3.2.3 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. This section considers two possibilities for the order in which the nodes of the graph are considered: *depth-first* and *breadth-first* search.

Consider the graph represented in Figure 3.15. States are labeled (A, B, C, . . .) so that they can be referred to in the discussion that follows. 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 are its siblings considered. Depth-first search examines the states in the graph of Figure 3.15 in the order A, B, E, K, S, L, T, F, M, C, G, N, H, O, P, U, D, I, Q, J, R. The backtrack algorithm of Section 3.2.2 implemented depth-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 deeper level. A breadth-first search of the graph of Figure 3.15 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.

We implement breadth-first search using lists, open and closed, 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.

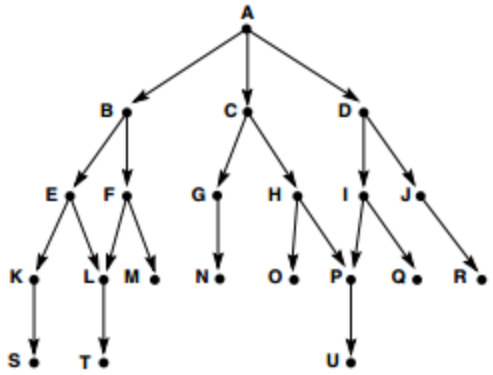

Figure 3.15 Graph for breadth- and depth-first search examples.

```
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.
```

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 and 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. This biases search toward the states that have been on open the longest, causing the search to be breadth-first. Child states that have already been discovered (already appear on either open or closed) are discarded. If the algorithm terminates because the condition of the "while" loop is no longer satisfied (open = [ ]) then it has searched the entire graph without finding the desired goal: the search has failed.

A trace of breadth_first_search on the graph of Figure 3.15 follows. 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 = [ ].

Figure 3.16 illustrates the graph of Figure 3.15 after six iterations of breadth_first_search. The states on open and closed are highlighted. States not shaded have not been discovered by the algorithm. Note that open records the states on the "frontier" of the search at any stage and that closed records states already visited.

Because breadth-first search considers every node at each level of the graph before going deeper into the space, all states are first reached along the shortest path from the start state. Breadth-first search is therefore guaranteed to find the shortest path from the start state to the goal. Furthermore, because all states are first found along the shortest path, any states encountered a second time are found along a path of equal or greater length. Because there is no chance that duplicate states were found along a better path, the algorithm simply discards any duplicate states.

It is often useful to keep other information on open and closed besides the names of the states. For example, note that breadth_first_search does not maintain a list of states on the current path to a goal as backtrack did on the list SL; all visited states are kept on closed. If a solution path is required, it can not be returned by this algorithm. The solution can be found by storing ancestor information along with each state. A state may be saved along with a record of its parent state, e.g., as a (state, parent) pair. If this is done in the search of Figure 3.15, the contents of open and closed at the fourth iteration would be:

open = [(D,A), (E,B), (F,B), (G,C), (H,C)]; closed = [(C,A), (B,A), (A,nil)]

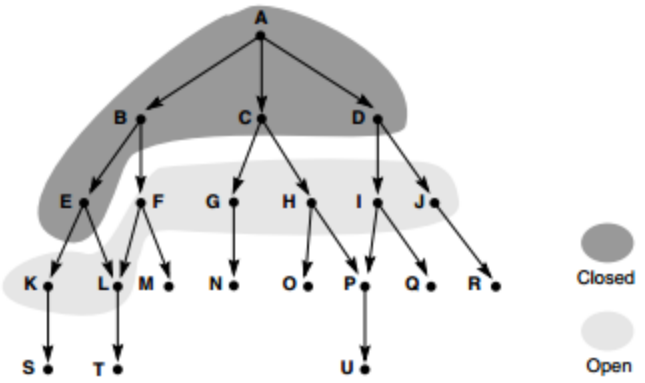

Figure 3.16 Graph of Figure 3.15 at iteration 6 of breadth-first search. States on open and closed are highlighted.

The path (A, B, F) that led from A to F could easily be constructed from this information. When a goal is found, the algorithm can construct the solution path by tracing back along parents from the goal to the start state. Note that state A has a parent of nil, indicating that it is a start state; this stops reconstruction of the path. Because breadth-first search finds each state along the shortest path and retains the first version of each state, this is the shortest path from a start to a goal.

Figure 3.17 shows the states removed from open and examined in a breadth-first search of the graph of the 8-puzzle. As before, arcs correspond to moves of the blank up, to the right, down, and to the left. The number next to each state indicates the order in which it was removed from open. States left on open when the algorithm halted are not shown.

Next, we create a depth-first search algorithm, a simplification of the backtrack algorithm already presented in Section 3.2.3. In this 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:

```
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.
```

A trace of depth_first_search on the graph of Figure 3.15 appears below. 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. Assume 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]

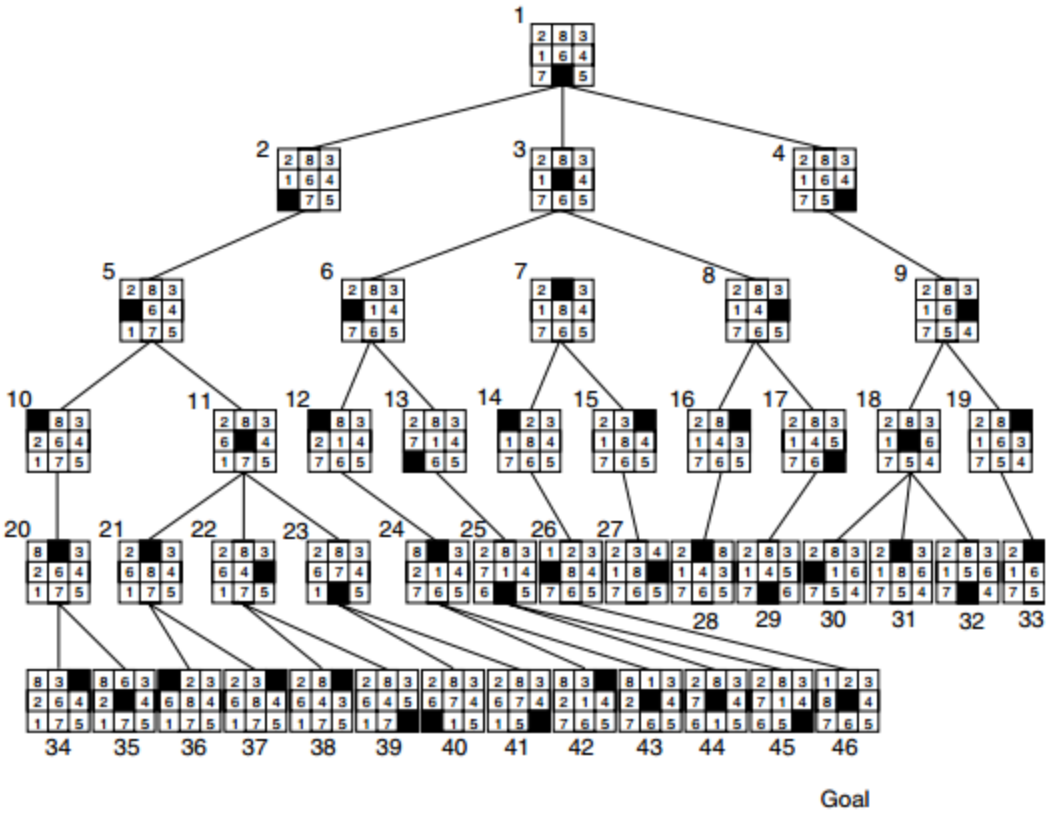

Figure 3.17 Breadth-first search of the 8-puzzle, showing order in which states were removed from open.

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]

and so on until either U is discovered or open = [ ].

As with breadth_first_search, open lists all states discovered but not yet evaluated (the current "frontier" of the search), and closed records states already considered. Figure 3.18 shows the graph of Figure 3.15 at the sixth iteration of the depth_first_search. The contents of open and closed are highlighted. As with breadth_first_search, the algorithm could store a record of the parent along with each state, allowing the algorithm to reconstruct the path that led from the start state to a goal.

Unlike breadth-first search, a depth-first search is not guaranteed to find the shortest path to a state the first time that state is encountered. Later in the search, a different path may be found to any state. If path length matters in a problem solver, when the algorithm encounters a duplicate state, the algorithm should save the version reached along the shortest path. This could be done by storing each state as a triple: (state, parent,

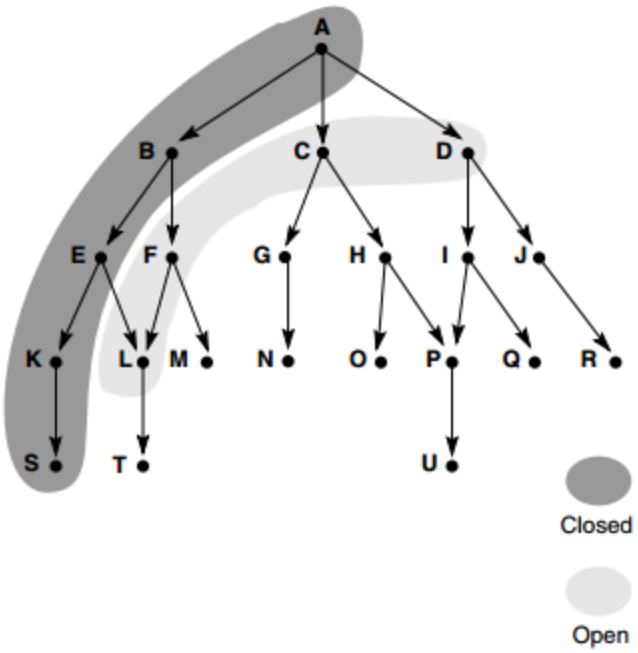

Figure 3.18 Graph of Figure 3.15 at iteration 6 of depth-first search. States on open and closed are highlighted.

length_of_path). When children are generated, the value of the path length is simply incremented by one and saved with the child. If a child is reached along multiple paths, this information can be used to retain the best version. This is treated in more detail in the discussion of *algorithm A* in Chapter 4. Note that retaining the best version of a state in a simple depth-first search does not guarantee that a goal will be reached along the shortest path.

Figure 3.19 gives a depth-first search of the 8-puzzle. As noted previously, the space is generated by the four "move blank" rules (up, down, left, and right). The numbers next to the states indicate the order in which they were considered, i.e., removed from open. States left on open when the goal is found are not shown. A depth bound of 5 was imposed on this search to keep it from getting lost deep in the space.

As with choosing between data- and goal-driven search for evaluating a graph, the choice of depth-first or breadth-first search depends on the specific problem being solved. Significant features include the importance of finding the shortest path to a goal, the branching factor of the space, the available compute time and space resources, the average length of paths to a goal node, and whether we want all solutions or only the first solution. In making these decisions, there are advantages and disadvantages for each approach.

Breadth-First Because it always examines all the nodes at level n before proceeding to level n + 1, breadth-first search 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. Unfortunately, if there is a bad branching factor, i.e., states have a high average number of children, the

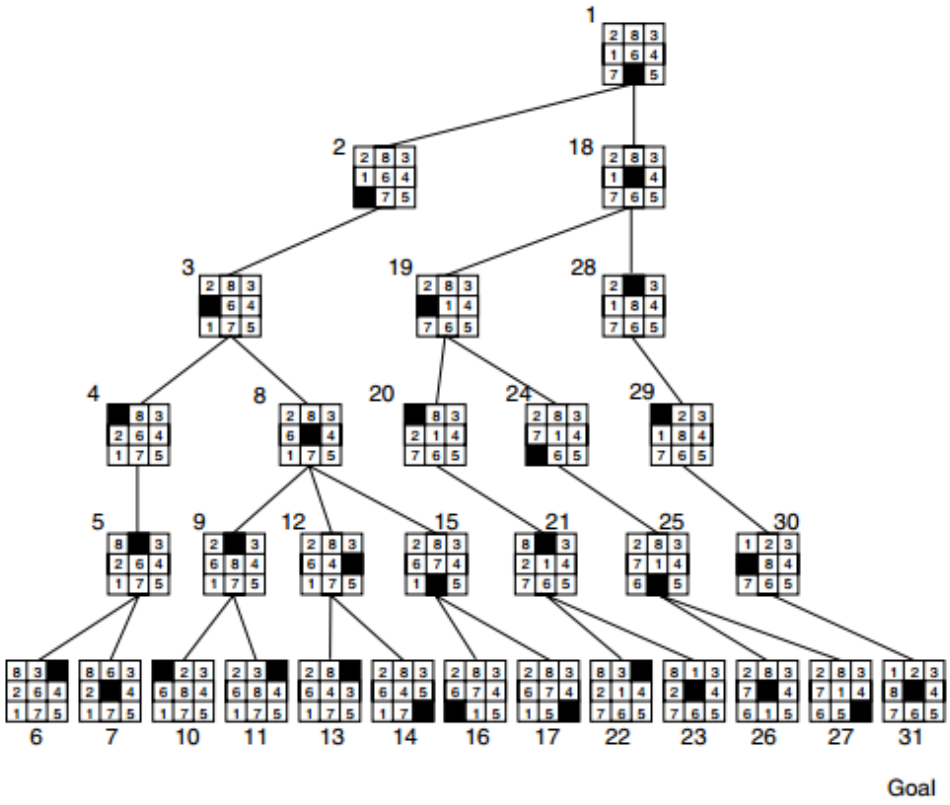

Figure 3.19 Depth-first search of the 8-puzzle with a depth bound of 5.

combinatorial explosion may prevent the algorithm from finding a solution using available memory. This is due to the fact that all unexpanded nodes for each level of the search must be kept on open. For deep searches, or state spaces with a high branching factor, this can become quite cumbersome.

The space utilization of breadth-first search, measured in terms of the number of states on 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 on the previous level. This gives B^n 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, in the game of chess, for example.

Depth-First Depth-first search gets quickly into a deep search space. If it is known that the solution path will be long, depth-first search will not waste time searching a large number of "shallow" states in the graph. On the other hand, 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 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. The space

usage of depth-first search is a linear function of the length of the path. 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 $B \times n$ states to go n levels deep into the space.

The best answer to the "depth-first versus breadth-first" issue is to examine the problem space and consult experts in the area. In chess, for example, breadth-first search simply is not possible. In simpler games, breadth-first search not only may be possible but, because it gives the shortest path, may be the only way to avoid losing.

3.2.4 Depth-First Search with Iterative Deepening

A nice compromise on these trade-offs is to use a depth bound on depth-first search. The depth bound forces a failure on a search path once it gets below a certain level. This causes a breadth-first like sweep of the search space at that depth level. When it is known that a solution lies within a certain depth or when time constraints, such as those that occur in an extremely large space like chess, limit the number of states that can be considered; then a depth-first search with a depth bound may be most appropriate. Figure 3.19 showed a depth-first search of the 8-puzzle in which a depth bound of 5 caused the sweep across the space at that depth.

This insight leads to a search algorithm that remedies many of the drawbacks of both depth-first and breadth-first search. *Depth-first iterative deepening* (Korf 1987) 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. No information about the state space is retained between iterations.

Because the algorithm searches the space in a level-by-level fashion, it is guaranteed to find a shortest path to a goal. Because it does only depth-first search at each iteration, the space usage at any level n is $B \times n$, where B is the average number of children of a node.

Interestingly, although it seems as if depth-first iterative deepening would be much less efficient than either depth-first or breadth-first search, its time complexity is actually of the same order of magnitude as either of these: $O(B^n)$. An intuitive explanation for this seeming paradox is given by Korf (1987):

> Since the number of nodes in a given level of the tree grows exponentially with depth, almost all the time is spent in the deepest level, even though shallower levels are generated an arithmetically increasing number of times.

Unfortunately, all the search strategies discussed in this chapter—depth-first, breadth-first, and depth-first iterative deepening—may be shown to have worst-case exponential time complexity. This is true for all *uninformed* search algorithms. The only approaches to search that reduce this complexity employ heuristics to guide search. *Best-first search* is a search algorithm that is similar to the algorithms for depth- and breadth-first search just presented. However, best-first search orders the states on the

open list, the current fringe of the search, according to some measure of their heuristic merit. At each iteration, it considers neither the deepest nor the shallowest but the "best" state. Best-first search is the main topic of Chapter 4.

4.2 The Best-First Search Algorithm

4.2.1 Implementing Best-First Search

In spite of their limitations, algorithms such as backtrack, hill climbing, and dynamic programming can be used effectively if their evaluation functions are sufficiently informative to avoid local maxima, dead ends, and related anomalies in a search space. In general,

however, use of heuristic search requires a more flexible algorithm: this is provided by *best-first search*, where, with a priority queue, recovery from these situations is possible.

Like the depth-first and breadth-first search algorithms of Chapter 3, best-first search uses lists to maintain states: open to keep track of the current fringe of the search and closed to record states already visited. An added step in the algorithm orders the states on open according to some heuristic estimate of their "closeness" to a goal. Thus, each iteration of the loop considers the most "promising" state on the open list. The pseudo-code for the function best_first_search appears below.

```
function best_first_search;

begin
   open := [Start];                                                    % initialize
   closed := [ ];
   while open ≠ [ ] do                                                 % states remain
      begin
         remove the leftmost state from open, call it X;
         if X = goal then return the path from Start to X
         else begin
                  generate children of X;
                  for each child of X do
                  case
                       the child is not on open or closed:
                         begin
                              assign the child a heuristic value;
                              add the child to open
                         end;
                       the child is already on open:
                         if the child was reached by a shorter path
                         then give the state on open the shorter path
                       the child is already on closed:
                         if the child was reached by a shorter path then
                           begin
                             remove the state from closed;
                             add the child to open
                           end;
                  end;                                                 % case
                  put X on closed;
                  re-order states on open by heuristic merit (best leftmost)
              end;
return FAIL                                                            % open is empty
end.
```

At each iteration, best_first_search removes the first element from the open list. If it meets the goal conditions, the algorithm returns the solution path that led to the goal. Note that each state retains ancestor information to determine if it had previously been reached by a shorter path and to allow the algorithm to return the final solution path. (See Section 3.2.3.)

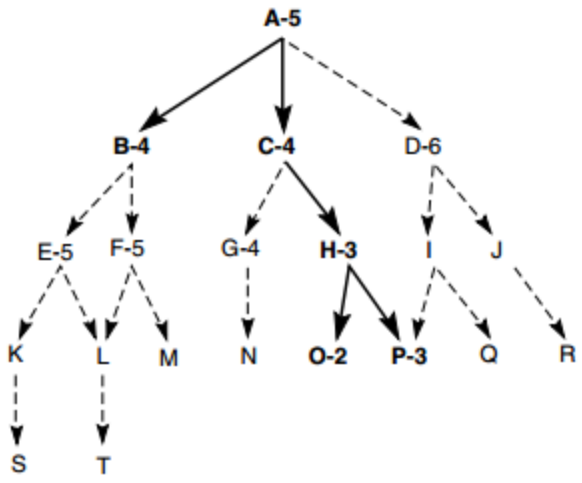

Figure 4.10 Heuristic search of a hypothetical state space.

If the first element on open is not a goal, the algorithm applies all matching production rules or operators to generate its descendants. If a child state is not on open or closed best_first_search applies a heuristic evaluation to that state, and the open list is sorted according to the heuristic values of those states. This brings the "best" states to the front of open. Note that because these estimates are heuristic in nature, the next "best" state to be examined may be from any level of the state space. When open is maintained as a sorted list, it is often referred to as a *priority queue*.

If a child state is already on open or closed, the algorithm checks to make sure that the state records the shorter of the two partial solution paths. Duplicate states are not retained. By updating the path history of nodes on open and closed when they are rediscovered, the algorithm will find a shortest path to a goal (within the states considered).

Figure 4.10 shows a hypothetical state space with heuristic evaluations attached to some of its states. The states with attached evaluations are those actually generated in best_first_search. The states expanded by the heuristic search algorithm are indicated in bold; note that it does not search all of the space. The goal of best-first search is to find the goal state by looking at as few states as possible; the more *informed* (Section 4.2.3) the heuristic, the fewer states are processed in finding the goal.

A trace of the execution of best_first_search on this graph appears below. Suppose P is the goal state in the graph of Figure 4.10. Because P is the goal, states along the path to P will tend to have lower heuristic values. The heuristic is fallible: the state O has a lower value than the goal itself and is examined first. Unlike hill climbing, which does not maintain a priority queue for the selection of "next" states, the algorithm recovers from this error and finds the correct goal.

1. open = [A5]; closed = [ ]
2. evaluate A5; open = [B4,C4,D6]; closed = [A5]

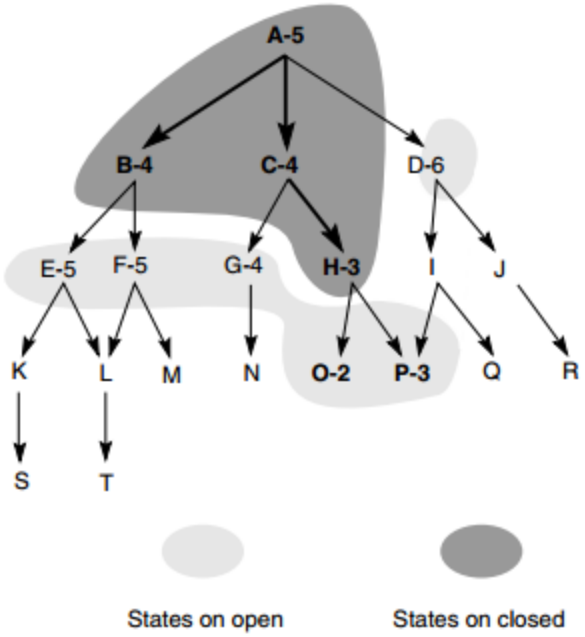

Figure 4.11 Heuristic search of a hypothetical state space with open and closed states highlighted.

3. evaluate B4; open = [C4,E5,F5,D6]; closed = [B4,A5]
4. evaluate C4; open = [H3,G4,E5,F5,D6]; closed = [C4,B4,A5]
5. evaluate H3; open = [O2,P3,G4,E5,F5,D6]; closed = [H3,C4,B4,A5]
6. evaluate O2; open = [P3,G4,E5,F5,D6]; closed = [O2,H3,C4,B4,A5]
7. evaluate P3; the solution is found!

Figure 4.11 shows the space as it appears after the fifth iteration of the while loop. The states contained in open and closed are indicated. open records the current frontier of the search and closed records states already considered. Note that the frontier of the search is highly uneven, reflecting the opportunistic nature of best-first search.

The best-first search algorithm selects the most promising state on open for further expansion. However, as it is using a heuristic that may prove erroneous, it does not abandon all the other states but maintains them on open. In the event a heuristic leads the search down a path that proves incorrect, the algorithm will eventually retrieve some previously generated, "next best" state from open and shift its focus to another part of the space. In the example of Figure 4.10, after the children of state B were found to have poor heuristic evaluations, the search shifted its focus to state C. The children of B were kept on open in case the algorithm needed to return to them later. In best_first_search, as in the algorithms of Chapter 3, the open list supports backtracking from paths that fail to produce a goal.

4.2.2 Implementing Heuristic Evaluation Functions

We next evaluate the performance of several different heuristics for solving the 8-puzzle. Figure 4.12 shows a start and goal state for the 8-puzzle, along with the first three states generated in the search.

The simplest heuristic counts the tiles out of place in each state when compared with the goal. This is intuitively appealing, because it would seem that, all else being equal, the state that had fewest tiles out of place is probably closer to the desired goal and would be the best to examine next.

However, this heuristic does not use all of the information available in a board configuration, because it does not take into account the distance the tiles must be moved. A "better" heuristic would sum all the distances by which the tiles are out of place, one for each square a tile must be moved to reach its position in the goal state.

Both of these heuristics can be criticized for failing to acknowledge the difficulty of tile reversals. That is, if two tiles are next to each other and the goal requires their being in opposite locations, it takes (several) more than two moves to put them back in place, as the tiles must "go around" each other (Figure 4.13).

A heuristic that takes this into account multiplies a small number (2, for example) times each direct tile reversal (where two adjacent tiles must be exchanged to be in the order of the goal). Figure 4.14 shows the result of applying each of these three heuristics to the three child states of Figure 4.12.

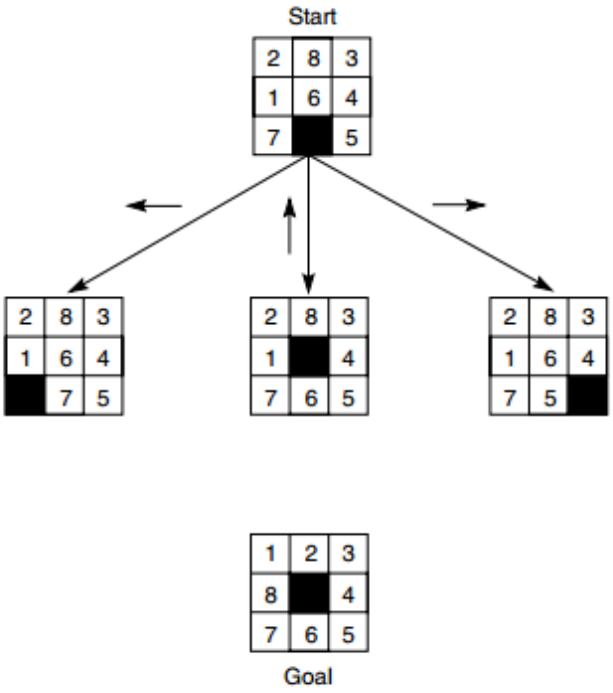

Figure 4.12 The start state, first moves, and goal state for an example 8-puzzle.

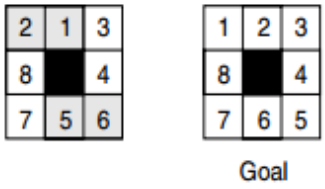

Figure 4.13 An 8-puzzle state with a goal and two reversals: 1 and 2, 5 and 6.

In Figure 4.14's summary of evaluation functions, the sum of distances heuristic does indeed seem to provide a more accurate estimate of the work to be done than the simple count of the number of tiles out of place. Also, the tile reversal heuristic fails to distinguish between these states, giving each an evaluation of 0. Although it is an intuitively appealing heuristic, it breaks down since none of these states have any direct reversals. A fourth heuristic, which may overcome the limitations of the tile reversal heuristic, adds the sum of the distances the tiles are out of place and 2 times the number of direct reversals.

This example illustrates the difficulty of devising good heuristics. Our goal is to use the limited information available in a single state descriptor to make intelligent choices. Each of the heuristics proposed above ignores some critical bit of information and is subject to improvement. The design of good heuristics is an empirical problem; judgment and intuition help, but the final measure of a heuristic must be its actual performance on problem instances.

If two states have the same or nearly the same heuristic evaluations, it is generally preferable to examine the state that is nearest to the root state of the graph. This state will have a greater probability of being on the *shortest* path to the goal. The distance from the

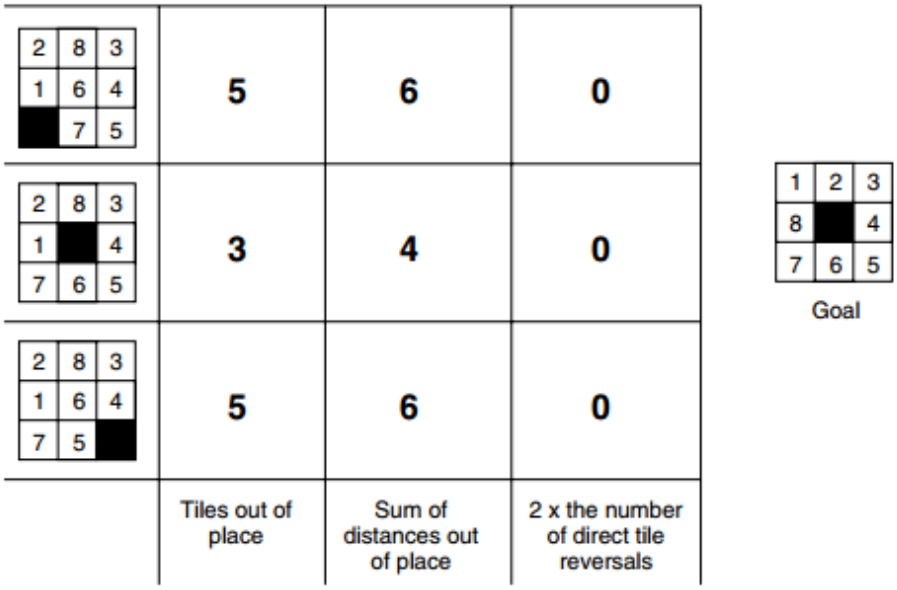

Figure 4.14 Three heuristics applied to states in the 8-puzzle.

starting state to its descendants can be measured by maintaining a depth count for each state. This count is 0 for the beginning state and is incremented by 1 for each level of the search. This depth measure can be added to the heuristic evaluation of each state to bias search in favor of states found shallower in the graph.

This makes our evaluation function, f, the sum of two components:

$$f(n) = g(n) + h(n)$$

where g(n) measures the actual length of the path from any state n to the start state and h(n) is a heuristic estimate of the distance from state n to a goal.

In the 8-puzzle, for example, we can let h(n) be the number of tiles out of place. When this evaluation is applied to each of the child states in Figure 4.12, their f values are 6, 4, and 6, respectively, see Figure 4.15.

The full best-first search of the 8-puzzle graph, using f as defined above, appears in Figure 4.16. Each state is labeled with a letter and its heuristic weight, f(n) = g(n) + h(n). The number at the top of each state indicates the order in which it was taken off the open list. Some states (h, g, b, d, n, k, and i) are not so numbered, because they were still on open when the algorithm terminates.

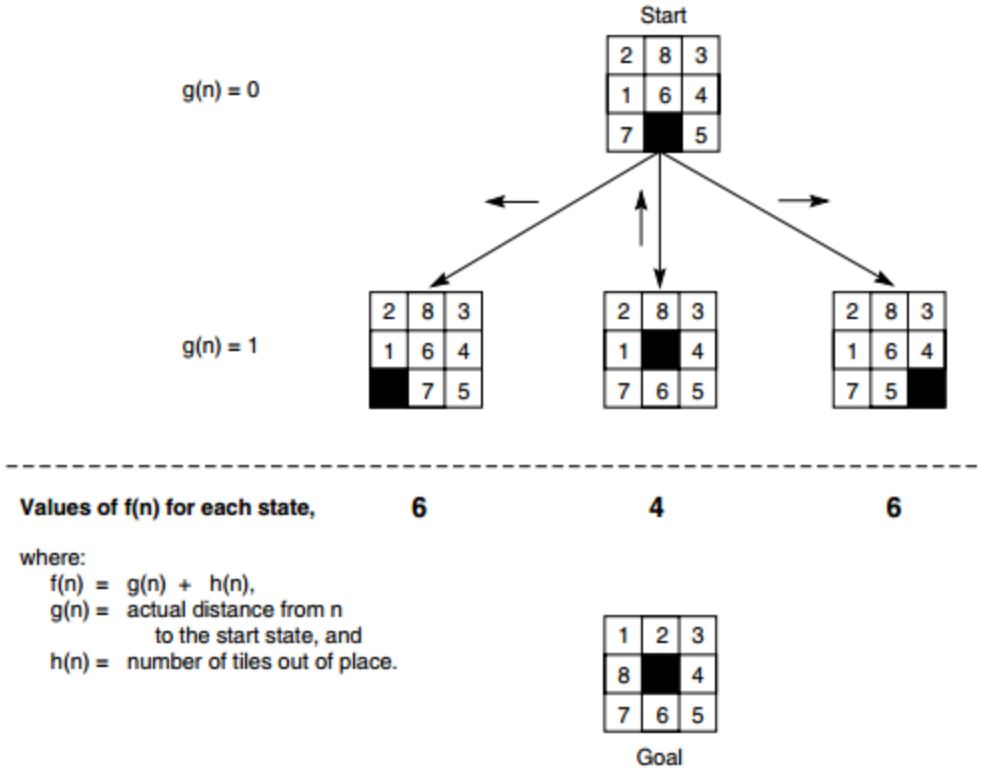

Figure 4.15 The heuristic f applied to states in the 8-puzzle.

The successive stages of open and closed that generate this graph are:

1. open = [a4];
   closed = [ ]
2. open = [c4, b6, d6];
   closed = [a4]
3. open = [e5, f5, b6, d6, g6];
   closed = [a4, c4]
4. open = [f5, h6, b6, d6, g6, i7];
   closed = [a4, c4, e5]
5. open = [j5, h6, b6, d6, g6, k7, i7];
   closed = [a4, c4, e5, f5]
6. open = [l5, h6, b6, d6, g6, k7, i7];
   closed = [a4, c4, e5, f5, j5]
7. open = [m5, h6, b6, d6, g6, n7, k7, i7];
   closed = [a4, c4, e5, f5, j5, l5]
8. success, m = goal!

In step 3, both e and f have a heuristic of 5. State e is examined first, producing children, h and i. Although h, the child of e, has the same number of tiles out of place as f, it is one level deeper in the space. The depth measure, g(n), causes the algorithm to select f for evaluation in step 4. The algorithm goes back to the shallower state and continues to the goal. The state space graph at this stage of the search, with open and closed highlighted, appears in Figure 4.17. Notice the opportunistic nature of best-first search.

In effect, the g(n) component of the evaluation function gives the search more of a breadth-first flavor. This prevents it from being misled by an erroneous evaluation: if a heuristic continuously returns "good" evaluations for states along a path that fails to reach a goal, the g value will grow to dominate h and force search back to a shorter solution path. This guarantees that the algorithm will not become permanently lost, descending an infinite branch. Section 4.3 examines the conditions under which best-first search using this evaluation function can actually be guaranteed to produce the shortest path to a goal.

To summarize:

1. Operations on states generate children of the state currently under examination.
2. Each new state is checked to see whether it has occurred before (is on either open or closed), thereby preventing loops.
3. Each state n is given an f value equal to the sum of its depth in the search space g(n) and a heuristic estimate of its distance to a goal h(n). The h value guides search toward heuristically promising states while the g value can prevent search from persisting indefinitely on a fruitless path.

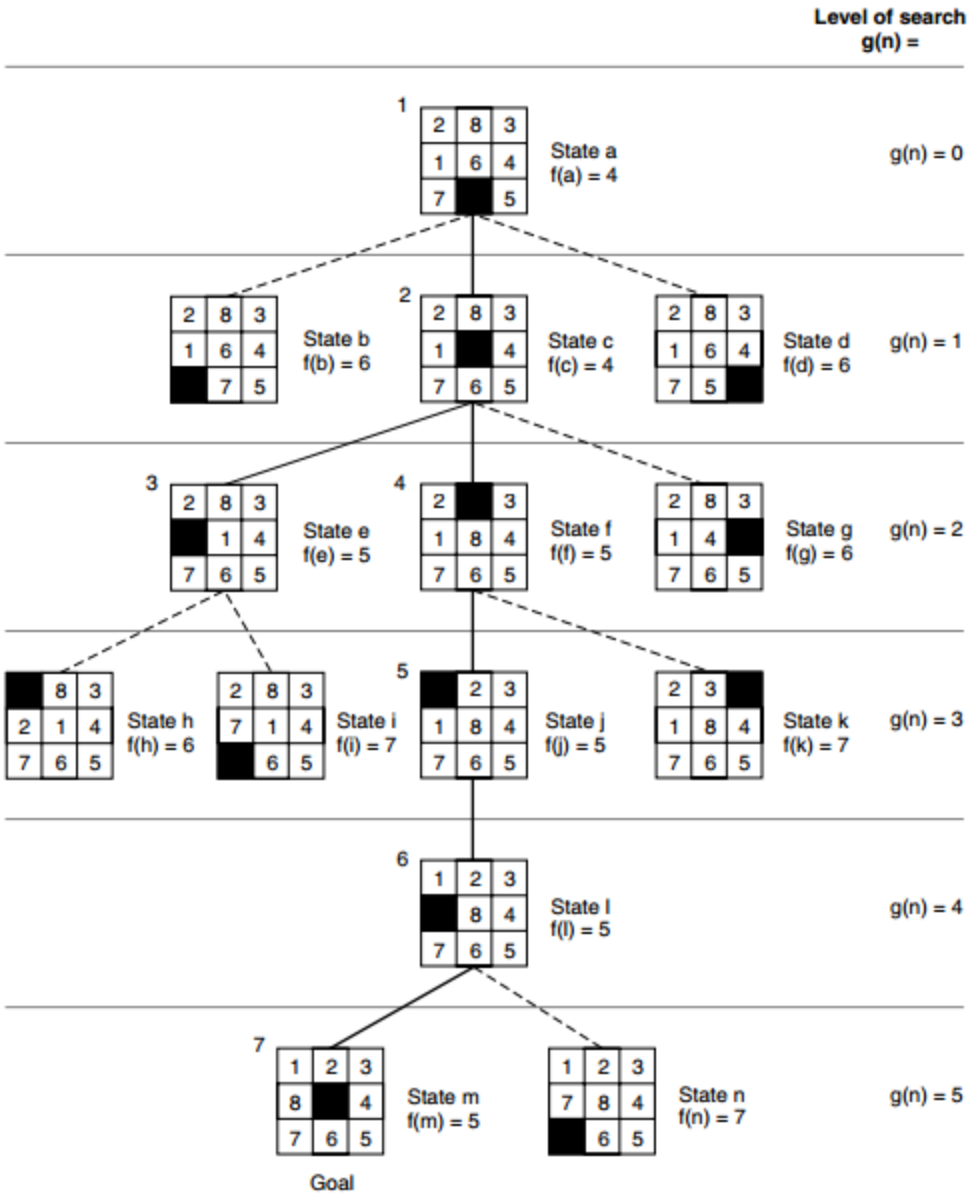

Figure 4.16 State space generated in heuristic search of the 8-puzzle graph.

4. States on open are sorted by their f values. By keeping all states on open until they are examined or a goal is found, the algorithm recovers from dead ends.
5. As an implementation point, the algorithm's efficiency can be improved through maintenance of the open and closed lists, perhaps as *heaps* or *leftist trees*.

Best-first search is a general algorithm for heuristically searching any state space graph (as were the breadth- and depth-first algorithms presented earlier). It is equally

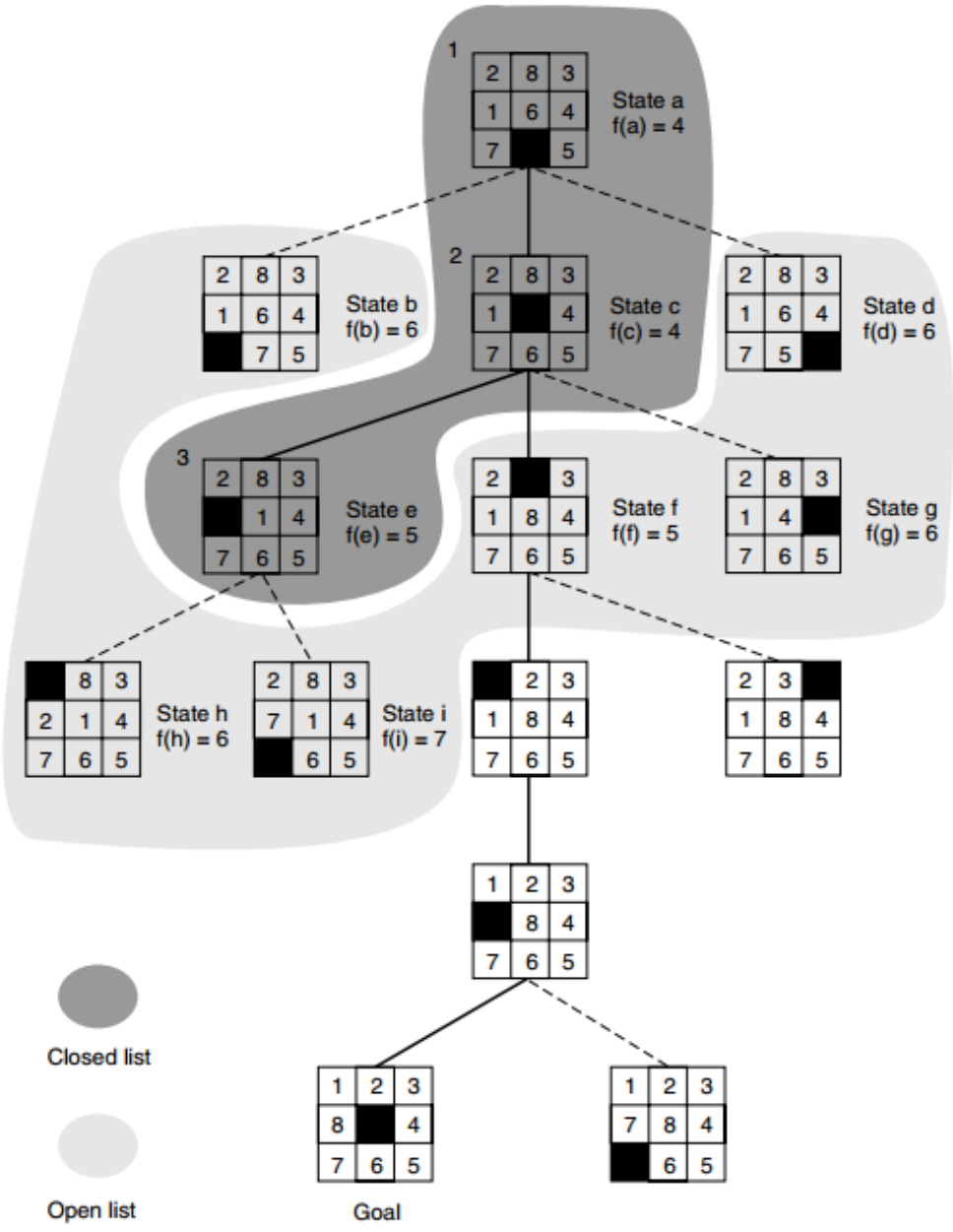

Figure 4.17 open and closed as they appear after the third iteration of heuristic search.

applicable to data- and goal-driven searches and supports a variety of heuristic evaluation functions. It will continue (Section 4.3) to provide a basis for examining the behavior of heuristic search. Because of its generality, best-first search can be used with a variety of heuristics, ranging from subjective estimates of state's "goodness" to sophisticated

measures based on the probability of a state leading to a goal. Bayesian statistical measures (Chapters 5 and 9) offer an important example of this approach.

Another interesting approach to implementing heuristics is the use of confidence measures by expert systems to weigh the results of a rule. When human experts employ a heuristic, they are usually able to give some estimate of their confidence in its conclusions. Expert systems employ *confidence measures* to select the conclusions with the highest likelihood of success. States with extremely low confidences can be eliminated entirely. This approach to heuristic search is examined in the next section.