3 Structures and Strategies for State Space Search

Additional references for the slides:

Russell and Norvig's AI book.

Robert Wilensky's CS188 slides:
www.cs.berkeley.edu/~wilensky/cs188/lectures/index.html

Chapter Objectives

- **Learn the basics of state space representation**
- **Learn the basics of search in state space**
- **The agent model: Has a problem, searches for a solution.**

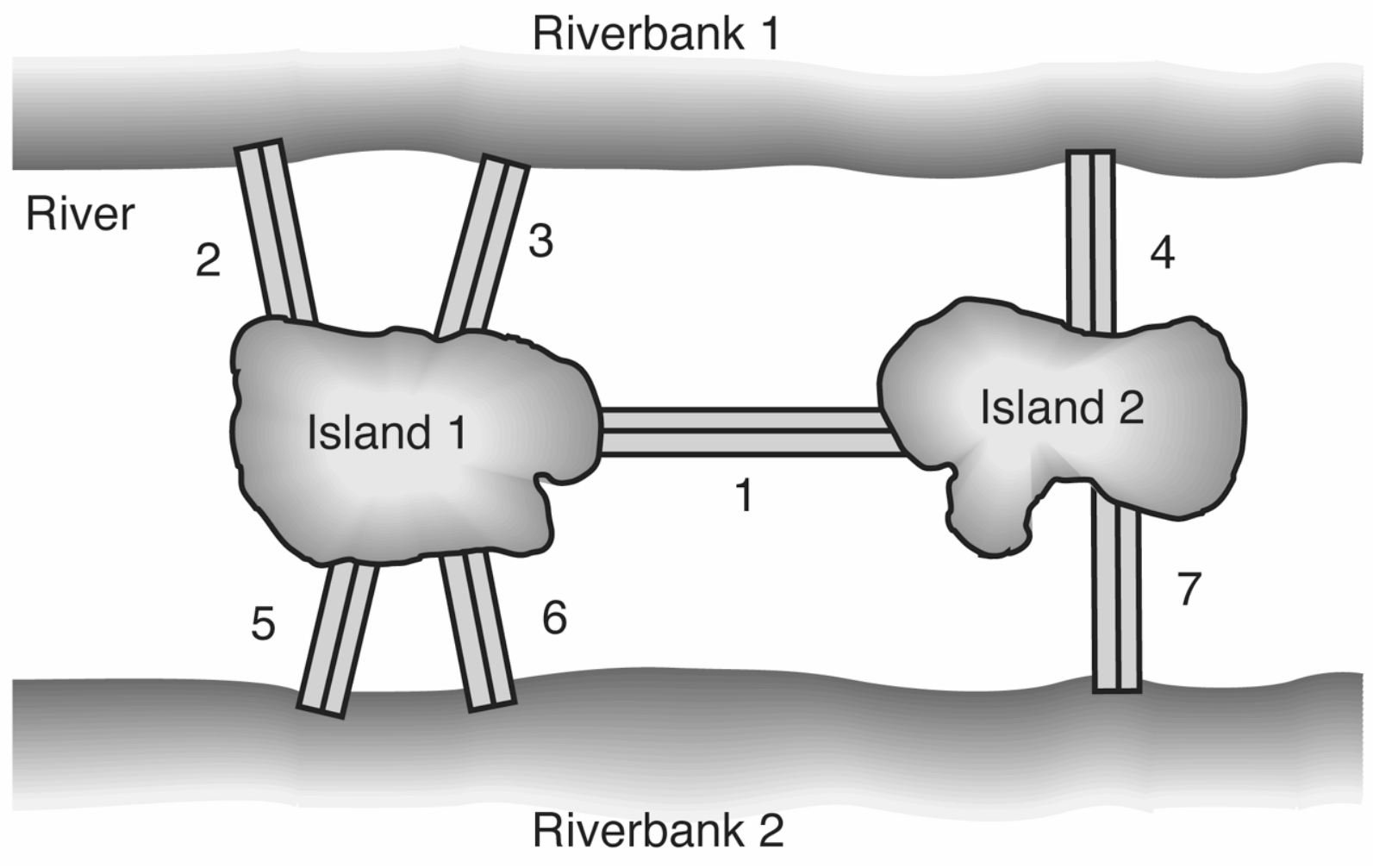
Graph theory: The city of Königsberg

The city is divided by a river. There are two islands at the river. The first island is connected by two bridges to both riverbanks and is also connected by a bridge to the other island. The second island two bridges each connecting to one riverbank.

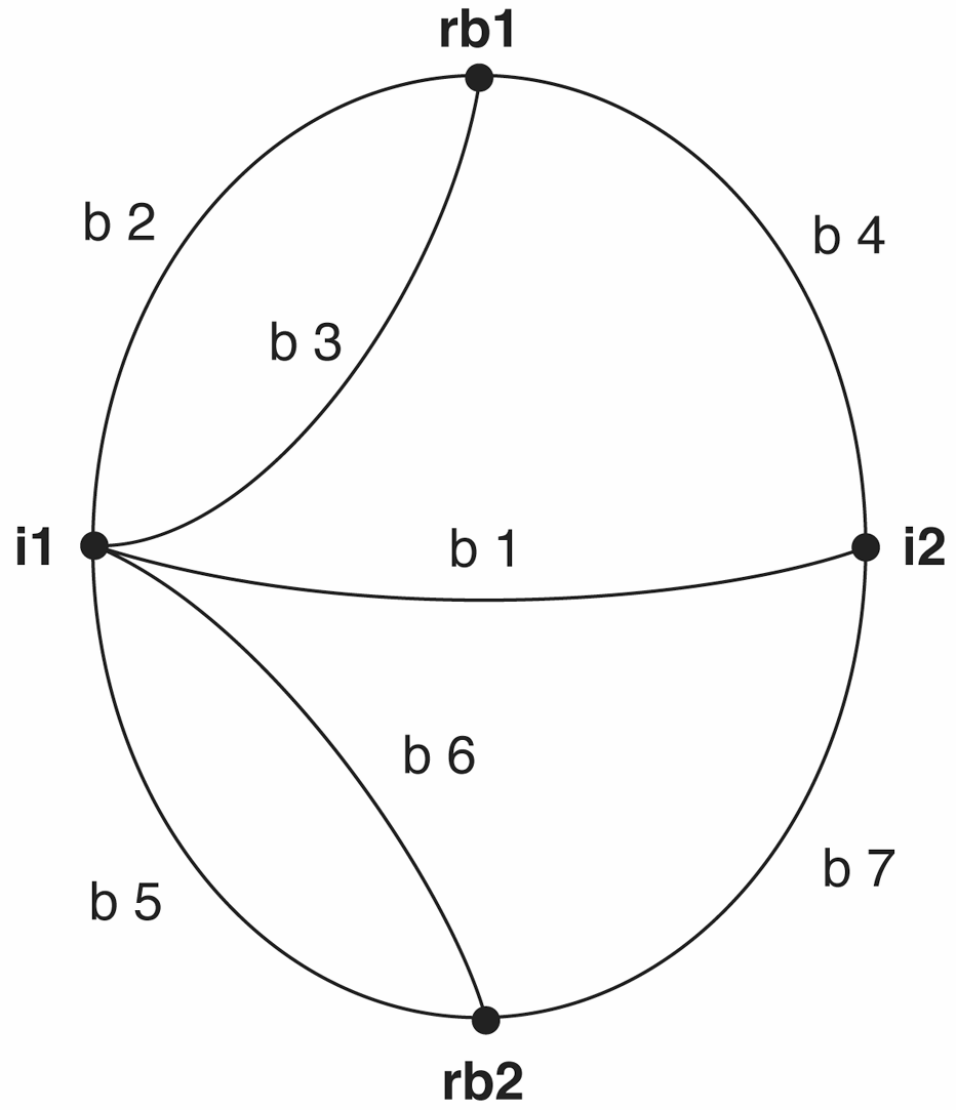
Question: Is there a walk around the city that crosses each bridge exactly once?

Swiss mathematician Leonhard Euler invented graph theory to solve this problem.

The city of Königsberg



Graph of the Königsberg bridge system



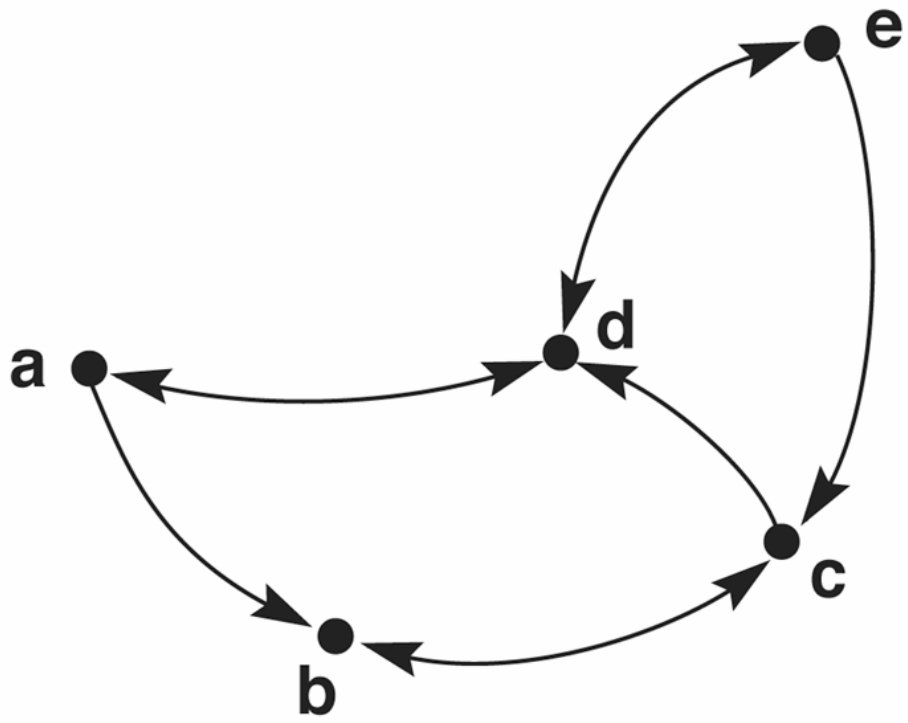
Definition of a graph

A graph consists of

- **A set of *nodes* (can be infinite)**
- **A set of *arcs* that connect pairs of nodes.**

An *arc* is an ordered pair, e.g.,
(i1, rb1),
(rb1, i1).

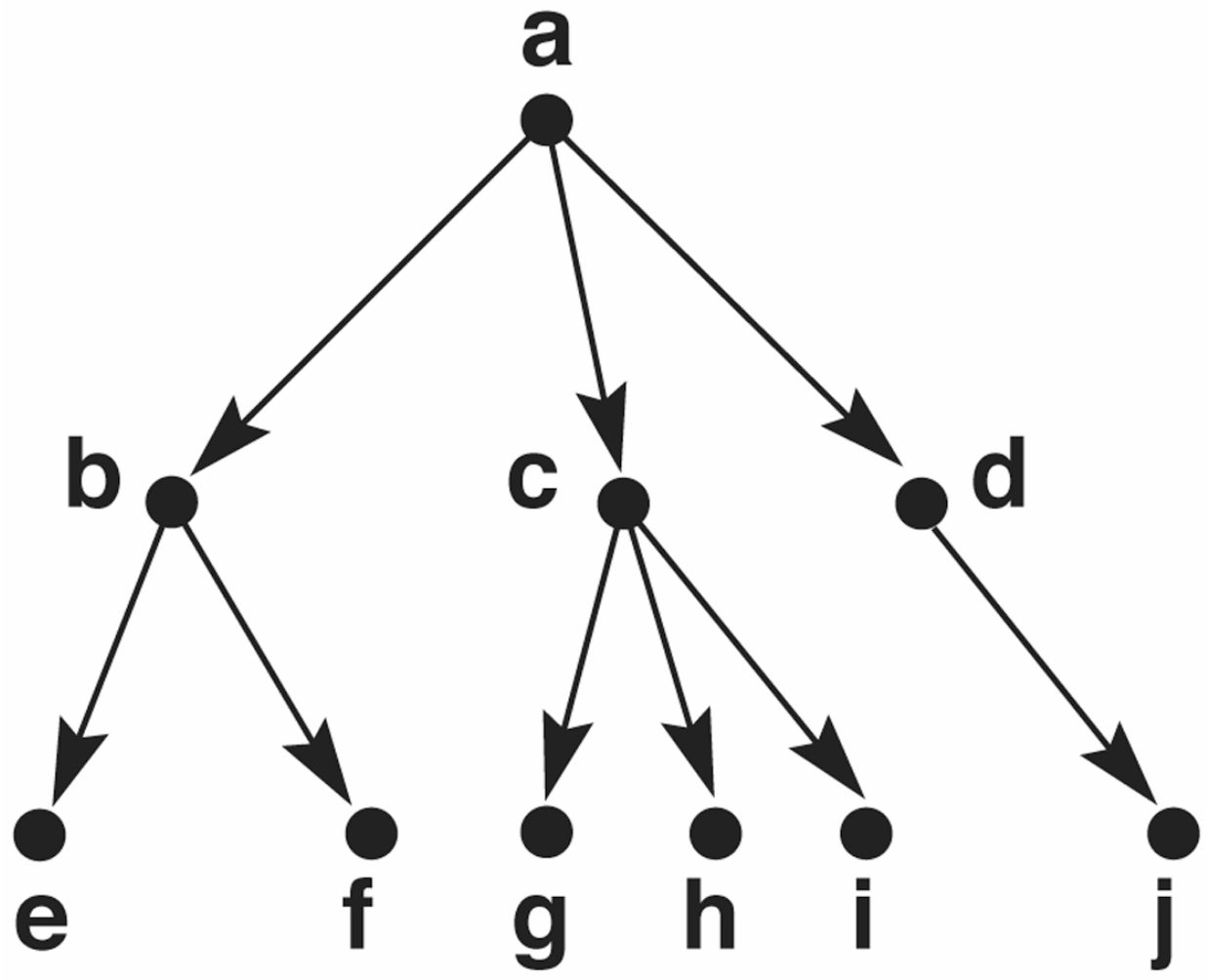
A labeled directed graph



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)}**

A rooted tree, exemplifying family relationships



Definition of a graph (cont'd)

A graph consists of nodes and arcs.

If a directed arc connects N and M, N is called the *parent*, and M is called the *child*. If N is also connected to K, M and K are *siblings*.

A *rooted tree* has a unique node which has no parents. The edges in a rooted tree are directed away from the root. Each node in a rooted tree has a unique parent.

Definition of a graph (cont'd)

A *leaf* or *tip* node is a node that has no children (sometimes also called a *dead end*).

A *path* of length n is an ordered sequence of n+1 nodes such that the graph contains arcs from each node to the following ones.

E.g., [a b e] is a path of length 2.

On a path in a rooted graph, a node is said to be an *ancestor* of all the nodes positioned after it (to its right), as well as a *descendant* of all nodes before it (to its left).

Definition of a graph (cont'd)

A path that contains any node more than once 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.

Two nodes are said to be *connected* if a path exists that includes them both.

A unifying view (Newell and Simon)

The problem space consists of:

- **a *state space* which is a set of states representing the possible configurations of the world**
- **a set of *operators* which can change one state into another**

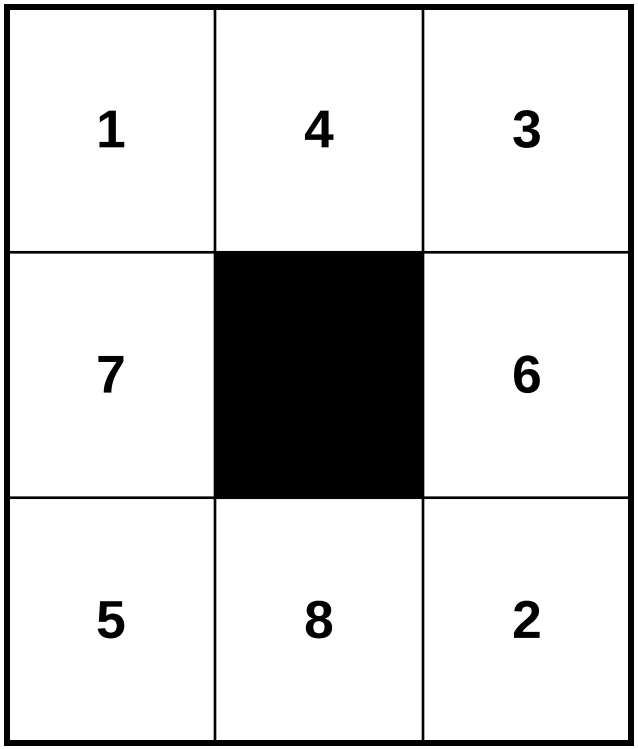
The problem space can be viewed as a graph where the states are the nodes and the arcs represent the operators.

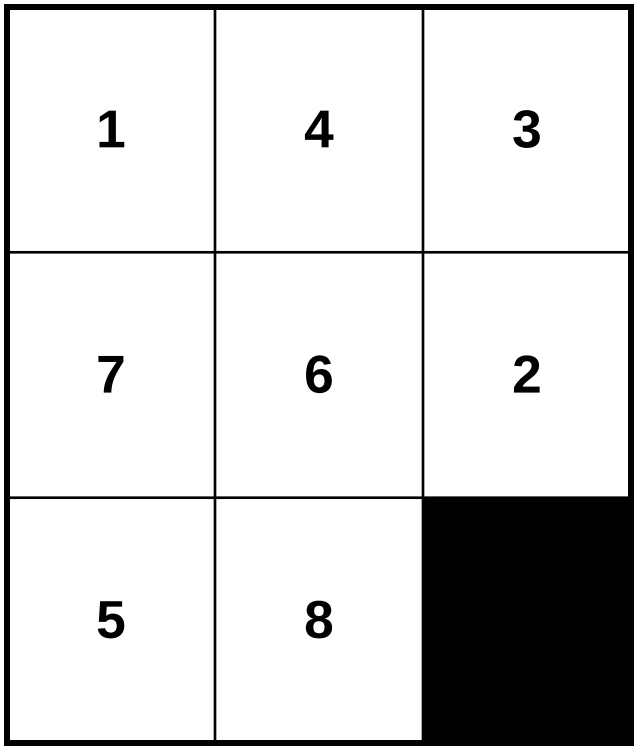
Searching on a graph (simplified)

Start with the initial state (root)

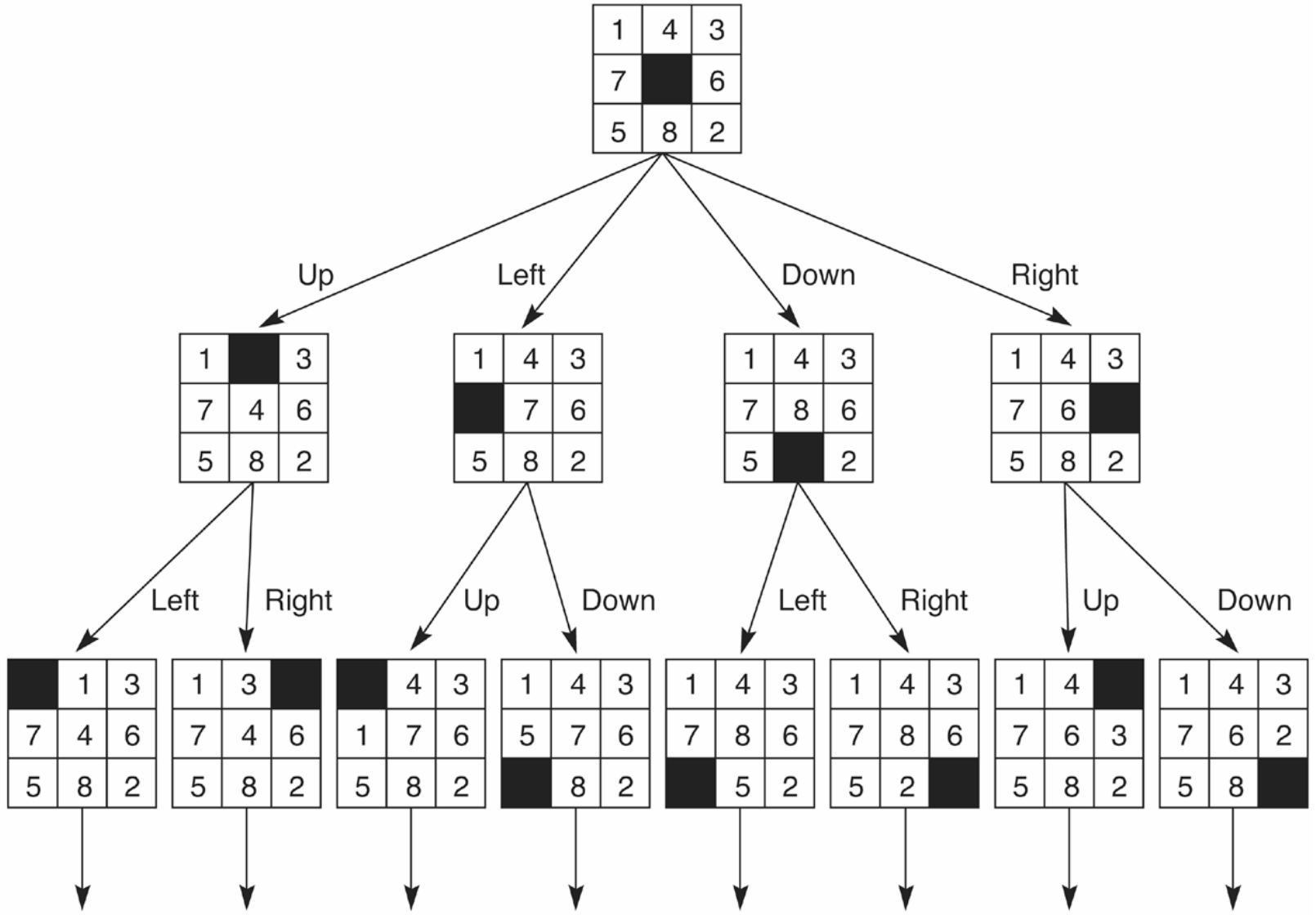
Loop until goal found

find the nodes accessible from the root


1
4
3
7
6
5
8
2


1
4
3
7
6
2
5
8

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



State space search

Represented by a four-tuple [*N*,*A*,*S*,*GD*], where:

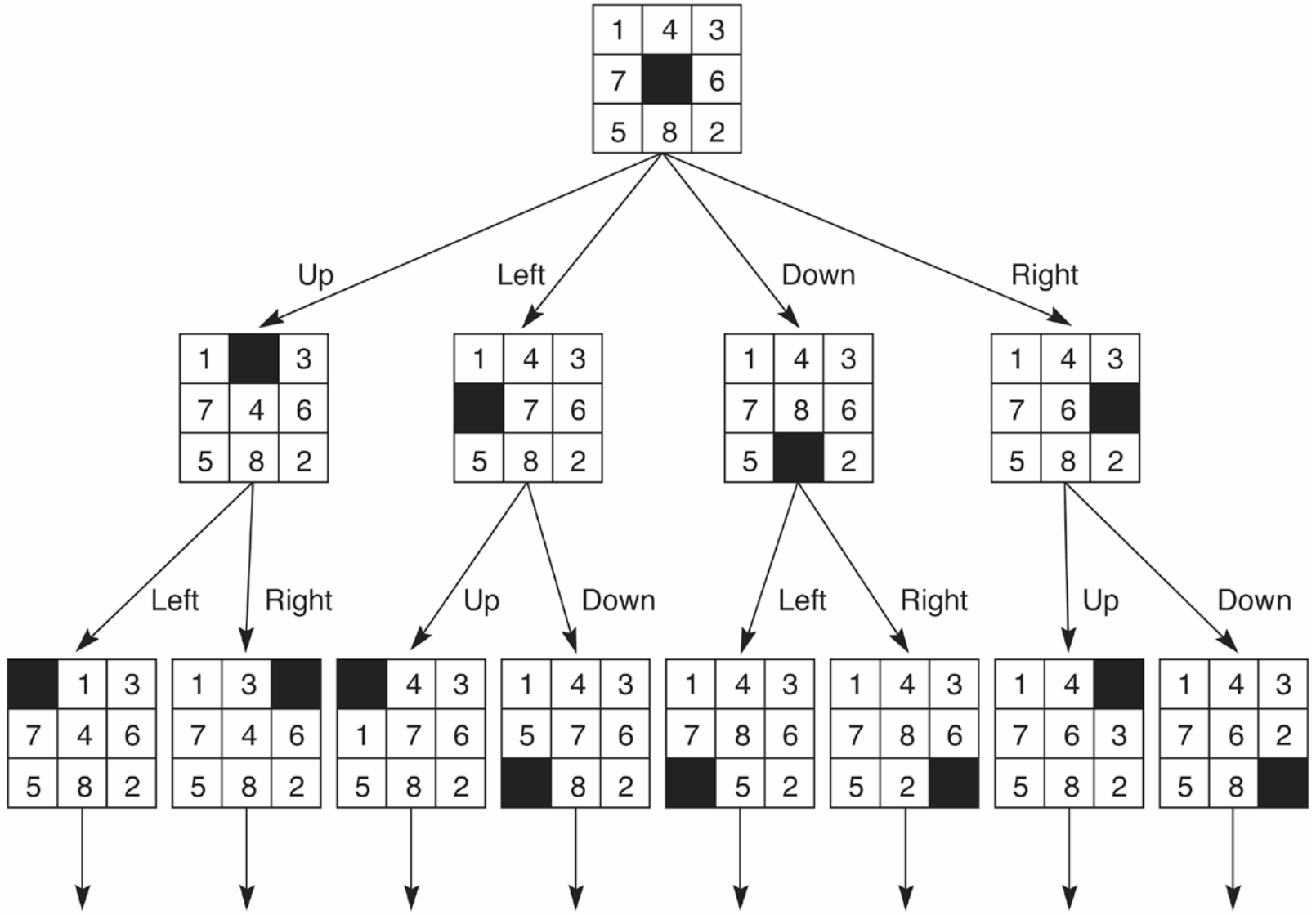
- ***N*** **is the problem space**
- ***A*** **is the set of arcs (or links) between nodes. These correspond to the operators.**
- ***S*** **is a nonempty subset of N. It represents the start state(s) of the problem.**
- ***GD*** **is a nonempty subset of N. It represents the goal state(s) of the problem. The states in GD are described using either:**
  - **a measurable property of the states**
  - **a property of the path developed in the search (a *solution path* is a path from node *S* to a node in *GD*)**

The 8-puzzle problem as state space search

- **states: possible board positions**
- **operators: one for sliding each square in each of four directions,**
  or, better, one for moving the blank square in each of four directions
- **initial state: some given board position**
- **goal state: some given board position**

Note: the “solution” is not interesting here, we need the path.

State space of the 8-puzzle (repeated)

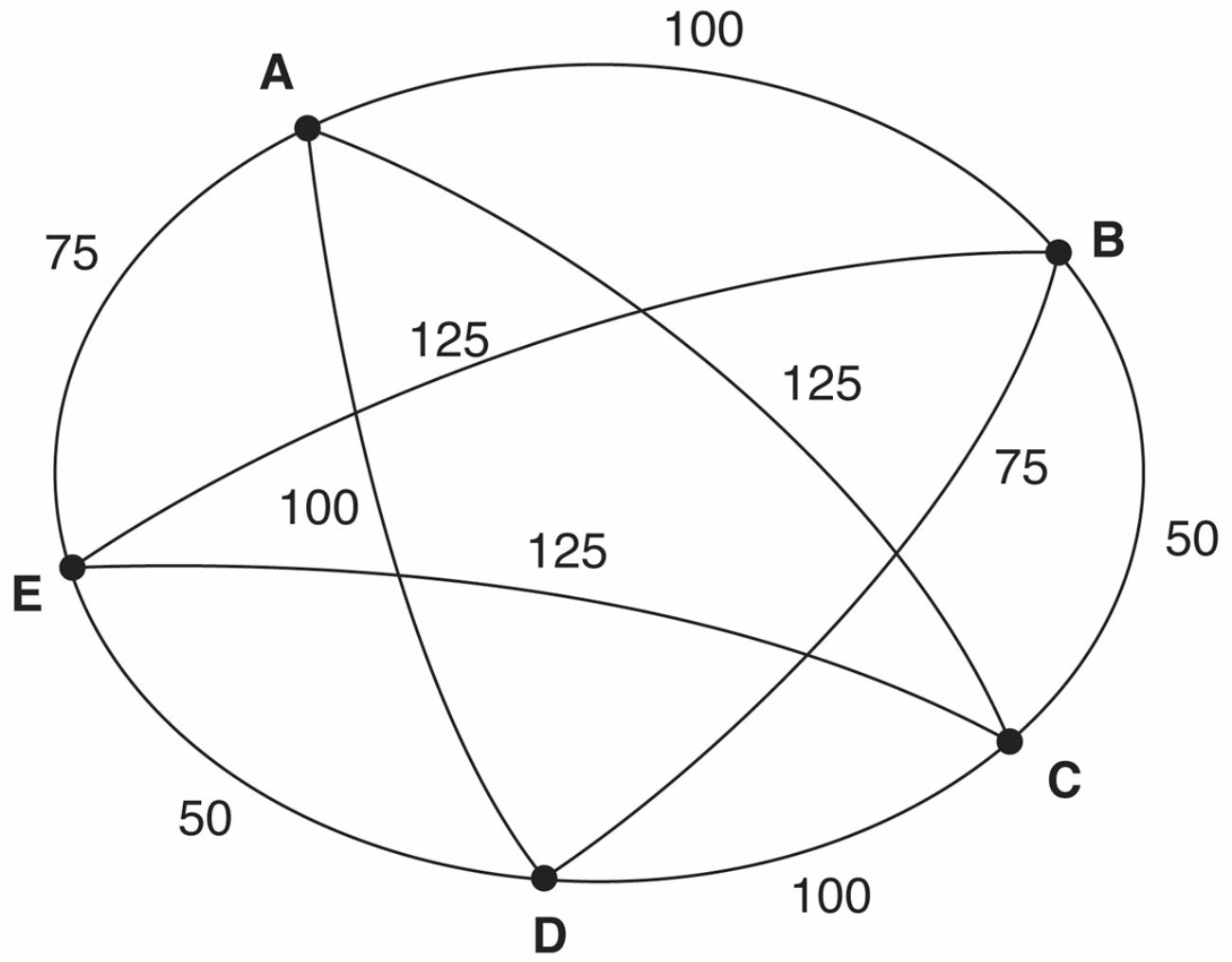


Traveling salesperson problem as state space search

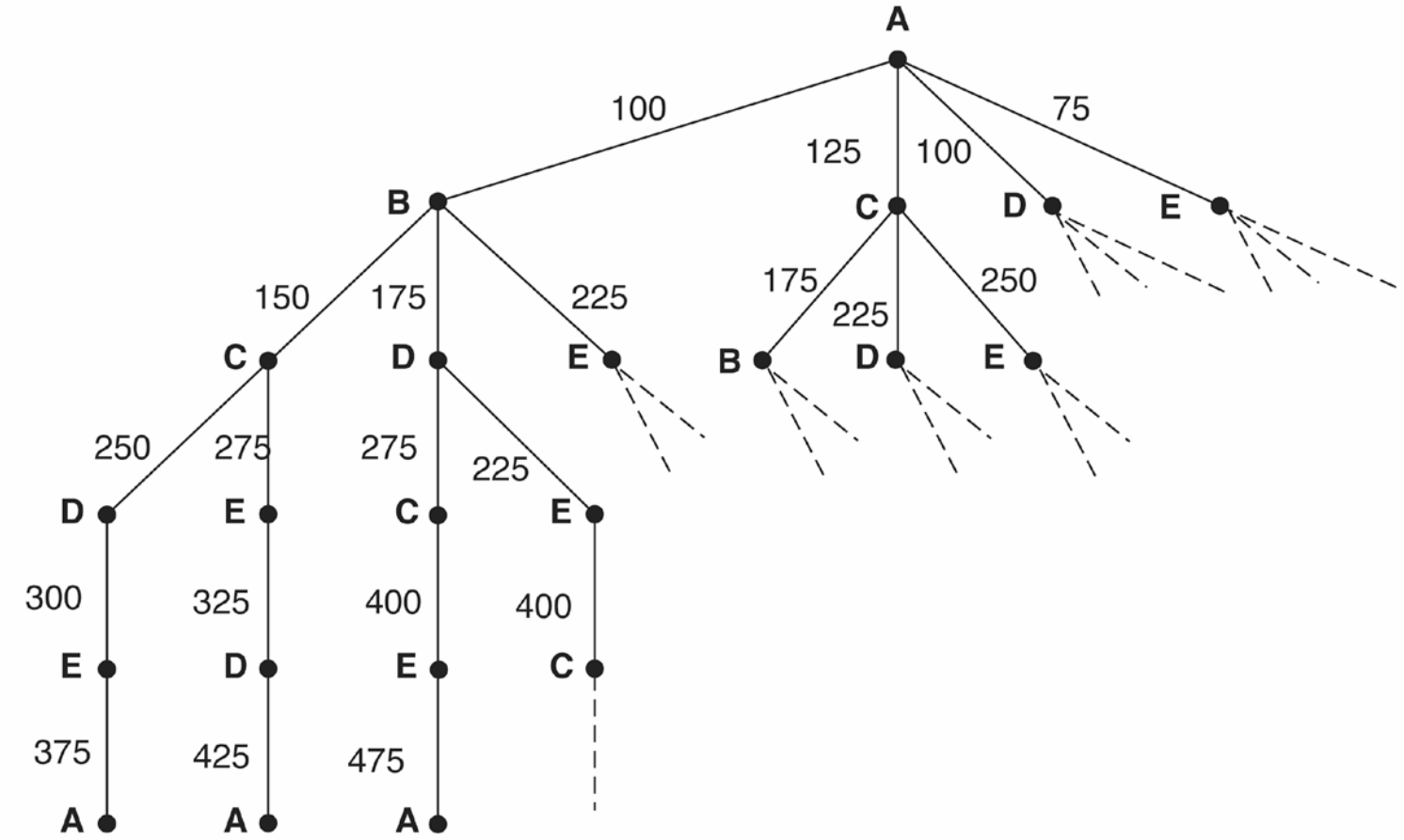
The salesperson has n cities to visit and must then return home. Find the shortest path to travel.

- **state space:**
- **operators:**
- **initial state:**
- **goal state:**

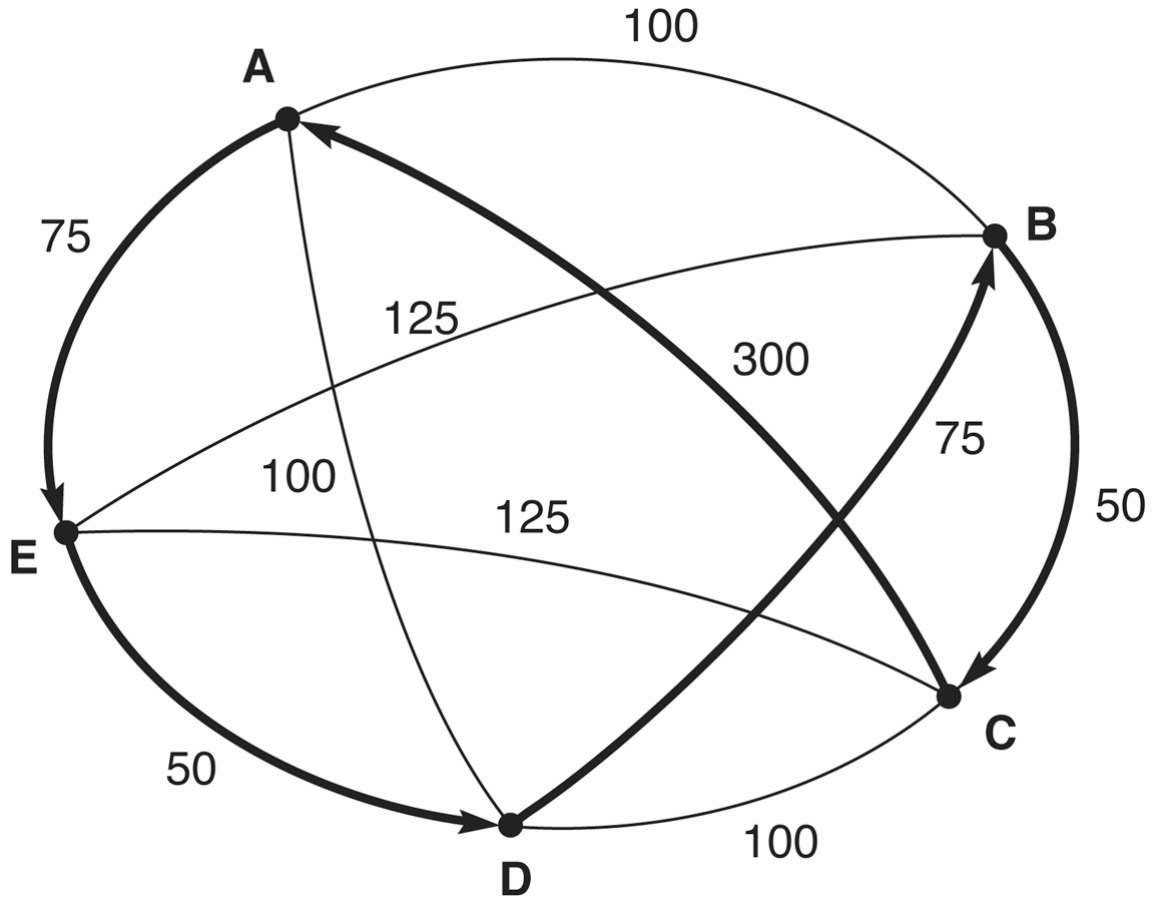
An instance of the traveling salesperson problem



Search of the traveling salesperson problem. (arc label = cost from root)



Nearest neighbor path



Nearest neighbor path = AEDBCA (550)

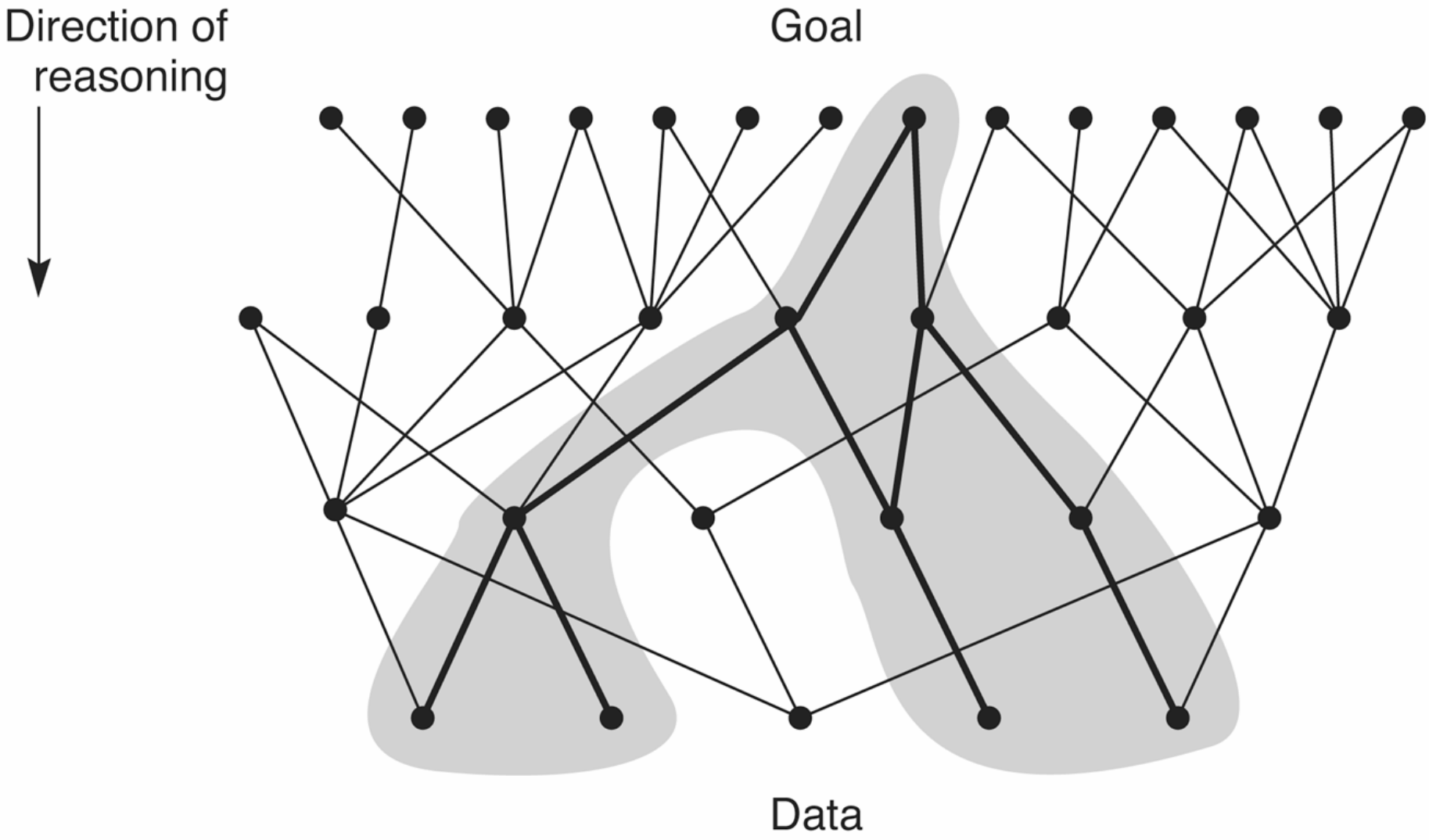
Minimal cost path = ABCDEA (375)

Tic-tac-toe as state space search

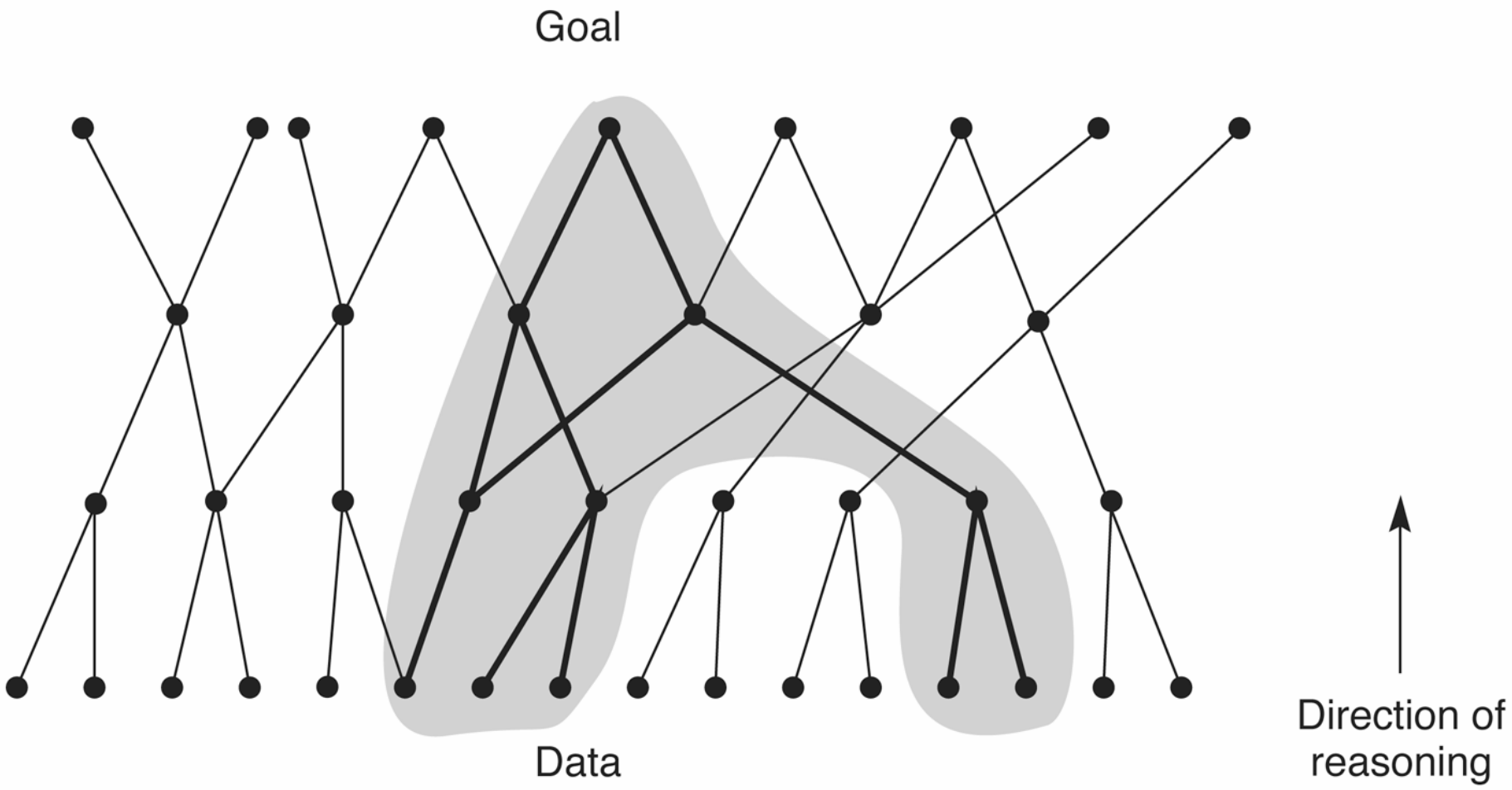
- **states:**
- **operators:**
- **initial state:**
- **goal state:**

Note: this is a “two-player” game

Goal-directed search



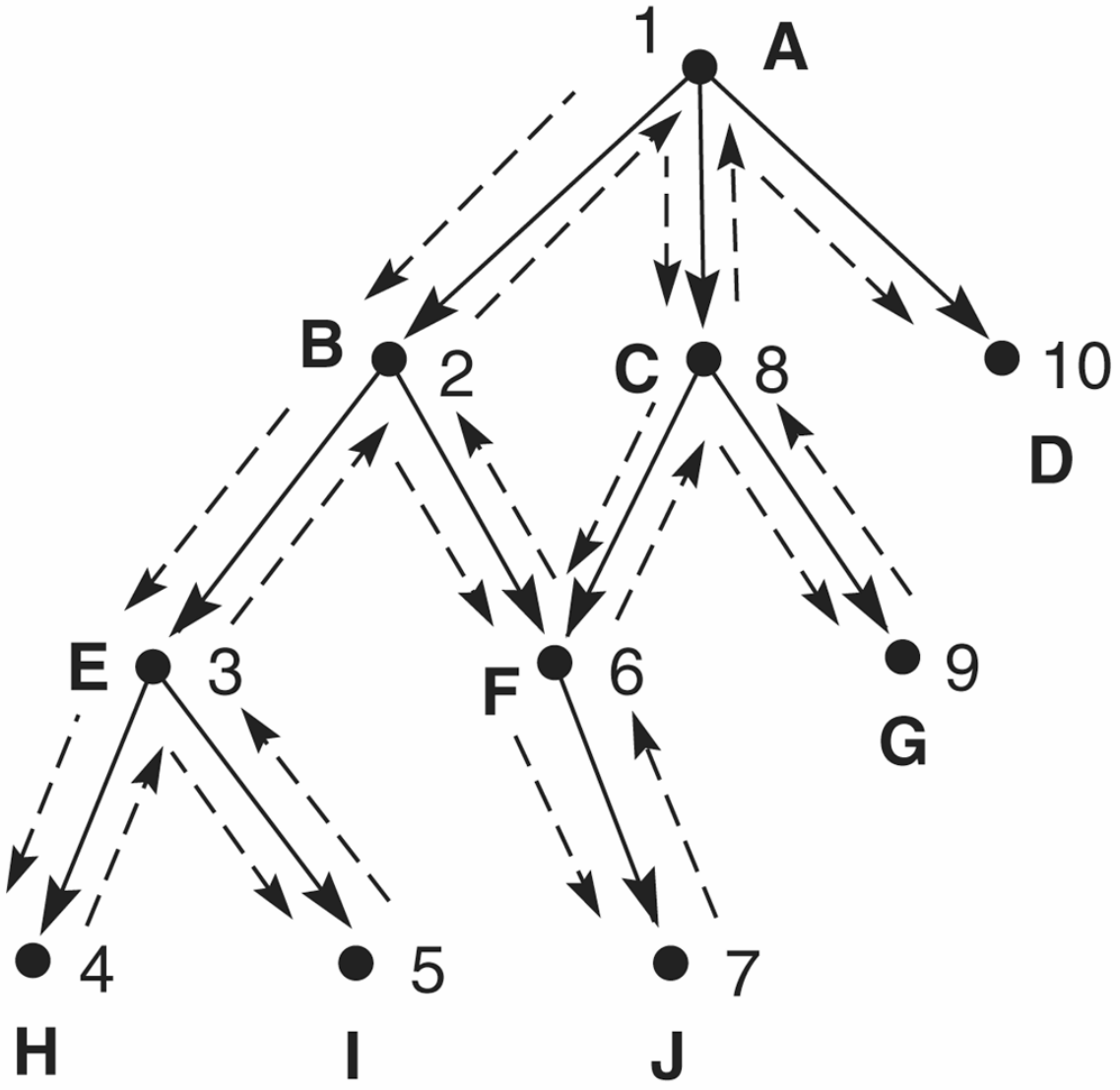
Data-directed search



```
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.
```

Trace of backtracking search (Fig. 3.12)

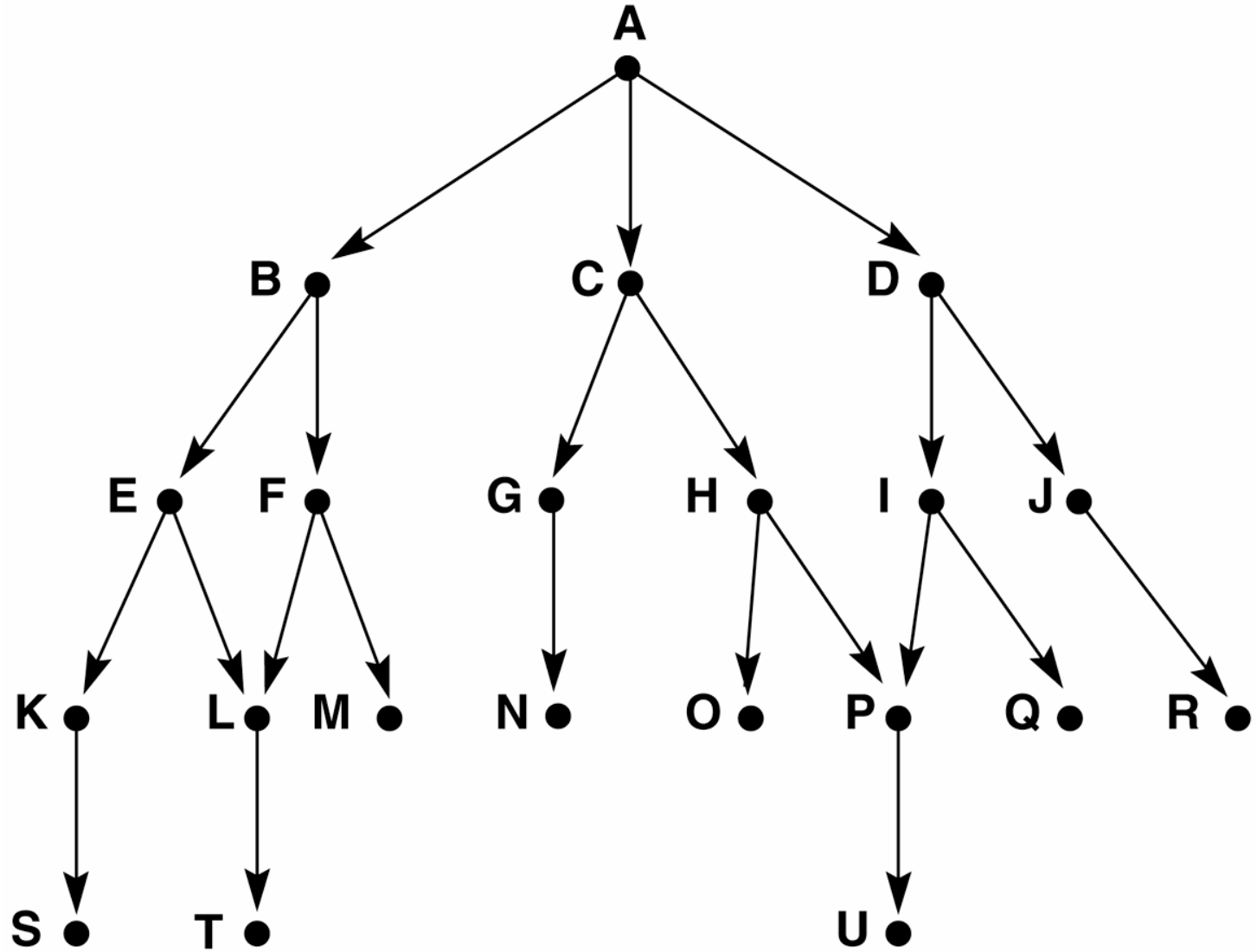


A trace of backtrack on the graph of Fig. 3.12

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]

Graph for BFS and DFS (Fig. 3.13)



Breadth_first search algorithm

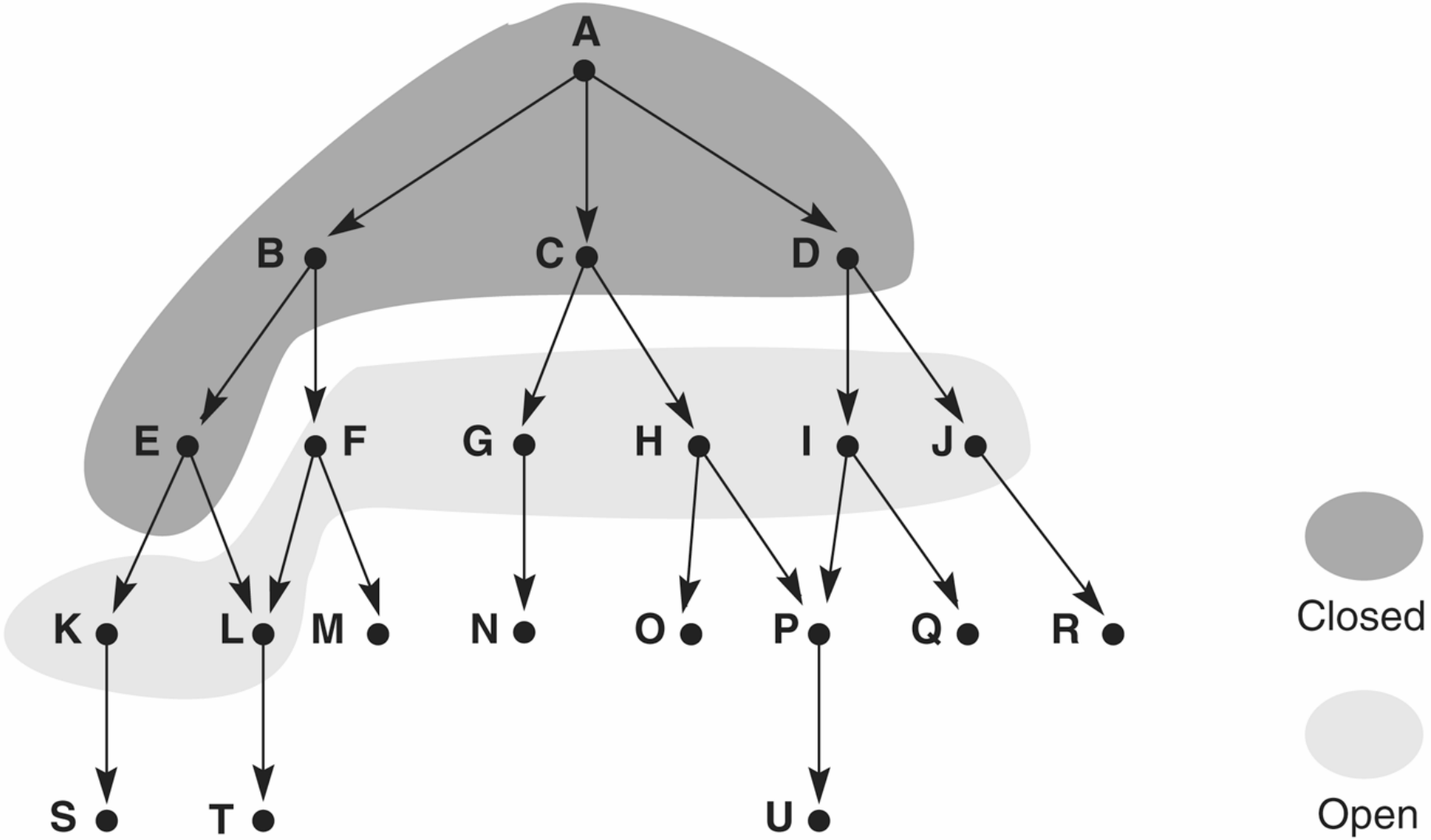
```
function breadth_first_search;

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.
```

Trace of BFS on the graph of Fig. 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** = [ ]

Graph of Fig. 3.13 at iteration 6 of BFS



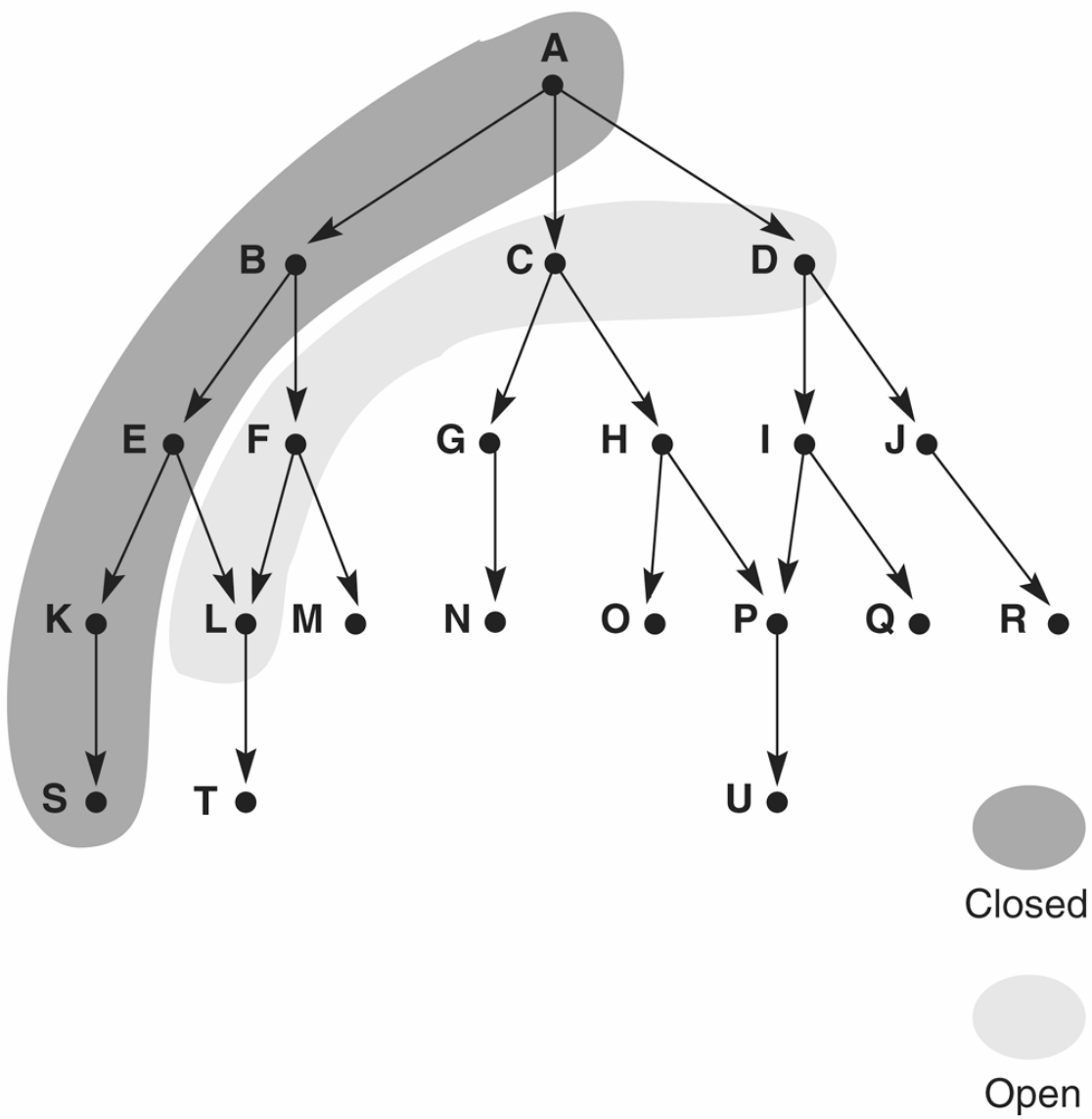
Depth_first_search algorithm

```
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.
```

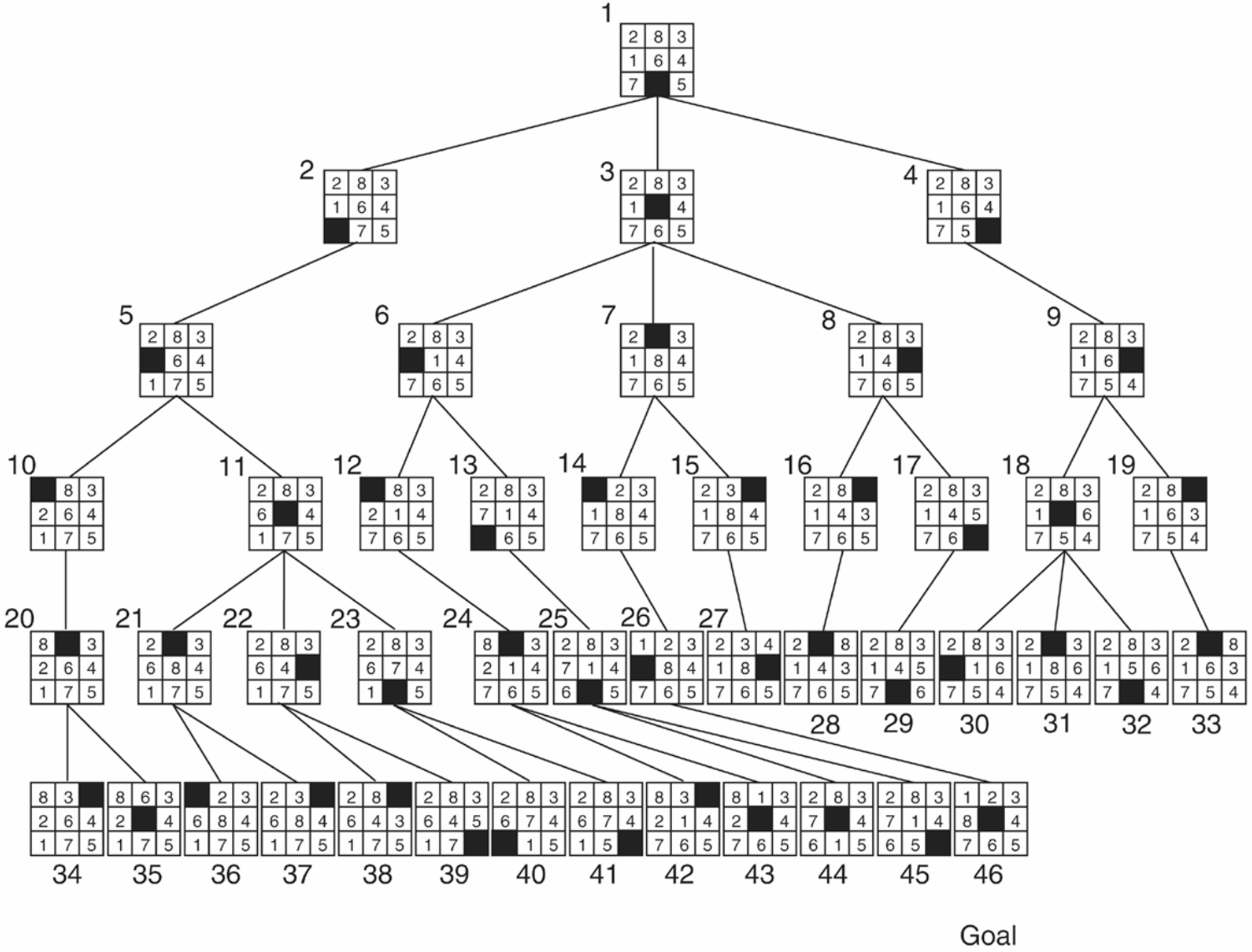
Trace of DFS on the graph of Fig. 3.13

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]**

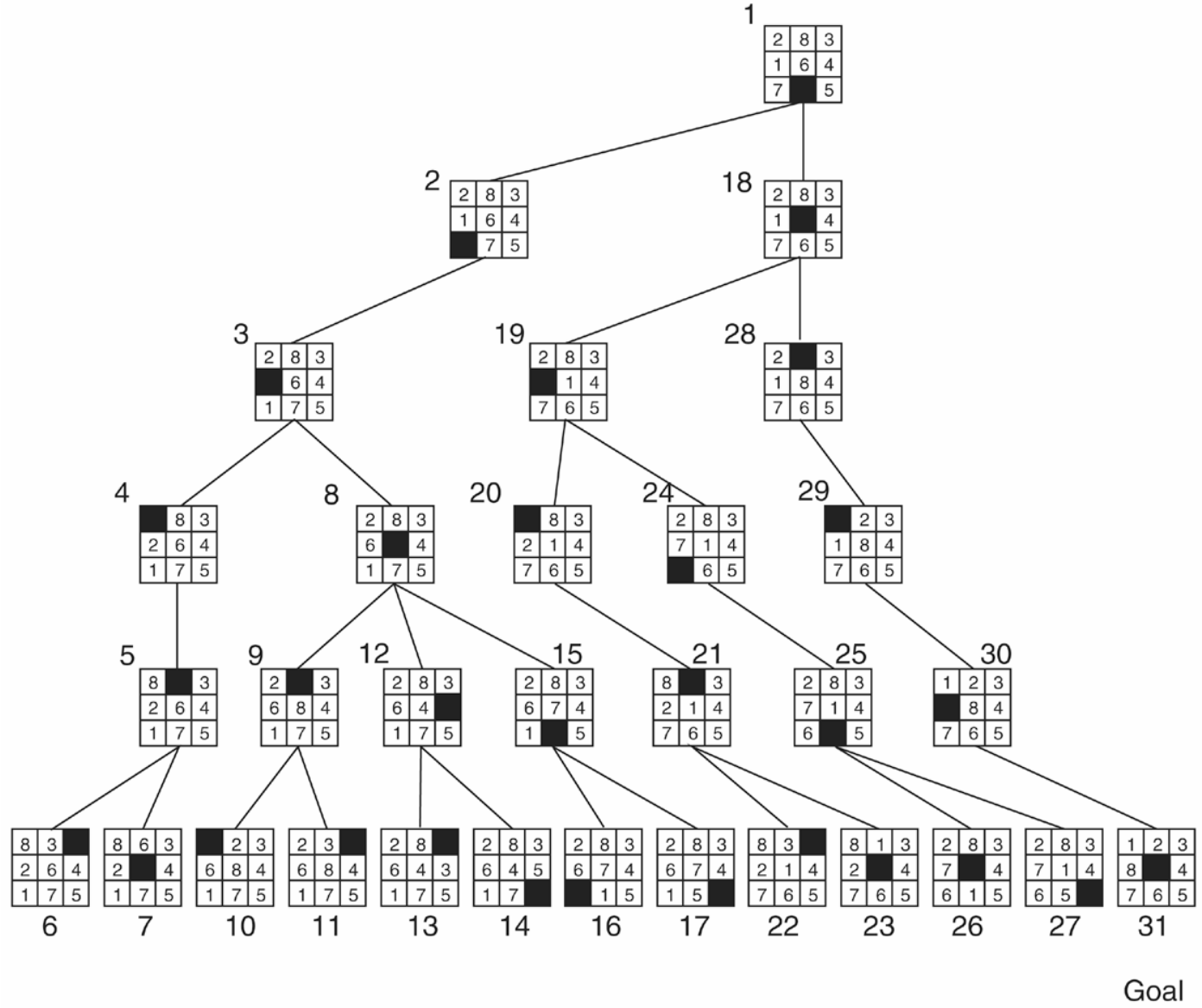
Graph of Fig. 3.13 at iteration 6 of DFS



BFS, label = order state was removed from OPEN



DFS with a depth bound of 5, label = order state was removed from OPEN



State space graph of a set of implications in propositional calculus

$q \rightarrow p$

$r \rightarrow p$

$v \rightarrow q$

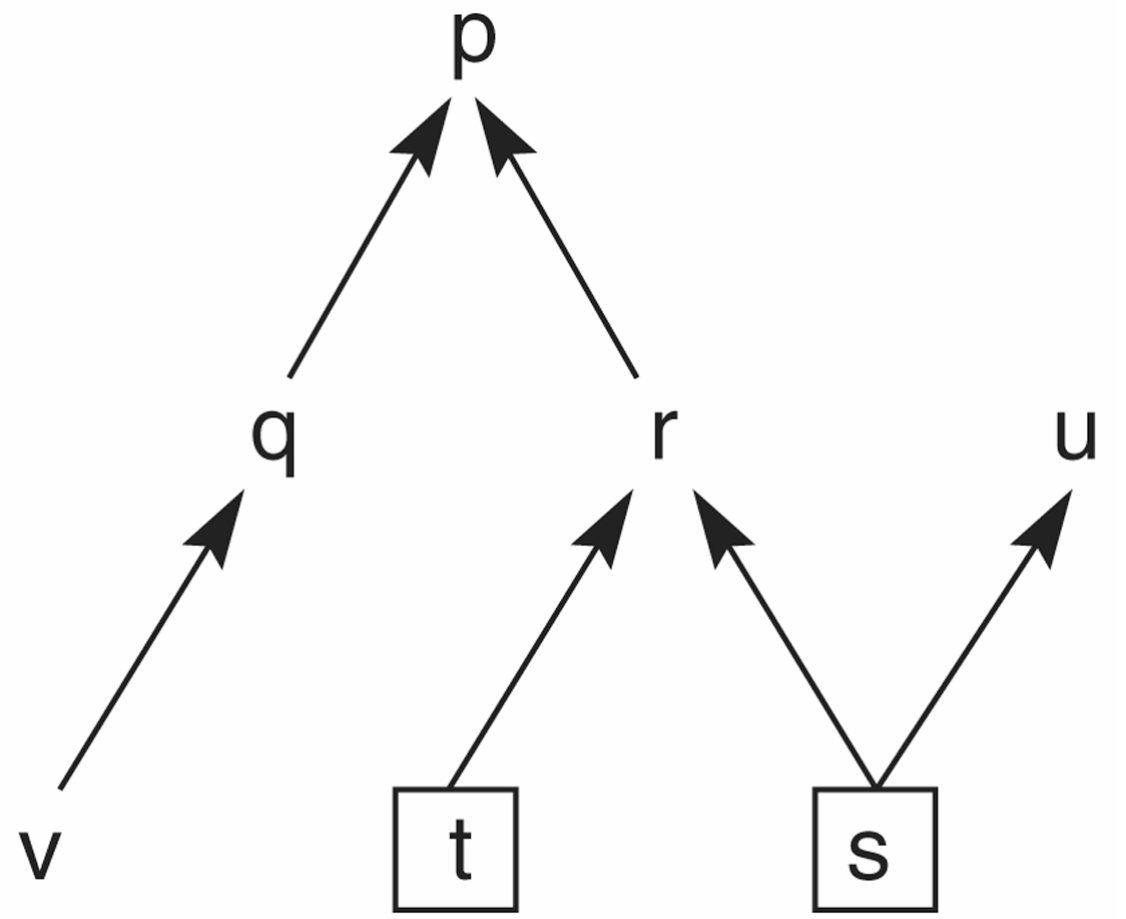
$s \rightarrow r$

$t \rightarrow r$

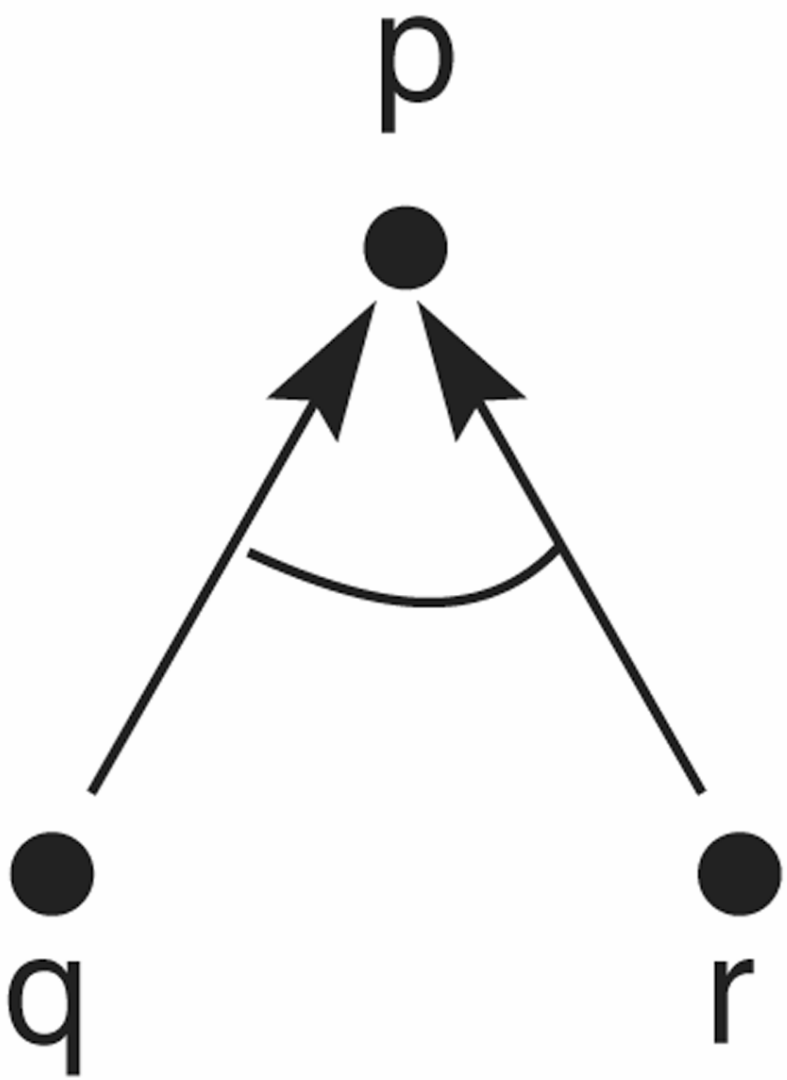
$s \rightarrow u$

s

t



And/or graph of the expression $q \wedge r \rightarrow p$



Hypergraph

A *hypergraph* consists of:

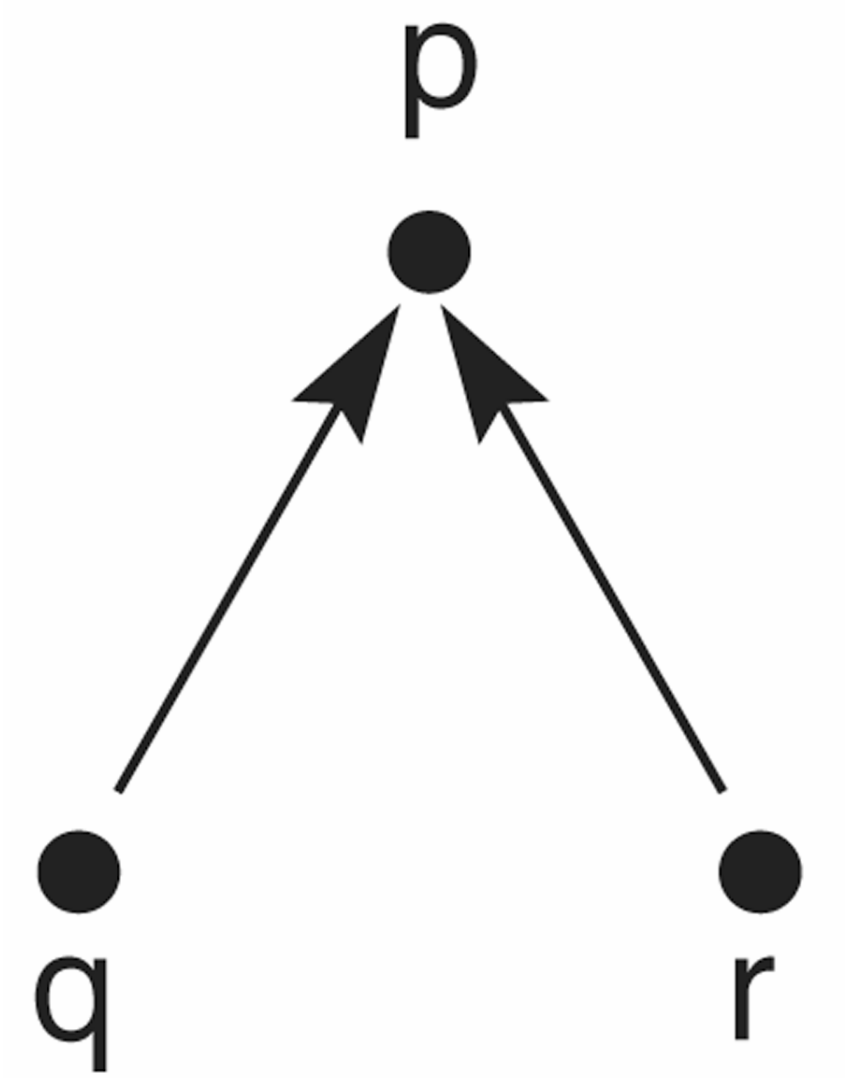
N: a set of nodes.

H: a set of *hyperarcs*.

***Hyperarcs* are defined by ordered pairs in which the first element of the pairs is a single node from N and the second element is a subset of N.**

An ordinary graph is a special case of hypergraph in which all the sets of descendant nodes have a cardinality of 1.

And/or graph of the expression q∨r→p



And/or graph of a set of propositional calculus expressions

a

b

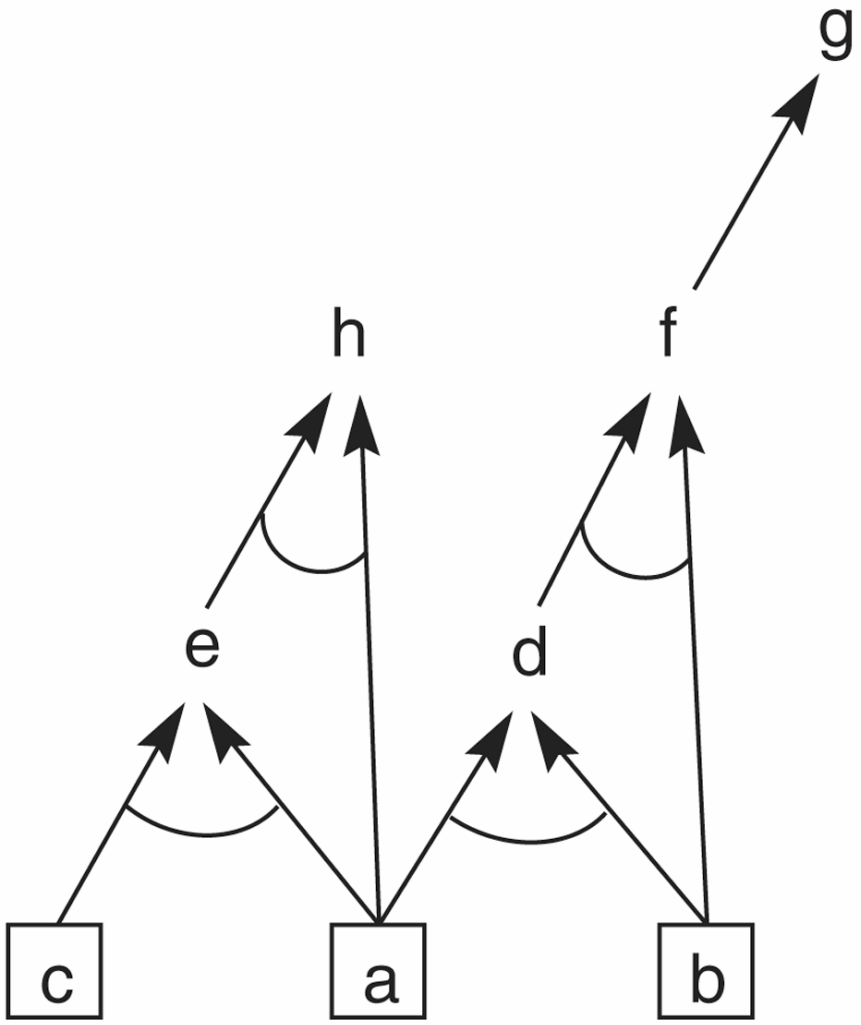
c

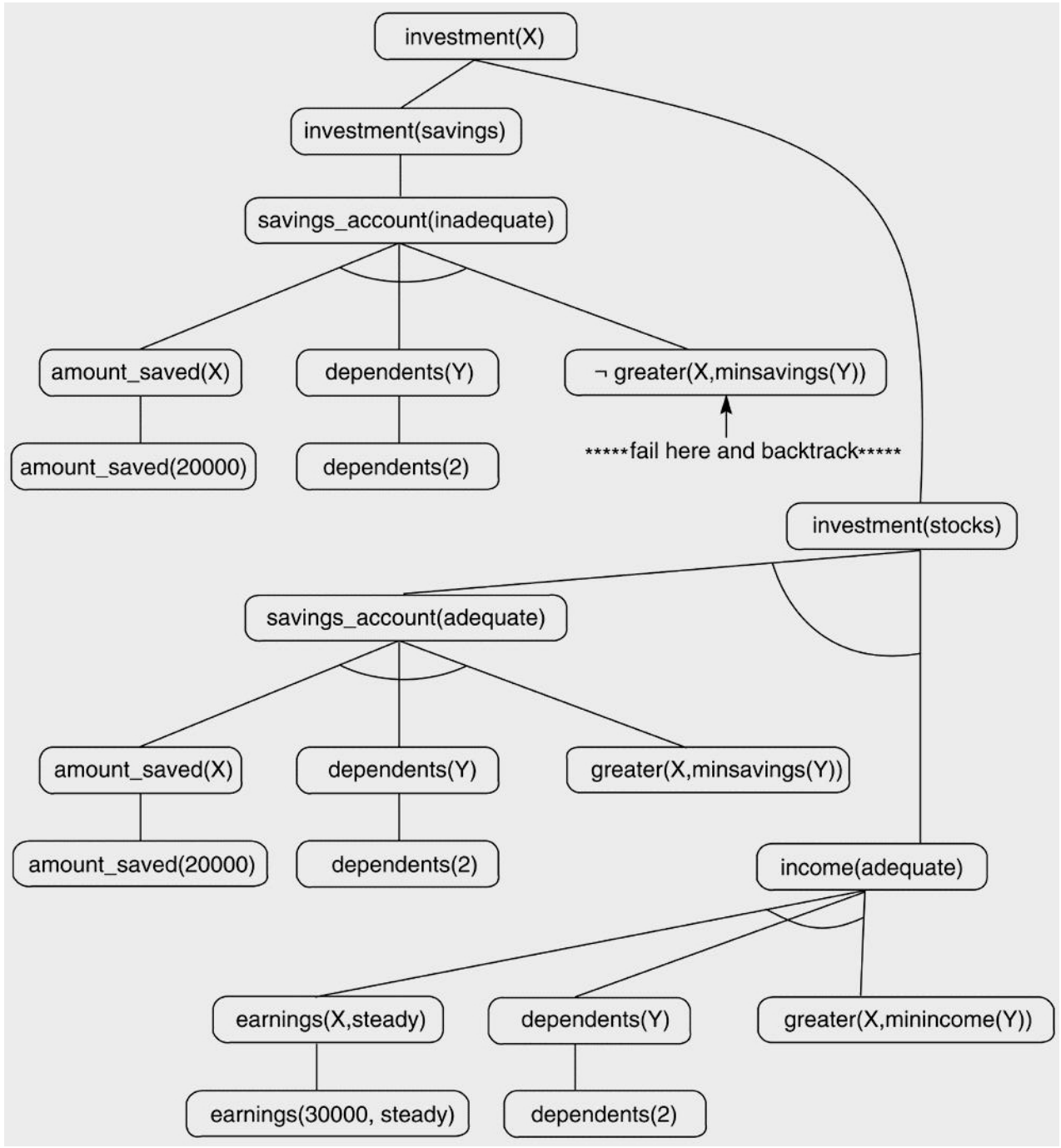
$a \wedge b \rightarrow d$

$a \wedge c \rightarrow e$

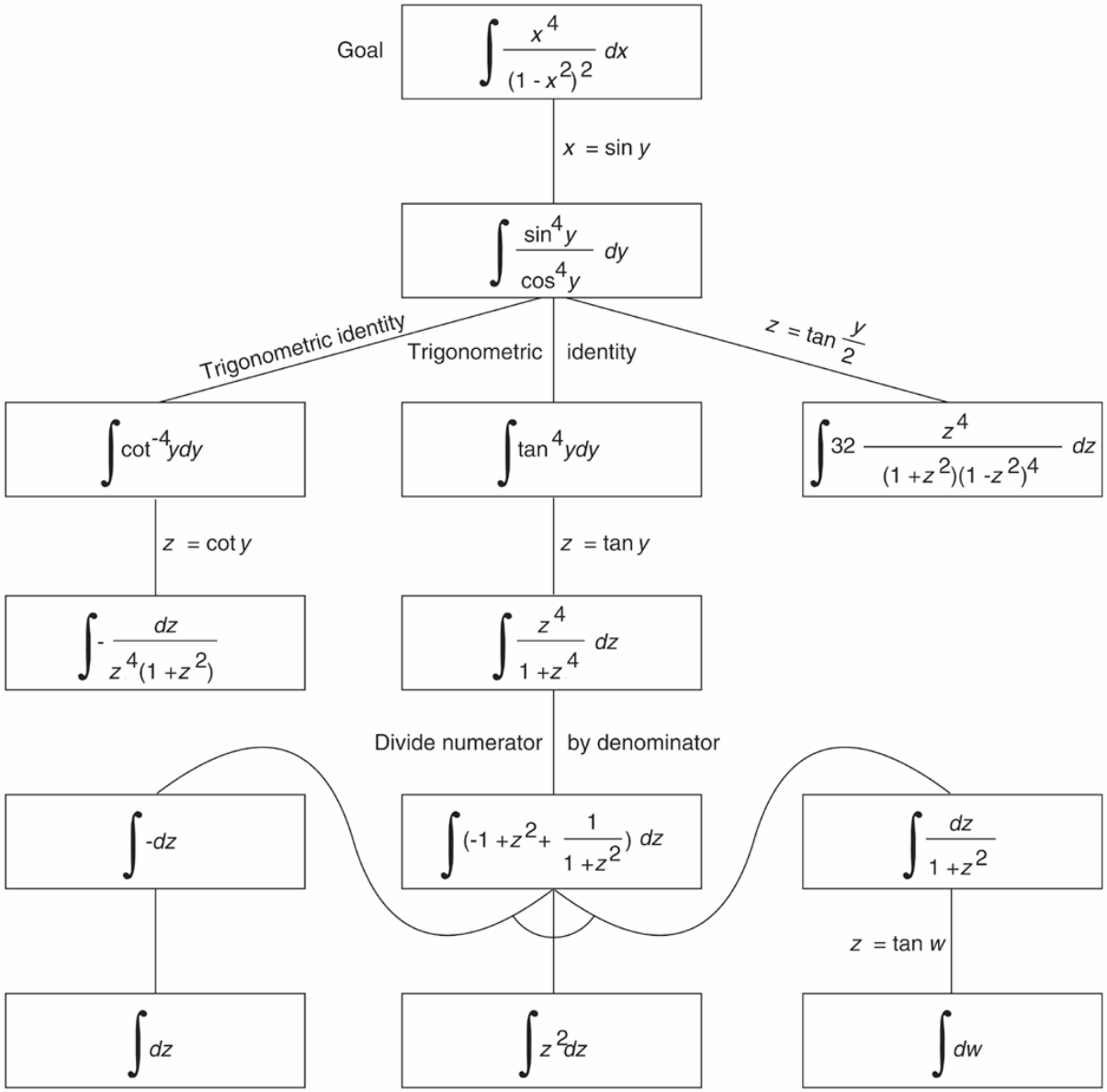
$b \wedge d \rightarrow f$

$f \rightarrow g$

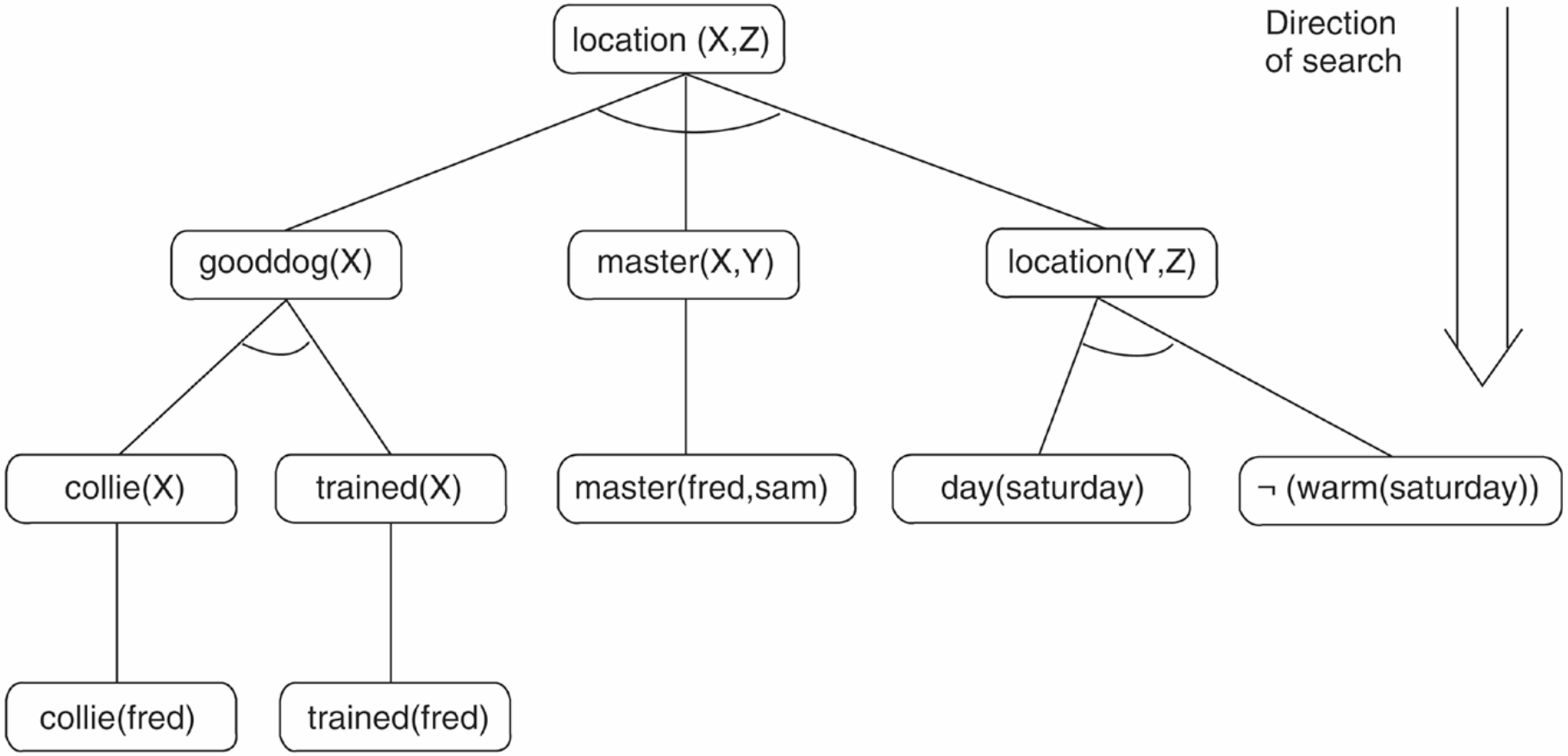



investment(X)
investment(savings)
savings_account(inadequate)
amount_saved(X)
dependents(Y)
¬ greater(X,minsavings(Y))
amount_saved(20000)
dependents(2)
*****fail here and backtrack*****
investment(stocks)
savings_account(adequate)
amount_saved(X)
dependents(Y)
greater(X,minsavings(Y))
amount_saved(20000)
dependents(2)
income(adequate)
earnings(X,steady)
dependents(Y)
greater(X,minincome(Y))
earnings(30000, steady)
dependents(2)

And/or graph of the part of the state space for integrating a function (Nilsson 1971)



The solution subgraph showing that fred is at the museum



Substitutions = {fred/X, sam/Y, museum/Z}

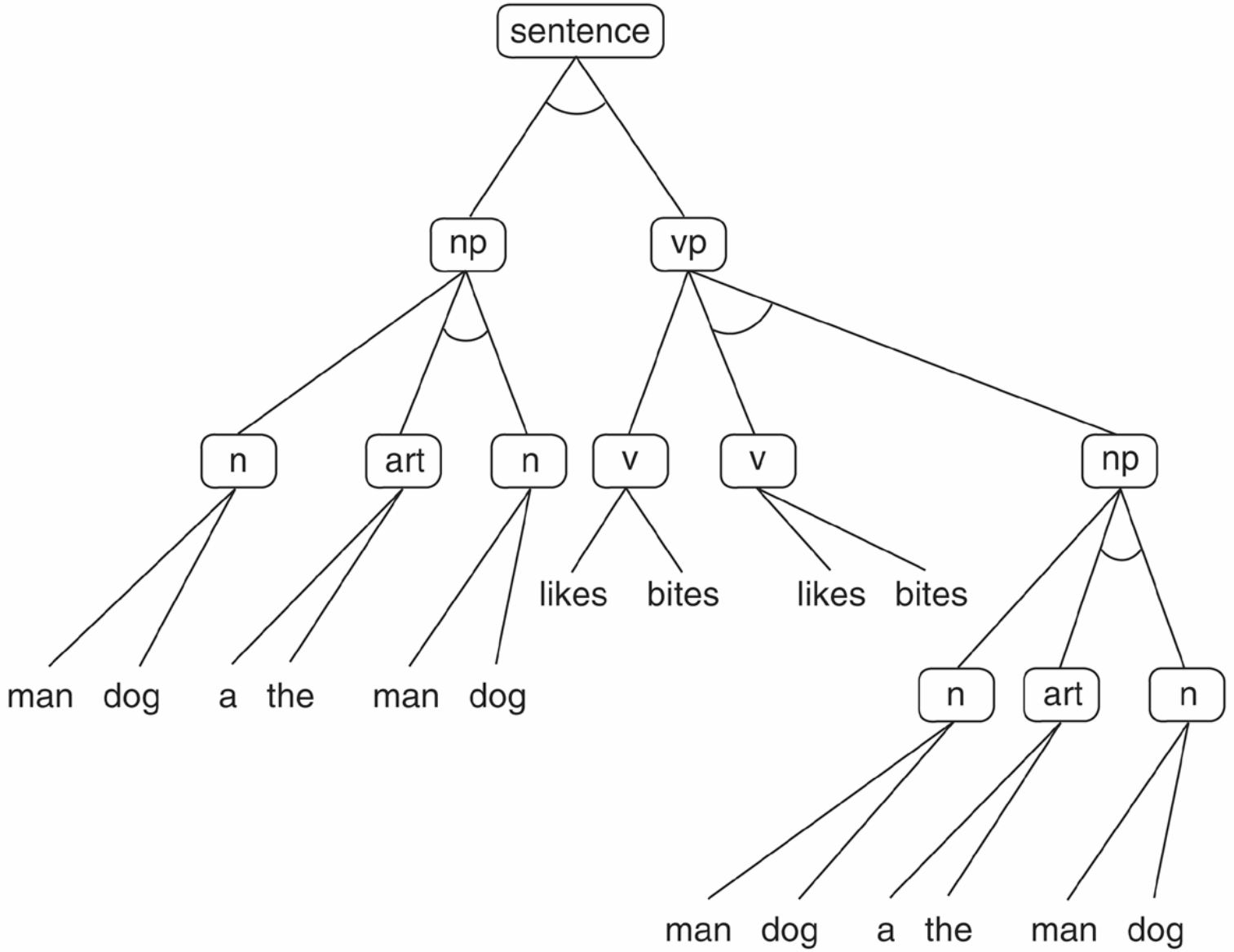
Facts and rules for the example

1. Fred is a collie.
   collie(fred).
2. Sam is Fred's master.
   master(fred,sam).
3. The day is Saturday.
   day(saturday).
4. It is cold on Saturday.
   $\neg$ (warm(saturday)).
5. Fred is trained.
   trained(fred).
6. Spaniels are good dogs and so are trained collies.
   $\forall$ X[spaniel(X) $\vee$ (collie(X) $\wedge$ trained(X)) $\rightarrow$ gooddog(X)]
7. If a dog is a good dog and has a master then he will be with his master.
   $\forall$ (X,Y,Z) [gooddog(X) $\wedge$ master(X,Y) $\wedge$ location(Y,Z) $\rightarrow$ location(X,Z)]
8. If it is Saturday and warm, then Sam is at the park.
   (day(saturday) $\wedge$ warm(saturday)) $\rightarrow$ location(sam,park).
9. If it is Saturday and not warm, then Sam is at the museum.
   (day(saturday) $\wedge$ $\neg$ (warm(saturday))) $\rightarrow$ location(sam,museum).

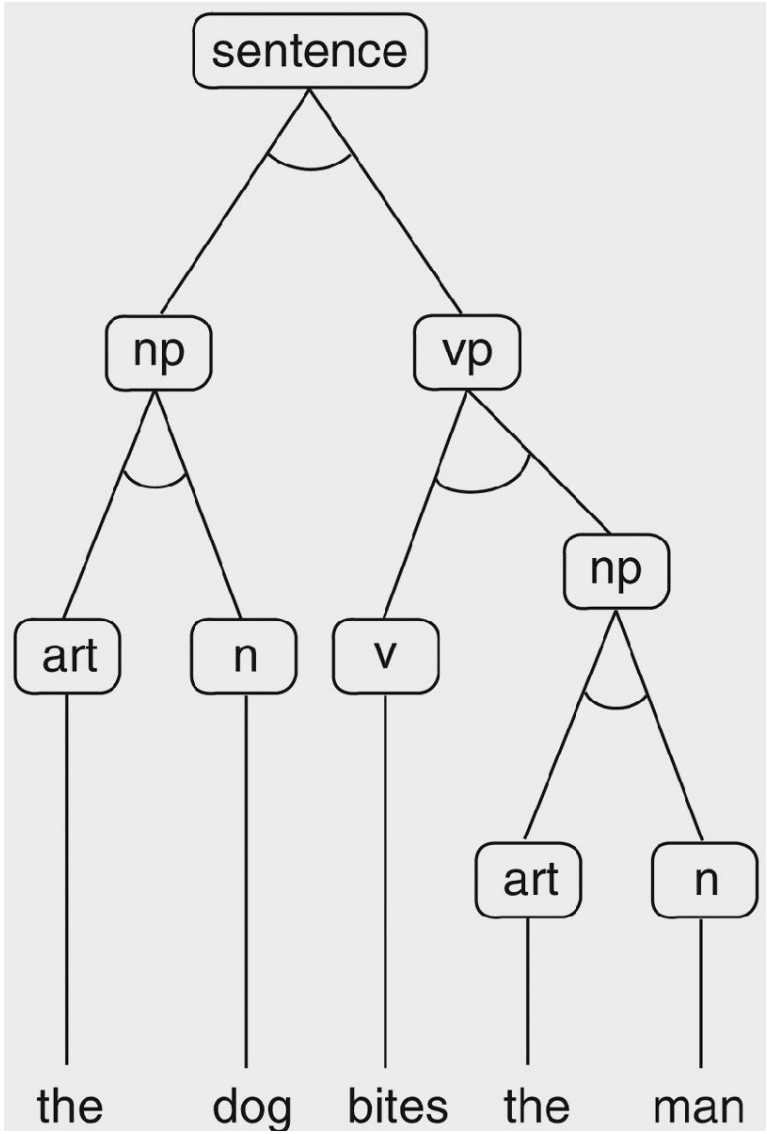
Five rules for a simple subset of English grammar

1. A sentence is a noun phrase followed by a verb phrase.
   sentence ↔ np vp
2. A noun phrase is a noun.
   np ↔ n
3. A noun phrase is an article followed by a noun.
   np ↔ art n
4. A verb phrase is a verb.
   vp ↔ v
5. A verb phrase is a verb followed by a noun phrase.
   vp ↔ v np
6. art ↔ a
7. art ↔ the
   ("a" and "the" are articles)
8. n ↔ man
9. n ↔ dog
   ("man" and "dog" are nouns)
10. v ↔ likes
11. v ↔ bites
    ("likes" and "bites" are verbs)

Figure 3.25: And/or graph for the grammar of Example 3.3.6. Some of the nodes (np, art, etc.) have been written more than once to simplify drawing the graph.



Parse tree for the sentence



“Blind search”

BFS and DFS are blind in the sense that they have no knowledge about the problem at all other than the problem space

Such techniques are also called brute-force search, uninformed search, or weak methods

Obviously, we can’t expect too much from these, but they provide

- **Worst-case scenarios**
- **A basis for more interesting algorithms later on**

BFS and DFS

Worst case scenarios are equally bad (exponential)

How to evaluate search algorithms

- **Completeness: a search algorithm is complete if it is guaranteed to find a solution when one exists**
- **Quality of solution: usually the path cost**
- **Time cost of finding the solution: usually in terms of number of nodes generated or examined**
- **Memory cost of finding the solution: how many nodes do we have to keep around during the search**

Evaluating BFS

- **Complete?**
  Yes
- **Optimal quality solution?**
  Yes
- **Time required in the worst case**
  $O(b^d)$
- **Memory required in the worst case (in OPEN)**
  $O(b^d)$

where *b* is the *branching factor*, *d* is the *depth of the solution*

Evaluating DFS

- **Complete?**
  Yes (only if the tree is finite)
- **Optimal quality solution?**
  No
- **Time required in the worst case**
  $O(b^m)$
- **Memory required in the worst case (in OPEN)**
  O(bm)

where *b* is the *branching factor*, *m* is the *maximum depth of the tree*

Example search problems

Puzzles: missionaries and cannibals, cryptarithmetic, 8-puzzle, 15-puzzle, Rubik's cube, n-queens, the Tower of Hanoi, …

2-player games: chess, checkers, Chinese Go, …

Proving theorems in logic and geometry

Path finding

"Industrial" problems: VLSI layout design, assembling a complex object

"AI problems": speech recognition, planning, …

The importance of the problem space

The choice of a problem space makes a big difference

- **in fact, finding a good abstraction is half of the problem**

Intelligence is needed to figure out what problem space to use

- **still poorly understood: the human problem solver is conducting a search in the space of problem spaces**