CS 550 Homework 2 - Search Problems

Student Name: Andrick Mercado

Are you a graduate student?

This is important. An assignment without a name, will be discarded. Please also include comments to explain your code. If you get full points, I will only check to make sure you didn't just trick the grading script. If you did not get full points, I need to evaluate for partial credit.

If you find errors in this notebook, please email me at wjwilson@sdsu.edu I will fix it and redistribute.

Problem 1

Write a recursive function to find a number in an unordered list and return the index. Assume that the number appears only once. **DO NOT** use built in Python list methods to find the index. The grading script is not intelligent enough to determine if followed the directions. I will manually check that you used a recursive search.

A recursive function might not be the best choice to search an unordered list. We are using it for a warm up exercise.

In [1]:
```
def number_search(num_list=[], num_find=0 ):
    """
    Returns the index where num_find was found in num_list
    Return -1 if the number is not in the list

    Arguments:
    num_list
    num_find
    Returns:
    num_index
    """
    ### DO NOT CHANGE THE FUNCTION NAME OR ARGUMENTS ###
    ### IT WILL BREAK THE GRADING SCRIPT ###

    index = -1
    ## Your code here
    def number_search_recursive(num_list, num):
        if num_list[0] == num:#if its the first element
            return 0
        return 1 + number_search_recursive(num_list[1:], num)
    try:
        return number_search_recursive(num_list, num_find)
    except IndexError: #when an indice is return thats not valid
        return -1
    ## End your code

    return index

# Use this portion if you want to add unit tests to validate your code
```

In [2]:
```
# Test function
# Do not change
def test1(num_list,num_find):
    inx_val = -1
    if num_find in num_list:
        inx_val = num_list.index(num_find)
    if number_search(num_list,num_find) == inx_val:
           return True
    else:
           return False



num_list = [6,9,1,0,-12,8]
num_find = 9
if test1(num_list,num_find):
    print("Test 1 worked on list {} and searching for {}".format(num_list,num_find))
else:
    print("Test 1 FAILED on list {} and searching for {}".format(num_list,num_find))
num_list = [6,9,1,0,-12,8]
num_find = 10
if test1(num_list,num_find):
    print("Test 1 worked on list {} and searching for {}".format(num_list,num_find))
else:
    print("Test 1 FAILED on list {} and searching for {}".format(num_list,num_find))
```

```
Test 1 worked on list [6, 9, 1, 0, -12, 8] and searching for 9
Test 1 worked on list [6, 9, 1, 0, -12, 8] and searching for 10
```

Problem 2

Write a function to solve a number puzzle. Undergraduate student may solve the problem in anyway they choose, other than using the uninformed BFS I included. I will use a "0" to represent the empty square.

Graduate Students may not use an uninformed depth first search. Some optimization must be included. I will manually validate the code.

All Make sure to watch for cycles. You do not want to loop on a set of moves infinitaly.

The first few cells are code to set up the framework. You do not need to do anything with them, even though it may be useful to skim them to help you with your section. I will try to write enough comments for this to make sense. If you prefer to write your own functions instead of using the helpers I provided, that is fine. You just need to make sure that when the test script calls the student_solve method that it returns the solution path.

You must run these cells before you run your code.

In [3]:
```
import random
import math
import copy


class EightPuzzle(object):
```

```
    def __init__(self,board=[],debug=False):
        self._parent = None
        if board != []:
            self.board = board
        else:
            self.generate_random()
        self.debug = debug
        self._depth = 0
        self.solved_board = [[0,1,2],[3,4,5],[6,7,8]]

    def generate_random(self):
        """ This function generates a random 3x3 board
        it is useful for testing
        """

        valid_board = False
        self.board = []
        while not valid_board:
            entries = [x for x in range(9)]
            random.shuffle(entries)
            for i in range(3):
                line =[]
                for j in range(3):
                    line.append(entries.pop())
                self.board.append(line)
            if  self.isSolvable():
                valid_board = True
            else:
                self.board = []


    ### TO DO - LOOP UNTIL YOU HAVE A SOLVABLE BOARD

    def _swap_and_clone(self,a):
        """
        This is borrowed from the same around as _clone
        It will is used when you make a move.
        You create a new board then swap the items in the move

        Since I will only ever swap with 0, I probably should just find 0 as part of th
        """
        b = self.find(0)
        p = copy.deepcopy(self)
        p.swap(a,b)
        p._depth = self._depth + 1
        p._parent = self
        return p

    def swap(self,pos_a,pos_b):
        """
        Basic swap function
        """
        temp = self.board[pos_a[0]][pos_a[1]]
        self.board[pos_a[0]][pos_a[1]] = self.board[pos_b[0]][pos_b[1]]
        self.board[pos_b[0]][pos_b[1]] = temp

    def manhattan_sum(self,new_board):
        """
        This is a poorly written manhattan sum function
        You can use it or write a better one yourself.
        I am not going to explain the code, because it is an embarasment.
```

```
        I am passing new_board as a parameter instead of using self.board because
          you might want to send a potenial board instead of the current one
        """

        board_size = len(new_board)
        m_sum = 0
        for vals in range(board_size **2):
            # Where is it in solution
            # Where is it in current board
            b_pos = []
            s_pos = []
            found_both = False
            for row in range(board_size):
                for col in range(board_size):
                    if new_board[row][col] == vals:
                        b_pos.append(row)
                        b_pos.append(col)
                    if self.solved_board[row][col] == vals:
                        s_pos.append(row)
                        s_pos.append(col)
                    if len(s_pos) > 0 and len(b_pos) > 0:
                        found_both = True
                        break
                if found_both:
                    break
            m_dist = abs(s_pos[0] - b_pos[0]) + abs(s_pos[1] - b_pos[1])
            m_sum += m_dist
            #if self.debug:
            #    print ("Manhattan Dist for {} is {}".format(vals,m_dist))

        return m_sum

    def check_solved(self,board=None):
        """
        This is a trivial function that just looks if the two boards are the same
        It returns true if the solved board matches the current board
        """
        if board == None:
            board = self.board
        return self.solved_board == board

    def find(self, value):
        """returns the row, col coordinates of the specified value
           on the board """

        if value < 0 or value > len(self.board) ** 2:
            raise Exception("value out of range")

        for row in range(len(self.board)):
            for col in range(len(self.board)):
                if self.board[row][col] == value:
                    return row, col

    def legal_moves(self):
        """
        This will return a list of tuples that are adjacent to the free square
        In our case, the free square is represented as a 0
        """

        blank_row, blank_col = self.find(0)
        peek_in = []
```

```
        if blank_row >= 0 and blank_row < (len(self.board) - 1):
            # Look below
            peek_in.append([blank_row+1,blank_col])
        if blank_row > 0 and blank_row < (len(self.board) ):
            # Look above
            peek_in.append([blank_row-1,blank_col])
        if blank_col >= 0 and blank_col < (len(self.board) - 1):
            # Look right
            peek_in.append([blank_row,blank_col+1])
        if blank_col > 0 and blank_col < (len(self.board) ):
            # Look left
            peek_in.append([blank_row,blank_col-1])

        return peek_in

    def print_board(self,board):
        for row in range(len(board)):
            print("-"*(4*len(board)+2))
            print("|",end=' ')
            for col in range(len(board)):
                if board[row][col] == 0:
                    print(' ',end=" | ")
                else:
                    print(board[row][col], end=' | ')
            print()
        print("-"*(4*len(board)+2))

    def print_path(self,path):
        for i in range(len(path)):
            print("\nBoard",i)
            self.print_board(path[i])

    def getInvCount(self,arr):
        inv_count = 0
        empty_value = 0
        for i in range(0, 9):
            for j in range(i + 1, 9):
                if arr[j] != empty_value and arr[i] != empty_value and arr[i] > arr[j]:
                    inv_count += 1
        return inv_count


    # This function returns true
    # if given 8 puzzle is solvable.
    def isSolvable(self) :
        # Count inversions in given 8 puzzle
        inv_count = self.getInvCount([j for sub in self.board for j in sub])
        # return true if inversion count is even.
        return (inv_count % 2 == 0)
```

Example Solution

This is my example solution. I am basing it on an uninformed breadth first search. Please do not reuse this. It is only meant to show an example of how to use the EightPuzzle class. The efficiency is very bad. If you run it, you will see an asterisk in the brackets for a long time. That means it thinks it is still running.

I found an interesting solver that I used to check their results against what I was doing: https://deniz.co/8-puzzle-solver/ When I tested one of my boards, it returned a result in 400 iterations using A+ and 40000 using breadth first.

During testing, I discovered that not every board is solvable. I found a post that gave me code on how to check it. https://www.geeksforgeeks.org/check-instance-8-puzzle-solvable/

In [4]:

```
from collections import deque

# Dont tell anyone I am using a global variable
visited = []

class BreadthFirst(EightPuzzle):
    def __init__(self,board=[],debug=False):
        # Constructor for my subclass
        #self.visited = [] # I am keeping track of visited boards separately than my pu
        # this is not space efficient, but I am being lazy
        super().__init__(board,debug)

    def expand_legal_moves(self):
        # We are using self.legal_moves() and self._swap_and_clone
        expansion = deque()
        global visited
        for move in self.legal_moves():
            p = self._swap_and_clone(move)
            if p.board not in visited: # Dont add the board if we have seen it board
                visited.append(p.board)
                expansion.append(p)

        #if len(expansion ) ==0 and self.debug:
        #    print("A board was a dead end")

        return expansion

    def solution(self):

        if not self.isSolvable():
            print("You gave me a bad puzzle")
            return []

        global visited
        visited = []
        puzzle_path= [] # Not sure if I need this. Test without it unless that doesnt w

        # self.board is my current state, I am going to push it on visited
        current_board = self
        visited.append(current_board.board)
        frontier = self.expand_legal_moves()

        visit_print = 10000 # I will use this later to print every 10000 board visits
        visit_print_inc = visit_print

        # Notice that I am not using a recursive function
        # If I was, I wouldn't be initializing things in the same way

        while len(frontier) > 0 and not self.check_solved(current_board):
            # If the frontier is empty, we quit with a fail
            # If it is solved, then we are done
```

```
            # Expand the first item in the frontier (if it is not the solution)
            q1 = frontier.popleft()
            if q1.check_solved():
                current_board = q1
                break # We can quit now. We found a solution
            else:
                # Expand q1 and add it to the frontier
                frontier.extend(q1.expand_legal_moves())

            # Just so I can see progress
            if len(visited) > visit_print:
                visit_print += visit_print_inc
                print("We have {} boards on the frontier now".format(len(frontier)))
                print("We have visited {} boards".format(len(visited)))
            if len(visited) > 360000:
                print("We shouldn't even get this big")
                current_board = q1
                break
             # temporary while writing

         # unwrap parents of current_board to get puzzle path
        puzzle_path= [current_board.board]
        while current_board._parent != None:
                current_board = current_board._parent
                puzzle_path.append(current_board.board)
        return puzzle_path

# I used this test case to see if it could handle a trivial change
puzzle = BreadthFirst(board=[[1,2,0],[3,4,5],[6,7,8]],debug=True)
#puzzle = BreadthFirst(board=[[4,2,0],[3,1,5],[7,6,8]],debug=True)
path = puzzle.solution()
print(puzzle.check_solved(path[0]))
puzzle.print_path(path)
```

```
True

Board 0
--------------
|   | 1 | 2 |
--------------
| 3 | 4 | 5 |
--------------
| 6 | 7 | 8 |
--------------

Board 1
--------------
| 1 |   | 2 |
--------------
| 3 | 4 | 5 |
--------------
| 6 | 7 | 8 |
--------------

Board 2
--------------
| 1 | 2 |   |
--------------
| 3 | 4 | 5 |
--------------
| 6 | 7 | 8 |
--------------
```

Your Solution

Put everything you do in the studentSolution subclass. You can define any methods you want and override my methods from the superclass You just must return the solution path from the the solution() method in a way that passes the test script.

Explain your solution Work through the logic and explain it here. It is good for planning and good for partial credit if your code has problems

- My current solution gets the job done it is not the most efficient as it doesnt go back to the parents to check other child, ,my current solution is linear so to speak.

In [5]:

```
import numpy as np
from itertools import chain
import time

class Node:

    def __init__(self,parent, child, f, g ,h):
        self.parent = parent
        self.child = child
        self.f = f
        self.g = g
        self.h = h

    '''Get and setters for node daTa'''
    def set_parent(self, parent):
        self.parent = parent

    def get_parent(self):
        return self.parent

    def set_h(self, hn):
        self.h = hn

    def get_h(self):
        return self.hn

    def set_g(self, gn):
        self.g = gn

    def get_g(self):
        return self.g

    def get_f(self):
        return self.g + self.h

    def get_board(self):
        return self.child

    '''DEtermines legal moves and their corresponding cost'''
    def expand_node(self,tree, explored_nodes, current_node, goal_node, location_zero,
        expand_node_list = [list(item.get_board()) for item in explored_nodes]

        explored_nodes.append(current_node)
        current_node_array = np.asarray(current_node.get_board())
        ''' works similar to legal moves method, but it checkks on a 1d array'''
```

```
            '''|0 x x|
               |3 x x|
               |6 x x|'''
            is_left_border = [location_zero != 0,location_zero != 3,location_zero != 6]
            if all(is_left_border):#location_zero != 0 and location_zero != 3 and location_
                node_copy = current_node_array.copy()
                temp = node_copy[location_zero - 1]
                node_copy[location_zero - 1] = current_node_array[location_zero]
                node_copy[location_zero] = temp#move the location of zero
                distance = self.manhattan_distance(node_copy, goal_node)#calculate cost of

                if not list(node_copy) in expand_node_list:#if its not a child we make it a
                    tree.append(Node(current_node, node_copy, 0, g_value, distance))

            '''|x x x|
               |x x x|
               |6 7 8|'''
            is_bottom_border = [location_zero != 6,location_zero != 7,location_zero != 8]
            if all(is_bottom_border):#location_zero != 6 and location_zero != 7 and locatio
                node_copy = current_node_array.copy()
                temp = node_copy[location_zero + 3]
                node_copy[location_zero + 3] = current_node_array[location_zero]
                node_copy[location_zero] = temp#move the location of zero
                distance = self.manhattan_distance(node_copy, goal_node)
                if not list(node_copy) in expand_node_list:#if its not a child we make it a
                    tree.append(Node(current_node, node_copy, 0, g_value, distance))#calcul

            '''|0 1 2|
               |x x x|
               |x x x|'''
            is_top_border = [location_zero != 0,location_zero != 1,location_zero != 2]
            if all(is_top_border):
                node_copy = current_node_array.copy()
                temp = node_copy[location_zero - 3]
                node_copy[location_zero - 3] = current_node_array[location_zero]
                node_copy[location_zero] = temp#move the location of zero
                distance = self.manhattan_distance(node_copy, goal_node)
                if not list(node_copy) in expand_node_list:#if its not a child we make it a
                    tree.append(Node(current_node, node_copy, 0, g_value, distance))#calcul

            '''|x x 2|
               |x x 5|
               |x x 8|'''
            is_right_border = [location_zero != 2,location_zero != 5,location_zero != 8]
            if all(is_right_border):
                node_copy = current_node_array.copy()
                temp = node_copy[location_zero + 1]
                node_copy[location_zero + 1] = current_node_array[location_zero]
                node_copy[location_zero] = temp#move the location of zero
                distance = self.manhattan_distance(node_copy, goal_node)
                if not list(node_copy) in expand_node_list:#if its not a child we make it a
                    tree.append(Node(current_node, node_copy, 0, g_value, distance))#calcul

''' Stores solution list of the moves from start to goal'''
solutionList = []

class studentSolution(EightPuzzle):

    def __init__(self, board=[], debug=False):
        super().__init__(board, debug)
```

```
    def least_cost_child(self, tree):
        cost_tree = []
        for i in range(len(tree)):
            cost_tree.append(tree[i].get_f())
        cost = min(cost_tree)
        index = cost_tree.index(cost)
        return index

    def solution_list(self,explored_nodes):
        path_to_solution = []
        cur = explored_nodes.pop()

        while explored_nodes[0] != cur:#while not root
            path_to_solution.append(cur)
            cur = cur.get_parent()

        path_to_solution.append(explored_nodes[0])#add the root
        path_to_solution.reverse()#reverse for start to end, instead of end to start

        for i in path_to_solution:
            solutionList.append(list(i.get_board()))

    def manhattan_distance(self,start, goal):
        start = np.asarray(start).reshape(3, 3)#if formatted this way we can use the wh
        goal = np.asarray(goal).reshape(3, 3)
        distance = 0

        for i in range(8):
            (a, b) = np.where(start == i + 1)
            (x, y) = np.where(goal == i + 1)
            distance = distance + abs((a - x)[0]) + abs((b - y)[0])

        return distance

    def solution(self):
        start = list(chain.from_iterable(self.board))#convert 2d array to 1d
        goal = list(chain.from_iterable(self.solved_board))#makes it easier to use othe
        solutionList.clear()#every time we call clears the solution from previous ones

        if not self.isSolvable():
            print ('You gave me a bad puzzle')
            return []

        #self.print_board(self.board)

        start = Node(None, start, 0, 0, 0)
        goal = Node(None, goal, 0, 0, 0)
        goal_board = np.asarray(goal.get_board())
        tree = [start]
        tree[0].set_h(self.manhattan_distance(start.get_board(),goal.get_board()))
        explored_nodes = []

        '''PRINT SOLUTION'''
        start = time.time()
        while True:#here we expand the nodes depending on f least value on all legal mo
            current_node = tree.pop(self.least_cost_child(tree))
            cur_g_value = current_node.get_g() + 1
            if np.array_equal(np.asarray(current_node.get_board()), goal_board):#if cur
                explored_nodes.append(current_node)#add the last node to the list
                self.solution_list(explored_nodes)
                break#we have gathered the solution so we leave
```

```
            elif not np.array_equal(current_node, goal_board):#if current node or board
                Node.expand_node(self,tree, explored_nodes, current_node,goal_board,
                                 np.where(np.asarray(current_node.get_board())== 0)[0][
        end = time.time()
        #print("Elapsed time: ", (end - start), "seconds")
        return solutionList

''' TESTING ZONE'''
'''
puzzle = studentSolution()#board=[[4, 2, 0], [3, 1, 5], [7, 6, 8]])  # board=[[1,2,0],[
puzzle_path = puzzle.solution()
print ('Solution to puzzle: ', puzzle_path)
print ('Puzzle length: ', len(puzzle_path))
'''
```

Out[5]:
```
"\npuzzle = studentSolution()#board=[[4, 2, 0], [3, 1, 5], [7, 6, 8]])  # board=[[1,2,
0],[3,4,5],[6,7,8]])\npuzzle_path = puzzle.solution()  \nprint ('Solution to puzzle: ',
puzzle_path)\nprint ('Puzzle length: ', len(puzzle_path))\n"
```

In [6]:
```
# This is my validation function for problem 2
# DO NOT EDIT
def problem2_checker():
    points = 0
    try:
        print("Testing with a simple board")
        puzzle = studentSolution(board=[[1,2,0],[3,4,5],[6,7,8]])
        puzzle_path = puzzle.solution()
        if 1 < len(puzzle_path) <= 4 :
            print("Your solution to the trivial puzzle has 4 or fewer in the path. +20
            points += 20
        else:
            print("You got a solution to the trivial puzzle. +10 points")
            points += 10
        puzzle = studentSolution(board=[[4,2,0],[3,1,5],[7,6,8]])
        puzzle_path = puzzle.solution()
        if 1 < len(puzzle_path) <= 150 :
            print("You solved the second puzzle with a path length of {}. +20 points ".
            points += 20
        else:
            print("You got a solution to the second puzzle with a path length of {}. +1
            points += 10
        print("Testing 3 random boards")
        for i in range(2):
            puzzle = studentSolution(  )
            print("Random board",i)
            puzzle.print_board(puzzle.board)
            puzzle_path = puzzle.solution()
            if 1 < len(puzzle_path) < 10000: # This is ridiculous limit. If the script
                print("Random puzzle {} was solved with a path length of {}. +20 point
                points +=20
    except:
        print("You got an exception. Returning {} points".format(points))
        #raise

    return points

total_points = 0
try:
    num_list = [6,9,1,0,-12,8]
    num_find = 9
```

```
    if test1(num_list,num_find):
        print("Problem 1 Test 1 worked on list {} and searching for {}".format(num_list
        total_points +=10
    else:
        print("Problem 1 Test 1 FAILED on list {} and searching for {}".format(num_list
    num_list = [6,9,1,0,-12,8]
    num_find = 10
    if test1(num_list,num_find):
        print("Problem 1 Test 2 worked on list {} and searching for {}".format(num_list
        total_points += 10
    else:
        print("Problem 1 Test 2 FAILED on list {} and searching for {}".format(num_list
except:
    print("You hit an exception at {} points.\n I am not handling errors.".format(total


total_points += problem2_checker()

print("The auto checker is assigning {} out of 100 points.".format(total_points))
print("If you see a flaw in the auto checker or assignment, email wjwilson@sdsu.edu")
```

```
Problem 1 Test 1 worked on list [6, 9, 1, 0, -12, 8] and searching for 9
Problem 1 Test 2 worked on list [6, 9, 1, 0, -12, 8] and searching for 10
Testing with a simple board
Your solution to the trivial puzzle has 4 or fewer in the path. +20 points
You solved the second puzzle with a path length of 19. +20 points
Testing 3 random boards
Random board 0
--------------
| 7 | 2 | 8 |
--------------
| 3 | 4 |   |
--------------
| 6 | 1 | 5 |
--------------
Random puzzle 1 was solved with a path length of 22. +20 points
Random board 1
--------------
| 5 | 3 | 1 |
--------------
| 4 | 7 |   |
--------------
| 6 | 2 | 8 |
--------------
Random puzzle 2 was solved with a path length of 22. +20 points
The auto checker is assigning 100 out of 100 points.
If you see a flaw in the auto checker or assignment, email wjwilson@sdsu.edu
```

if the program doesnt execute we most likely have an unsolvable board, stop kernel and re-run

first run always takes longer

In [ ]: