CS152 N-puzzle Assignment 1

Tomer Eldor

Minerva Schools

PuzzleNode Class

First, I built a PuzzleNode Class to store the state of each tile. Here I include all attributes and elements needed to implement an A* search, including parent and string methods.

Data structure for representing each board state

The data structure used to represent tiles state is a simple 1D array (python list). I'm working with this flat lists since it is the simplest datastracture which contains all the needed information yet lowest in complexity and computational resources (supposedly). It is also easier to use for some of the verifications and operations. So I'm flatteinning the given list.

String(self) methods for printing

Since, I read that a Str() impelmentation is needed, I interpreted it initially to represent the board visually in the most conveient way possible. Therefore this code first implements a visual nxn (square) representation of the board, to see most clearly the actual board state and inspect the validity of the moves and solutions. Later I read that the program should also print each step as sublists specifically. Therefore, when the *printit* flag is True, this program will print both the request single lines of nested sublist, followed by the visual representation fo the board along with the step number.

In [5]:

```
# Importing and setup
import heapq
import numpy as np

# I'm working with flattened lists as the tiles board state, I'll first
 define a flattenning funciton
def flatten(board):
    # if it's nested lists, flatten them. I do this with list comprehens
ion taking each tile at a time from each sublist
    if type(board[1])==list:
        board = [item for sublist in board for item in sublist]
    # else, it should be a list of ints or floats
    elif type(board[1])==int or type(board[1])==float:
        board = board
    # if it's neither, it's a wrong input and will raise an error.
    else:
        raise ValueError("Class 'PuzzleNode' got values that are not a s
ublist of ints nor a flat list of ints.")
    return board
```

```
class PuzzleNode():
    """ defining a class for each puzzle board node, used for each state
 in the frontier"""
    def __init__(self, n, values, cost, parent, heuristic_id):

        # initializing values for the PuzzleNode: n and the actual tiles
 values.
        self.n = n

        self.tiles = flatten(values)

        # Defining costs: we need to define the current step's cost so f
ar,
        # the huristic estimate for the cost to get to the goal,
        # and the total cost (g+h) of the full path using that node and
 getting to the goal
        self.cost = cost
        self.heuristic = heuristic_id(self.tiles)
        self.total_cost = self.cost + self.heuristic

        # The parent will be specified as an input when creating each bo
ard.
        # This is a pointer to that parent, for reconstructing the solut
ions.
        self.parent = parent

        # We'll want to store each board eventually in a "visited" set o
r dicitionary
        # (since they have O(1) search time and best for this lookup tab
le use.)
        # Therefore, we define how to hash a PuzzleNode object.
        self.hashvalue = hash(tuple(self.tiles))

    def __hash__(self):
        return self.hashvalue

    def __print__(self):
        # converting a print statement of these as a grid
        # first, I need to convert each digit to a string
        strings_list = [str(x) for x in self.tiles]
        # now split to rows of length n, by indexing from i to i + n per
 row,
        # and looping that for each in xrange(), which starts from the t
op 0, stops at the goal_state(length=n**2),and skips by n- dimension of
 the board
        rows = [" ".join(strings_list[i:i + self.n]) for i in xrange(0,
self.n**2, self.n)]
        return "\n".join(rows)

    ### add str method as list of lists.
    def __str__(self):
        # I've been working with the tiles values list as a flat 1D arra
y.
        # I later saw that we're asked to print them in the format of
 [[1,2,3][4,5,6,][7,8,0]].
        # I preferreed to print it as a board would actually look like,
```

```
 for which I have the next method for pretty printing.
        # However, if this is the requirement for output, here it will b
e converted to look like that.
        nested_list = [self.tiles[i : i + self.n] for i in range(0, 
self.n**2, self.n)]
        return str(nested_list)

    # add equating method: for checking if 2 PuzzleNodes are equal
    def __eq__(self, other):
        return self.tiles == other.tiles

print flatten(test1)
print flatten(test2)
```

```
[2, 3, 7, 1, 8, 0, 6, 5, 4]
[7, 0, 8, 4, 6, 1, 5, 3, 2]
```

SolvePuzzle function

Below we define the function solvePuzzle that accepts three arguments and returns three values. It is callable by using: steps, frontierSize, err = solvePuzzle(n, state, heuristic, print).

I'm starting by defining the input verification function that I'll later call from within the SolvePuzzle function.

In [6]:
```
from time import time ## This is for my own will to check performance ti
me; not in the specified instructions

#solvePuzzle(n, state, heuristic, print)
#goal_state = PuzzleNode(n = n, values = range(n**2), cost = 100, parent
=None, heuristic_id = heuristics[0])

def verify_input(initial_state,n):
    """As the problem is defined:
    'This problem easily generalizes to boards of size n^2 - 1, for any
 natural number n > 3.
    'Your program should work correctly for arbitrary n-by-n boards (for
 any 2 ≤ n < 128)'
    """
    err = 0 #assuming best intent... that there are no errors until foun
d guilty
    reason = "input was valid"
    initial_state = flatten(initial_state)     # flatten starting state
 if needed

    #verifying valid size range
    if n<2 or n>=128:
        err = -1
        #print "size error" #debug
        reason = "N is not between 2 and 128"

    #verifying correct size (square)
    if len(initial_state) != n**2:
        err = -1
        reason = "board size isn't N^2"
```

```
    # verifying that there are only numbers, only the numbers form 0 to
 n**2, and they all appear once
    sorted_initial_state = sorted(initial_state)
    valid_list = range(n**2)
    if sorted_initial_state != valid_list:
        err = -1
        reason = "Are you sure your N and tiles match? The tiles aren't
 from 0 to N^2." #debug
        #raise ValueError("The numbers aren't from 0 to your specified N
^2. Please enter numbers from 0 to N^2 -1 as the initial state with corr
esponding N, formatted as a list of sublists of equal size n, or as a fl
attened down list.")

    #debug: testing edge cases, I want to raise an error if input is inv
alid and not wait for the program to finish.
    if err == -1:
        raise ValueError(reason)
    return err, initial_state, reason


def solvePuzzle(n, initial_state, heuristic_id, printit=True):
    start_time = time()

    ### Let's initialize reporistories: ###
    # a Heap where we'll store our boards, our "frontier"
    frontier_tree = []
    # dictionary for costs of boards
    cost = {}
    # a dictionary for visited nodes. * why dictionary/set? these are gr
eat for adding non-duplicates unique sets, and use hashing, thus have co
nstant lookup time of O(1) so they're great for these lookup tables. The
refore I implemented a hash method
    visited = {}

    ### Let's verify out input (and get the corrected initail state and
 error code) ###
    err, initial_state, reason = verify_input(initial_state,n)
    #DATA STRUCTURE FOR STORING EACH BOARD: a 1D array. I'm working with
 flat lists since it is simpler on complexity and computational power (s
upposedly) and contains all the needed information.  So I'm flatteinning
 the given list.

    # Initialize our initial state as a board node
    starting_state = PuzzleNode(n=n, values=initial_state, cost=0, paren
t=None, heuristic_id = heuristic_id)

    # Printing initial state at start (if we want to print)
    if printit == True:
        print "Starting to solve from: "
        print str(starting_state)
        print "Solving..."

    # GOAL: If we are to generalize from the goal of ordered numbers 0-8
 for the 3x3 grid, then the goal shuold always be range(n**2)
    goal_state = PuzzleNode(n = n, values = range(n**2), cost = 100, par
ent=None, heuristic_id = heuristic_id)
```

```
    # Initializing our frontier with the starting state and its cost
    # DATA STRUCTURE TO STORE frontier: HEAPQ (AS SPECIFIED IN THE EMAI
L).
    # HEAPQ is good built-in library to serve for trees and heaps struct
ures.
    # We really want a PriorityQueue, but since this wasn't explicitly a
llowed,
    # we should be able to imitate a priority queue by storing tuples of
 the (cost, board)
    # Thus the HeapQ reorders it as a priority queue by the cost.
    heapq.heappush(frontier_tree, (starting_state.total_cost, starting_s
tate))

    ### A* (A-STAR) SEARCH ALGORITHM ###
    # Let's search the frontier_tree for solutions as long as it still h
as nodes! (While Loop)
    # initializing counters and holdkeepers
    inspected_states_counter = 0 # I want to keep track of total inspect
ed steps counter
    # Max Frontier size is the maximal size that our frontier has been a
t any stage.
    # for that, I set the initial frontierSize to 0, and whenever I have
 a longer priorityqueue I'll reset the frontierSize to that. That way it
 will end up as the maximal.
    frontierSize = 0
    # we want to traverse our entire frontier_tree until exausting it
    while frontier_tree:
        # Pop the last state from the frontier and work from there
        curr_board_and_cost = heapq.heappop(frontier_tree) # my tree con
tains the board and cost attached as a tuple
        current_state = curr_board_and_cost[1] # the board is the second
 element of the tuple (cost, board)
        inspected_states_counter += 1 #increment counter of inspected st
ates
        #print("tree size: {}, curr state popped: \n{}".format(len(front
ier_tree), str(current_state))) #debug

        # A* checks if we're finished (at goal), and break if we are!
        # print("Heuristic to goal: {}".format(heuristic_id(current_stat
e.tiles) )) #debug
        if heuristic_id(current_state.tiles) == 0:
            #goal_state = current_state # debug
            break # we are done!

        #### INSPECT LEAF NODES ####

        # Defining the position of the empty tile, 0
        index_0 = current_state.tiles.index(0) #finding the index of the
 "0", the empty slot
        row0 = index_0 / n # the "X axis" index of it is the index divid
ed by n (number of columns/rows)
        col0 = index_0 % n # the "Y axis" index of it is the index modul
o n (number of columns/rows, the remainder translates to how many spots
 to the right..)

        # check possible next moves (where can we swap the empty slot wi
th). Starting with the current index of 0, checking which neighboring in
```

```
dexes are availalbe for it to swap with.
        moves_list = []
        if(col0 - 1 >= 0): moves_list.append([row0, col0 - 1])  # if we
 can move left ,add that move
        if(col0 + 1 <  n): moves_list.append([row0, col0 + 1])  # if we
 can move right, add that move
        if(row0 - 1 >= 0): moves_list.append([row0 - 1, col0])  # if we
 can go down ,add that move
        if(row0 + 1 <  n): moves_list.append([row0 + 1, col0])  # if we
 can move up ,add that move

        # now check suitability for each possible move
        for move in moves_list:
            new_state = current_state.tiles[:] #copy values of tiles int
o the new state
            # after this move, the new index for 0 will be just the line
ar combination of the indexes: x*n (row number*amount of items per row)
 + y (position within row, like remainder).
            index_0_new = move[0]*n + move[1]
            # SWAP tiles, by simultaneous multiple "= assignment
            new_state[index_0], new_state[index_0_new] = new_state[index
_0_new], new_state[index_0]
            # make a new PuzzleNode class of it. we'll define the cost a
s +1 more than current, since we define the cost of each step as 1
            new_PuzzleNode = PuzzleNode(n = n, values = new_state, cost
= current_state.cost + 1, parent = current_state, heuristic_id = heurist
ic_id)
            new_cost = new_PuzzleNode.total_cost
            #debug: #print "new_PuzzleNode: \n", new_PuzzleNode.__print_
_() #debug

            # checking that the new board is NOT A PREVIOUSLY VISITED BO
ARD, OR that the new cost is SMALLER than an equal existing state's cost
 (to find a better path to a previously-visited state)
            # by having a hashing method to PuzzleNode class, we can ver
ify efficiently if it exists in the dictionary such as visited or
            if (new_PuzzleNode not in visited) or (new_cost < cost[new_P
uzzleNode]):
                #debug: #print("appending a new board \n {}".format(str
(new_PuzzleNode()))) #debug
                cost[new_PuzzleNode] = new_cost #reassign new cost
                visited[new_PuzzleNode] = 1 # insert an indicator that t
his board has been visited to the hashed location in the visited list.
                new_PuzzleNode.parent = current_state #setting current s
tate as parent
                heapq.heappush(frontier_tree,
(new_PuzzleNode.total_cost, new_PuzzleNode))

            # update frontier size if it's larger than the last maximal
 frontier size, by taking the max of them both.
            frontierSize = max(frontierSize,len(frontier_tree))

    ### RECONSTRUCTING THE SOLUTION
    ### Backtracking: we start from the goal node and backtrack through
 parents to recreate the path
    solution_steps = [] # initializing a list to contain solution steps
    curr_boardstate = current_state #starting with the last state we wer
```

```
e in = the goal state
    while curr_boardstate != starting_state: #backtracking back from the
 goal through parents until reaching the starting state
        #print curr_boardstate.__str__()
        solution_steps.insert(0,curr_boardstate) #intersting the parent
 before on the solution steps list; so that our solution is eventually o
rdered from start->goal
        curr_boardstate = curr_boardstate.parent #reassinging the curren
t step to its parent and iterating
    solution_steps.insert(0,starting_state) #now add the actual initial
 state as the first step (since the while loop stops when it reaches it
 and doesn't add it)
    steps = len(solution_steps)

    ### Printing our solution nicely, if asked: ###
    if printit == True:
        print("Took {} steps to reach solution.".format(steps))
        print("Max Frontier Size was {} (branching factor).".format(fron
tierSize))
        print("Finished with Error Code of {}: {}".format(err, reason))
        print("\n_____________________\nHere are the stages in sublists
 format: ")
        for step_index in range(steps):
            #print("\nStep {}:".format(step_index))
            print(str(solution_steps[step_index]))
        print("\n_____________________\nAnd here are the steps in a pret
ty visual square format")
        for step_index in range(steps):
            #print("")
            print("\nStep {}:".format(step_index))
            print(solution_steps[step_index].__print__())

    runtime = time() - start_time

    return steps, frontierSize, err, inspected_states_counter, runtime
```

Heuristics

Below I'm defining heuristics, starting with manhattan distance and then misplaced tiles. I'm first defining a function for calcualting the manhatten distance *of each tile* from its origin, then using it in a manhattan distance wrapper function. I then wrap them up under "heuristics" wrapper handler.

In [7]:
```
def misplacedTiles(tiles):
    tiles = flatten(tiles) # handle if input is a nested list and not a
 flat list
    misplaced = 0
    goal_state_list = range(len(tiles))
    # check for each tile if it is the same as the goal
    for tile in tiles:
        # mark misplaced tiles and add to counter
        if goal_state_list.index(tile) != tiles.index(tile) and tile!=0:
            misplaced += 1
            #print tile, ", misplaced:", misplaced
```

```
    return misplaced


def manhattanDist_per_tile(index, tile,n):
    # get indices of our tile
    tile_x = index / n
    tile_y = index % n

    # define goal state and where does this tile needs to reach (its goa
l indicies)
    goal_state_list = range(n**2) # generate goal state list
    goal_index = goal_state_list.index(tile) # find the desired indices
 for this tile
    goal_state_x = goal_index / n
    goal_state_y = goal_index % n

    # calculate manhattan distance: summing the horizontal (x axis) and
 vertical (y axis) distances
    manhattan_dist_tile = 0 # initialize manhattan distance measure per
 one tile
    manhattan_dist_tile += abs(tile_x - goal_state_x)
    manhattan_dist_tile += abs(tile_y - goal_state_y)
    return manhattan_dist_tile


def manhattanDist(tiles):
    tiles = flatten(tiles) # handle if input is a nested list and not a
 flat list
    manhatten_dist = 0     # initializing values

    #getting n and goal state
    n = int(len(tiles)**0.5)
    goal_state_list = range(len(tiles))

    # Calculating distance tile by tile and summing up
    for tile in tiles:
        # calculating the manhatten distance for misplaced tiles which a
ren't the empty slot 0:
        if goal_state_list.index(tile) != tiles.index(tile) and tile!=0:
            manhatten_dist += manhattanDist_per_tile(tiles.index(tile),
tile, n)
        #print tile, "manhatten_dist = ", manhatten_dist #debug

    return manhatten_dist

# WRAPPER / Function Handling list
heuristics = [misplacedTiles, manhattanDist]
```

Comparing Heuristics

In [73]:

```
## Wrapper function to test all test cases with all heuristics and compa
re
test1 = [[2,3,7],[1,8,0],[6,5,4]]
test2 = [[7,0,8],[4,6,1],[5,3,2]]
test3 = [[5,7,6],[2,4,3],[8,1,0]]

def test_heuristics(n,printit=False):
    global test1, test2, test3
    test_list = [test1,test2,test3]

    for test_board in test_list: #debug [:1]
        print("\nTesting Heuristics for board: {}".format(test_board))
        # run with Heuristic 0: misplacedTiles
        steps0, frontierSize0, err0, inspected0, runtime0 =
solvePuzzle(n = n, initial_state = test_board, heuristic_id =
heuristics[0], printit = printit)
        # run with Heuristic 1: manhattanDist
        steps1, frontierSize1, err1, inspected1, runtime1 =
solvePuzzle(n = n, initial_state = test_board, heuristic_id =
heuristics[1], printit = printit)
        # printing results in a mock-table style (since we're not allowe
d to import more libraries like Pandas for nicely presenting tables, oth
erwise I would)
        print("\t        Misplaced Tiles  vs  Manhattan Distance ")
        print("Steps:          \t    {} \t \t
{}".format(steps0,steps1))
        print("Frontier size:  \t    {} \t  {}".format(frontierSize0,fro
ntierSize1))
        print("Inspected total:\t    {} \t  {}".format(inspected0,inspec
ted1))
        print("Runtime (sec):\t       {0:10.3f}    {0:10.3f}".format(run
time0,runtime1))

from random import shuffle  # IMPORTING SHUFFLE ONLY TO RANDOMLY SHUFFLE
 TEST STARTING BOARDS FOR MYSELF
def test_random(n):
    # For robustness: RANDOMLY SHUFFLED NEW INITIAL STRATING BOARDS
    rand_start = range(n)
    shuffle(rand_start)
    test_heuristics(n)

# TEST ALL TEST CASES WITH ALL HEURISTICS
test1 = [[2,3,7],[1,8,0],[6,5,4]]
test2 = [[7,0,8],[4,6,1],[5,3,2]]
test3 = [[5,7,6],[2,4,3],[8,1,0]]
test_heuristics(n = 3, printit=False)
```

```
Testing Heuristics for board: [[2, 3, 7], [1, 8, 0], [6, 5, 4]]
                Misplaced Tiles  vs  Manhattan Distance
Steps:                      18            18
Frontier size:              565          67
Inspected total:            917          126
Runtime (sec):              0.153        0.153

Testing Heuristics for board: [[7, 0, 8], [4, 6, 1], [5, 3, 2]]
                Misplaced Tiles  vs  Manhattan Distance
Steps:                      26            26
Frontier size:              13066        640
Inspected total:            28725        1110
Runtime (sec):              4.000        4.000

Testing Heuristics for board: [[5, 7, 6], [2, 4, 3], [8, 1, 0]]
                Misplaced Tiles  vs  Manhattan Distance
Steps:                      29            29
Frontier size:              22177        1140
Inspected total:            67824        2070
Runtime (sec):              8.609        8.609
```

In [9]:
```
## MARKDOWN TABLE
```

Test for edge cases

1. TEST FOR PYTHON 3
2. Test for larger N - V
3. Test for smaller N - V
4. Test for wrong Ns or wrong inputs -V

In [95]:

```
# test for smaller boards: n=2
test4board = [[3,2],[1,0]]
smallboard = [[3,1],[2,0]]

print "Solving a 3block Puzzle, n=2"
verify_input(smallboard, n=2)
solvePuzzle(n = 2, initial_state = smallboard, heuristic_id =
heuristics[1], printit = True)
```

```
Solving a 3block Puzzle, n=2
Starting to solve from:
[[3, 1], [2, 0]]
Solving...
Took 7 steps to reach solution.
Max Frontier Size was 3 (branching factor).
Finished with Error Code of 0: input was valid

_____________________
Here are the stages in sublists format:
[[3, 1], [2, 0]]
[[3, 1], [0, 2]]
[[0, 1], [3, 2]]
[[1, 0], [3, 2]]
[[1, 2], [3, 0]]
[[1, 2], [0, 3]]
[[0, 2], [1, 3]]

_____________________
And here are the steps in a pretty visual square format

Step 0:
3 1
2 0

Step 1:
3 1
0 2

Step 2:
0 1
3 2

Step 3:
1 0
3 2

Step 4:
1 2
3 0

Step 5:
1 2
0 3

Step 6:
0 2
1 3
```

```
Out[95]: (7, 3, 0, 13, 0.0023069381713867188)
```

In [11]:

```
## TESTING FOR LARGER Ns:   4x4, "15-Puzzle"
# The complexity increases exponentially, or more than that, with
test15easy = [[1,2,3,6], [4,7,0,5], [8,9,10,11],[12,13,14,15]]
test15 = [[2,3,6,4], [1,15,0,8], [5,7,12,14],[10,9,13,11]]
print "Solving a 15 Puzzle"
solvePuzzle(n = 4, initial_state = test15easy, heuristic_id =
heuristics[1], printit = True)
```

```
Solving a 15 Puzzle
Starting to solve from:
[[1, 2, 3, 6], [4, 7, 0, 5], [8, 9, 10, 11], [12, 13, 14, 15]]
Solving...
Took 16 steps to reach solution.
Max Frontier Size was 177 (branching factor).
Finished with Error Code of 0: input was valid
```

```
Here are the stages in sublists format:
[[1, 2, 3, 6], [4, 7, 0, 5], [8, 9, 10, 11], [12, 13, 14, 15]]
[[1, 2, 3, 6], [4, 0, 7, 5], [8, 9, 10, 11], [12, 13, 14, 15]]
[[1, 0, 3, 6], [4, 2, 7, 5], [8, 9, 10, 11], [12, 13, 14, 15]]
[[1, 3, 0, 6], [4, 2, 7, 5], [8, 9, 10, 11], [12, 13, 14, 15]]
[[1, 3, 7, 6], [4, 2, 0, 5], [8, 9, 10, 11], [12, 13, 14, 15]]
[[1, 3, 7, 6], [4, 2, 5, 0], [8, 9, 10, 11], [12, 13, 14, 15]]
[[1, 3, 7, 0], [4, 2, 5, 6], [8, 9, 10, 11], [12, 13, 14, 15]]
[[1, 3, 0, 7], [4, 2, 5, 6], [8, 9, 10, 11], [12, 13, 14, 15]]
[[1, 0, 3, 7], [4, 2, 5, 6], [8, 9, 10, 11], [12, 13, 14, 15]]
[[1, 2, 3, 7], [4, 0, 5, 6], [8, 9, 10, 11], [12, 13, 14, 15]]
[[1, 2, 3, 7], [4, 5, 0, 6], [8, 9, 10, 11], [12, 13, 14, 15]]
[[1, 2, 3, 7], [4, 5, 6, 0], [8, 9, 10, 11], [12, 13, 14, 15]]
[[1, 2, 3, 0], [4, 5, 6, 7], [8, 9, 10, 11], [12, 13, 14, 15]]
[[1, 2, 0, 3], [4, 5, 6, 7], [8, 9, 10, 11], [12, 13, 14, 15]]
[[1, 0, 2, 3], [4, 5, 6, 7], [8, 9, 10, 11], [12, 13, 14, 15]]
[[0, 1, 2, 3], [4, 5, 6, 7], [8, 9, 10, 11], [12, 13, 14, 15]]
```

```
And here are the steps in a pretty visual square format

Step 0:
1 2 3 6
4 7 0 5
8 9 10 11
12 13 14 15

Step 1:
1 2 3 6
4 0 7 5
8 9 10 11
12 13 14 15

Step 2:
1 0 3 6
4 2 7 5
8 9 10 11
12 13 14 15

Step 3:
1 3 0 6
4 2 7 5
8 9 10 11
12 13 14 15

Step 4:
1 3 7 6
4 2 0 5
```

```
8 9 10 11
12 13 14 15

Step 5:
1 3 7 6
4 2 5 0
8 9 10 11
12 13 14 15

Step 6:
1 3 7 0
4 2 5 6
8 9 10 11
12 13 14 15

Step 7:
1 3 0 7
4 2 5 6
8 9 10 11
12 13 14 15

Step 8:
1 0 3 7
4 2 5 6
8 9 10 11
12 13 14 15

Step 9:
1 2 3 7
4 0 5 6
8 9 10 11
12 13 14 15

Step 10:
1 2 3 7
4 5 0 6
8 9 10 11
12 13 14 15

Step 11:
1 2 3 7
4 5 6 0
8 9 10 11
12 13 14 15

Step 12:
1 2 3 0
4 5 6 7
8 9 10 11
12 13 14 15

Step 13:
1 2 0 3
4 5 6 7
8 9 10 11
12 13 14 15
```

```
Step 14:
1 0 2 3
4 5 6 7
8 9 10 11
12 13 14 15

Step 15:
0 1 2 3
4 5 6 7
8 9 10 11
12 13 14 15
```

Out[11]:
```
(16, 177, 0, 165, 0.053732872009277344)
```

In [ ]:

Extension: Is the Puzzle Solvable?

Many have already studied this and found that this depends on the number of inversions. As stated by Mark Ryan (2004), in https://www.cs.bham.ac.uk/~mdr/teaching/modules04/java2/TilesSolvability.html (https://www.cs.bham.ac.uk/~mdr/teaching/modules04/java2/TilesSolvability.html): An inversion is when a tile precedes another tile with a lower number on it. The solution state has zero inversions. The formula says:

1. If the grid width is odd, then the number of inversions in a solvable situation is even.
2. If the grid width is even, and the blank is on an even row counting from the bottom (second-last, fourth-last etc), then the number of inversions in a solvable situation is odd.
3. If the grid width is even, and the blank is on an odd row counting from the bottom (last, third-last, fifth-last etc) then the number of inversions in a solvable situation is even.

This is because, as they explain: Fact 1: For a grid of odd width, the polarity of the number of inversions is invariant. That means: all legal moves preserve the polarity of the number of inversions. Fact 2: For a grid of even width, the following is invariant: (#inversions even) == (blank on odd row from bottom).

Fact 3: If the width is odd, then every solvable state has an even number of inversions. If the width is even, then every solvable state has an even number of inversions if the blank is on an odd numbered row counting from the bottom; an odd number of inversions if the blank is on an even numbered row counting from the bottom;

Fact 4: (the converse of fact 3) If a state is such that: If the width is odd, then the state has an even number of inversions. If the width is even and the blank is on an odd numbered row counting from the bottom, then the state has an even number of inversions If the width is even and the blank is on an even numbered row counting from the bottom, then the state has an odd number of inversions.

** Currently, it seems that these formulas consider the goal state as having the empty tile as being on the bottom right instead of top left corner, therefore I need to change it as:

- instead of counting rows from the bottom, count rows from the top, since that is where 0 should reach.

In [80]:

```
# first defining an inversion counting function:
def countInversions(n, initial_state):
    initial_state = flatten(initial_state) # flatten initial state list
 if needed
    inversions = 0 #initialize the number of inversions to 0, as in the
```

```
solved state
    # now we'll count the number of inversions:
    # for each tile, we iterate thruogh the tiles following it and count
 how many of them are smaller and thus need to be inverted
    for i in range(0,n**2-1): #iterating through tile1
        for j in range(1,n**2):  #iterating through
            if initial_state[i] > initial_state[j]:
                inversions += 1
                #debug: print("[i]: {} > [j]: {}".format(initial_state
[i] , initial_state[j])) #debug
    #debug print "inversions", inversions
    return inversions

def isSolvable(n, board):
    if type(board) != list:
        board = board.tiles #3if it's a PuzzleNode
    # flatten initial state list
    board = flatten(board)

    inversions = 0 #initialize the number of inversions to 0, as in the
 solved state
    # now we'll count the number of inversions:
    # for each tile, we iterate thruogh the tiles following it and count
 how many of them are smaller and thus need to be inverted
    for i in range(0, n**2 -1): #iterating through tile1
        if board[i] != 0: #the blank tile doesn't count as an inversion
            #debug: print "inversions", inversions, "inspecting for i=",
i, "which is ", board[i]
            for j in range(i, n**2):  #iterating through from i to the e
nd
                # if the first one is larger than the second,
                # plus, the blank tile doesn't count as an inversion
                if (board[i] > board[j]) and board[j] != 0:
                    inversions += 1
                    #print("[{}]: {} > [{}]: {}".format(i,board[i] ,j, b
oard[j])) #debug
    print "Total inversions:", inversions
    inversions_even = (inversions % 2 == 0) # is num of inverstions eve
n? (boolean value)

    # rule 1: If the grid width is odd, then the number of inversions in
 a solvable situation is even.
    if n % 2 != 0: # odd grid:
        solvable = inversions_even # is num of inverstions even? in our
 case, because the 0 should be at the top and not the bottom row, it's i
nverse: if odd: solvable. if even, not solvable.

    # rule 2: If the grid width is even, and the blank is on an even row
 counting from the bottom (second-last, fourth-last etc), then the numbe
r of inversions in a solvable situation is odd.
    # rule 3: If the grid width is even, and the blank is on an odd row
 counting from the bottom (last, third-last, fifth-last etc) then the nu
mber of inversions in a solvable situation is even.
    if n % 2 == 0:
        index_0 = board.index(0)
        row0 = index_0 / n
        #print "index_0, row0", index_0, row0 #debug
```

```
        #(n - row0) = how many rows from bottom. if it's even, it's modu
lo 2 shuold be 0,
        # thus row_odd will be 0 (meaning even; if odd, it's modulo 2 sh
ould be 1, thus 1.
        row0_odd = row0 % 2
        # print n, row0, n - row0, row0_odd #debug
        # according to rules 2,3, when n is even: solvable situations ar
e when row0=even & inversions=odd, OR row0=odd & inversions=even.
        # hence, this comes down to: if row0_odd==True and inversions_ev
en==True, or the opposite (both False), then it's solvable. Therefore, I
 can just compare the values of row0_odd and inversions_even!
        solvable =  (row0_odd == inversions_even)

    return solvable
```

In [97]:

```
# Testing IsSolvable?
print "*** BOARDS I KNOW ARE SOLVABLE: ***"
print "N=3"
print test1
print("Is test1, n=3 solvable? {}".format( isSolvable(n=3, board=test1)
) )
print test2
print("Is test2, n=3 solvable? {}".format( isSolvable(n=3, board=test2)
) )
print test3
print("Is test3, n=3 solvable? {}".format( isSolvable(n=3, board=test3)
) )

print "\nN=2"
smallboard = [[3,1],[2,0]]
print("Is board with n=2 solvable? {}".format( isSolvable(n=2, board=sma
llboard)))

print "\nN=4"
print("Is board with n=4 solvable? {}".format( isSolvable(n=4, board=tes
t15) ) )

print "\n\n *** UNSOLVABLE PUZZLES: ***"
unsolvable3A = [1,0,3,2,4,5,6,7,8] #board I verified to be unsolvable
print("Is unsolvalbe board A with n=3 solvable? {}".format(
isSolvable(n=3, board=unsolvable3A) ) )
unsolvable3B = [7,0,2,8,5,3,6,4,1] ##board I verified to be unsolvable
print("Is unsolvalbe board B with n=3 solvable? {}".format(
isSolvable(n=3, board=unsolvable3B) ) )
```

```
*** BOARDS I KNOW ARE SOLVABLE: ***
N=3
[[2, 3, 7], [1, 8, 0], [6, 5, 4]]
Total inversions: 12
Is test1, n=3 solvable? True
[[7, 0, 8], [4, 6, 1], [5, 3, 2]]
Total inversions: 22
Is test2, n=3 solvable? True
[[5, 7, 6], [2, 4, 3], [8, 1, 0]]
Total inversions: 18
Is test3, n=3 solvable? True

N=2
Total inversions: 2
Is board with n=2 solvable? True

N=4
Total inversions: 26
Is board with n=4 solvable? True


 *** UNSOLVABLE PUZZLES: ***
Total inversions: 1
Is unsolvalbe board A with n=3 solvable? False
Total inversions: 19
Is unsolvalbe board B with n=3 solvable? False
```

In [98]:

```
### Integrating into solvePuzzle function
```

In [108]:

```
def solvePuzzle_verified(n, initial_state, heuristic_id, printit=True):

    if isSolvable(n, board) == False:
        err = -2


    return steps, frontierSize, err, inspected_states_counter, runtime


def solvePuzzle_verified(n, initial_state, heuristic_id, printit=True):
    start_time = time()
    frontier_tree = []
    cost = {}
    visited = {}
    # verify input is valid #
    err, initial_state, reason = verify_input(initial_state,n)

    ### VERIFY THAT INITIAL STATE IS SOLVABLE ###
    if isSolvable(n, initial_state) == False:
        err = -2  # I would also break after this, since otherwise it mi
ght encounter an infinite loop
        steps, frontierSize = None, None
        raise ValueError("The board inserted is unsolvable. err = -2")


    ### FROM HERE, SAME AS BEFORE.....


    #DATA STRUCTURE FOR STORING EACH BOARD: a 1D array. I'm working with
 flat lists since it is simpler on complexity and computational power (s
upposedly) and contains all the needed information.  So I'm flatteinning
 the given list.

    # Initialize our initial state as a board node
    starting_state = PuzzleNode(n=n, values=initial_state, cost=0, paren
t=None, heuristic_id = heuristic_id)

    # Printing initial state at start (if we want to print)
    if printit == True:
        print "Starting to solve from: "
        print str(starting_state)
        print "Solving..."
    goal_state = PuzzleNode(n = n, values = range(n**2), cost = 100, par
ent=None, heuristic_id = heuristic_id)
    heapq.heappush(frontier_tree, (starting_state.total_cost, starting_s
tate))

    ### A* (A-STAR) SEARCH ALGORITHM ###
    inspected_states_counter = 0 # I want to keep track of total inspect
ed steps counter
    frontierSize = 0
    while frontier_tree:
        curr_board_and_cost = heapq.heappop(frontier_tree) # my tree con
tains the board and cost attached as a tuple
```

```
        current_state = curr_board_and_cost[1] # the board is the second
 element of the tuple (cost, board)
        inspected_states_counter += 1 #increment counter of inspected st
ates
        #print("tree size: {}, curr state popped: \n{}".format(len(front
ier_tree), str(current_state))) #debug
        # A* checks if we're finished (at goal), and break if we are!
        if heuristic_id(current_state.tiles) == 0:
            #goal_state = current_state # debug
            break # we are done!

        #### INSPECT LEAF NODES ####
        # Defining the position of the empty tile, 0
        index_0 = current_state.tiles.index(0) #finding the index of the
 "0", the empty slot
        row0 = index_0 / n # the "X axis" index of it is the index divid
ed by n (number of columns/rows)
        col0 = index_0 % n # the "Y axis" index of it is the index modul
o n (number of columns/rows, the remainder translates to how many spots
 to the right..)

        # check possible next moves (where can we swap the empty slot wi
th). Starting with the current index of 0, checking which neighboring in
dexes are availalbe for it to swap with.
        moves_list = []
        if(col0 - 1 >= 0): moves_list.append([row0, col0 - 1])  # if we
 can move left ,add that move
        if(col0 + 1 <  n): moves_list.append([row0, col0 + 1])  # if we
 can move right, add that move
        if(row0 - 1 >= 0): moves_list.append([row0 - 1, col0])  # if we
 can go down ,add that move
        if(row0 + 1 <  n): moves_list.append([row0 + 1, col0])  # if we
 can move up ,add that move

        # now check suitability for each possible move
        for move in moves_list:
            new_state = current_state.tiles[:] #copy values of tiles int
o the new state
            # after this move, the new index for 0 will be just the line
ar combination of the indexes: x*n (row number*amount of items per row)
 + y (position within row, like remainder).
            index_0_new = move[0]*n + move[1]
            # SWAP tiles, by simultaneous multiple "= assignment
            new_state[index_0], new_state[index_0_new] = new_state[index
_0_new], new_state[index_0]
            # make a new PuzzleNode class of it. we'll define the cost a
s +1 more than current, since we define the cost of each step as 1
            new_PuzzleNode = PuzzleNode(n = n, values = new_state, cost
= current_state.cost + 1, parent = current_state, heuristic_id = heurist
ic_id)
            new_cost = new_PuzzleNode.total_cost
            #debug: #print "new_PuzzleNode: \n", new_PuzzleNode.__print_
_() #debug

            # checking that the new board is NOT A PREVIOUSLY VISITED BO
ARD, OR that the new cost is SMALLER than an equal existing state's cost
 (to find a better path to a previously-visited state)
```

```
            # by having a hashing method to PuzzleNode class, we can ver
ify efficiently if it exists in the dictionary such as visited or
            if (new_PuzzleNode not in visited) or (new_cost < cost[new_P
uzzleNode]):
                #debug: #print("appending a new board \n {}".format(str
(new_PuzzleNode()))) #debug
                cost[new_PuzzleNode] = new_cost #reassign new cost
                visited[new_PuzzleNode] = 1 # insert an indicator that t
his board has been visited to the hashed location in the visited list.
                new_PuzzleNode.parent = current_state #setting current s
tate as parent
                heapq.heappush(frontier_tree,
(new_PuzzleNode.total_cost, new_PuzzleNode))

            # update frontier size if it's larger than the last maximal
 frontier size, by taking the max of them both.
            frontierSize = max(frontierSize,len(frontier_tree))

    ### RECONSTRUCTING THE SOLUTION
    ### Backtracking: we start from the goal node and backtrack through
 parents to recreate the path
    solution_steps = [] # initializing a list to contain solution steps
    curr_boardstate = current_state #starting with the last state we wer
e in = the goal state
    while curr_boardstate != starting_state: #backtracking back from the
 goal through parents until reaching the starting state
        #print curr_boardstate.__str__()
        solution_steps.insert(0,curr_boardstate) #intersting the parent
 before on the solution steps list; so that our solution is eventually o
rdered from start->goal
        curr_boardstate = curr_boardstate.parent #reassinging the curren
t step to its parent and iterating
    solution_steps.insert(0,starting_state) #now add the actual initial
 state as the first step (since the while loop stops when it reaches it
 and doesn't add it)
    steps = len(solution_steps)

    ### Printing our solution nicely, if asked: ###
    if printit == True:
        print("Took {} steps to reach solution.".format(steps))
        print("Max Frontier Size was {} (branching factor).".format(fron
tierSize))
        print("Finished with Error Code of {}: {}".format(err, reason))
        print("\n_____________________\nHere are the stages in sublists
 format: ")
        for step_index in range(steps):
            #print("\nStep {}:".format(step_index))
            print(str(solution_steps[step_index]))
        print("\n_____________________\nAnd here are the steps in a pret
ty visual square format")
        for step_index in range(steps):
            #print("")
            print("\nStep {}:".format(step_index))
            print(solution_steps[step_index].__print__())

    runtime = time() - start_time
```