3 STRUCTURES AND STRATEGIES FOR STATE SPACE SEARCH

Figure 3.1: The city of Königsberg.

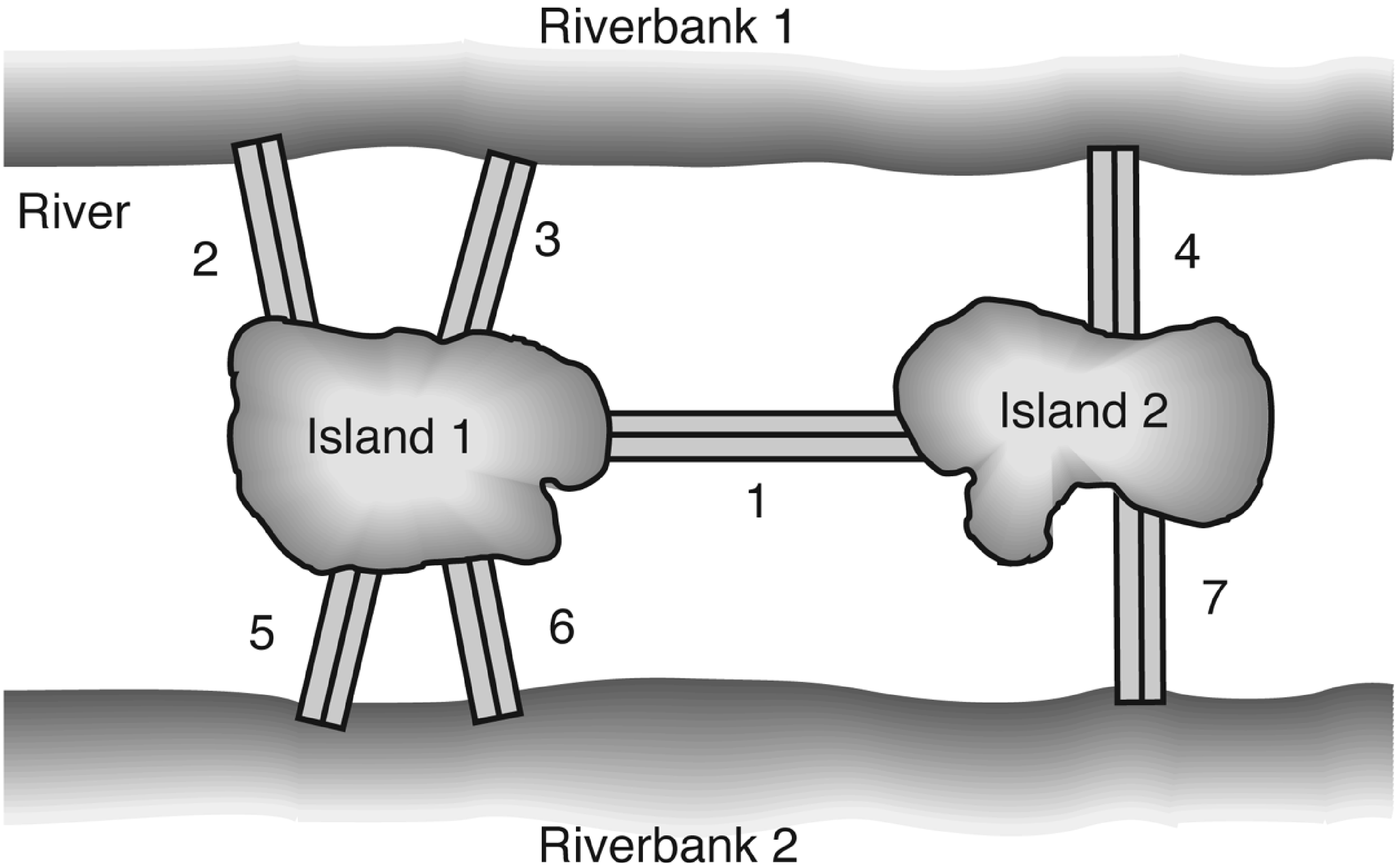


Figure 3.2: Graph of the Königsberg bridge system.

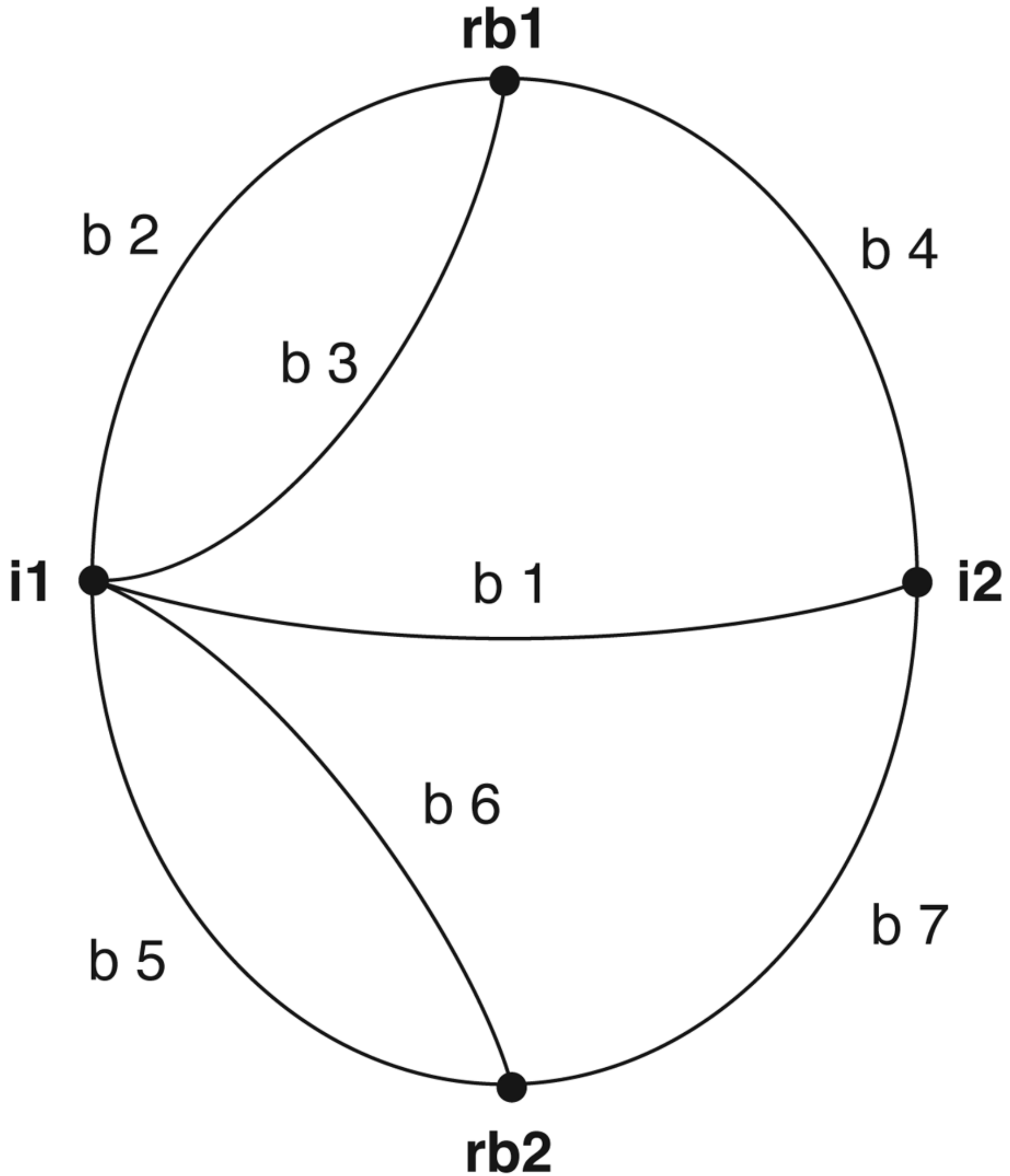
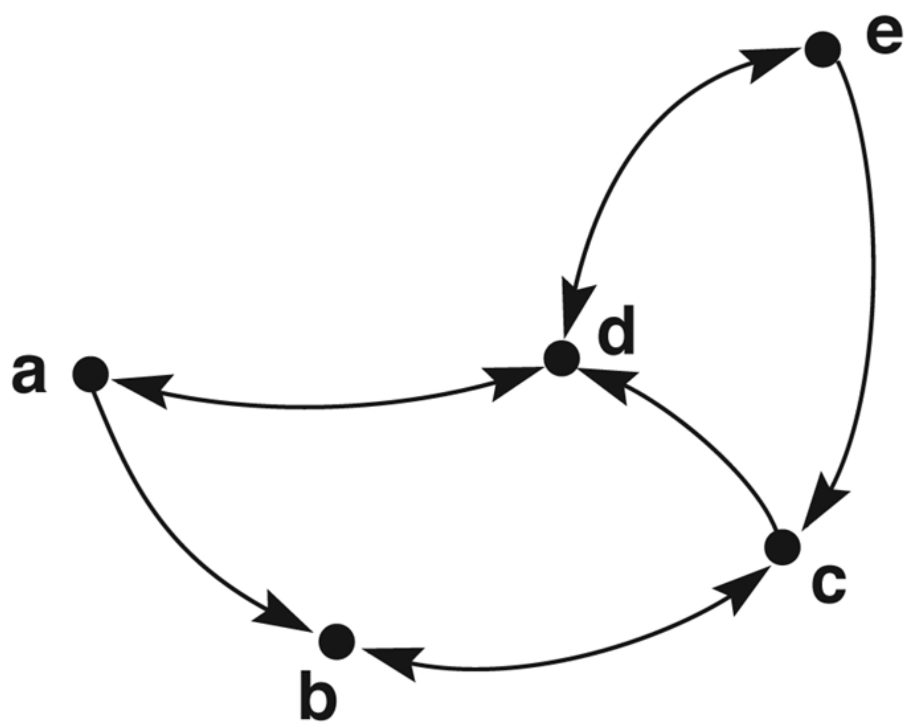


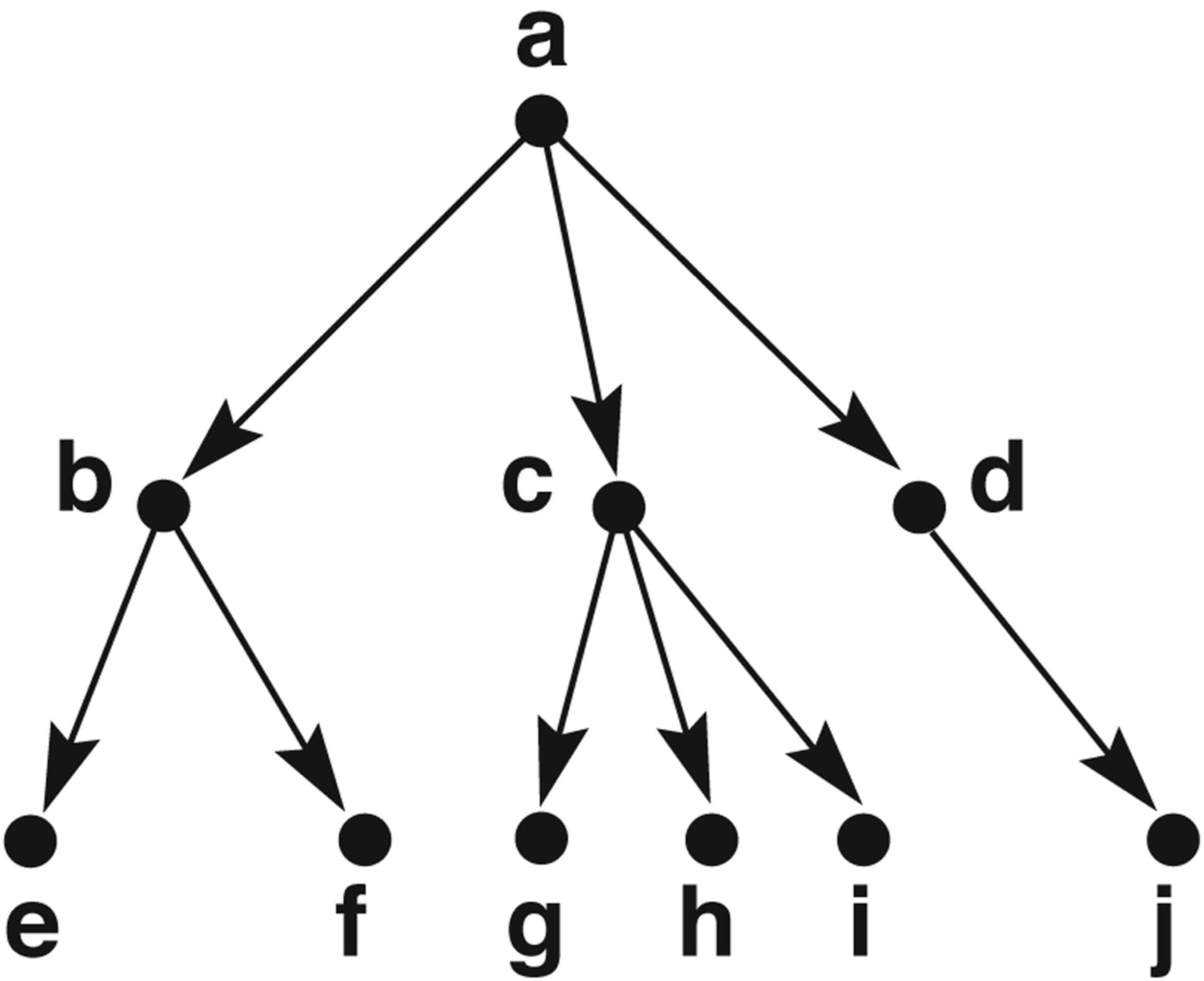
Figure 3.3: A labeled directed graph.



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.



DEFINITION

GRAPH

A graph consists of:

A set of *nodes* $\mathbf{N}_1$, $\mathbf{N}_2$, $\mathbf{N}_3$, … $\mathbf{N}_n$ …, which need not be finite.

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

Arcs are ordered pairs of nodes; i.e., the arc ($\mathbf{N}_3$, $\mathbf{N}_4$) connects node $\mathbf{N}_3$ to node $\mathbf{N}_4$. This would indicate a direct connection from node $\mathbf{N}_3$ to $\mathbf{N}_4$ but not from $\mathbf{N}_4$ to $\mathbf{N}_3$, unless ($\mathbf{N}_4$, $\mathbf{N}_3$) is also an arc, in which case the arc joining $\mathbf{N}_3$ and $\mathbf{N}_4$ is undirected.

If a directed arc connects $\mathbf{N}_j$ and $\mathbf{N}_k$, then $\mathbf{N}_j$ is called the *parent* of $\mathbf{N}_k$ and $\mathbf{N}_k$, the *child* of $\mathbf{N}_j$. If the graph also contains an arc ($\mathbf{N}_j$, $\mathbf{N}_l$), then $\mathbf{N}_k$ and $\mathbf{N}_l$ are called *siblings*.

A *rooted* graph has a unique node $\mathbf{N}_S$ 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 [$\mathbf{N}_1$, $\mathbf{N}_2$, $\mathbf{N}_3$, …, $\mathbf{N}_n$], where each pair $\mathbf{N}_i$, $\mathbf{N}_{i+1}$ in the sequence represents an arc, i.e., ($\mathbf{N}_i$, $\mathbf{N}_{i+1}$), is called a *path* of length $\mathbf{n - 1}$ in the graph.

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 (to its left).

A path that contains any node more than once (some $\mathbf{N}_j$ 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.

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**.

Figure 3.6: State space of the 8-puzzle generated by "move blank" operations.

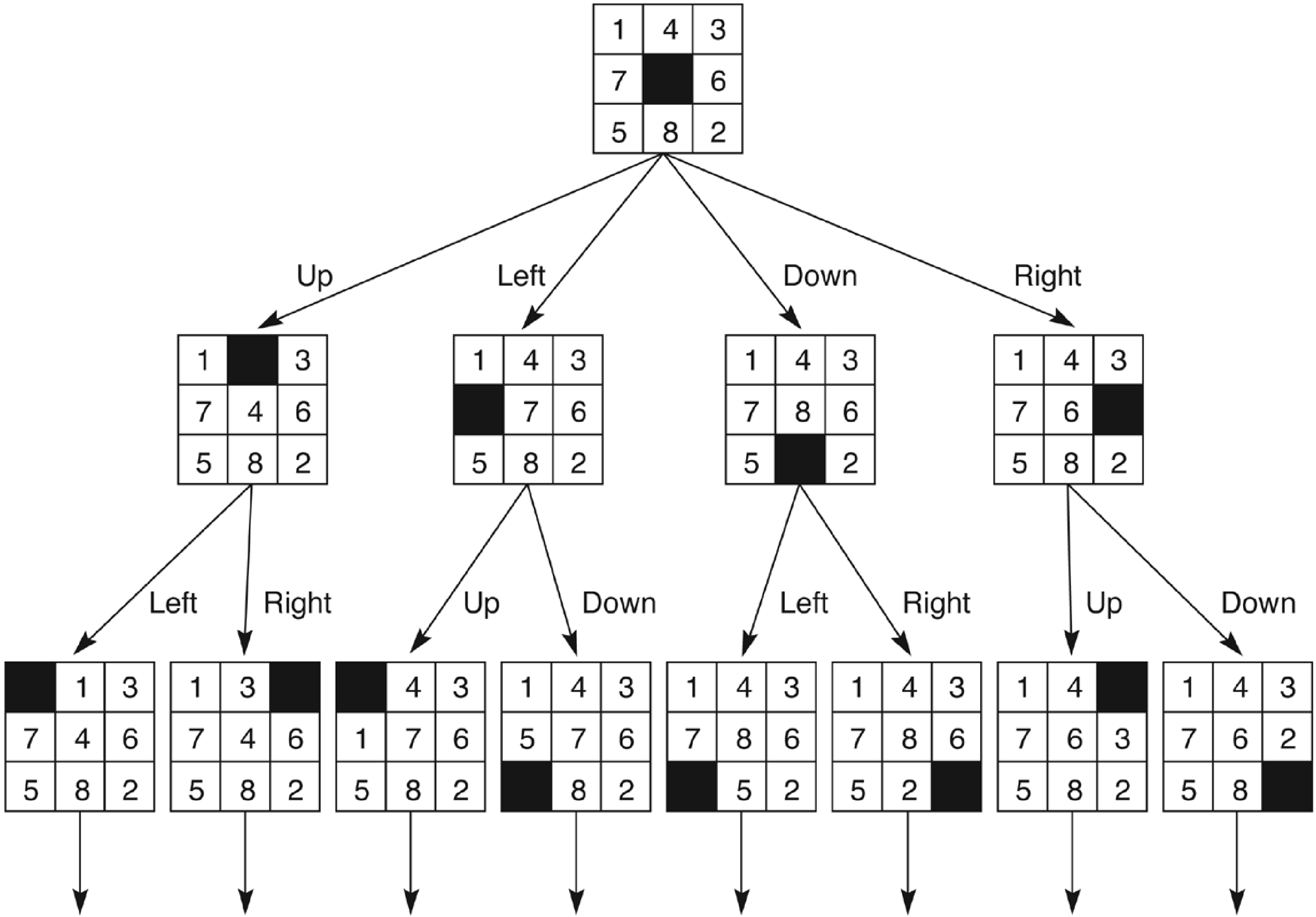


Figure 3.7: An instance of the traveling salesperson problem.

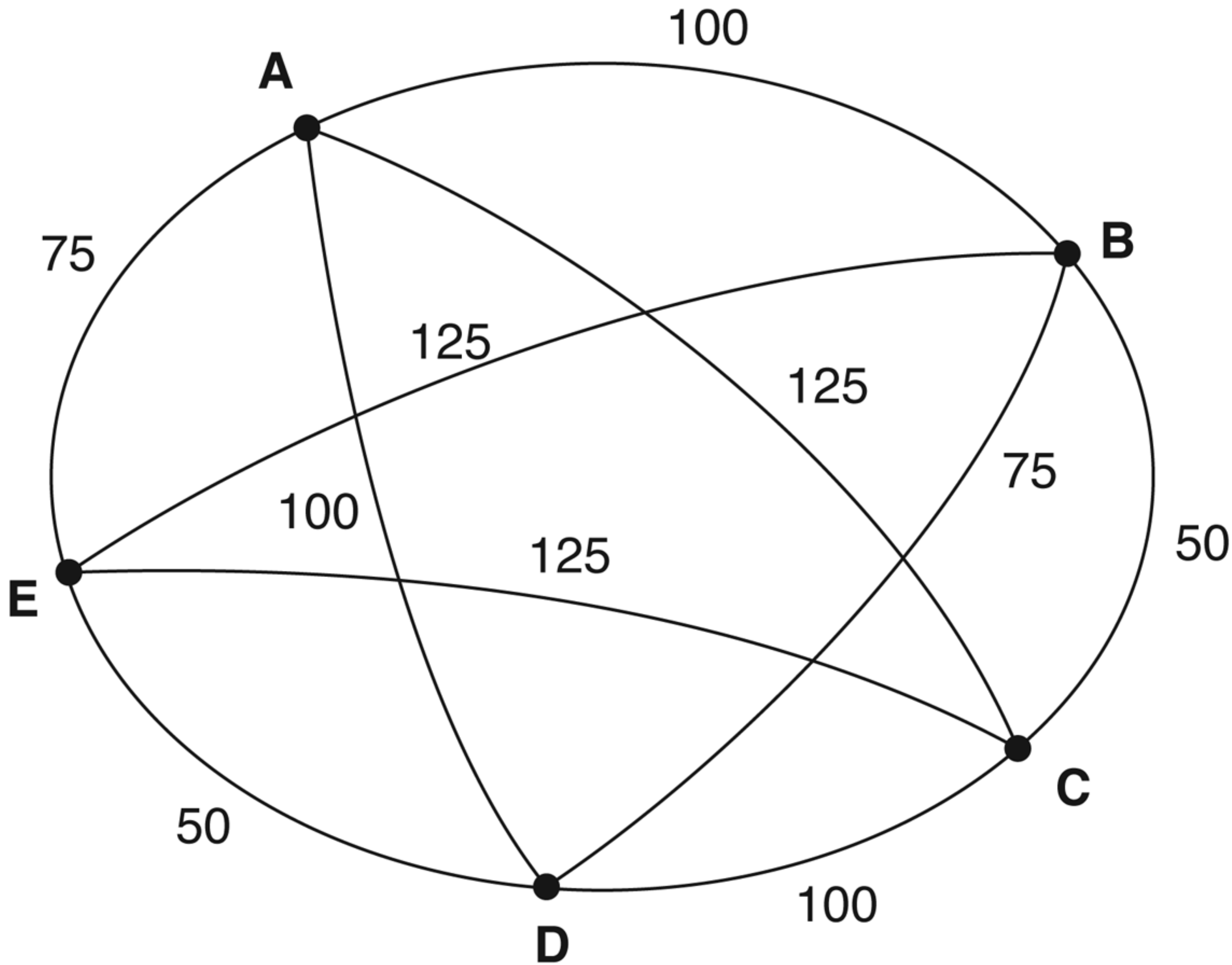


Figure 3.8: Search of the traveling salesperson problem. Each arc is marked with the total weight of all paths from the start node (A) to its endpoint.

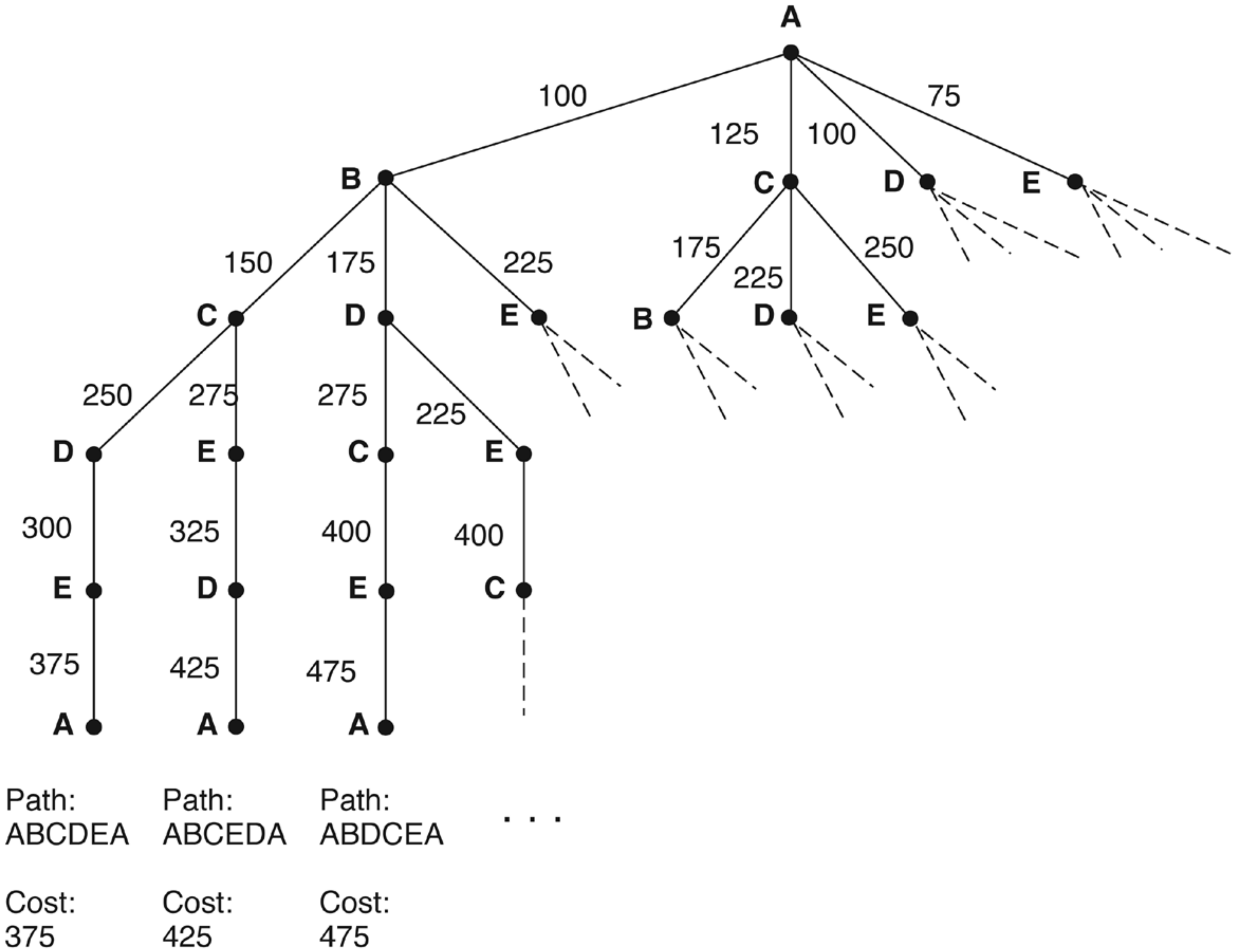


Figure 3.9: An instance of the traveling salesperson problem with the nearest neighbor path in bold. Note that this path (A, E, D, B, C, A), at a cost of 550, is not the shortest path. The comparatively high cost of arc (C, A) defeated the heuristic.

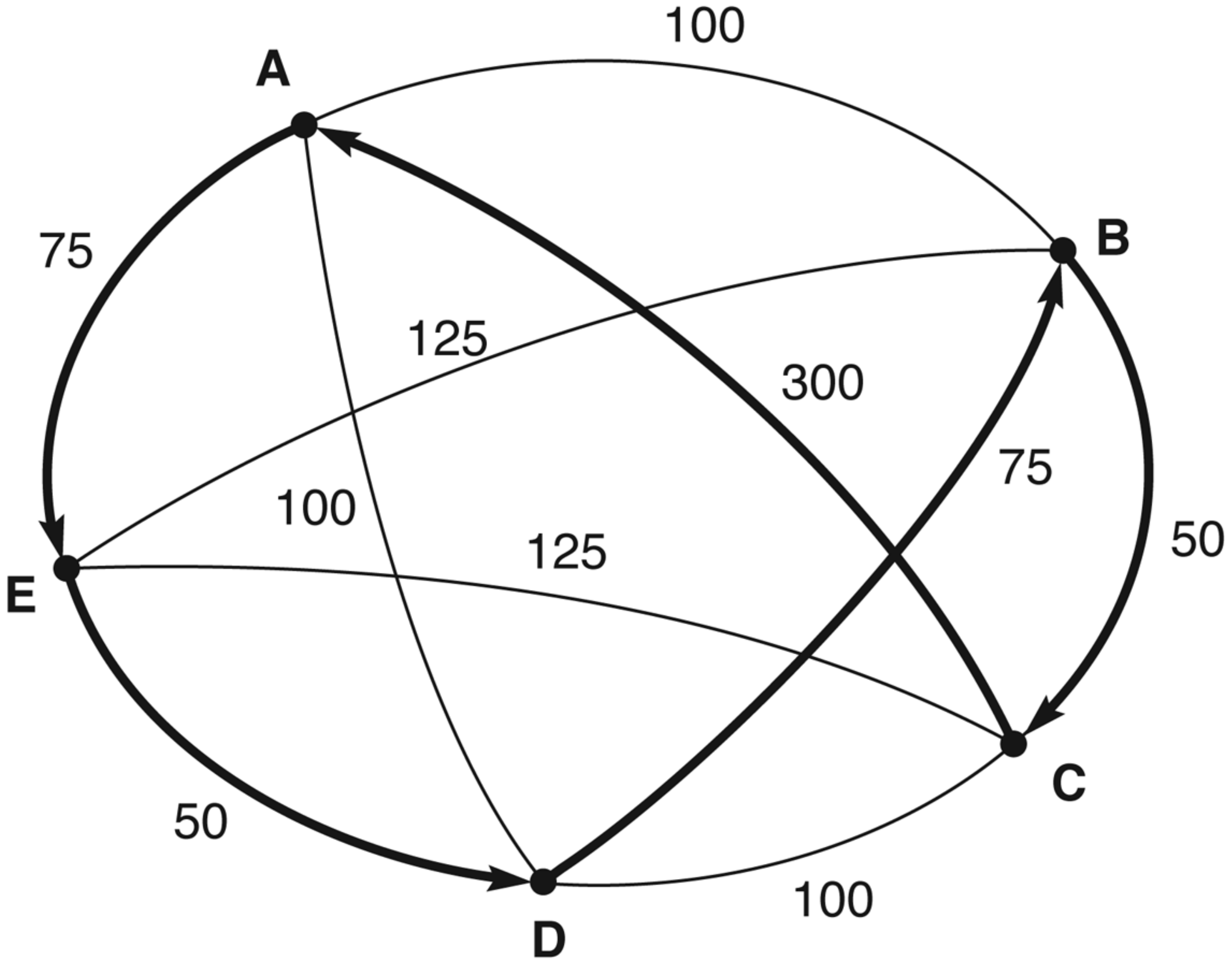


Figure 3.10: State space in which goal-directed search effectively prunes extraneous search paths.

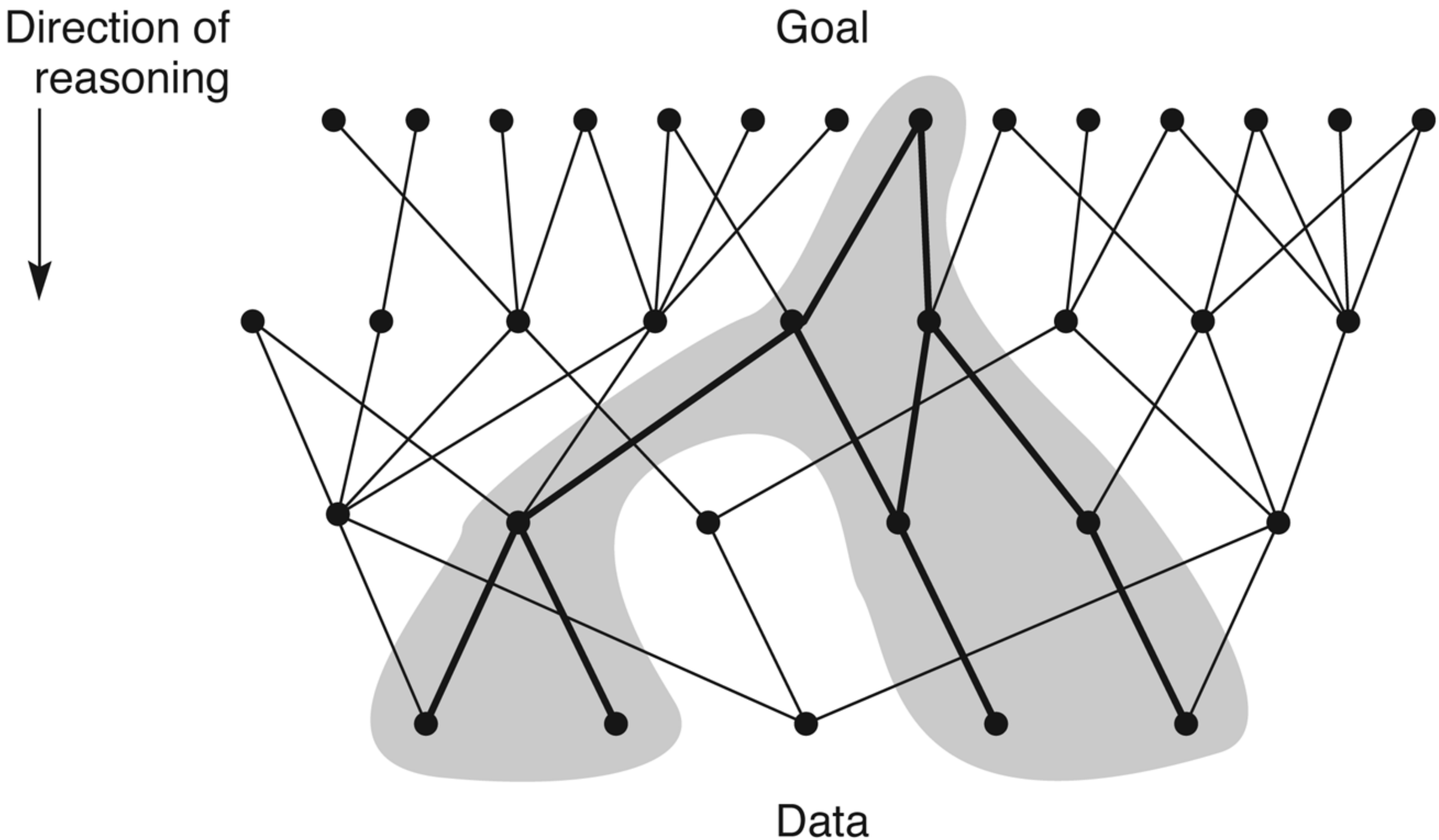
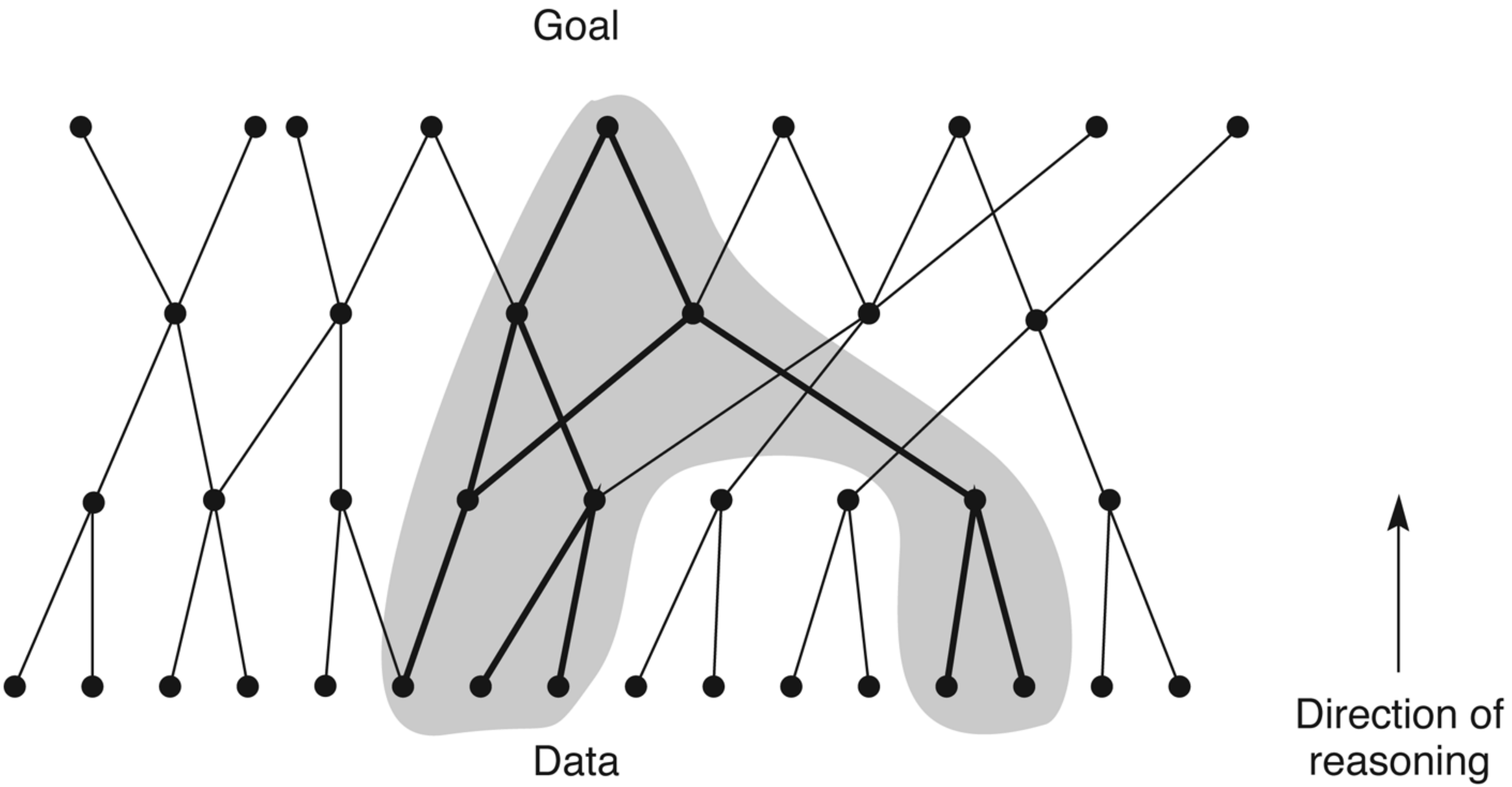


Figure 3.11: State space in which data-directed search prunes irrelevant data and their consequents and determines one of a number of possible goals.



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]

Figure 3.12: Backtracking search of a hypothetical state space.

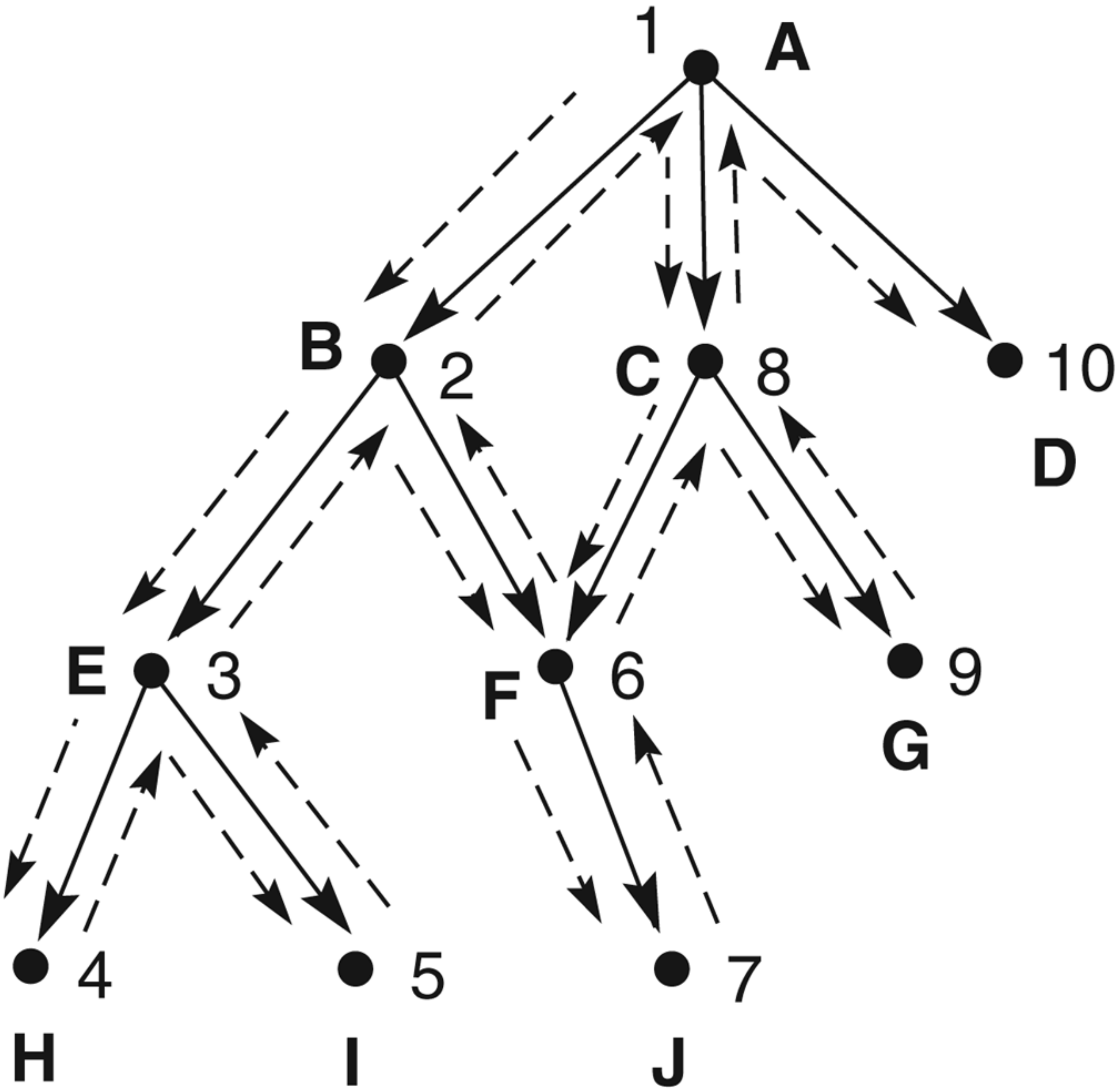
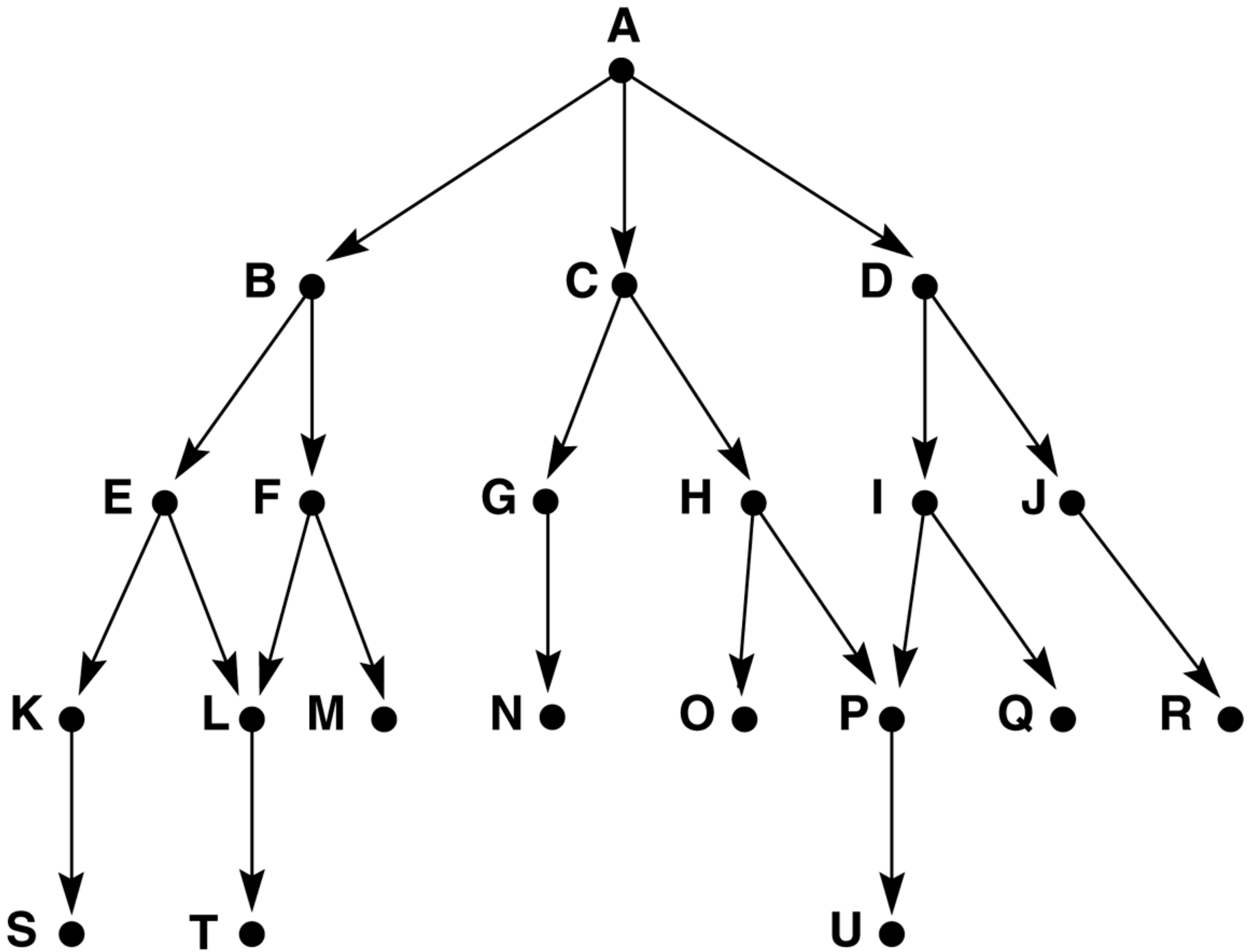


Figure 3.13: Graph for breadth- and depth-first search examples.



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** = [ ]

