$$
\begin{aligned}
f(n_2) &= g(n_2) + h(n_2) \\
&= g^*(n_2) + h(n_2) \qquad \text{(RESULT 7)} \\
&= g^*(n_1) + c(n_1,n_2) + h(n_2) \\
&= g(n_1) + c(n_1,n_2) + h(n_2) \\
&\quad \text{(RESULT 7)}
\end{aligned}
$$

Since the monotone restriction implies

$$c(n_1,n_2) + h(n_2) \geq h(n_1),$$

we have

$$f(n_2) \geq g(n_1) + h(n_1) = f(n_1).$$

Since this fact is true for any adjacent pair of nodes in the sequence of nodes expanded by **A***, we have

RESULT 8: If the monotone restriction is satisfied, the f values of the sequence of nodes expanded by **A*** is nondecreasing.

When the monotone restriction is not satisfied, it is possible that some node has a smaller f value at expansion than that of a previously expanded node. We can exploit this observation to improve the efficiency of **A*** under this condition. By RESULT 5, when node n is expanded, $f(n) \leq f^*(s)$. Suppose, during the execution of **A***, we maintain a global variable, F, as the maximum of the f values of all nodes so far expanded. Certainly $F \leq f^*(s)$ at all times. If ever a node, n, on *OPEN* has $f(n) < F$, we know by the corollary to RESULT 3 that it will eventually be expanded. In fact, there may be several nodes on *OPEN* whose f values are strictly less than F. Rather than choose, from these, that node with the smallest f value, we might rather choose that node with the smallest g value. (All of them must eventually be expanded anyway.)

The effect of this altered node selection rule is to enhance the chances that the first path discovered to a node will be an optimal path. Thus, even when the monotone restriction is not satisfied, this alteration will diminish the need for pointer redirection in step 7 of the algorithm. (Note that when the monotone restriction is satisfied, RESULT 8 implies that there will never be a node on *OPEN* whose f value is less than F.)

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
```