CS152 Assignment 2: The 8-puzzle

Before you turn in this assignment, make sure everything runs as expected. First, **restart the kernel** (in the menubar, select Kernel→Restart) and then run the test cells for each of the questions you have answered. Note that a grade of 3 for the A* implementation requires all tests in the "Basic Functionality" section to be passed. The test cells pass if they execute with no errors (i.e. all the assertions are passed).

Make sure you fill in any place that says `YOUR CODE HERE`. Be sure to remove the `raise NotImplementedError()` statements as you implement your code - these are simply there as a reminder if you forget to add code where it's needed.

Question 1

Define your `PuzzleNode` class below. Ensure that you include all attributes that you need to implement an A* search. If you wish, you can even include member functions, such as a function to generate successor states. Alternatively, you can code up this functionality later in the `solvePuzzle` function.

In [5]:

```
class PuzzleNode:
    """
    Class PuzzleNode:
    Provides a structure for performing A* search for the n^2-1 puzzle

    Attributes:
    fval -- Represents the heuristic value for each state
    gval -- Represents the state level
    parent -- Represents the parent node of a given node

    Methods:
    It -- Aids comparisons for the priority queue
    Print -- Prints the board (for testing)

    """
    def __init__(self, board, fval, gval, parent):

        self.fval = fval
        self.gval = gval
        self.parent = parent
        self.board = board
        self.pruned = False


    def __lt__(self, other):
        return self.fval < other.fval

    def print(self):
        for row in self.board:
            print(row)
        return
```

Question 2

Define your heuristic functions using the templates below. Ensure that you extend the `heuristics` list to include all the heuristic functions you implement. Note that state will be given as a list of lists, so ensure your function accepts this format. You may use packages like numpy if you wish within the functions themselves.

In [49]:

```
def h1(state):
    """
    This function returns the number of misplaced tiles, given the board state
    Input:
        -state: the board state as a list of lists
    Output:
        -h: the number of misplaced tiles
    """
    displacement = 0

    #iterate through each position on the state
    for i in range(len(state)):
        for j in range(len(state)):
            #compares the value in the position to the calculated goal state
            if state[i][j] != len(state)*i+j and state[i][j] != 0:
                #increment displacement
                displacement += 1

    return displacement

def h2(state):
    """
    This function returns the Manhattan distance from the solved state, given the board sta
    Input:
        -state: the board state as a list of lists
    Output:
        -h: the Manhattan distance from the solved configuration
    """
    #initialize the distance counter
    distance = 0

    #iterate through the positions in the grid
    for i in range(len(state)):
        for j in range(len(state)):
            if state[i][j] != 0:
                x, y = divmod(state[i][j], len(state))
                distance += abs(x - i) + abs(y - j)

    return distance

# Extra heuristic for the extension.  If implemented, modify the function below
def h3(state):
    """
    This function returns a heuristic that dominates the Manhattan distance, given the boar
    Input:
        -state: the board state as a list of lists
    Output:
        -h: the Heuristic distance of the state from its solved configuration
    """
    return 0

# If you implement more than 3 heuristics, then add any extra heuristic functions onto the
heuristics = [h1, h2, h3]
```

Question 3

Code up your A* search using the SolvePuzzle function within the template below. Please do not modify the function header, otherwise the automated testing will fail. You may define other functions or import packages as needed in this cell or by adding additional cells.

In [33]:

```python
def is_incorrect_state(state):
    """
    This function returns boolean for duplicate values in a state and if the state
    is n*n, which is an indicator of incorrectness.

    Input:
        -state: the board state as a list of lists
    Output:
        -h: boolean
    """
    #initialize set
    all_nodes = set()

    #checks that state is n*n
    if len(state) != len(state[0]):
        return True

    #iterate through the positions in the grid
    for i in range(len(state)):
        for j in range(len(state)):

            #check for duplicate in set
            if state[i][j] not in all_nodes:
                all_nodes.add(state[i][j])
            else:
                return True

    return False
```

In [61]:

```
from queue import PriorityQueue
import itertools

def solvePuzzle(state, heuristic):
    """This function should solve the n**2-1 puzzle for any n > 2
    (although it may take too long for n > 4)).
    Inputs: n
        -state: The initial state of the puzzle as a list of lists
        -heuristic: a handle to a heuristic function.
    Outputs:
        -steps: The number of steps to optimally solve the puzzle (excluding the initial st
        -exp: The number of nodes expanded to reach the solution
        -max_frontier: The maximum size of the frontier over the whole search
        -opt_path: The optimal path that represents  the state of the board at the ith step
        -err: An error code.  If state is not of the appropriate size and dimension, return
        For the extention task, if the state is not solvable, then return -2
    """
    exp = 0
    max_frontier = 0
    err = 0
    path = []

    # checks if the given state is eligible for the puzzle
    if is_incorrect_state(state):
        return 0, 0, 0, [], -1

    #creates a start node with initial attributes from class
    start_node = PuzzleNode(state, heuristic(state), 0, None)

    #initialize priority queue with start node
    frontier = PriorityQueue()
    frontier.put(start_node)

    #initialize the visited dictionary with a string board key and node value
    visited_states = {}
    visited_states[str(start_node.board)] = start_node

    #generates the goal state for the puzzle grid
    n = len(state)
    initial = 0
    goal = [[0 for _ in range(n)] for _ in range(n)]
    for i in range(n):
        for j in range(n):
            goal[i][j] = initial
            initial += 1

    cur_frontier = 1
    #iterate through the frontier
    while not frontier.empty():
        max_frontier = max(max_frontier, cur_frontier)

        #collect the priority node for next operation
        cur_node = frontier.get()
        cur_frontier -= 1

        #check if a given node is marked for removal
        if cur_node.pruned:
            continue
        # Check if a given node is the goal state
        if cur_node.board == goal:
            #if it is the goal state, then generate the sequence of paths reached
            path = [cur_node.board]
            while cur_node.parent != None:
                cur_node = cur_node.parent
                path.append(cur_node.board)
            steps = len(path) - 1
            path.reverse()
            return (steps, exp, max_frontier, path , err)

        exp += 1
        # iterate through all positions in the puzzle grid
        for i, j in itertools.product(range(n), range(n)):
            #define the top, down, left, and right positions for swap
            directions = ((i + 1, j),(i, j + 1),(i - 1, j),(i, j - 1))

            for direction in directions:
                r, c = direction

                #check that the swap positions are within the grid bounds
                #check that we only consider swaps around the the black node
                if r >= 0 and c >= 0 and r < n and c < n and cur_node.board[r][c] == 0:

                    #create a copy of the state into new variable board
                    board = []
                    for row in cur_node.board:
                        board.append([x for x in row])

                    #swap
                    board[i][j], board[r][c] = board[r][c], board[i][j]

                    #calculating the fvalue with the gval (level)
                    child_fval = heuristic(board) + cur_node.gval + 1

                    #update the fval for each child
                    child = PuzzleNode(board, child_fval, cur_node.gval + 1, cur_node)

                    # prune nodes with less efficient paths
                    #this is determined through the gval (step count)
                    if str(child.board) in visited_states:
                        if visited_states[str(child.board)].gval > child.gval:
                            visited_states[str(child.board)].pruned = True
                        else:
                            continue

                    #add child to a frontier
                    frontier.put(child)
                    cur_frontier += 1
                    visited_states[str(board)] = child

    return 0, exp, max_frontier, path, -2

state = [[2,3,7],[1,8,0],[6,5,4]]
print(solvePuzzle(state, h1))
```

```
(17, 764, 481, [[[2, 3, 7], [1, 8, 0], [6, 5, 4]], [[2, 3, 7], [1, 8, 4],
[6, 5, 0]], [[2, 3, 7], [1, 8, 4], [6, 0, 5]], [[2, 3, 7], [1, 0, 4], [6,
8, 5]], [[2, 0, 7], [1, 3, 4], [6, 8, 5]], [[0, 2, 7], [1, 3, 4], [6, 8,
5]], [[1, 2, 7], [0, 3, 4], [6, 8, 5]], [[1, 2, 7], [3, 0, 4], [6, 8, 5]],
[[1, 2, 7], [3, 4, 0], [6, 8, 5]], [[1, 2, 0], [3, 4, 7], [6, 8, 5]], [[1,
0, 2], [3, 4, 7], [6, 8, 5]], [[1, 4, 2], [3, 0, 7], [6, 8, 5]], [[1, 4,
2], [3, 7, 0], [6, 8, 5]], [[1, 4, 2], [3, 7, 5], [6, 8, 0]], [[1, 4, 2],
[3, 7, 5], [6, 0, 8]], [[1, 4, 2], [3, 0, 5], [6, 7, 8]], [[1, 0, 2], [3,
4, 5], [6, 7, 8]], [[0, 1, 2], [3, 4, 5], [6, 7, 8]]], 0)
```

Extension Questions

The extensions can be implemented by modifying the code from Q2-3 above appropriately.

1. **Initial state solvability:** Modify your SolvePuzzle function code in Q3 to return -2 if an initial state is not solvable to the goal state.
2. **Extra heuristic function:** Add another heuristic function (e.g. pattern database) that dominates the misplaced tiles and Manhattan distance heuristics to your Q2 code.
3. **Memoization:** Modify your heuristic function definitions in Q2 by using a Python decorator to speed up heuristic function evaluation

There are test cells provided for extension questions 1 and 2.

Basic Functionality Tests

The cells below contain tests to verify that your code is working properly to be classified as basically functional. Please note that a grade of **3** on #aicoding and #search as applicable for each test requires the test to be successfully passed. **If you want to demonstrate some other aspect of your code, then feel free to add additional cells with test code and document what they do.**

In [47]:

```
## Test for state not correctly defined

incorrect_state = [[0,1,2],[2,3,4],[5,6,7]]
_,_,_,_,err = solvePuzzle(incorrect_state, lambda state: 0)
assert(err == -1)
```

In [67]:

```
## Test for non n*n grids

incorrect_state = [[0,1,2],[2,3,4]]
_,_,_,_,err = solvePuzzle(incorrect_state, lambda state: 0)
assert(err == -1)
```

In [50]:

```
## Heuristic function tests for misplaced tiles and manhattan distance

# Define the working initial states
working_initial_states_8_puzzle = ([[2,3,7],[1,8,0],[6,5,4]], [[7,0,8],[4,6,1],[5,3,2]], [[

# Test the values returned by the heuristic functions
h_mt_vals = [7,8,7]
h_man_vals = [15,17,18]

for i in range(0,3):
    h_mt = heuristics[0](working_initial_states_8_puzzle[i])
    h_man = heuristics[1](working_initial_states_8_puzzle[i])
    assert(h_mt == h_mt_vals[i])
    assert(h_man == h_man_vals[i])
```

In [51]:

```
## A* Tests for 3 x 3 boards
## This test runs A* with both heuristics and ensures that the same optimal number of steps
## with each heuristic.

# Optimal path to the solution for the first 3 x 3 state
opt_path_soln = [[[2, 3, 7], [1, 8, 0], [6, 5, 4]], [[2, 3, 7], [1, 8, 4], [6, 5, 0]],
                 [[2, 3, 7], [1, 8, 4], [6, 0, 5]], [[2, 3, 7], [1, 0, 4], [6, 8, 5]],
                 [[2, 0, 7], [1, 3, 4], [6, 8, 5]], [[0, 2, 7], [1, 3, 4], [6, 8, 5]],
                 [[1, 2, 7], [0, 3, 4], [6, 8, 5]], [[1, 2, 7], [3, 0, 4], [6, 8, 5]],
                 [[1, 2, 7], [3, 4, 0], [6, 8, 5]], [[1, 2, 0], [3, 4, 7], [6, 8, 5]],
                 [[1, 0, 2], [3, 4, 7], [6, 8, 5]], [[1, 4, 2], [3, 0, 7], [6, 8, 5]],
                 [[1, 4, 2], [3, 7, 0], [6, 8, 5]], [[1, 4, 2], [3, 7, 5], [6, 8, 0]],
                 [[1, 4, 2], [3, 7, 5], [6, 0, 8]], [[1, 4, 2], [3, 0, 5], [6, 7, 8]],
                 [[1, 0, 2], [3, 4, 5], [6, 7, 8]], [[0, 1, 2], [3, 4, 5], [6, 7, 8]]]

astar_steps = [17, 25, 28]
for i in range(0,3):
    steps_mt, expansions_mt, _, opt_path_mt, _ = solvePuzzle(working_initial_states_8_puzzl
    steps_man, expansions_man, _, opt_path_man, _ = solvePuzzle(working_initial_states_8_pu
    # Test whether the number of optimal steps is correct and the same
    assert(steps_mt == steps_man == astar_steps[i])
    # Test whether or not the manhattan distance dominates the misplaced tiles heuristic in
    assert(expansions_man < expansions_mt)
    # For the first state, test that the optimal path is the same
    if i == 0:
        assert(opt_path_mt == opt_path_soln)
```

In [55]:

```
## A* Test for 4 x 4 board
## This test runs A* with both heuristics and ensures that the same optimal number of steps
## with each heuristic.

working_initial_state_15_puzzle = [[1,2,6,3],[0,9,5,7],[4,13,10,11],[8,12,14,15]]
steps_mt, expansions_mt, _, _, _ = solvePuzzle(working_initial_state_15_puzzle, heuristics[
steps_man, expansions_man, _, _, _ = solvePuzzle(working_initial_state_15_puzzle, heuristic
# Test whether the number of optimal steps is correct and the same
assert(steps_mt == steps_man == 9)
# Test whether or not the manhattan distance dominates the misplaced tiles heuristic in eve
assert(expansions_mt >= expansions_man)
```

Extension Tests

The cells below can be used to test the extension questions. Memoization if implemented will be tested on the final submission - you can test it yourself by testing the execution time of the heuristic functions with and without it.

In [63]:

```
## Puzzle solvability test

unsolvable_initial_state = [[7,5,6],[2,4,3],[8,1,0]]
_,_,_,_,err = solvePuzzle(unsolvable_initial_state, lambda state: 0)
assert(err == -2)
```

In [ ]:

```
## Extra heuristic function test.
## This tests that for all initial conditions, the new heuristic dominates over the manhatt

dom = 0
for i in range(0,3):
    steps_new, expansions_new, _, _, _ = solvePuzzle(working_initial_states_8_puzzle[i], he
    steps_man, expansions_man, _, _, _ = solvePuzzle(working_initial_states_8_puzzle[i], he
    # Test whether the number of optimal steps is correct and the same
    assert(steps_new == steps_man == astar_steps[i])
    # Test whether or not the manhattan distance is dominated by the new heuristic in every
    # the number of nodes expanded
    dom = expansions_man - expansions_new
    assert(dom > 0)
```

In [ ]:

```
## Memoization test - will be carried out after submission
```