Heuristics

The two heuristics (h1 and h2 respectively) used are:

- The number of tiles in incorrect positions (relative to the desired final state of the board). This total does not include the blank tile
- The distance (in terms of cells) of a cell from its intended final position. The difference in rows and columns for each cell is added. This calculation once again does not include the blank tile.

Both are consistent. Now, every time we look for the successors of a board state, we can see that we increase the previous g(n) by 1. In the ideal case when the successor is a goal state, the resulting f(n) is the same as the previous state, i.e., only one tile needed to be readjusted. But there is no way where the current f(n) can be less than the previous state.

In case of the either heuristic, the value of h(n) can only change by a factor of 1 at max. A non-blank cell reaches/doesn't reach its final state. Since the tile moves only one step, the value of h(n) changes by 1.

Also, it is obvious that the Manhattan distance of a tile from its current to final state will always be a value greater than or equal to 1, if not in the correct position. The other heuristic always has a max value of 1. Now we can see that for every state of the board, the Manhattan distance heuristic will dominate the incorrect positions heuristic.

Memory Issues

In my implementation, every state of the board is an object. That object contains information about the state of the board, the path traversed to get to that state from the start state, f(n) value, g(n) value, h(n) value, and the list of possible moves from that position. Now this is a lot of information, but is fairly easy for a computer to handle when the number of such objects to be maintained in the memory is not too large. But the nature of the A* algorithm forces me to maintain a list containing the states being considered. Also, to optimize the algorithm, another list containing the states already explored fully must be maintained. Every time, I insert a state to be considered, I must first look through the 1st list to see if it is already evaluated. Also, if it is in the "currently considered states" list, I must update the corresponding entry in the list.

At every iteration of the code, until the final state has been reached, several states (max 3 after optimization) are added to the list. Now, since this is a list of objects, and each object containing a couple of lists, the amount of memory keeps increasing for every iteration. Even if a state is fully explored, it still cannot be thrown away, since it is needed to check before insertion into the other list.

For example, in the 15 puzzle situation, the A* algorithm can expand to a maximum of 4 states (without any optimization, moving the blank tile up, down, left or right), every iteration. In cases when the number of iterations is more than 10000 (not uncommon in certain configurations), we would have to maintain info for all these states, and go through all of them (even multiple times during an iteration) before we proceed to the next step.

Iterative Deepening A*

In this, the program goes through iterations until it reaches the goal state.

At each iteration, all the child nodes are recursively expanded. This is a depth first search. The expansion stops when it results in a goal state or when we encounter a state which results in the value of the f(n) of the state being greater than a threshold. In the beginning the threshold is simply calculate from the start state. At each iteration, the threshold used for the next iteration is the minimum cost of all values that exceeded the current threshold.

Using the previous heuristics that are consistent and admissible, we obtain the goal state without having to maintain the objects representing various states in the memory. This is because at each iteration, we recalculate in a recursive fashion. We do end up using more computing power but consume less memory.

Time complexity is similar to BFS (O(b^d)) where b is the branching factor and d is the depth of the goal state.

The space complexity is O(d), where d is the depth of the goal state. As stated previously, we don't need to maintain any state in memory. Only saving the value of f(n) for calculating the new threshold suffices.

The algorithm is obviously complete, since it is a variation of BFS and DFS combined.

Some sample results

Input

2,1,6

3,5,

4,8,7

Output

L,D,R,U,L,L,U,R,D,L,D,R,R,U,U,L,D,D,R

Input

7,6,2

4,1,5

,8,3

Output

U,R,U,L,D,D,R,U,U,L,D,R,R,D,L,L,U,U,R,R,D,D