MODULE 2

Syllabus

- Search Methods:
  - data driven and goal driven search.
  - Depth first and breadth first search,
  - DFS with iterative deepening.
  - Heuristic search-best first search,
  - A * algorithm.
  - AO* algorithm,
  - Constraint Satisfaction.
  - Crypt Arithmetic Problems

The State Space Representation of Problems

- Nodes of a graph====> partial problem solution states
- Arcs===> to steps in a problem-solving process
- One or more initial states, corresponding to the given information in a problem instance, form the root of the graph.
- The graph also defines one or more goal conditions, which are solutions to a problem instance.
- State space search characterizes problem solving as the process of finding a solution path from the start state to a goal.

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.
  - 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 measurable property of the path developed in the search, for example, the sum of 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.

EXAMPLE: TIC-TAC-TOE

- Start state===>empty board
- Goal state===> a board state having three Xs in a row, column, or diagonal
- The path from the start state to a goal state gives the series of moves in a winning game.
- states in the space===> are all the different configurations of Xs and Os that the game can have

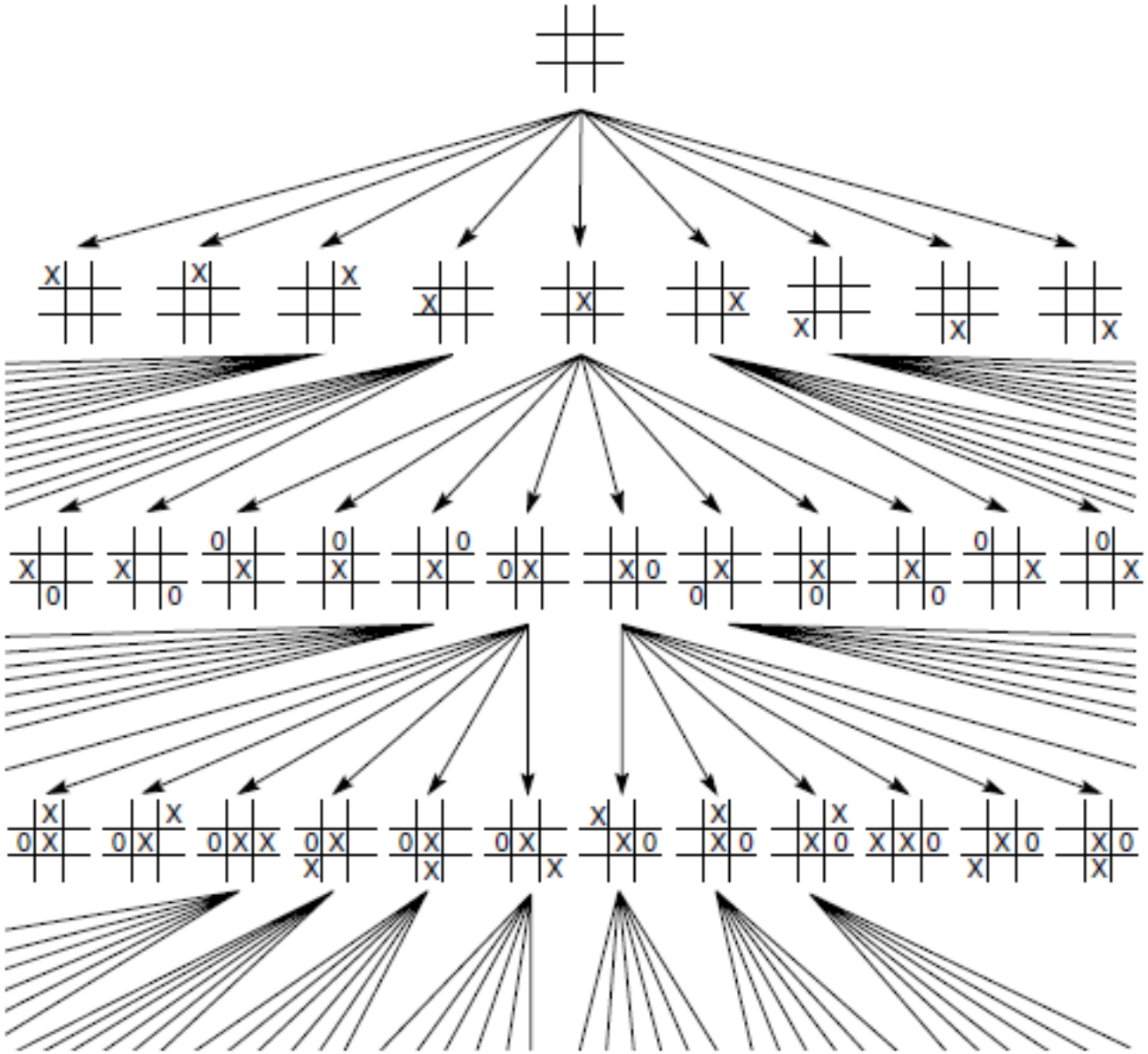


Figure II.5 Portion of the state space for tic-tac-toe.

- although there are 39 ways to arrange {blank, X, O} in nine spaces, most of them would never occur in an actual game.
- Arcs are generated by legal moves of the game, alternating between placing an X and an O in an unused location.
- The state space is a graph rather than a tree, as some states on the third and deeper levels can be reached by different paths.
- However, there are no cycles in the state space, because the directed arcs of the graph do not allow a move to be undone.
- It is impossible to “go back up” the structure once a state has been reached.
- No checking for cycles in path generation is necessary.
- A graph structure with this property is called a directed acyclic graph, or DAG, and is common in state space search and in graphical models.

THE 8-PUZZLE

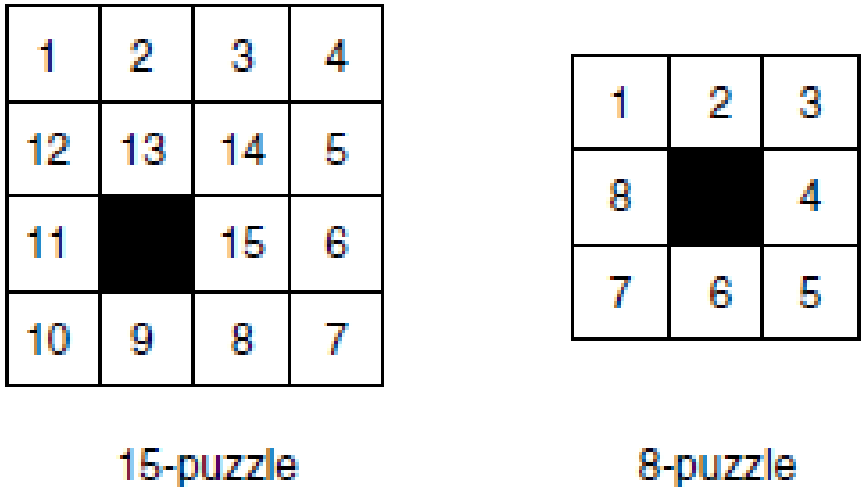


Figure 3.7 The 15-puzzle and the 8-puzzle.

- The legal moves are:

move the blank up ↑
move the blank right →
move the blank down ↓
move the blank left ←

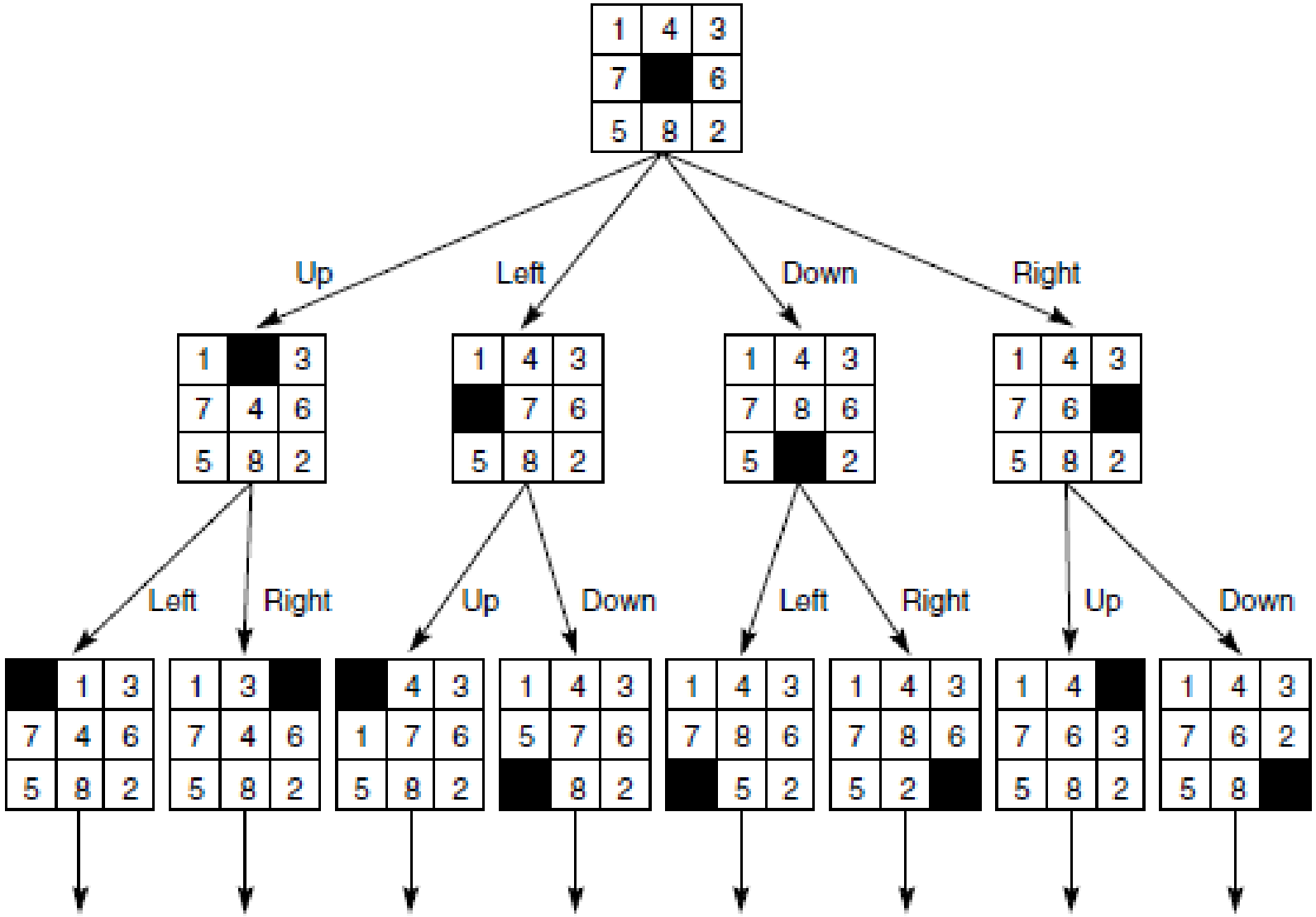


Figure 3.8 State space of the 8-puzzle generated by move blank operations.

Strategies for State Space Search

- Data driven Search
  - 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 (we hope!) it generates a path that satisfies the goal condition.

Cont..

- Goal driven Search
  - backward chaining (solve a maze by working back from the finish to the start)
  - take the goal that we want 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.
  - Search continues, working backward through successive subgoals until (we hope!) it works back to the facts of the problem.
  - This finds the chain of moves or rules leading from data to a goal, although it does so in backward order.

Data driven search

- It uses the knowledge and constraints found in the given data to guide search along lines known to be true
- All or most of the data are given in the initial problem statement. Interpretation problems often fit this mold by presenting a collection of data and asking the system for interpretation.
- There are large no. 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.

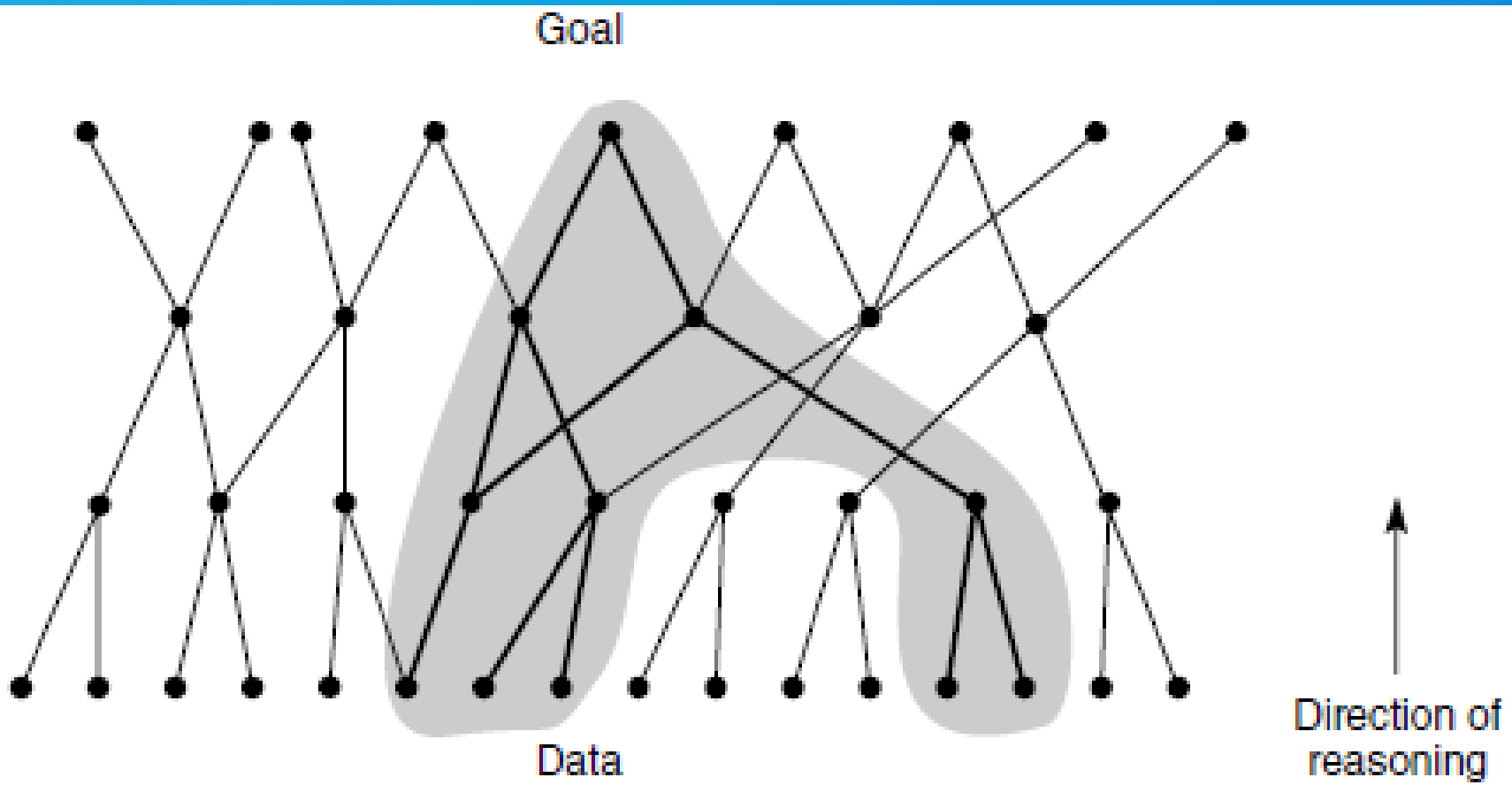


Figure 3.13 State space in which data-directed search prunes irrelevant data and their consequents and determines one of a number of possible goals.

EXAMPLE

“I AM DESCENDENT OF PAZHASSI RAJA”

Goal driven search Usage

- A goal is given in the problem statement or can easily be formulated.
  - In a mathematics theorem prover, for example, the goal is the theorem to be proved.
- There are large no. of rules that match the facts of the problem and thus produce an increase in no of conclusions or goals.
  - In a mathematics theorem prover, for example, the total no of rules that may be applied to the entire set of axioms.
- Problem data are not given and must be acquired by the problem solver.
  - In a medical diagnosis program,for example, a wide range of diagnostic tests can be applied. Doctors order only those that are necessary to confirm or deny a particular hypothesis.

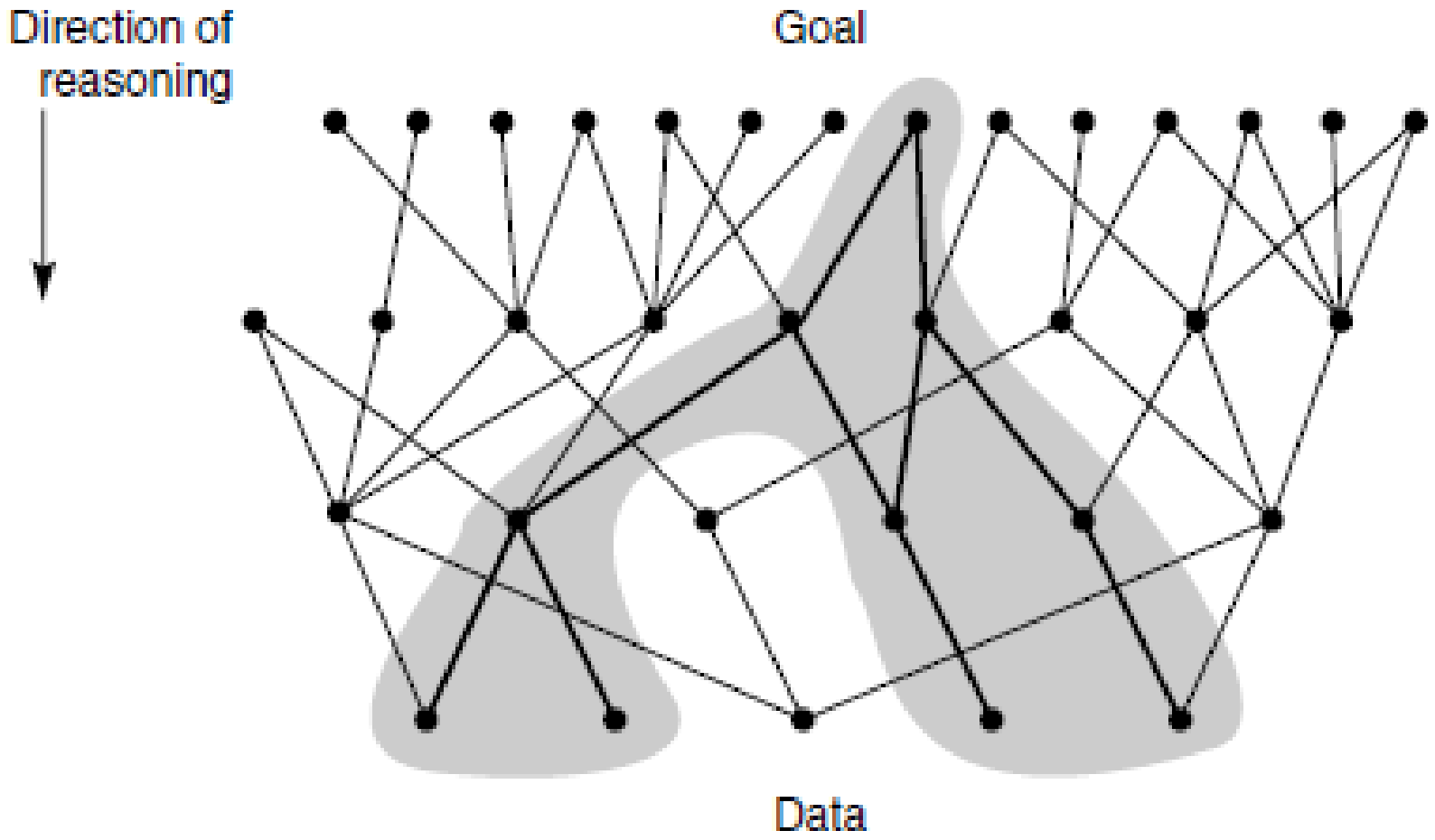
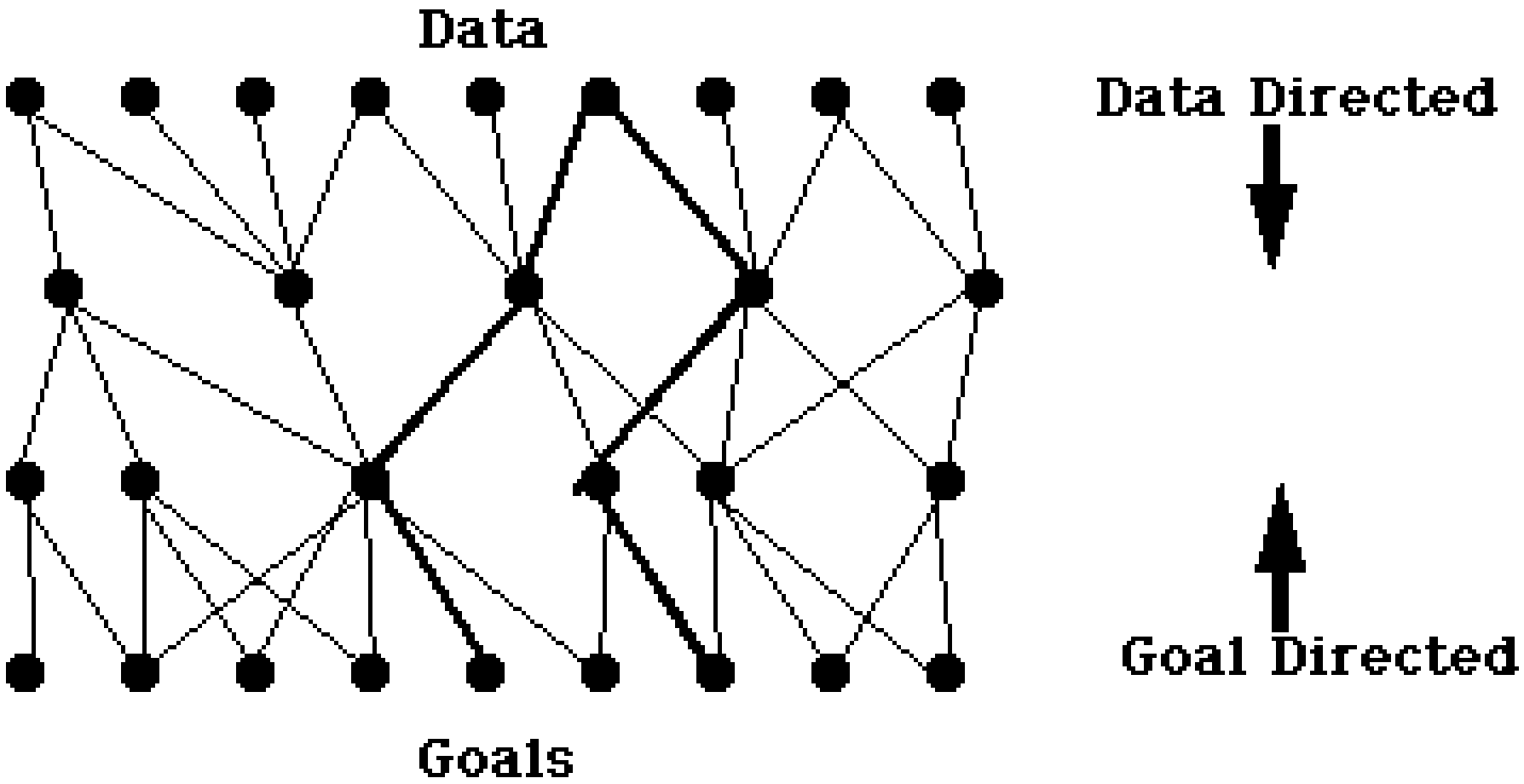


Figure 3.12 State space in which goal-directed search effectively prunes extraneous search paths.


Data
Data Directed
Goal Directed
Goals

Uninformed search methods

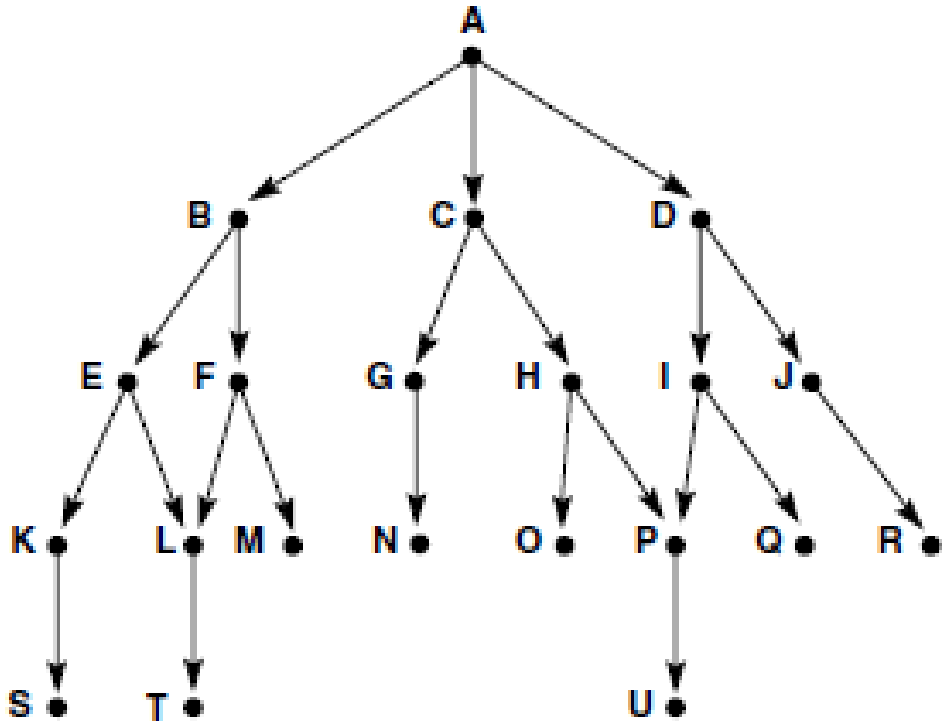
- Depth-first search
- Breadth-first search
- DFS with iterative deepening

Depth-First Search

- 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 arc its siblings considered.

Cont..

- Depth-first search examines the states in the graph of Figure in the order A, B, E, K, S, L, T, F, M, C, G. N, H, 0, P, U, D, I, Q, J, R. The backtrack algorithm of implemented Depth-First search.



DFS-Algorithm

```
function depth_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 left end of open            % stack
            end
      end;
   return FAIL                                               % no states left
end.
```

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]
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]

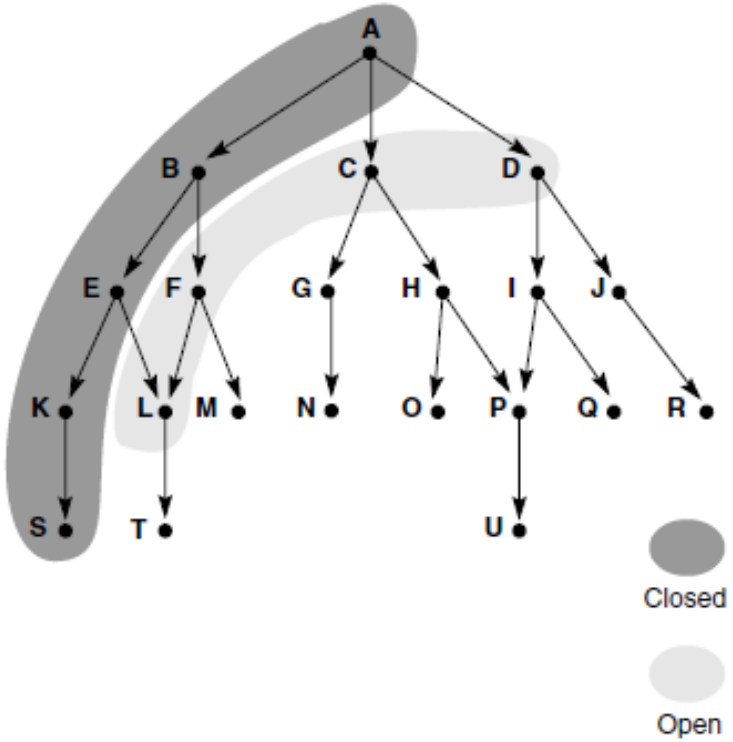
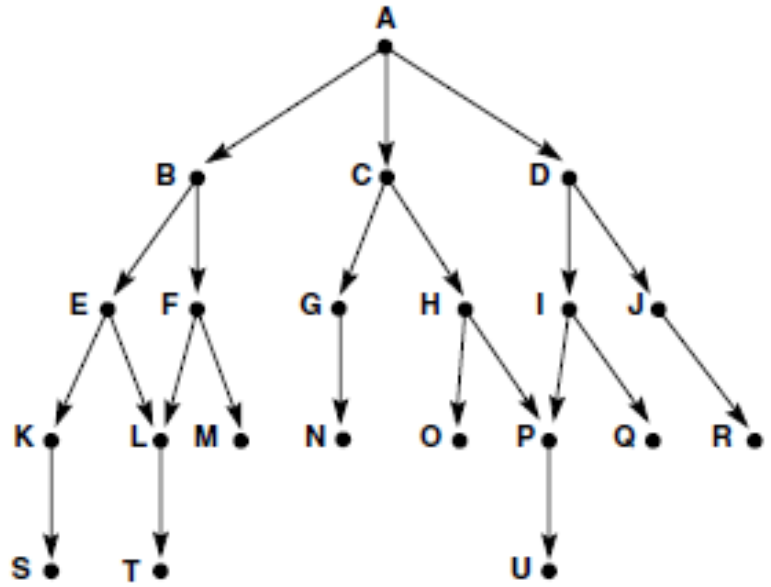


Figure 3.18 Graph of Figure 3.15 at iteration 6 of depth-first search. States on open and closed are highlighted.

Breadth-First Search

- Breadth-first search, in contrast, explores the space in a level-by-level fashion. Only when there are no more states to be explored at a given level does the algorithm move on to the next level.
- A breadth-first search of the graph of Figure considers the states in the order A, B, C, D, E, F, G, H, I, J, K, L, M, N, 0, P, Q, R, S, T, U.



BFS-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.
```

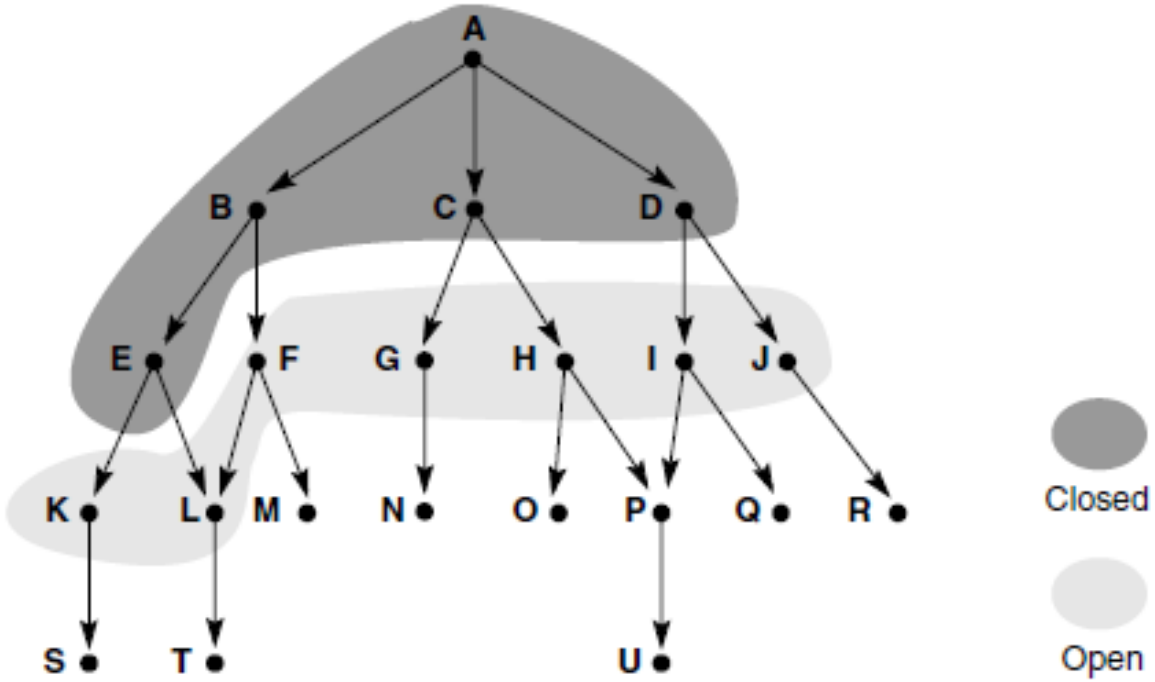


Figure 3.16 Graph of Figure 3.15 at iteration 6 of breadth-first search. States on open and closed are highlighted.

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 = [ ].

BFS in 8-puzzle-an example

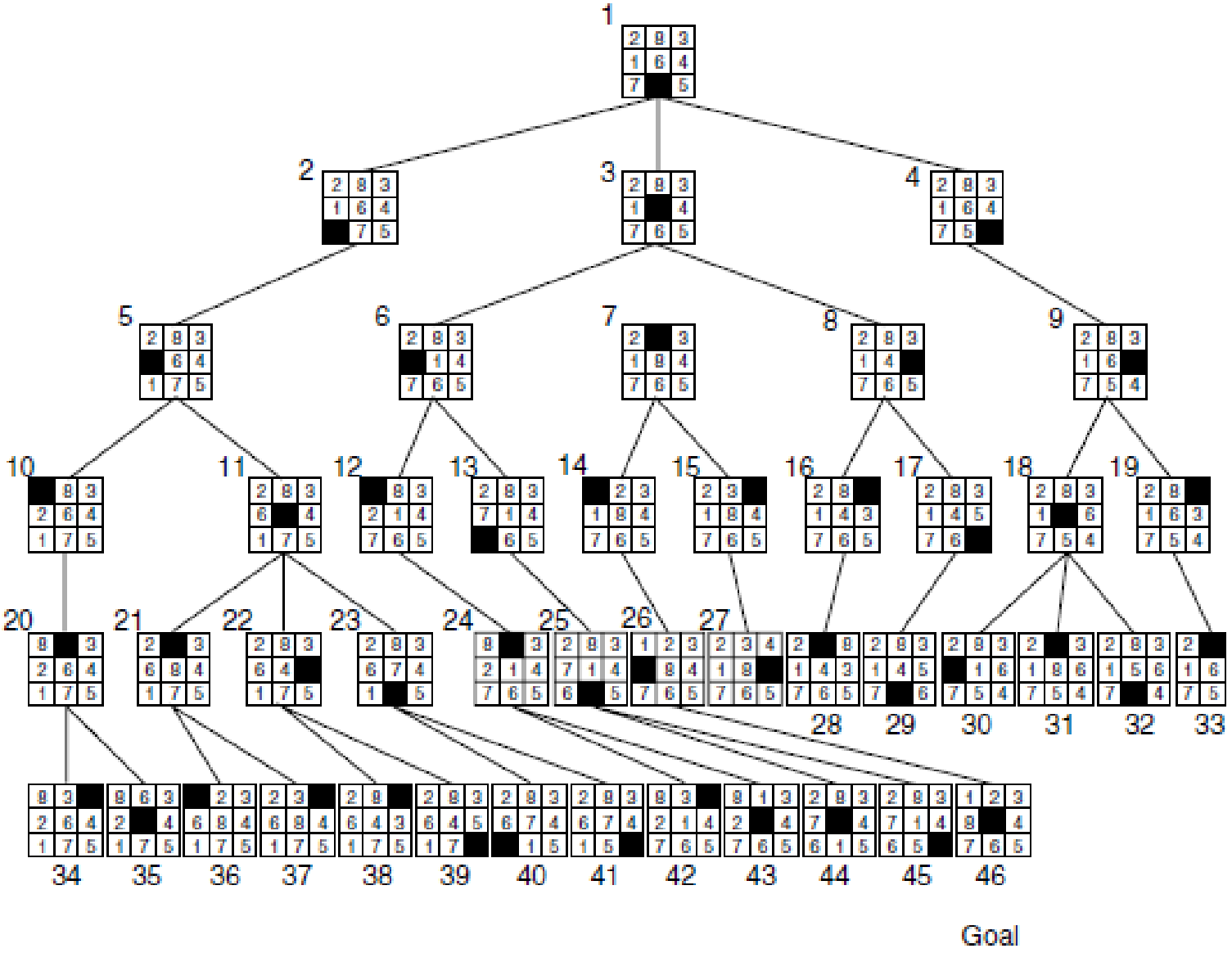


Figure 3.17 Breadth-first search of the 8-puzzle, showing order in which states were removed from open.

DFS in 8-puzzle-an example

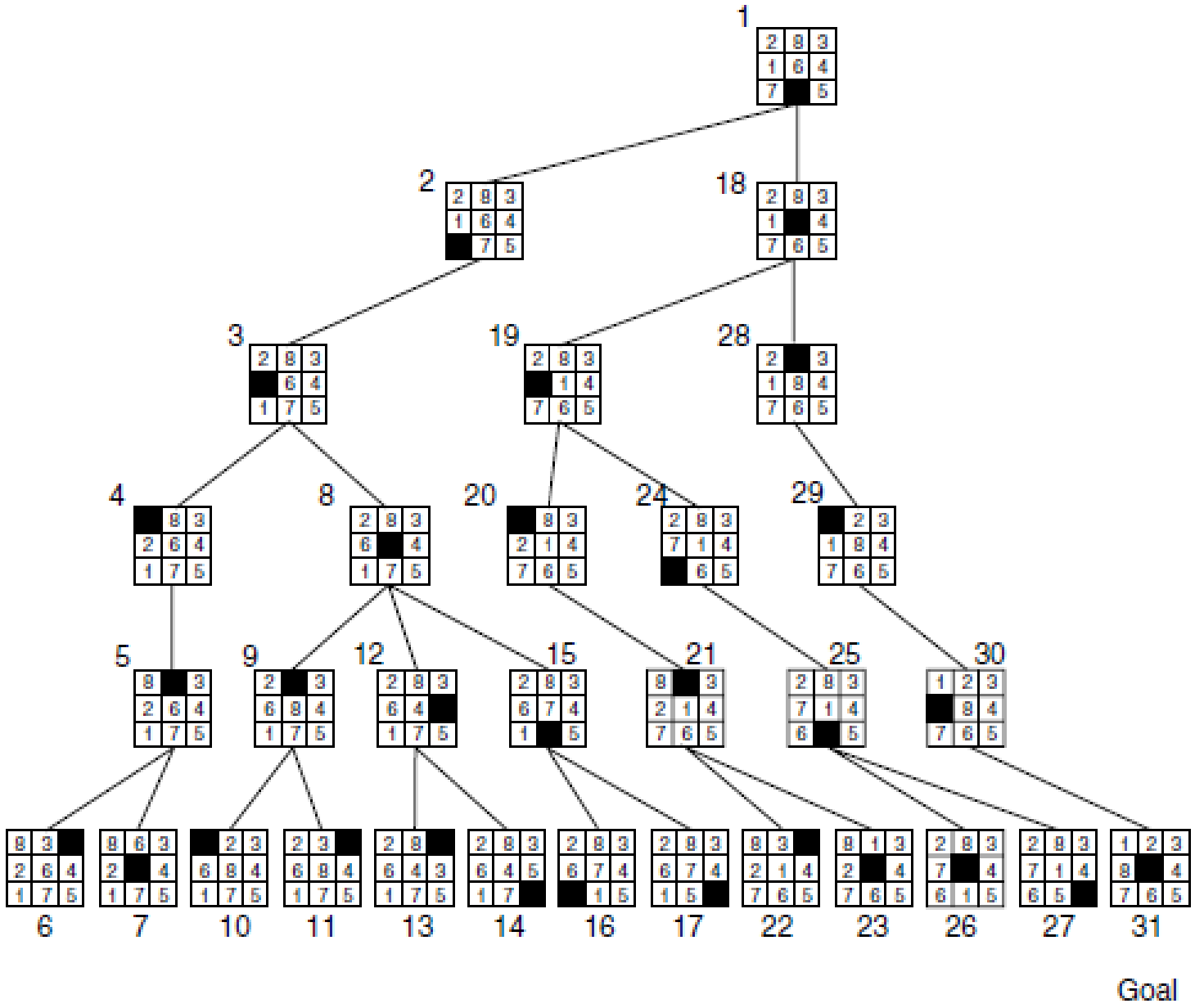


Figure 3.19 Depth-first search of the 8-puzzle with a depth bound of 5.

BFS

- **Advantages:**
  - Breadth-First search always examines all the nodes at level n before proceeding to level n + 1,
  - BFS always finds the shortest path to a goal node.
  - In a problem where it is known that a simple solution exists, this solution will be found.
- **Limitations**:
  - The space utilization of breadth-first search, measured in terms of the number of states in open, is an exponential function of the length of the path at any time. If each state has an average of B children, the number of states on a given level is B times the number of states in the previous level. This gives B" states on level n.
  - Breadth-first search would place all of these on open when it begins examining level n. This can be prohibitive if solution paths are long.

DFS

- **Advantages**:
  - Depth-first search is much more efficient for search spaces with many branches because it does not have to keep all the nodes at a given level on the open list.
  - At each level, open retains only the children of a single state. If a graph has an average of B children per state, this requires a total space usage of B x n states to go n levels deep into the space.
- **Limitations**:
  - depth-first search can get “lost” deep in a graph, missing shorter paths to a goal or even becoming stuck in an infinitely long path that does not lead to a goal.

Depth-First Search with Iterative Deepening(IDS or IDDFS)

Limit the number of states that can be considered

- use a depth bound on depth-first search.
- Performs a depth-first search of the space with a depth bound of 1.
- If it fails to find a goal, it performs another depth-first search with a depth bound of 2.
- This continues, increasing the depth bound by one each time.
- At each iteration, the algorithm performs a complete depth-first search to the current depth bound.

Heuristic search

- Heuristic search is an AI search technique that employs heuristic for its moves.
- A heuristic is only an informed guess of the next step to be taken in solving a problem.
- A heuristic can lead a search algorithm to a sub-optimal solution or fail to find any solution at all.
  - the heuristic measure
  - an algorithm that uses heuristics to search the state space.

Best First Search

“Best First Search is a way of combining the advantages of both depth first search and breadth first search method.”

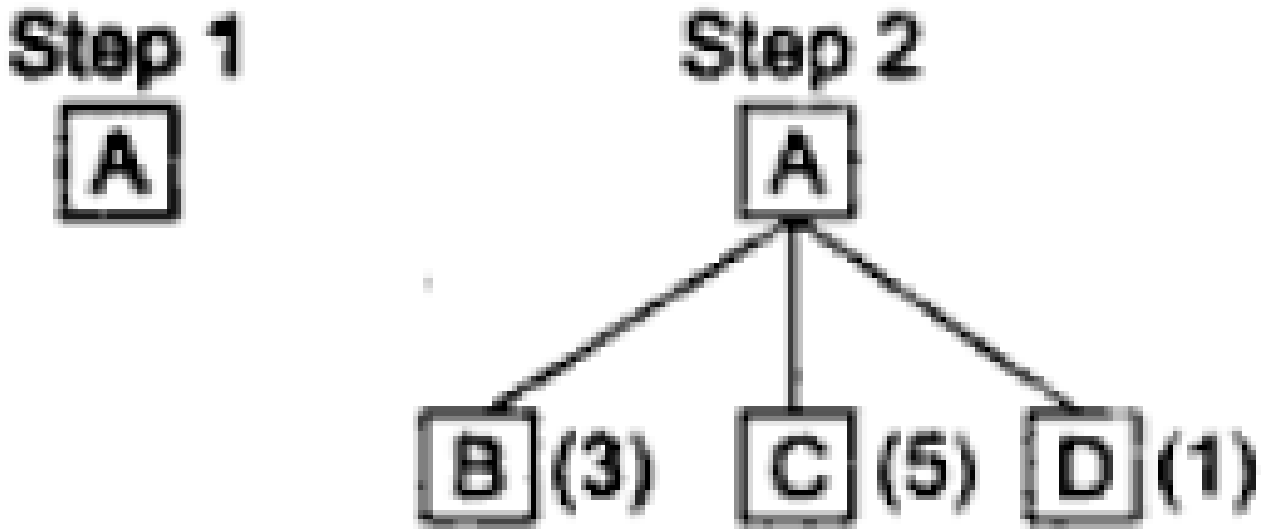
OR Graphs

- Depth first search is good because it allows a solution to be found without all competing branches having to be expanded.
- Breadth First Search is good because it does not get trapped in dead end paths.
- One way of combining the two is to follow a single path at a time, but to switch paths whenever some competing path looks more promising than the current one.

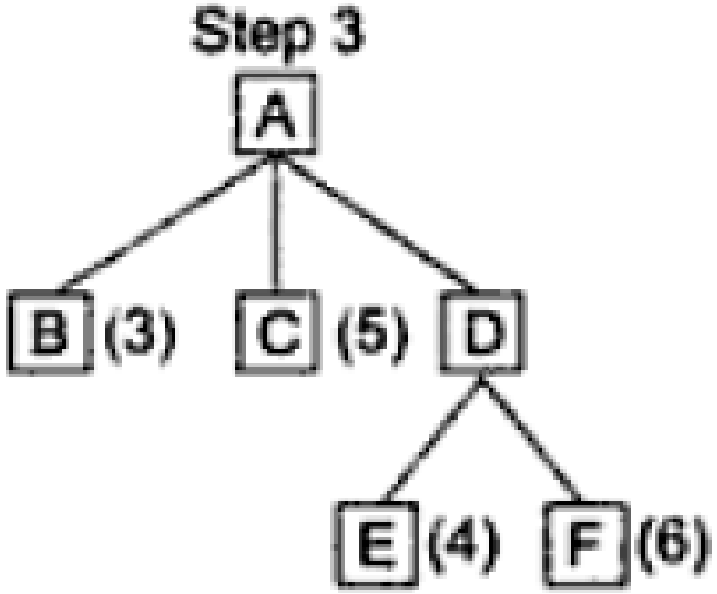
Cont..

- At each step of select the most promising of the nodes by applying an appropriate heuristic function to each of them.
- Then expand the choosen node by using rules to generate its sucessors.
  - If one of them is a solution we can quit.
  - If not all those new nodes are added to set of nodes generated so far.
  - A bit of Depth First Searching occurs as the most promising branch is explored.
  - If a solution is not found, that branch will start to be less promising than one of the top level branches that had been ignored.
  - The old branch is not forgotten.
  - Its last node remain in the set of generated but unexpected nodes.

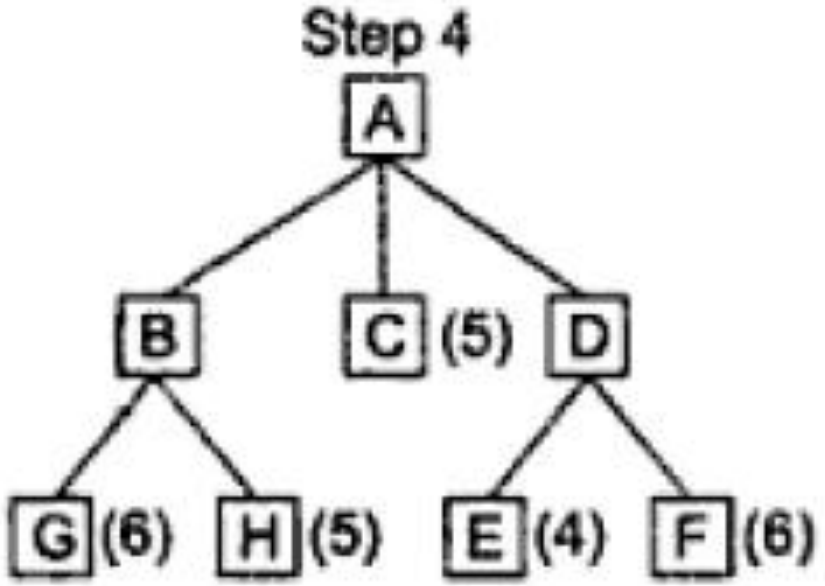
Cont..



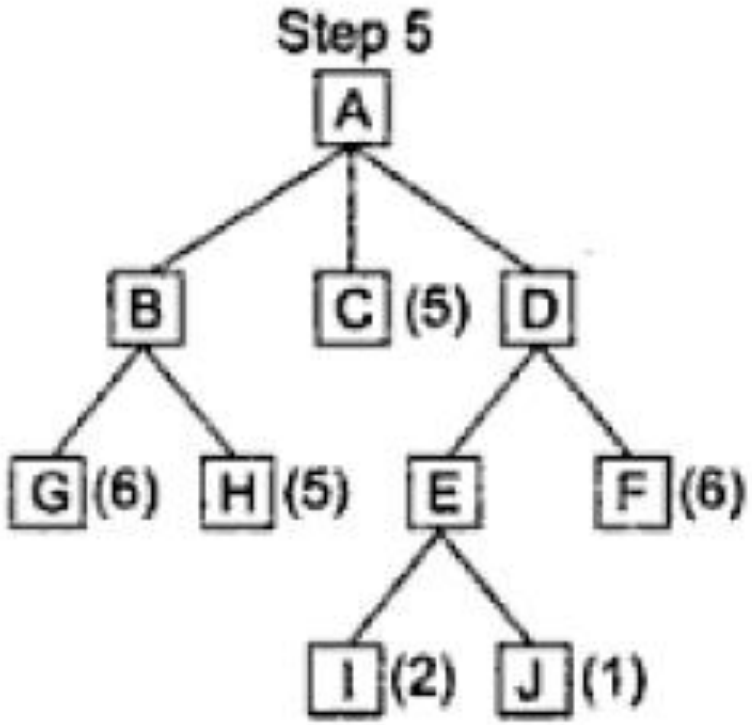
Cont..



Cont..



Cont..



Hill Climbing vs Best Fit

- This procedure is very similar to steepest ascent hill climbing with two exceptions:
  - In hill climbing one move is selected all others are rejected never reconsidered
  - Best available state is selected in best fit search even if state has value lower than value of state just explored. Whereas hill climbing will stop if there is no successor states with better value than current state

Directed Graph Searching (OR Graph)

- search a graph a graph instead of a tree so that duplicate paths will not be taken.
- searching a directed graph in which each node represent a point in problem space.
- Each node will contains:
  - addition to a description of problem state it represents,
  - an indication of how promising it is,
  - a parent link that points back to best node it came from, and
  - a list of nodes that were genenrated from it

Cont..

- The parent link will make it possible to recover the path to goal once goal is found.
- The list of successors will make it possible if a better path is found to an already existing node, to propogate improvement down to its successors.

We call this an OR-Graph since each of this branch represents an alternate problem solving path

Cont..

- To implement this we need two list of nodes
  1. OPEN
     - Nodes that have been generated and have had the heuristic function applied to them but which have not yet been examined(ie had their successors generated).
     - OPEN is a priority queue in which the element with highest priority are those with most promising value of heuristic function.
  2. CLOSED
     - Nodes that have already been examined.
     - We need to keep these in memory if we want to search a graph rather than a tree, since whenever a new node is generated we need to check if it is generated before.

Heuristic Function

- The heuristic function must be able to search more promising paths first.

$$f'=g+h'$$

- *g*
  - measure of cost of getting from initial state to current node.
  - It is not an estimation
  - It is exact sum of cost of applying each of the rules that were applied along the best path to the node
  - Must be non negative else graph will traverse in a cycles which will appear to get better but will be getting longer

Cont..

- *h'*
  - It is an estimate of additional cost of getting from current node to goal state.
  - Good one get low value, bad one get high value
- *f' = g + h'*
  - f' represents an estimate of cost of getting from initial state to goal state along the path generated from current node
  - If more than one path generated the node, the best path is recorded

Actual Operation

- Proceed in steps, expanding each node at each step, until it generate a goal state node
- At each step pick the most promising node that have so far been generated but not expanded.
  - It generate sucessor of the choosen node
  - Apply heuristic function to them
  - Add them to list of open nodes, after checking to see if any of them have been generated before. By doing above check we guarentee that each node appear exactly once in the graph, although many nodes may point to its successors

Algorithm Best First Search

1. Start OPEN containing initial state
2. Until goal is found or there are no nodes present in OPEN do:
   a) Pick up the best node on OPEN
   b) Generate its successors
   c) For each successor do
      i. If it has not been generated before, evaluate it add it to open, record its parent
      ii. If it has been generated before, change the parent if the new path is better than previous one. In that case update the cost of getting to this node and to any of its successors that this node may already have.

The A* Algorithm

- The best first search algorithm is a simplification of A* algorithm.
- This algorithm uses the same heuristic functions f’, g and h’ and also the list OPEN and CLOSED.

Algorithm A*

1. Start with OPEN containing only the initial node. Set that nodes g value to 0, and its h' value to whatever it is. And its f' value to h'+0 ie h'. Set CLOSED to an empty list
2. Until a goal node is found, repeat the following procedure:

   If there is no node on OPEN, report failure

   Otherwise pick a node on OPEN with lowest f' value. Call it BESTNODE. Remove it from OPEN. Place it on CLOSED.

   See if BESTNODE is goal node. If so exit and report solution.

   Otherwise generate successor of BESTNODE but do not set BESTNODE to point to them yet.

Cont..

For each such SUCCESOR do the following:

a) Set SUCCESOR to point back to BESTNODE. These backward links will make it possible to recover the path once a solution is found.

b) Compute g(SUCCESOR) = g(BESTNODE) + cost of getting from BESTNODE to SUCCESOR

Cont..

c) See if successor is same as any node in OPEN

If so call that node OLD throw SUCCESOR away and add OLD to list of BESTNODE's successors.

Now we must decide wheather it is cheaper to get to OLD via its current parent or to SUCCESOR via BESTNODE comparing their g values.

If OLD is cheaper then do nothing.

else if SUCCESOR then reset OLD's parent link to point to BESTNODE, record new cheaper path in g(OLD), and update f'(OLD).

Cont..

d) If SUCCESSOR was not on OPEN, see if it is on CLOSED. If so so call the node on CLOSED OLD node and add it to list of BESTNODES succesors.

Check to see if new path or old path is better and set parent link and g and f' function accordingly.

If we have found a better path to CLOSED OLD we must propogate improvements to OLD's Successor. This is a tricky process.

OLD points to its successors. Each successor inturn points to its Successors and so forth until each branch terminates with a node that is either is still on OPEN or has no successors.

while propogation check if path through current parent is still better than new path then stop propogation

Cont..

e) if SUCCESSOR was not already on either OPEN or CLOSED, then put it on OPEN add it to list of BESTNODES's successor.

Compute
f'(SUCCESSOR)=g(SUCCESSOR)+h'(SUCCESSOR)

Problem of under estimation and over estimation in A*

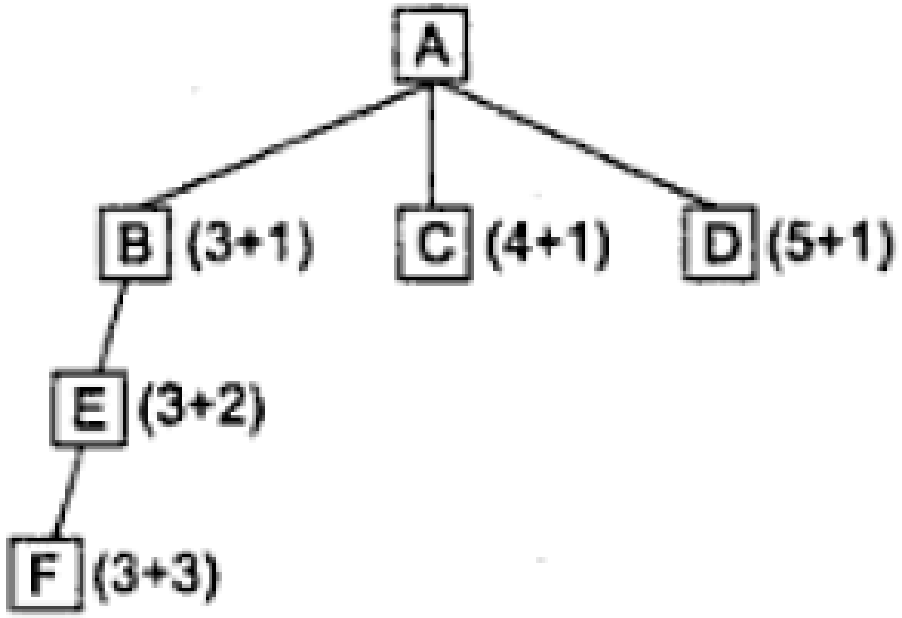


Fig. 3.4 *h' Underestimates h*

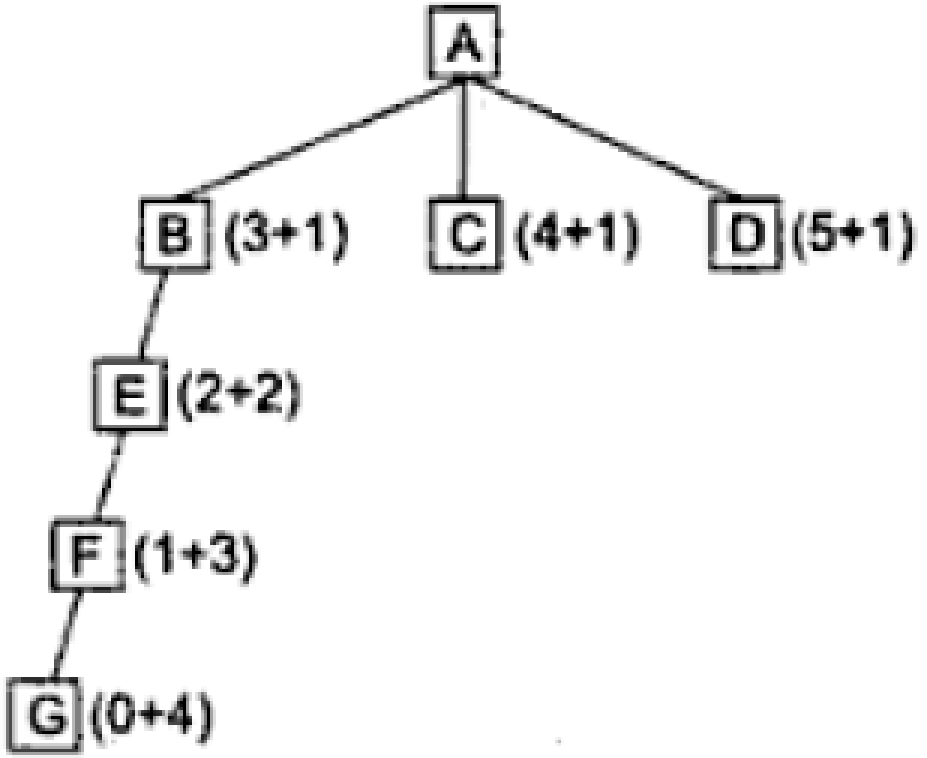


Fig. 3.5 *h' Overestimates h*

- Better h’ closer we will get to that direct approach
- If value of h’ is always 0, the search will be controlled by g
- If value of g is also 0 search strategy will be random
- If value of g is always 1 then search will be breadth first
  - All nodes on one level will have lower g value and thus lower f’ value than will all nodes on next level
- H’ is either perfect nor 0
  - Then we can guarantee that h’ never overestimates h
  - Then A* algorithm guarantees to find optimal path to goal,if one exists

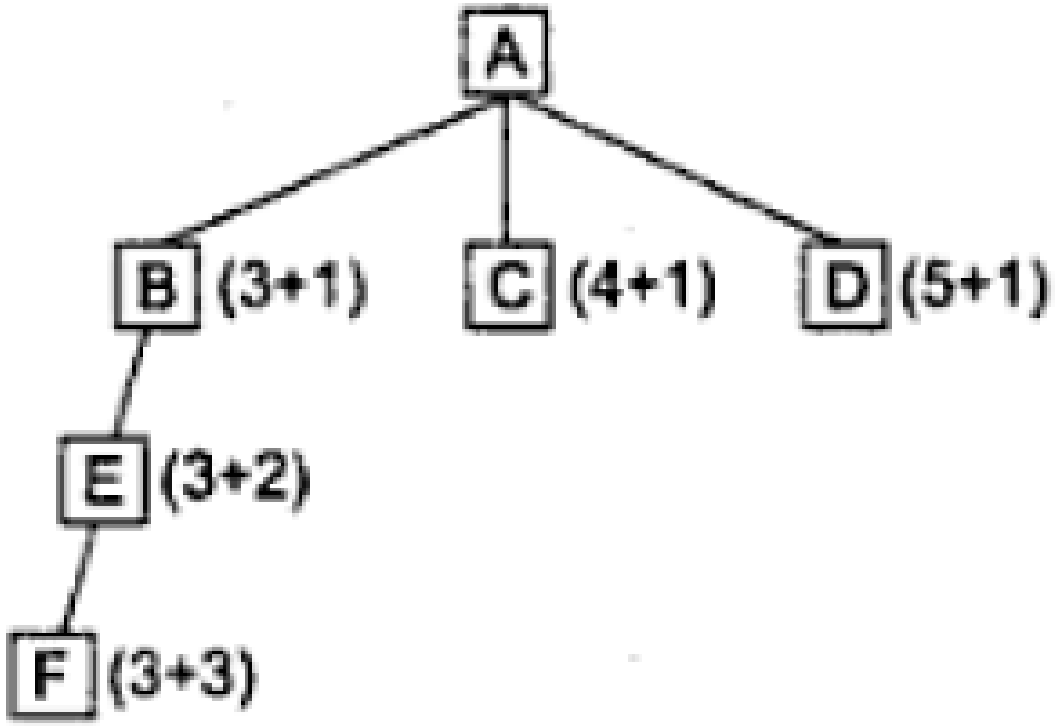


Fig. 3.4 *h' Underestimates h*

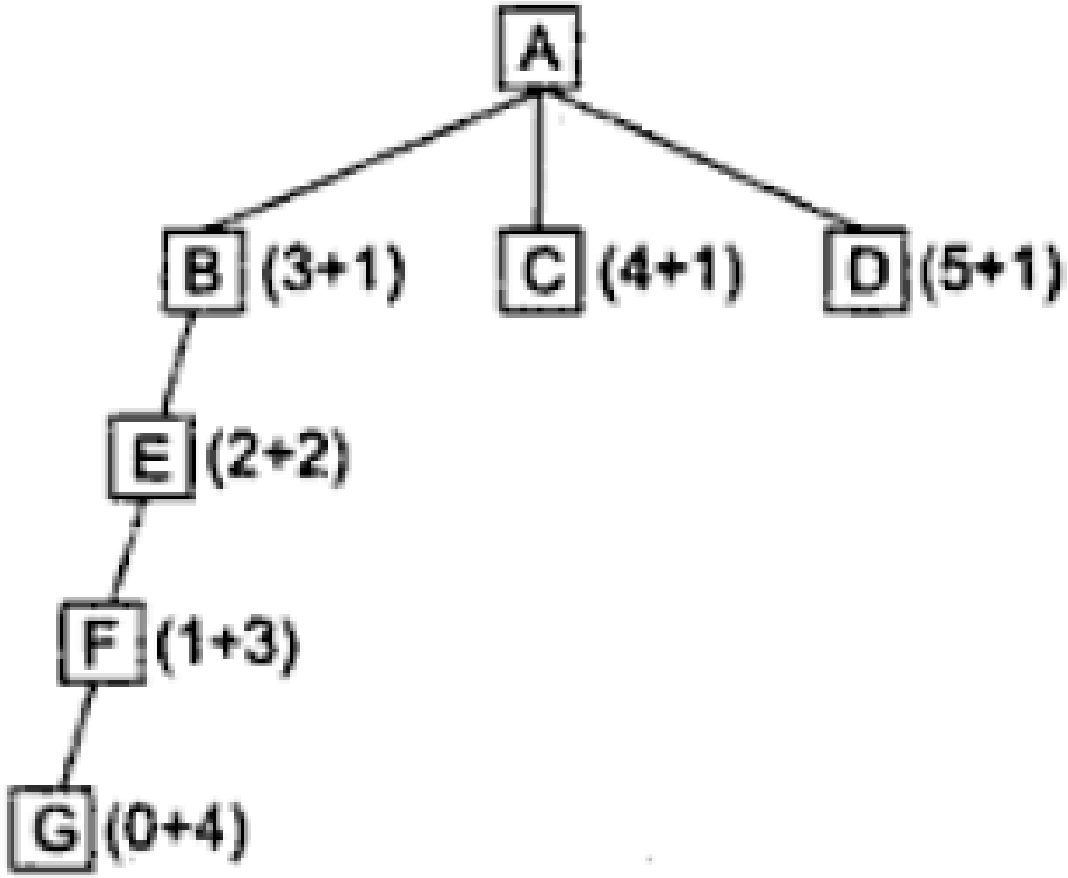


Fig. 3.5 *h' Overestimates h*

GRACEFUL DECAY OF ADMISSIBILITY

Graceful Decay of Admissibility: If h' rarely overestimates h by more than δ, then the A* algorithm will rarely find a solution whose cost is more than δ greater than the cost of the optimal solution.

Problem Reduction

- AND-OR Graphs
  - AND-OR graph is useful for representing the solution of problem that can be solved by decomposing them into a set of smaller problems, all of which must then be solved.
  - This decomposition, or reduction, generates arcs that we call AND arcs.
  - One AND arc may point to any number of successor nodes, all of which must be solved in order for the arc to point to a solution.
  - Just as in an OR graph, several arcs may emerge from a single node, indicating a variety of ways in which the problem might be solved.

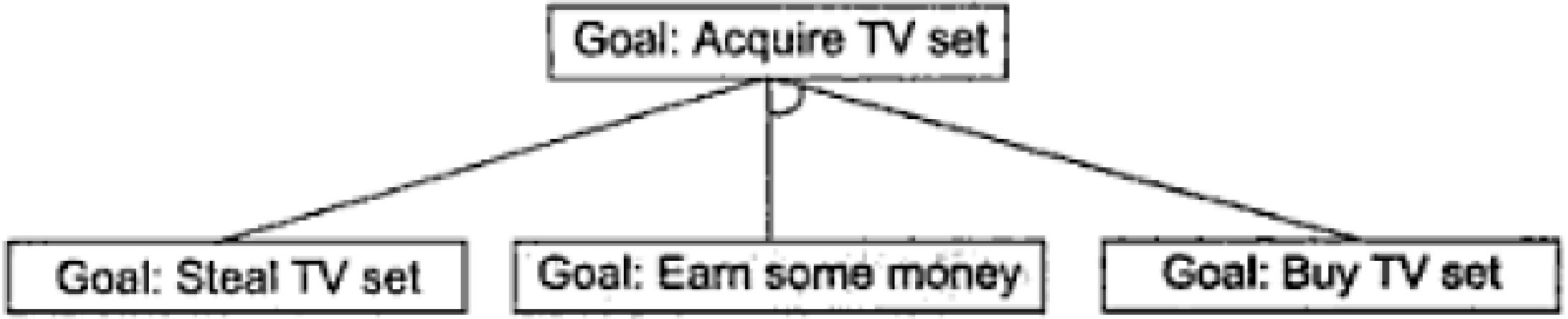
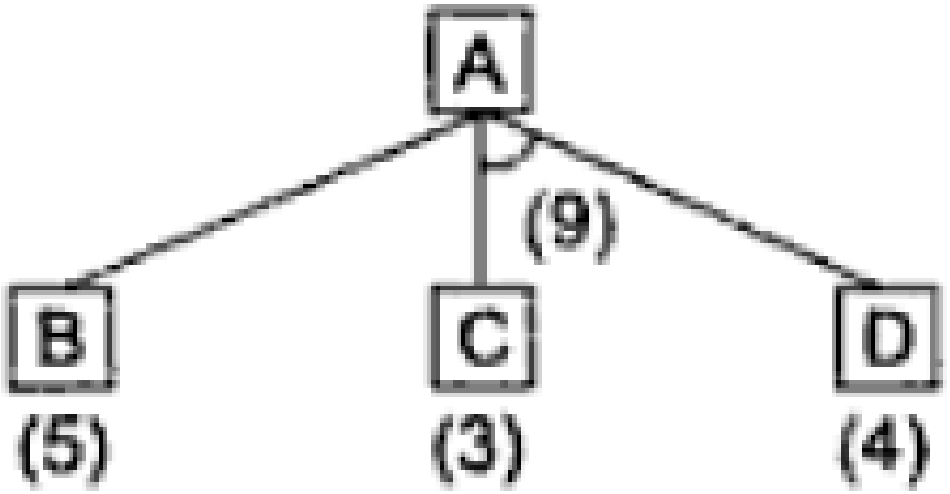


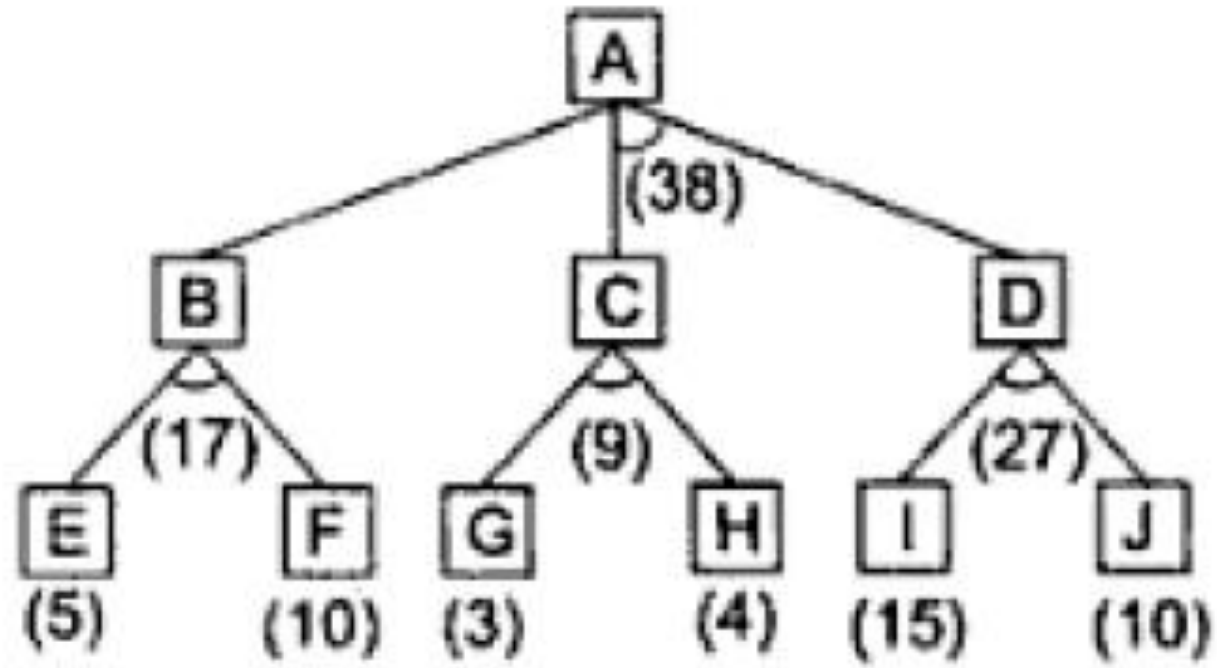
Fig. 3.6 *A Simple AND-OR Graph*

Cont..

- In order to find solutions in an AND-OR graph we need an algorithm similar to best-first search but with the ability to handle the AND arcs appropriately.
- This algorithm should find a path from starting node of the graph to a set of nodes representing solution state.
- It may be necessary to get more than one solution state since each arm of an AND arc must lead to its own solution node.



(a)



(b)

Cont..

- the top node A has been expanded producing two area one leading to B and leading to C-D .
- the numbers at each node represent the value of f ' at that node
- it is assumed that every operation has unit cost,
  - i.e., each are with single successor will have a cost of 1 and each of its components.
- it appears that C is the most promising node to expand since its f ' = 3 the lowest but going through B would be better since to use C we must also use D' and the cost would be 9(3+4+1+1). Through B it would be 6(5+1).

- the choice of the next node to expand depends not only n a value but also on whether that node is part of the current best path form the initial mode.
- In figure the node G appears to be the most promising node, with the least f ' value.
- But G is not on the current beat path, since to use G we must use GH with a cost of 9 and again this demands that arcs be used (with a cost of 27).
- The path from A through B, E-F is better with a total cost of (17+1=18).

- Thus we can see that to search an AND-OR graph, the following three things must be done.
  1. traverse the graph starting at the initial node and following the current best path, and accumulate the set of nodes that are on the path and have not yet been expanded.
  2. Pick one of these unexpanded nodes and expand it. Add its successors to the graph and computer f ' (cost of the remaining distance) for each of them.
  3. Change the f ' estimate of the newly expanded node to reflect the new information produced by its successors. Propagate this change backward through the graph. Decide which of the current best path.

- in order to describe the algorithm we need to exploit a value called FUTILITY
- If estimated cost of a solution become greater than value of FUTILITY, then we abandon the search

Algorithm: Problem Reduction

1. Initialize the graph to the starting node.
2. Loop until the starting node is labeled *SOLVED* or until its cost goes above *FUTILITY*:
   (a) Traverse the graph, starting at the initial node and following the current best path, and accumulate the set of nodes that are on that path and have not yet been expanded or labeled as solved.
   (b) Pick one of these unexpanded nodes and expand it. If there are no successors, assign *FUTILITY* as the value of this node. Otherwise, add its successors to the graph and for each of them compute f' (use only h' and ignore g, for reasons we discuss below). If of any node is 0, mark that node as *SOLVED*.
   (c) Change the f' estimate of the newly expanded node to reflect the new information provided by its successors. Propagate this change backward through the graph. If any node contains a successor arc whose descendants are all solved, label the node itself as *SOLVED*. At each node that is visited while going up the graph, decide which of its successor arcs is the most promising and mark it as part of the current best path. This may cause the current best path to change. This propagation of revised cost estimates back up the tree was not necessary in the best-first search algorithm because only unexpanded nodes were examined. But now expanded nodes must be reexamined so that the best current path can be selected. Thus it is important that their f' values be the best estimates available.

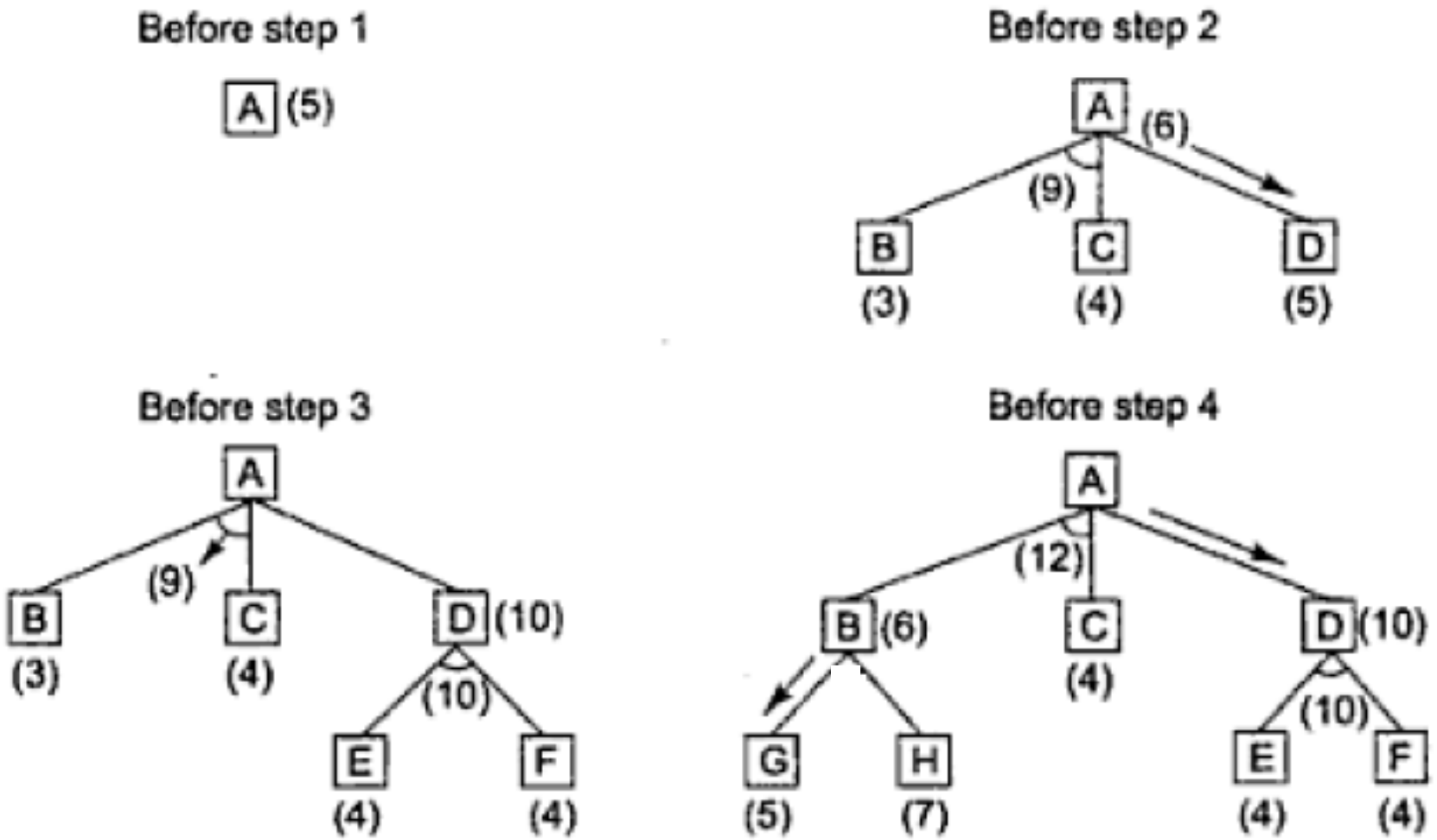


Fig. 3.8 *The Operation of Problem Reduction*

AO* Algorithm

- uses a single structure GRAPH, representing the part of the search graph that has been explicitly generated so far.
- Each node in the graph will point both down to its immediate successors and up to its immediate predecessors.
- Each node in the graph will also have associated with it an h’ value, an estimate of cost of a path from itself to a set of solution nodes.
- We will not store g as it is not possible to compute a single such value since there may be many path to the same state.

Algorithm AO*

1. Let GRAPH consist only of the node representing the initial state. Call this node INIT. Compute h'(INIT).

2. Until INIT is labelled SOLVED or until INIT's h' value become greater than FUTILITY, repeat the following procedure:
   a) trace the labeled arcs from INIT and select for expansion one of the as yet unexpanded nodes that occurs on this path. Call this the selected node NODE

b) Generate the successor of NODE. If there are none, then assign FUTILITY as the h' value of NODE. This is equivalent to saying that NODE is not solvable . If there are successors, then for each one that is not also an ancestor of NODE do the following:

i. Add SUCCESSOR to the GRAPH

ii. If the SUCCESSOR is a terminal(goal) node, label it SOLVED and assign it an h' value of 0

iii. If SUCCESSOR is not a terminal node, compute its h' value

c) Propagate the newly discovered information up the graph by doing the following. Let S be a set of nodes that have been labelled SOLVED or whose h’ value have been changed and so need to have values propagated back to their parents. Initialize S to NODE. Until S is empty, repeat the following procedures

i. If possible, select from S a node, none of whose descendants in the GRAPH occurs in S. If there is no such node, select any node from S. Call this node CURRENT and remove it from S

ii. Compute the cost of each of the arcs emerging from CURRENT. The cost of each arc is equal to the sum of the h' values of each of the nodes at the end of the arc plus whatever the cost of the arc itself is. Assign as CURRENT's new h' value the minimum of costs just computed for the arcs emerging from it.

iii. Mark the best path out of current by marking the arc that had the minimum cost as computed in previous step.

iv. Mark CURRENT as SOLVED if all of the nodes connected to it through the new labeled arc have been labelled SOLVED.

v. If CURRENT has been labelled SOLVED or if the cost of CURRENT was changed then its new status must be propagated back up the GRAPH. So add all of the ancestors of CURRENT to S.

Unnecessary backward Propogation

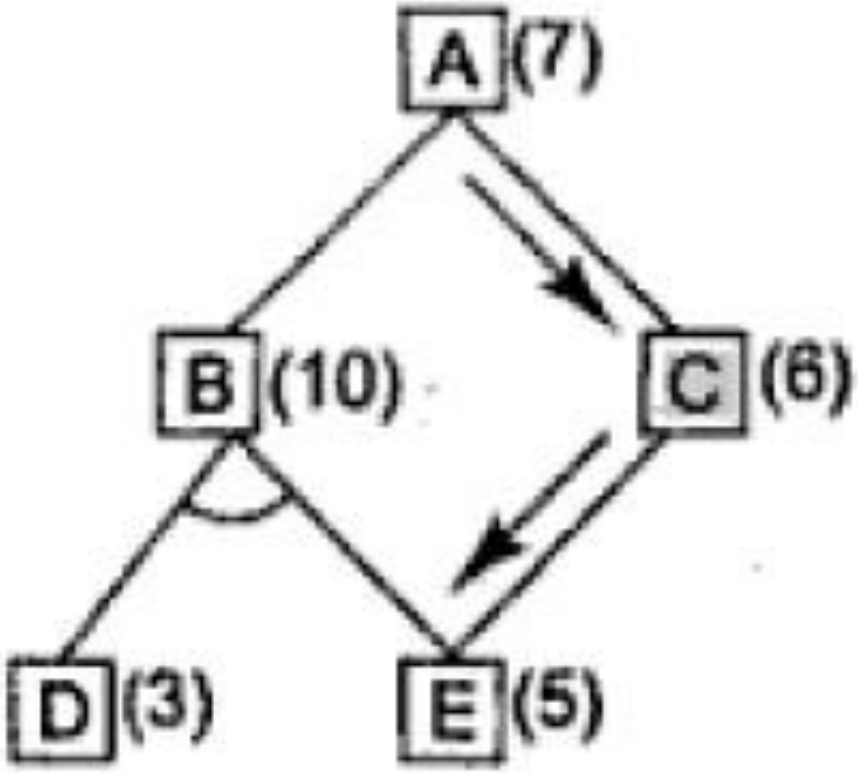
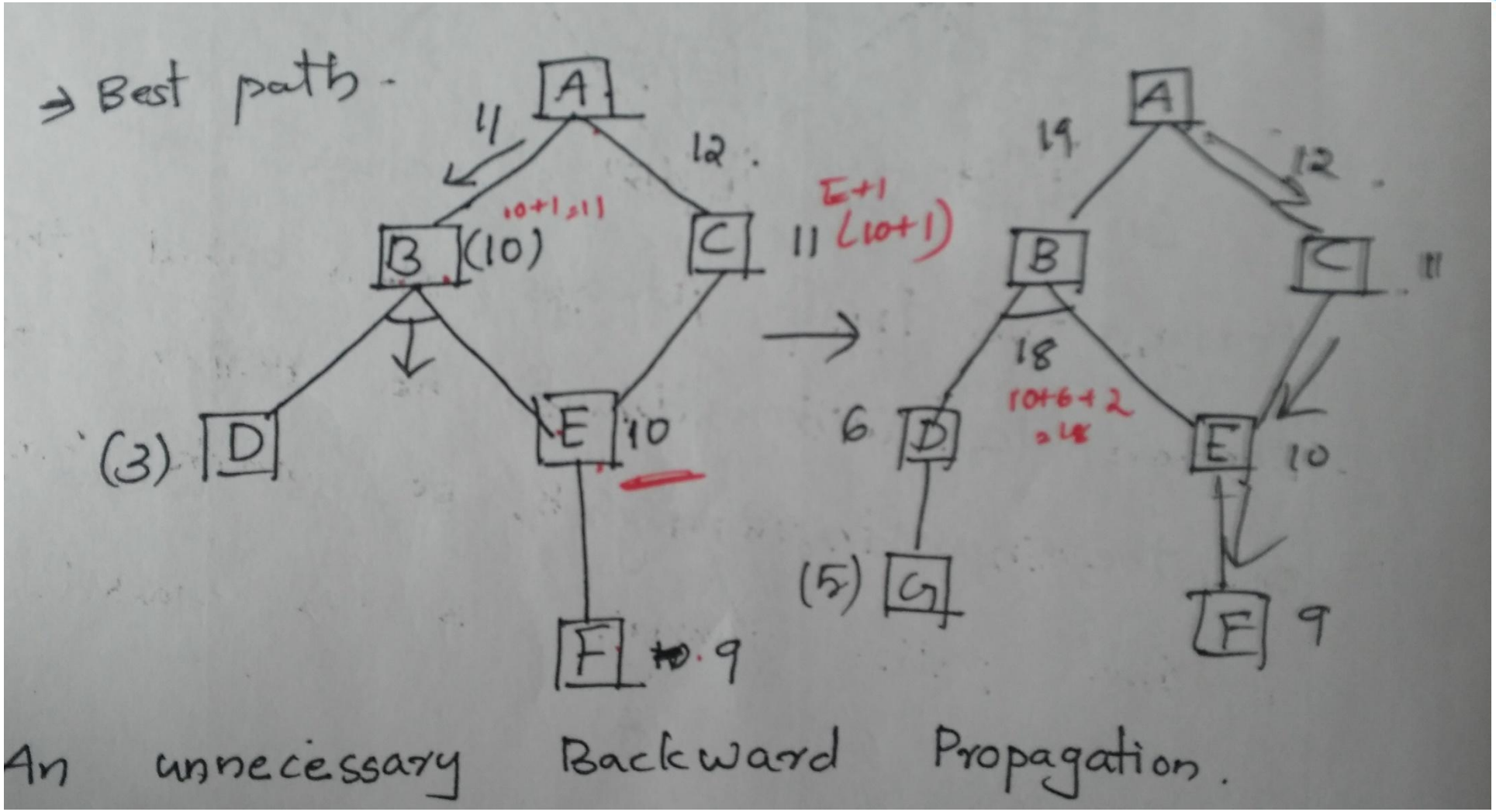


Fig. 3.11 *An Unnecessary Backward Propagation*


⇒ Best path.
A
11
12
B
(10)
10+1=11
C
11
E+1
(10+1)
(3)
D
E
10
F
9
A
19
12
B
C
11
18
10+6+2
=18
6
D
E
10
(5)
G
F
9
An unnecessary Backward Propagation.

Constraint Satisfaction

- Discover some problem state that satisfies a given set of constraints

Constraint satisfaction is a search procedure that operates in a space of constraint sets

- Initial state:
  - contains the constraint that are originally given in the problem description.
- Goal state:
  - is any state that has been constrained “enough” where “enough” must be defined for each problem.

Cont..

- Constraint satisfaction is a two step process.
  1. First, constraints are discovered or propagated as far as possible throughout the system. Then if there is still not a solution search begins.
  2. A guess about something is made and added as a new constraint. Propagation can then occur with this new constraint.

Cont..

- eg, one constraint, N=E+1
- added constraint N=3
- We could propagate that to get a stronger constraint on E, E=2

Cont..

- Constraint propagation also arise from the presence of inference rules that allow additional constraint to be inferred from the ones that are given.
- Constraint propagation terminates for one of the two reasons:
  - A contradiction may be detected. If the contradiction involves only those constraints that were given as part of problem specification then no solution exists.
  - A propagation has run out of stream and there are no further changes that can be made on the basis of current knowledge.
- At this point, the second step begins. Some hypothesis about a way to strengthen the constraints must be made.

Cryptarithmetic Problem

```
   S  E  N  D
+  M  O  R  E
---------------
M  O  N  E  Y
```

- No two letter have same value.
- The sum of digits must be as shown in problem.

Cont..

0 1 2 3 4 5 6 7 8 9

S E N D
M O R E

M O N E Y

Cont..

0	1	2	3	4	5	6	7	8	9
O	**M**								**S**

M is a carry so M=1
S+M+C3>9
S+1+C3>9
S+C3>8
S=8/9 (C3=0/1)
S=8 eliminated(8+1+0=9)
so C3=0,S=9, O=0

	C3=0	C2	C1	
	S	E	N	D
	M	O	R	E
M	O	N	E	Y

Cont..

0	1	2	3	4	5	6	7	8	9
O	**M**							**R**	**S**

0+E+C2=N

E+C2=N (C2=0/1)

C2=0 Eliminated (N+R=E, makes R=0 is not true)

So C2=1 And E+1=N -(1)

R+N+C1=E+10XC2 (C2=1)

R+N+C1=E+10 -(2)

R+C1=9

C1=0/1 TAKE C1=1, R=8

C3=0 C2=1 C1=1

	9	E	N	D
	1	0	R	E
1	0	N	E	Y

Cont..

0	1	2	3	4	5	6	7	8	9
O	**M**							**R**	**S**

D+E=Y+10

Possible combinations

7+3=10 ✖

7+4=11 ✖

7+5=12 possible making Y=2

E=5, N=6, D=7

C3=0 C2=1 C1=1

	9	E	N	D
	1	0	8	E
1	0	N	E	Y

Cont..

0	1	2	~~3~~	~~4~~	5	6	7	8	9
O	**M**	**Y**			**E**	**N**	**D**	**R**	**S**

C3=0 C2=1 C1=1

9	5	6	7
1	0	8	5

1	0	6	5	2

Algorithm: Constraint Satisfaction

1. Propagate available constraints. To do this first set OPEN to set of all objects that must have values assigned to them in a complete solution. Then do until an inconsistency is detected or until OPEN is empty.
   a) Select an object OB from OPEN. Strengthen as much as possible the set of constraints that apply to OB.
   b) If this set is different from the set that was assigned the last time OB was examined or if this is the first time OB has been examined, then add to OPEN all objects that share any constraint with OB
   c) Remove OB from OPEN

Cont..

2. If the union of constraint discovered above defines a solution, then quit and report the solution.
3. if the union of the constraints discovered above define a contradiction, then return failure .

Cont..

4. If neither of the above occurs, then it is necessary to make a guess at something in order to proceed. To do this loop until a solution is found or all possible solution have been eliminated.
   a) Select an object whose value is not yet determined and select a way of strengthening the constraints on that object.
   b) recursively invoke constraint satisfaction with the current set of constraints augmented by strengthening constraint just selected