Principles of Artificial Intelligence.

2.4.6. THE HEURISTIC POWER OF EVALUATION FUNCTIONS

The selection of the heuristic function is crucial in determining the heuristic power of search algorithm A. Using $h \equiv 0$ assures admissibility but results in a breadth-first search and is thus usually inefficient. Setting h equal to the highest possible lower bound on h^* expands the fewest nodes consistent with maintaining admissibility.

Often, heuristic power can be gained at the expense of admissibility by using some function for h that is not a lower bound on h^*. This added heuristic power then allows us to solve much harder problems. In the 8-puzzle, the function $h(n) = W(n)$ (where $W(n)$ is the number of tiles in the wrong place) is a lower bound on $h^*(n)$, but it does not provide a very good estimate of the difficulty (in terms of number of steps to the goal) of a tile configuration. A better estimate is the function $h(n) = P(n)$, where $P(n)$ is the sum of the distances that each tile is from "home" (ignoring intervening pieces). Even this estimate is too coarse, however, in that it does not accurately appraise the difficulty of exchanging the positions of two adjacent tiles.

An estimate that works quite well for the 8-puzzle is

$$h(n) = P(n) + 3S(n).$$

The quantity $S(n)$ is a *sequence score* obtained by checking around the noncentral squares in turn, allotting 2 for every tile not followed by its proper successor and allotting 0 for every other tile; a piece in the center scores one. We note that this h function does not provide a lower bound for h^*. With this heuristic function used in the evaluation function $f(n) = g(n) + h(n)$, we can easily solve much more difficult 8-puzzles than the one we solved earlier. In Figure 2.9 we show the search tree resulting from applying **GRAPHSEARCH** with this evaluation function to the problem of transforming

```
2 1 6
4   8
7 5 3
```

into

```
1 2 3
8   4
7 6 5
```


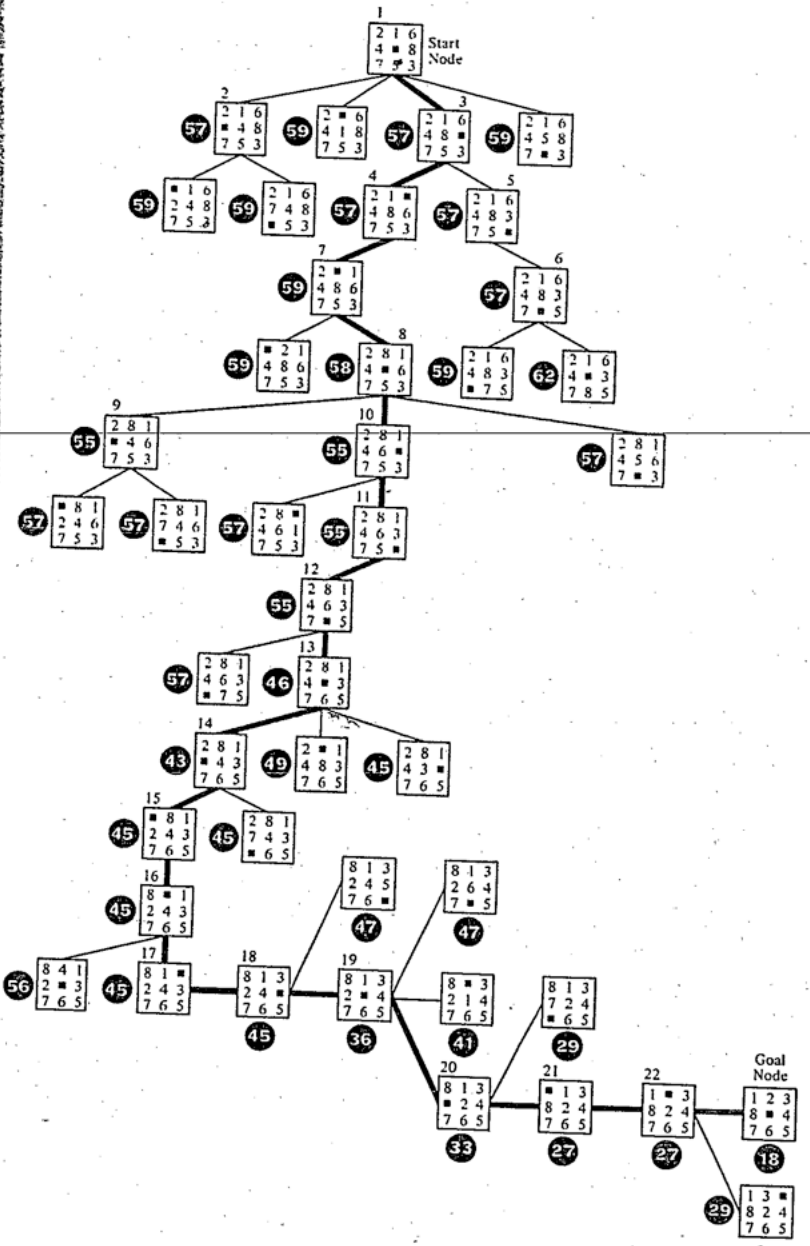

Fig. 2.9 A search tree for the 8-puzzle.

Again, the f values of each node are circled in the figure, and the uncircled numbers show the order in which nodes are expanded. (In the search depicted in Figure 2.9, ties among minimal f values are resolved by selecting the deepest node in the search tree.)

The solution path found happens to be of minimal length (18 steps); although, since the h function is not a lower bound for h^*, we were not guaranteed of finding an optimal path. Note that this h function results in a focused search, directed toward the goal; only a very limited spread occurred, near the start.

Another factor that determines the heuristic power of search algorithms is the amount of effort involved in calculating the heuristic function. The best function would be one identically equal to h^*, resulting in an absolute minimum number of node expansions. (Such an h could, for example, be determined as a result of a separate complete search at every node; but this obviously would not reduce the total computational effort.) Sometimes an h function that is not a lower bound on h^* is easier to compute than one that is a lower bound. In these cases, the heuristic power might be doubly improved—because the total number of nodes expanded can be reduced (at the expense of admissibility) and because the computational effort is reduced.

In certain cases the heuristic power of a given heuristic function can be increased simply by multiplying it by some positive constant greater than one. If this constant is very large, the situation is as if $g(n) \equiv 0$. In many problems we merely desire to find *some* path to a goal node and are unconcerned about the cost of the resulting path. (We are, of course, concerned about the amount of search effort required to find a path.) In such situations, we might think that g could be ignored completely since, at any stage during the search, we don't care about the costs of the paths developed thus far. We care only about the remaining seach effort required to find a goal node. This search effort, while possibly dependent on the h values of the nodes on *OPEN*, would seem to be independent of the g values of these nodes. Therefore, for such problems, we might be led to use $f \equiv h$ as the evaluation function.

To ensure that *some* path to a goal will eventually be found, g should be included in f even when it is not essential to find a path of minimal cost. Such insurance is necessary whenever h is not a perfect estimator; if the node with minimum h were always expanded, the search process might expand deceptive nodes forever without ever reaching a goal node.

Including g tends to add a breadth-first component to the search and thus ensures that no part of the implicit graph will go permanently unsearched.

The relative weights of g and h in the evaluation function can be controlled by using $f = g + wh$, where w is a positive number. Very large values of w overemphasize the heuristic component, while very small values of w give the search a predominantly breadth-first character. Experimental evidence suggests that search efficiency is often enhanced by allowing the value of w to vary inversely with the depth of a node in the search tree. At shallow depths, the search relies mainly on the heuristic component, while at greater depths, the search becomes increasingly breadth-first, to ensure that some path to a goal will eventually be found.

To summarize, there are three important factors influencing the heuristic power of Algorithm **A**:

(a) the cost of the path,

(b) the number of nodes expanded in finding the path, and

(c) the computational effort required to compute h.

The selection of a suitable heuristic function permits one to balance these factors to maximize heuristic power.

2.5. RELATED ALGORITHMS

2.5.1. BIDIRECTIONAL SEARCH

Some problems can be solved using production systems whose rules can be used in either a forward or a backward direction. An interesting possibility is to search in both directions simultaneously. The graph-searching process that models such a bidirectional production system can be viewed as one in which search proceeds outward simultaneously from both the start node and from a set of goal nodes. The process terminates when (and if) the two search frontiers meet in some appropriate fashion.