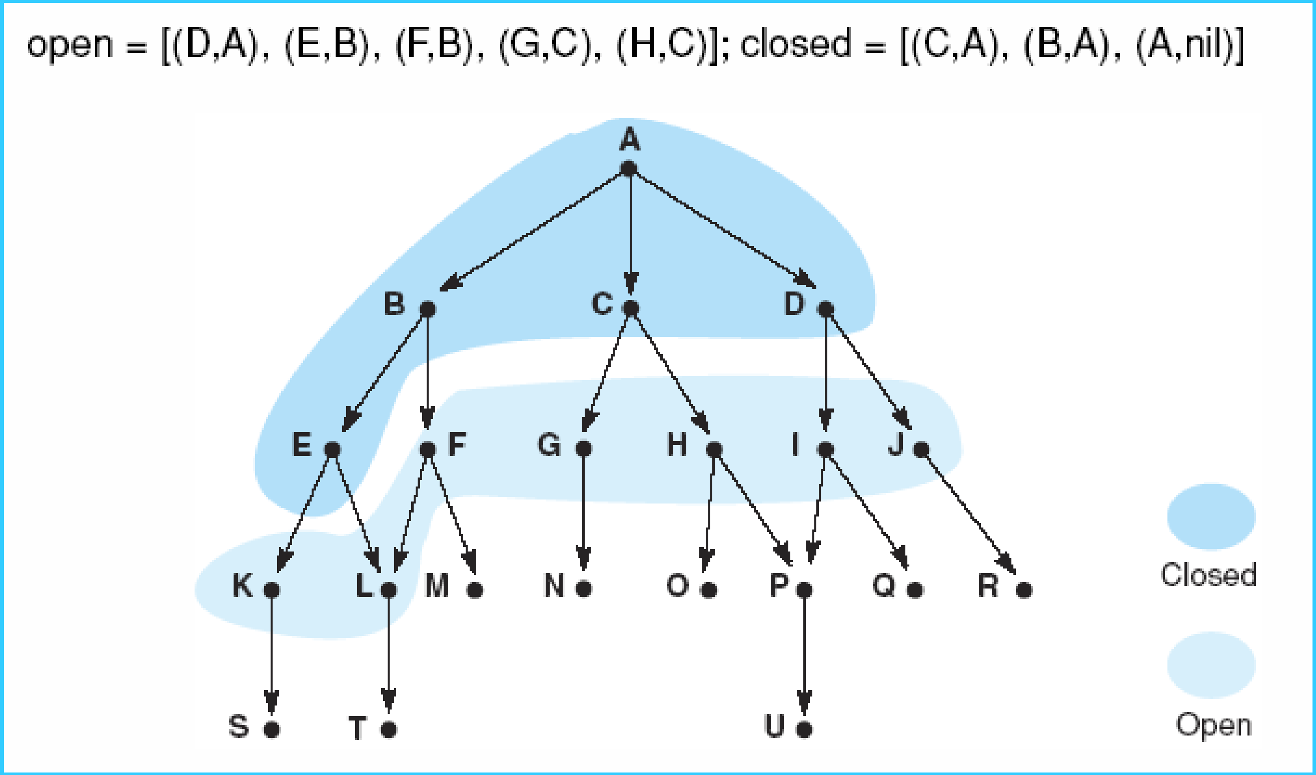
```
function breadth_first_search;

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 on the graph of Figure 3.13

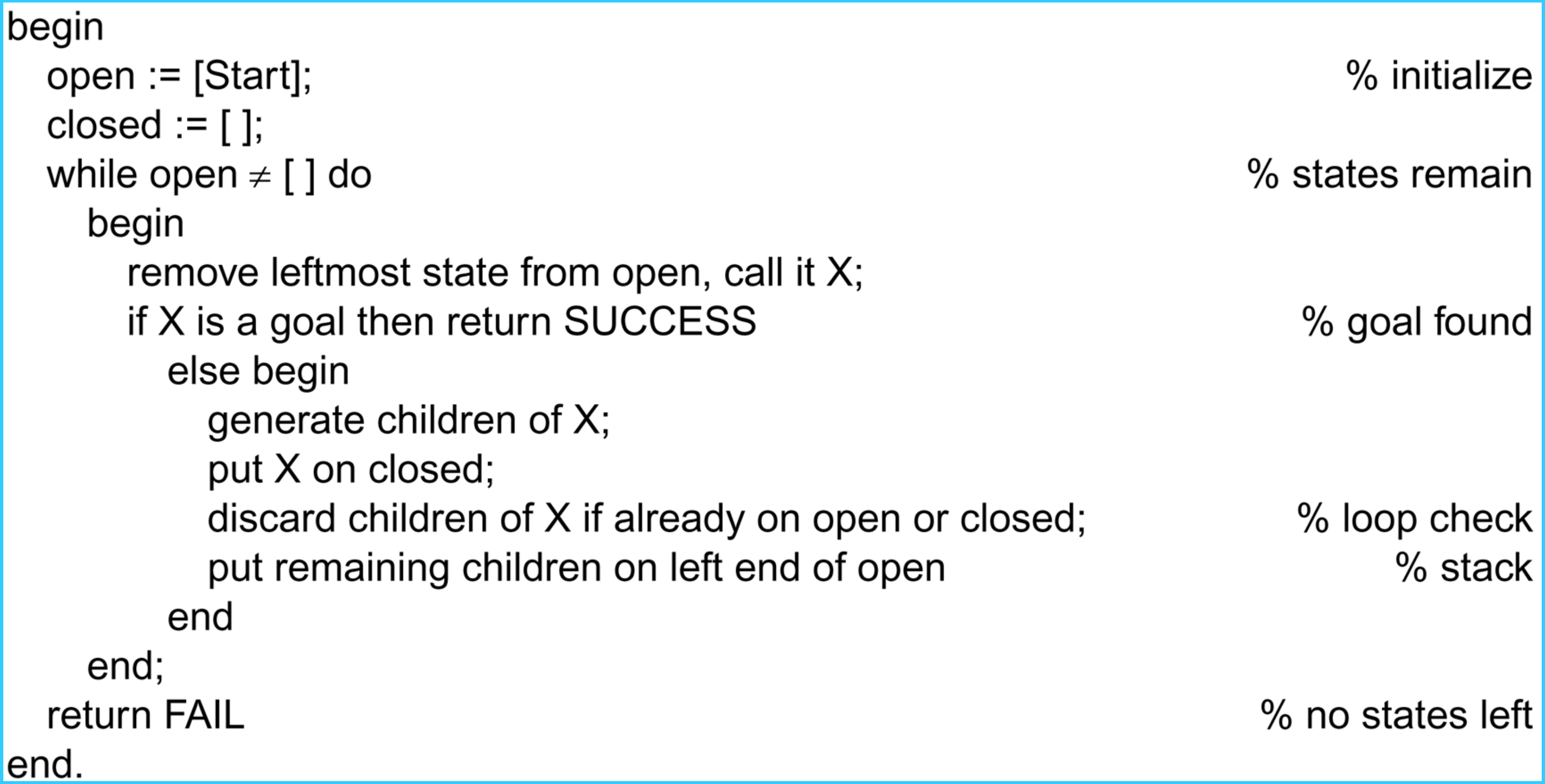


1. **open = [A]; closed = [ ]**
2. **open = [B,C,D]; closed = [A]**
3. **open = [C,D,E,F]; closed = [B,A]**
4. **open = [D,E,F,G,H]; closed = [C,B,A]**
5. **open = [E,F,G,H,I,J]; closed = [D,C,B,A]**
6. **open = [F,G,H,I,J,K,L]; closed = [E,D,C,B,A]**
7. **open = [G,H,I,J,K,L,M]** (as L is already on 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** = [ ]

Fig 3.16 Graph of Fig 3.15 at iteration 6 of breadth-first search. States on open and closed are highlighted.

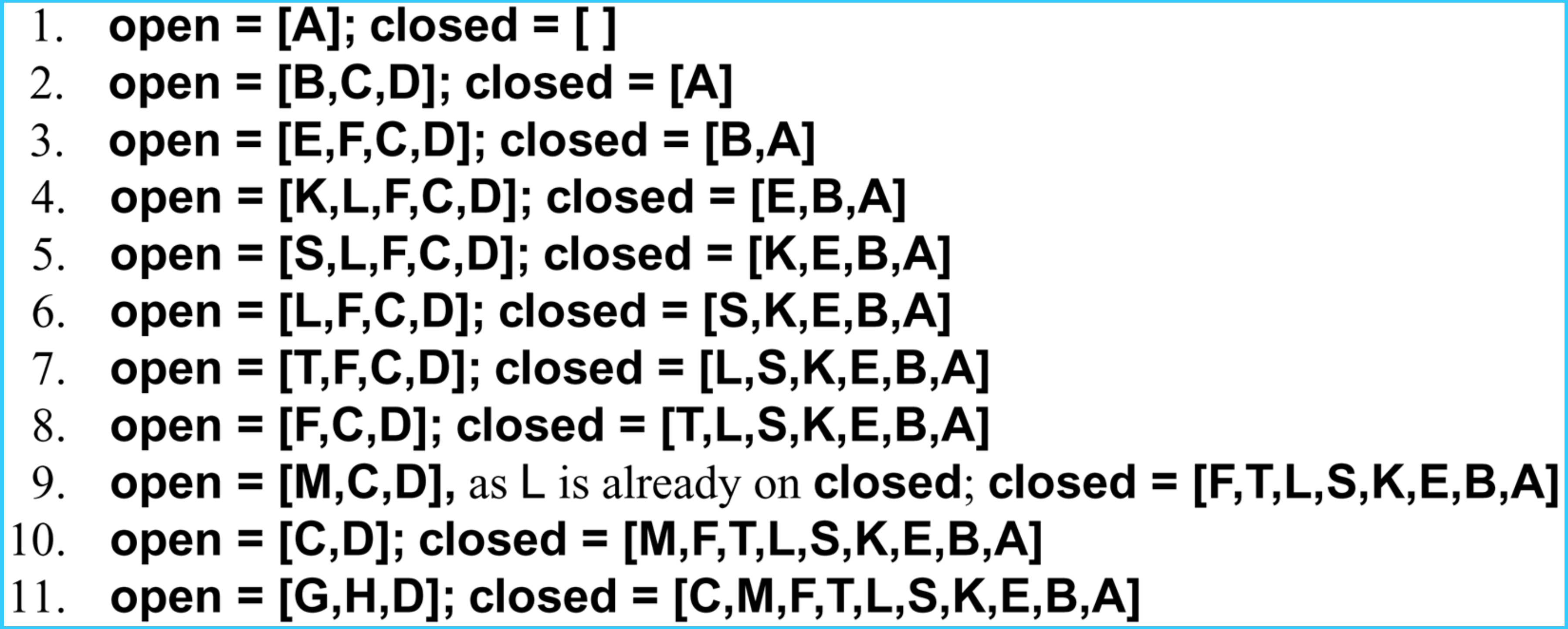

open = [(D,A), (E,B), (F,B), (G,C), (H,C)]; closed = [(C,A), (B,A), (A,nil)]
A
B
C
D
E
F
G
H
I
J
K
L
M
N
O
P
Q
R
S
T
U
Closed
Open

Function 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 on the graph of Figure 3.13



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]**

Fig 3.17 Breadth-first search of the 8-puzzle, showing order in which states were removed from open.

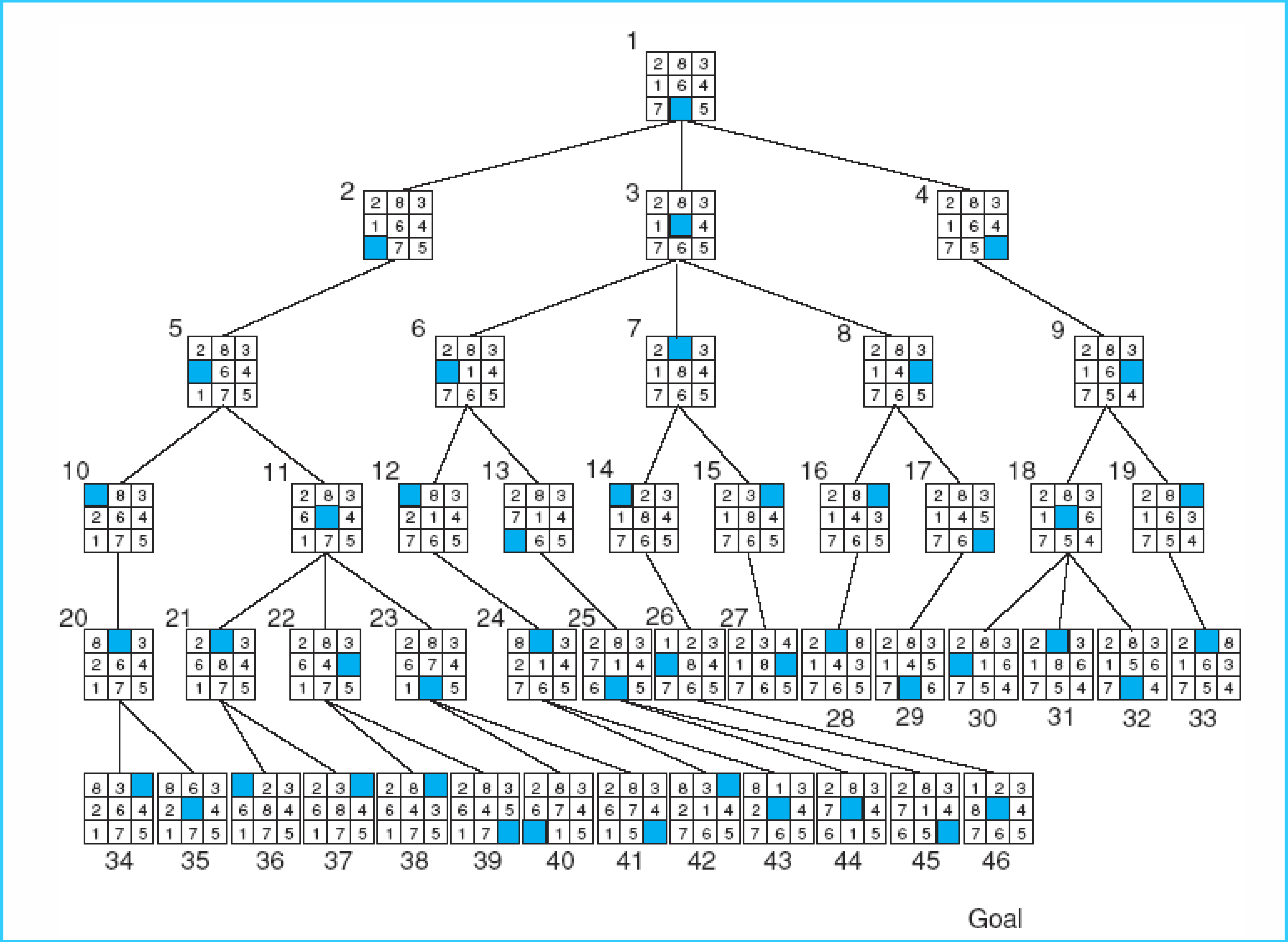


Fig 3.18 Graph of fig 3.15 at iteration 6 of depth-first search. States on open and closed are highlighted.

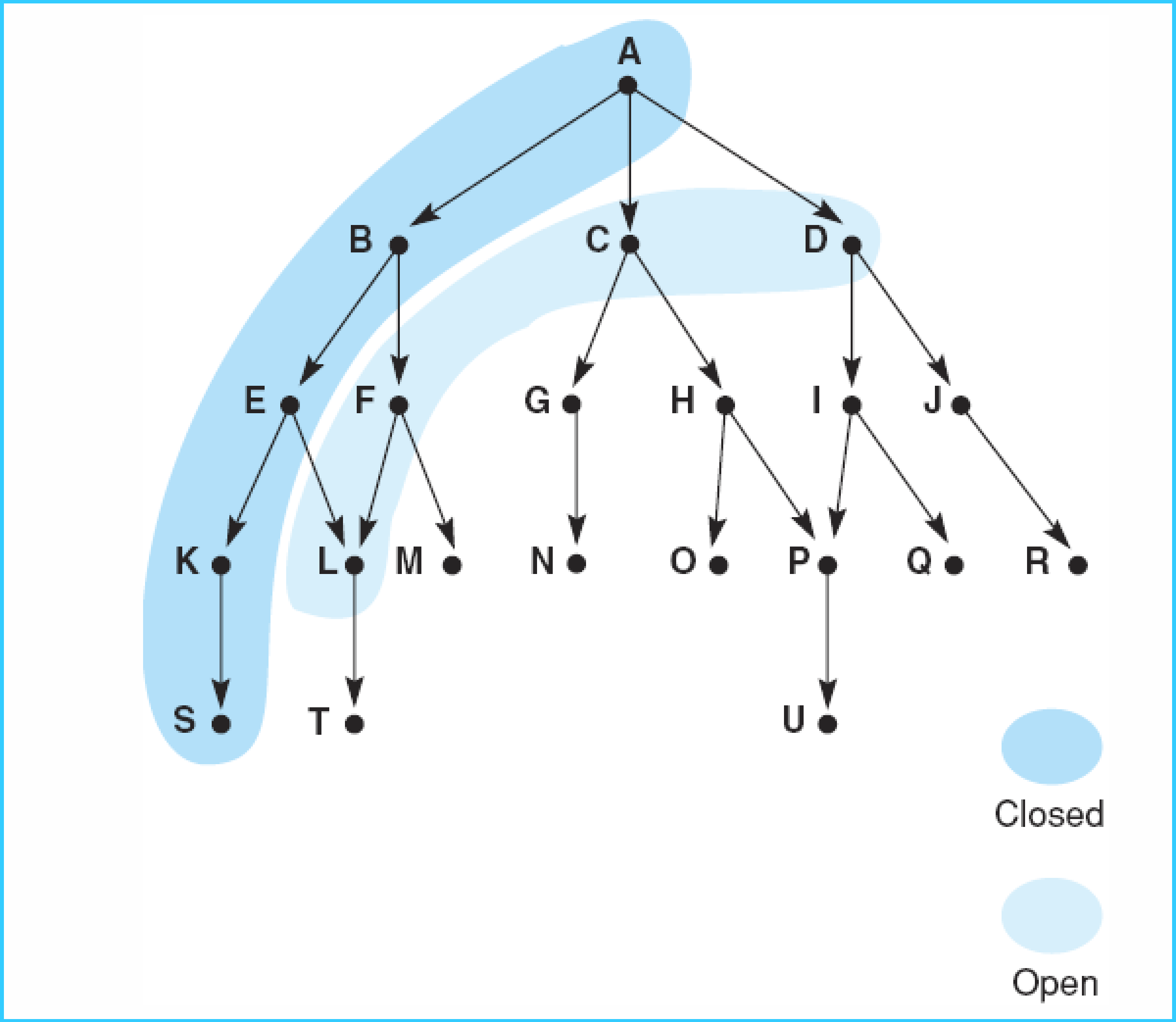
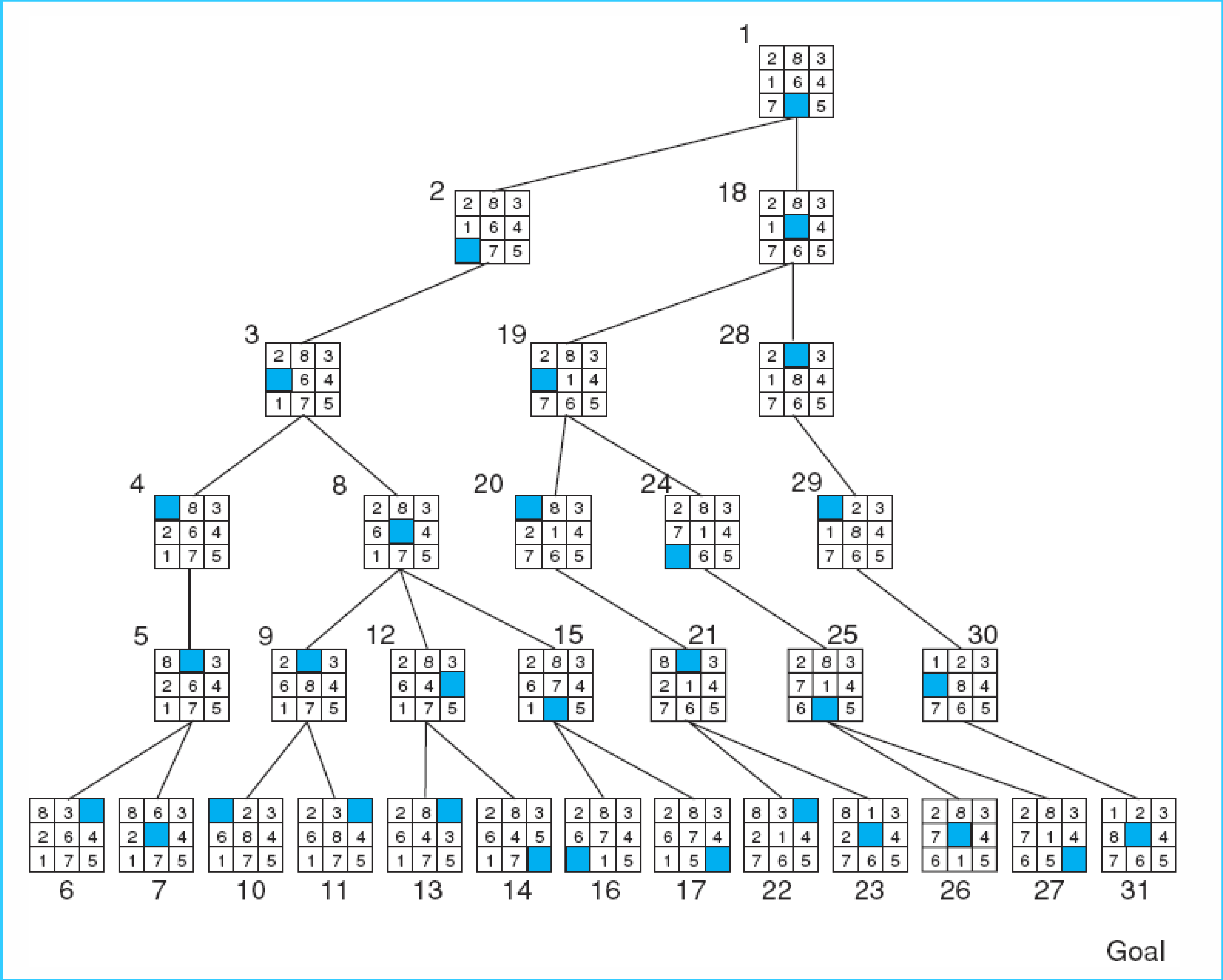


Fig 3.19 Depth-first search of the 8-puzzle with a depth bound of 5.



- Besides the names of states, other information may be stored in OPEN and CLOSE lists
  – In breadth-first search, if the path is required for a solution, the ancestor information needs to be stored with each state.
  – In depth-first search, if path length matters in a problem solver, each state should be stored as a triple (state, parent, length-of-path).

The choice of depth-first or breadth-first search depends on the problem

- the importance of finding the optimal path to a goal
- the branching of the state space
- the available time and space resources
- the average length of paths to a goal node

A trade-off – Depth-first search with a depth bound

- The depth bound forces a failure on a search path once it gets below a certain level. This causes a breadth like sweep of the space at that depth level.
- It is preferred when time constraints exist or a solution is known within a certain depth.
- It is guaranteed to find a shortest path to a goal.
- Depth-first iterative deepening (Korf 1987) -- depth bound increases one at each iteration

- For all uniformed search algorithms, the worst-case time complexity is exponential.