Output 1A
****

7 1 6
8 3 5
2 0 4

8 7 6
1 0 5
2 3 4

5
12
U U L D R
5 5 5 5 5

****

Output1B
****

7 1 6
8 3 5
2 0 4

8 7 6
1 0 5
2 3 4

5
12
U U L D R
5 5 5 5 5

****

Output 2A
****

2 6 0
1 3 4
7 5 8

1 2 3
4 5 6
7 8 0

10
24
L D R U L L D R D R
10 10 10 10 10 10 10 10 10 10

****

Output 2B
****

2 6 0
1 3 4
7 5 8

1 2 3
4 5 6
7 8 0

10
21
L D R U L L D R D R
10 10 10 10 10 10 10 10 10 10

****

Output 3A
****

5 4 3
2 6 7
1 8 0

1 2 3
4 5 6
7 8 0

22
1229
U L U L D D R U U L D D R R U L L D R U
R D
12 12 12 12 12 12 12 14 16 16 16 16 18 18
18 20 22 22 22 22 22 22

****

Output 3B
****

5 4 3
2 6 7
1 8 0

1 2 3
4 5 6
7 8 0

22
804
U L U L D D R U U L D D R R U L L D R U
R D
12 14 14 14 14 14 14 16 18 20 20 20 20 18
18 20 22 22 22 22 22 22

****

Output 4A
****

8 7 3
0 4 5
6 2 1

1 2 3
4 5 6
7 8 0

23
821
U R D D R U L D L U U R D R D L L U U R
D R D
17 17 17 19 19 19 21 23 23 23 23 23 23 23
23 23 23 23 23 23 23 23 23

****

Output 4B
****

8 7 3
0 4 5
6 2 1

1 2 3
4 5 6
7 8 0

23
986
U R D D R U L D L U U R D R D L L U U R
D R D
19 19 19 19 19 19 21 23 23 23 23 23 23 23
25 25 23 23 23 23 23 23 23

****

HOW TO RUN:

In command line with Python3 Installed, navigate to directory with both

`8Puzzle.py`
and
`Astar.py`

Run by using
`Python3 8Puzzle.py`

When prompted, input the name of the INPUT FILE. Then, type A or B for heuristic type.

A: Sum of Manhattan Distance
B: A + 2 * Linear Conflicts

Then, press ENTER after the search is done to skip saving. Or type in the name of file for saving output. Done.

SOURCE:

8Puzzle.py

```
import AStar

# Produce entry points for search algorithm from file.
def InitFromFile(targetFile, heuType):
	initialMap = [];
	goalMap = [];

	f = open(targetFile);
	fLines = f.readlines();
	f.close();

	# Getting initial state (line 0-2) and appending to initialMap
by row.
	for i in range(0,3):
		curLine = fLines[i];
		nums = curLine.split();
		initialMap.append([  int(nums[0]), int(nums[1]),
int(nums[2])  ]);

	# Getting goal state (line 4-6) by row.
	for i in range(4,7):
		curLine = fLines[i];
		nums = curLine.split();
		goalMap.append([  int(nums[0]), int(nums[1]), int(nums[2])
]);

	# Generates a root node where search can begin.
	rootNode = AStar.StateNode(initialMap, goalMap, 0, heuType,
None, None);

	return rootNode, goalMap

# Puts result into desired output format.
def GenerateOutput(roNode, searchObj, searchResult):
	out = "";
	out += roNode.getOutput();
	out += "\n";
	out += searchResult.getOutput();
```

```
	out += 
"\n"+str(searchResult.dep)+"\n"+str(searchObj.totalNodes)+"\n";

	moves = "";
	costs = "";
	pathList = [];
	GetPath(searchResult, pathList);


	for curStep in pathList:
		moves += str(curStep[0]) + " ";
		costs += str(curStep[1].fCost) + " ";


	out += moves + "\n" + costs
	return out;

def GetPath(tNode, pList):
	if tNode.lastAction != None and tNode.lastNode != None:
		GetPath(tNode.lastNode, pList);
		pList.append((tNode.lastAction, tNode.lastNode));
	else:
		return

	return

def main():

	# Usability
	fname = input("File Name -> ");
	heurT = input("Heuristic Type (A or B) -> ")

	# Initialize from file and create a search manager object.
	rootNode, goalMap = InitFromFile(fname, heurT);
	search = AStar.AStarSearch(rootNode, goalMap);
	print("Initial Heuristic:", rootNode.heu);

	# Some terminal output for verbose operation
	print("--- GOAL ---");
	print(goalMap);
	print();
	print(rootNode);
```

```
        result = search.doSearch();
        output = GenerateOutput(rootNode, search, result);
        print("\n", "********************************", "\n");
        print(output);
        print("\n", "********************************", "\n");

        #Write results to file if output name is specified.
        fname = "";
        fname = input("Output File Name (or ENTER to skip) -> ");
        if(fname != ""):
              writeout = open(fname, "w+");
              writeout.write(output);
              writeout.close();
              print("Saved.");
        else:
              print("Not saved.");

def testing():
        # Usability
        fname = input("File Name -> ");
        heurT = input("Heuristic Type (A or B) -> ")

        # Initialize from file and create a search manager object.
        rootNode, goalMap = InitFromFile(fname, heurT);
        print(rootNode);

        tList = rootNode.GenerateChildren();
        print(rootNode.GetEmptyPos());
        for e in tList:
              print(e);

#testing();
main();
#AStar.ImpPrintT();
```

AStar.py

```
# This file contains the implementations of the classes
#     and methods pretaining to the A* algorithm used
#     in the 8Puzzle solution script.

import copy

# This class keeps track of nodes and variables related to the
search.
class AStarSearch:
     ###   CLASS ATTRIBUTES   ###
     ###    list explored     ###
     ###    list goalMap      ###
     ###    list priorityQ    ###

     def __init__(self, inRoot, goMap):
          self.priorityQ = [inRoot];
          self.explored = [];
          self.goalMap = goMap;
          self.totalNodes = 1;
          self.expandedNodes = 0;

     def doSearch(self):
          done = False;

          while not done:

               self.expandNode();
               # If next node to be expanded is solution, the search
is done.
               if(len(self.priorityQ) == 0):
                    print("Priority Q Empty!")
                    done = True;
                    return
```

```
                if(self.priorityQ[-1].tMap == self.goalMap):
                        done = True;

            print("TOTAL EXPANDED:", self.expandedNodes);
            # Return the solution node
            return self.priorityQ[-1];

    # Expands the first node in priority queue and adds children to
the priority queue.
    def expandNode(self):
#           print("ASTAR EXPANDING NODE")
            match = False

            # Only expand the node if it has not already been
explored.
            nNode = self.priorityQ.pop();
#           if(nNode in self.explored):
#                   return 1;

            for e in self.explored:
                    if e.tMap == nNode.tMap:
                            return 1;

            potChildren = nNode.GenerateChildren();
            self.expandedNodes += 1;

            self.explored.append(nNode);

            # Add children to queue
            for i in potChildren:
                    match = False
                    for j in self.explored:
                            if i.tMap == j.tMap:
                                    match = True;


                    if not match:
                            self.priorityQ.append(i)
                            self.totalNodes += 1;
                            if(self.totalNodes % 100 == 0):
                                    print("Nodes Generated:",
self.totalNodes);
```

```
#                    print("Seen Nodes: " ,len(self.explored));
#                    print(i);

          self.priorityQ.sort(key=self.getNodeCost, reverse=True);
          return 0;

     def getNodeCost(self, inNode):
          return inNode.fCost;

class StateNode:
     ###  CLASS ATTRIBUTES  ###
     ###     list tMap      ### >>> Element: [ [R1], [R2]. [R3] ]
     ###      int heu       ### (stands for heuristic)
     ###      int dep       ### (stands for depth)
     ###      str heu T     ### (Stands for heu type)
     ###      int fCost     ###
     ###      str lastAction###

     ##### CLASS METHODS ####
     def __init__(self, inMap, goMap, depth, heuType="A",
lastAction=None, lastNode=None):
          self.tMap = inMap;
          self.gMap = goMap;
          self.dep = depth;

          self.heuT = heuType;
          self.heu = getHeuristicMan(self.tMap, goMap);
          if(self.heuT != "A"):
               self.heu += (2*getLinearConflicts(self.tMap,
self.gMap));
          self.fCost = self.heu + self.dep;

          self.lastAction = lastAction;
          self.lastNode = lastNode;


     # Gets location of empty tile slot.
     def GetEmptyPos(self):
          found = False;
          xp = 0;
          yp = 0;

          while not found and (yp < 3):
```

```
                if(self.tMap[yp][xp] == 0):
                    found = True;
                else:
                    xp += 1;
                    if(xp > 2):
                        xp = 0;
                        yp += 1;

            return (yp, xp);

        # Gets list of potential children of current state.
        def GenerateChildren(self):
            zPos = self.GetEmptyPos();
            potChildren = [];

            # Check LEFT and swap for new state if valid.
            if(zPos[1] > 0):
                nMap = copy.deepcopy(self.tMap);
                nMap[zPos[0]][zPos[1]] = nMap[zPos[0]][zPos[1]-1];
                nMap[zPos[0]][zPos[1]-1] = 0;
                potChildren.append(StateNode(nMap, self.gMap,
self.dep+1, self.heuT, "L", self));

            # Check RIGHT and swap for new state if valid.
            if(zPos[1] < 2):
                nMap = copy.deepcopy(self.tMap);
                nMap[zPos[0]][zPos[1]] = nMap[zPos[0]][zPos[1]+1];
                nMap[zPos[0]][zPos[1]+1] = 0;
                potChildren.append(StateNode(nMap, self.gMap,
self.dep+1, self.heuT, "R", self));

            # Check UP and swap for new state if valid.
            if(zPos[0] > 0):
                nMap = copy.deepcopy(self.tMap);
                nMap[zPos[0]][zPos[1]] = nMap[zPos[0]-1][zPos[1]];
                nMap[zPos[0]-1][zPos[1]] = 0;
                potChildren.append(StateNode(nMap, self.gMap,
self.dep+1, self.heuT, "U", self));

            # Check DOWN and swap for new state if valid.
            if(zPos[0] < 2):
                nMap = copy.deepcopy(self.tMap);
                nMap[zPos[0]][zPos[1]] = nMap[zPos[0]+1][zPos[1]];
```

```
                nMap[zPos[0]+1][zPos[1]] = 0;
                potChildren.append(StateNode(nMap, self.gMap,
self.dep+1, self.heuT, "D", self));

        return potChildren;

    # Gives info on self in desired output format.
    def getOutput(self):
        out = "";
        for row in self.tMap:
            for each in row:
                out += str(each) + " ";
            out += "\n";

        return out;

    def __str__(self):
        out = "";
        for row in self.tMap:
            for each in row:
                out += str(each) + " ";
            out += "\n";

        out += "Cost: " + str(self.fCost) + "\n";

        return out;


# Get number of linear conflicts
def getLinearConflicts(tMap, goalMap):
    conflicts = 0;

    for i in range(len(tMap)):
#       print("ROW", i)
        # Check row i for linear conflicts.
        for xx in range(len(tMap)):
#           print("->", xx)
            gp = getgoalPos(tMap, (i, xx), goalMap);

            # If current tile on row has goal on this row.
            if(gp[0] == i):
                for xxx in range(len(tMap)):
```

```
                              if(xxx > xx) and tMap[i][xx] != 0 and
tMap[i][xxx] != 0:
                                    gp = getgoalPos(tMap, [i, xxx],
goalMap);
                                    # If tile goal is also in the same
row
                                    if (gp[0] == i):
                                          if (xxx > xx and gp[1] <= xx)
or (xxx < xx and gp[1] >= xx):
                                                conflicts += 1;
#                                               print("---", tMap[i][xx],
"<->", tMap[i][xxx],"conflicts:", conflicts)

#           print("COL", i)
            # Check row i for linear conflicts.
            for yy in range(len(tMap)):
#                 print("->", yy)
                  gp = getgoalPos(tMap, [yy, i], goalMap);

                  # If current tile on column has goal on this column.
                  if(gp[1] == i):
                        for yyy in range(len(tMap)):
                              if(yyy > yy) and tMap[yy][i] != 0 and
tMap[yyy][i] != 0:
                                    gp = getgoalPos(tMap, (yyy, i),
goalMap);
                                    # If tile goal is also in the same
column
                                    if (gp[0] == i):
                                          if (yyy > yy and gp[0] <= yy)
or (yyy < yy and gp[0] >= yy):
                                                conflicts += 1;
#                                               print("---", tMap[yy][i],
"<->", tMap[yyy][i],"conflicts:", conflicts)

      return conflicts;

# Get goal position of tile at startPos in tMap
def getgoalPos(tMap, startPos, goalMap):
      tFound = False;
      cX = 0;
      cY = 0;
```

```
    targetVal = tMap[startPos[0]][startPos[1]];
    # Find targetVal position in goalMap
    while not tFound:
        if(goalMap[cY][cX] == targetVal):
            tFound = True;
        else:
            if(cX < 2):
                cX += 1;
            else:
                cX = 0;
                cY += 1;

    return (cY, cX);

# Sums manhattan distances of tiles in inMap to goMap
def getHeuristicMan(inMap, goalMap):
    retSum = 0;
    # Get manhattan distance of each tile and add to sum.
    for i in range(3):
        for j in range (3):
            if(inMap[i][j] != 0):
                gp = getgoalPos(inMap, (i, j), goalMap);
                retSum += (abs(gp[1] - j) + abs(gp[0] - i));

    return retSum
```