```
#!/usr/bin/python
import os, sys, copy
clear = lambda: os.system('clear')

#final goal state to find
goal_state = [['1', '2', '3'], ['4', '5', '6'], ['7', '8', ' ']]
values = ['1', '2', '3', '4', '5', '6', '7', '8']

def initMenu():
	#default puzzle needed
	default = [['1', '2', '3'], ['4', '8', ' '], ['7', '6', '5']]

	print "8-Puzzle Game Developed by Rashid!"
	print "First choose \"default\" or \"create your own\" puzzle: \n"
	print "1. Default Puzzle: "
	printPuzzle(default)
	print "\n2. Create your own\n"

	temp = []
	while 1:
		choice = raw_input("Choice: ")
		if(choice == "1"):
			temp = default
			return default
		elif(choice == "2"):
			temp = ownPuzzle()
			return temp


#basic function to get user alg choice
def algorithmChoice():
	print "Please choose an algorithms to use:"
	print "1. Uniform Cost Search"
	print "2. A*, misplaced tile heuristic"
	print "3: A*, Manhattan distance heuristic\n"
	print "4. Exit\n"

	while(1):
		choice = raw_input("Choice: ")
		if(choice == '1'):
			return "ucostSearch"
		elif(choice == '2'):
			return "mtileHeuristic"
		elif(choice == '3'):
			return "manhattan"
		elif(choice == "4"):
			print "Thanks for playing!\nExiting Program..."
			sys.exit(0)
		else:
			print "Error: Entry invalid, please try again."
			print "Please choose an algorithms to use:"
			print "1. Uniform Cost Search"
			print "2. A*, misplaced tile heuristic"
			print "3: A*, Manhattan distance heuristic\n"
		return choice

#function to make own puzzle
def ownPuzzle():
	print "Please create your own puzzle and use a 0 for a blank spot:\n"

	blankUsed = 0
```

```
	while (blankUsed == 0):
		#will take input and check if it has the 0
		row_1 = raw_input("Please enter the first row. Use spaces between numbers: ")
		row_1 = row_1.split(' ')
		if(row_1.count('0')):
			for x in range(len(row_1)):
				if row_1[x] == '0':
					row_1[x] = ' '
					blankUsed = 1
		#will take input and check if it has the 0
		row_2 = raw_input("Please enter the second row. Use spaces between numbers: ")
		row_2 = row_2.split(' ')
		if(row_2.count('0')):
			for x in range(len(row_2)):
				if row_2[x] == '0':
					row_2[x] = ' '
					blankUsed = 1

		#will take input and check if it has the 0
		row_3 = raw_input("Please enter the third row. Use spaces between numbers: ")
		row_3 = row_3.split(' ')
		if(row_3.count('0')):
			for x in range(len(row_3)):
				if row_3[x] == '0':
					row_3[x] = ' '
					blankUsed = 1

	#takes all input and forms it into a temp array
	temp = []
	temp.append(row_1)
	temp.append(row_2)
	temp.append(row_3)

	return puzzle


def printPuzzle(array):
	print array[0]
	print array[1]
	print array[2]


class node:
	def __init__(self):
		self.heuristic = 0
		self.depth = 0
	def printNode(self):
		print " ___ ___ ___ "
		print "| "+self.originalPuzzle[0][0]+" | "+self.originalPuzzle[0][1]+" |
"+self.originalPuzzle[0][2]+" |"
		print "| "+self.originalPuzzle[1][0]+" | "+self.originalPuzzle[1][1]+" |
"+self.originalPuzzle[1][2]+" |"
		print "| "+self.originalPuzzle[2][0]+" | "+self.originalPuzzle[2][1]+" |
"+self.originalPuzzle[2][2]+" |"
		print " ___ ___ ___ "
		return ""
	def newPuzzle(self, puzzle):
		self.originalPuzzle = puzzle
	def puzzleLenght(self, puzzle):
		return len(puzzle)
```

```
def copyNodes(puzzle):
	#copy function does work
	#need deepcopy to copy object through whole array
	newNode = copy.deepcopy(puzzle)
	return newNode


def expandUp(puzzle):
	newNodes = []
	directionUP = copyNodes(puzzle)
	for x in puzzle:
		blankIndex = 0
		#see if there is a blank tile in current row
		if(x.count(' ') == 1):
			#see if it isnt the same row in directionUp matrix
			if(x != directionUP[0]):
				#get index of blank space
				blankIndex = x.index(' ')
				#check to see if x is same in second row
				if(x == puzzle[1]):
					#does following calcs to switch blank with above tile
					directionUP[1][blankIndex] = directionUP[0][blankIndex]
					directionUP[0][blankIndex] = ' '
					newNodes.append(directionUP)
				else:
					#does following calcs to switch blank with above tile
					directionUP[2][blankIndex] = directionUP[1][blankIndex]
					directionUP[1][blankIndex] = ' '
					newNodes.append(directionUP)
	return newNodes

def expandDown(puzzle):
	newNodes = []
	directionDown = copyNodes(puzzle)
	for x in puzzle:
		blankIndex = 0
		#see if there is a blank tile in current row
		if(x.count(' ') == 1):
			#see if it isnt the same row in directionUp matrix
			if(x != directionDown[2]):
				#get index of blank space
				blankIndex = x.index(' ')
				#check to see if x is same in second row
				if(x == puzzle[0]):
					#does following calcs to switch blank with down tile
					directionDown[0][blankIndex] = directionDown[1][blankIndex]
					directionDown[1][blankIndex] = ' '
					newNodes.append(directionDown)
				else:
					#does following calcs to switch blank with down tile
					directionDown[1][blankIndex] = directionDown[2][blankIndex]
					directionDown[2][blankIndex] = ' '
					newNodes.append(directionDown)
	return newNodes

def expandLeft(puzzle):
	newNodes = []
	directionLeft = copyNodes(puzzle)
	for x in directionLeft:
```

```
            #see if there is a blank tile in current row
            if(x.count(' ') == 1):
                  #make sure there isnt a blank tile in left space
                  if(x.index(' ') != 0):
                        #switches blank space with left index
                        tempindex = x.index(' ')
                        x[tempindex] = x[tempindex - 1]
                        x[tempindex - 1] = ' '
                        newNodes.append(directionLeft)
      return newNodes

def expandRight(puzzle):
      newNodes = []
      directionRight = copyNodes(puzzle)
      for x in directionRight:
            #see if there is a blank tile in current row
            if(x.count(' ') == 1):
                  #make sure there isnt a blank tile in right space
                  if(x.index(' ') != 2):
                        #switches blank space with left index
                        tempindex = x.index(' ')
                        x[tempindex] = x[tempindex + 1]
                        x[tempindex + 1] = ' '
                        newNodes.append(directionRight)
      return newNodes


def organizeList(puzzleList):
      #using bubblesort algorithm, organize by first higher cost to lower costs
      for x in xrange(len(puzzleList)-1, 0, -1):
            for i in xrange(x):
                  cost1 = puzzleList[i].heuristic + puzzleList[i].depth
                  cost2 = puzzleList[i+1].heuristic + puzzleList[i+1].depth
                  if(cost1 > cost2):
                        #swap puzzles
                        #use temp to store puzzle be overwritten
                        temp = puzzleList[i]
                        puzzleList[i] = puzzleList[i+1]
                        puzzleList[i+1] = temp

      return puzzleList

def mtile(puzzle):
      count = 0
      for row in range(len(puzzle)):
            for col in range(len(puzzle)):
                  #make sure we arent looking at blank space
                  if(puzzle[row][col] != ' '):
                        if(puzzle[row][col] != goal_state[row][col]):
                              #need to determine heuristic with all posbilities that arent in the right
place,
                              #such that we increase worst case by 1
                              count = count + 1
      return count

def manhattan(puzzle, values):
      count = 0
      index_row1 = 0
      index_col1 = 0
      index_row2 = 0
      index_col2 = 0
```

```
	for x in values:
		for row in range(len(puzzle)):
			for col in range(len(puzzle)):
				if(x == puzzle[row][col]):
				#check to ensure that the value we're looking for is equal to current puzzle index
					index_row1 = row;
					index_col1 = col;
				if(x == goal_state[row][col]):
				#check to see that the values we're looking for is equal to "goal_state" index
					index_row2 = row;
					index_col2 = col;

		#takes the abs valies of the two with the difference from the goal state from the current state
		temp2 = abs(index_row2 - index_row1)
		temp1 = abs(index_col2 - index_col1)
		count += temp1 + temp2
	return count

def calcHeuristic(puzzleList, choice):
	#function to determine heuristic instead of recalling over and over.
	heuristic = 0
	if(choice == "ucostSearch"):
		heuristic = 1
	if(choice == "mtileHeuristic"):
		heuristic = mtile(puzzleList.originalPuzzle)
	if(choice == "manhattan"):
		heuristic = manhattan(puzzleList.originalPuzzle, values)

	return heuristic


def search(puzzle, choice):
	#holds the temporary nodes we'll use to check
	temp = []
	#number of nodes we iterate through on the output screen
	totalNodes = 0
	#queue size to output for goal results
	queueSize = 0

	#going to be our temp list of puzzles to iterate through
	puzzleList = node()
	puzzleList.newPuzzle(puzzle)
	puzzleList.depth = 0

	#calculate heuristics of current node and iterate through this within while(1) loop
	puzzleList.heuristic = calcHeuristic(puzzleList, choice)
#	print puzzleList.heuristic

	#append the first puzzle passed in from function call, usually default or typed puzzle from user input
	temp.append(puzzleList)

#	print temp[0].originalPuzzle

	#while we havent reached goal state yet
	while 1:
		#if no more puzzles in the list to iterate, then puzzle not solvable
		if(len(temp) == 0):
			print "no more possible moves"
			sys.exit(0)

		#take front node from the puzzleList
```

```
		frontNode = node()
		#determine depth and heuristic
		frontNode.originalPuzzle = temp[0].originalPuzzle
		frontNode.depth = temp[0].depth
		frontNode.heuristic = temp[0].heuristic


		#default required output from documentation
		t = frontNode.originalPuzzle
		print t
		print "Expand node with g(n) = ", frontNode.depth
		print "and h(n) = ", frontNode.heuristic, "is: \n", frontNode.printNode()
		print "Expanding...\n"

		#pop the first node from the list, iterate through next one
		temp.pop(0)

		#check to see if goal state has been found
		if(goal_state == t):
			print "Goal State Reached"
			print "Number of nodes expanded: ", totalNodes
			print "Size of the queue: ", queueSize
			print "Final State Depth: ", frontNode.depth
			break

		#check all different possibilities of moving left, right, up, and down
		allMovesPossible = expandUp(frontNode.originalPuzzle)
		allMovesPossible += expandDown(frontNode.originalPuzzle)
		allMovesPossible += expandLeft(frontNode.originalPuzzle)
		allMovesPossible += expandRight(frontNode.originalPuzzle)

		#from all the possibilities, see what the heuristics is
		for x in allMovesPossible:
			nodeToCheck = node()
			nodeToCheck.newPuzzle(x)
			nodeToCheck.heuristic = calcHeuristic(nodeToCheck, choice)
			nodeToCheck.depth = frontNode.depth + 1
			temp.append(nodeToCheck)
			totalNodes = totalNodes + 1

			#update queue size to determine output for goal state result
			if(len(temp) > queueSize):
				queueSize = len(temp)
		#rearrange by best option first to worst option last using BUBBLESORT
		temp = organizeList(temp)

if __name__ == "__main__":
	while 1:
		print "\nRashid Goshtasbi AI 8\-Puzzle Solver"
		#input check asking for choice of algorithm
		choice = algorithmChoice()
		print "User choice: " + choice
		#input check to ask to use default or make own puzzle
		puzzle = initMenu()
		print "Starting Search...\n"
		#runs program with algorithm choice and puzzle
		search(puzzle, choice)
```