Principal data structure used to hold the eight tile initial state and goal state is a list. Any eight puzzle generated by the program is of the EightPuzzle class, which contains typical methods like \_\_init\_\_(), \_\_eq\_\_(), \_\_str\_\_(), among others that make up the meat of the program. Below will be summaries of methods that are not necessarily self-explanatory by my count.

**def** **\_generate\_moves(self)**:

\_generate\_moves() first invokes an instance of the utility class in a secondary file named util.py. It then uses util.py’s methods to find the free space of the current puzzle instance as well as all possible switches that can be made with that free space and its surrounding tiles (done using a function called \_get\_legal\_moves() that checks the row and column that the free space is currently in and then returns a list of tuples that represent the possible legal moves that can be made with the free space). \_generate\_moves() also contains a submethod named swap\_and\_clone() that is used when returning from \_generate\_moves(). swap\_and \_clone clones the current matrix and uses the swap function in util.py to swap the free space and whatever adjacent index is being moved to the free space’s place. Whenever a swap\_and\_clone() is called, the depth of the search tree iterates by one and the parent of that next depth is the configuration of the matrix that initially called \_generate\_moves().

The final thing to discuss is the return statement of \_generate\_moves(), which returns a map data structure that applies swap\_and\_clone() to every possible move generated by \_get\_legal\_moves() and returns all possible legal matrices that can be formed.

**def** **solve(self, heur)**:

solve() is the behemoth function that does all of the heavy lifting for the transformation of the initial matrix to the goal matrix. solve() creates two lists. The first list (referred to as openlist) is made from the calling EightPuzzle instance by simply inserting the current configuration into the list. The second list (called closedlist) is an empty list that will be filled as solve() solves the EightPuzzle with possible moves. An integer for the number of moves is also initialized to 0.

The first thing that solve() does following these declarations is checking to see if the length of openlist is greater than zero, and then pops the first element of openlist (which is in itself the configuration of the calling matrix). solve() then checks to see if the current configuration of the matrix is equal to the goal state- if it is, then the path to the solution is generated and the number of moves taken to get there is returned. If the matrix configuration *isn’t* equal to the goal state, the current configuration of the matrix is returned for further examination.

The possible successor matrices (referred to by the variable name move in the code) are generated by performing \_generate\_moves() on the current matrix configuration. solve() iterates through all of the successor matrices checking if the successor matrices have been seen before. It does this by checking if the current successor is equivalent to the initial matrix stored in openlist or any of the successor matrices in closedlist. The heuristic value of the successor matrix being examined is generated (simple manhattan distances of all tiles from their current location to their goal location) and the complete cost (measured as the depth of the node- the lower the depth, the higher the cost) + heuristic is calculated as well.

If the current successor is *not* found in either openlist nor closed list, then that successor matrix is appended to the openlist as the new current configuration of the calling matrix.

If the current successor *is* found in openlist already, then a copy of that successor’s location within openlist is made. An examination of the examined successor’s utility (its cost plus its heuristic) versus the copy currently in openlist is made. If the current examined successor does have a better overall utility, then the currently examined move has its values- its heuristic value, its parent within the search tree, and its depth- to the copy of itself located within openlist.

If the current successor *is* found in closedlist already, then the copy of the successor matrix is made again like in the paragraph above. Again, the utility of the copy is checked versus the utility of the currently examined successor matrix. This time, if the currently examined successor matrix has a better utility than the copy’s, the successor matrix is *removed* from closedlist and *added* to openlist. This if statement is the last block before the for loop of iterations through the successor matrices is completed.

Following that for loop above, the currently examined *initial* matrix from which the successors are derived (READ: NOT THE SUCCESSOR MATRIX) is added to closedlist. The openlist is sorted by the utility of all of its indexes, and the solve() function returns an empty list and 0 if the goal state could not be reached.

In short, the solve() function looks at the calling matrix’s configuration, generates its possible successors, adds those successors to a separate list of successors, and constructs a tree by replacing the correct (read: best utility) successor as the ‘parent node’ each time and finding the best successor all over again.

def **heur**(puzzle, item\_total\_calc, total\_calc):

heur() is the function which calculates the ‘total utility’ (again defined as the cost + heuristic) of a matrix. heur() iterates through each tile within the eight tile puzzle that called it and checks to see where its proper column and row should be. Because computer code counts from 0, the tile being examined has to be equal to itself minus 1. We’d normally say that the number 8 belongs in row 3 column 2 of the 8 puzzle if we had it in front of us in our hands, but in compute code, 8 belongs in row 2 column 1. The target column and row are generated by performing modulo and division on the tile being examined. For example, if the tile we’re looking at currently has a 3 contained in it, 3’s value is subtracted by 1, resulting in 2. 2 modulo 3 is equal to 2 and 2 divide 3 is equal to 0. Therefore, the target column and row of the 3 tile is row 0 column 2.

A check for if the current tile being examined is the free space is done by checking if the target row for the currently examined tile is less than 0. The target row of 0 within the 8 tile puzzle is then set to the last row, 2. After each tile’s target row and column are deduced, the total utility of the calling EightPuzzle has the distance of the current row and destination row added to its value. heur() returns the total utility of the EightPuzzle after iterating through all 9 indices of the EightPuzzle.

def **main**():

main() invokes an instance of util.py and creates an instance of EightPuzzle p. The created puzzle p is shuffled 20 times, and then printed to the user. The path to the goal state and the number of steps taken to get there are performed on the shuffled EightPuzzle and then printed to the user.