Assignment2

October 30, 2022

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.

[1]:
```
# My code for question 1 and the A* search algorithm is adapted from the A*␣
↪code from class.
# It can be found here: https://sle-collaboration.minervaproject.com/?
↪minervaNotebookId=cl7xiia6601b00j1v3k12f6i0&userId=10834&name=Vicki+Petrova&avatar=https%3A
↪/s3.amazonaws.com/picasso.fixtures/
↪Victoria_Petrova_10834_2020-09-11T21%3A29%3A05.
↪741Z&readOnly=1&isInstructor=0&signature=074dc12f2cf5719cef6d50f8949019f69d19b58e4b64214bd7




#Now, define the class PuzzleNode:
class PuzzleNode:
    """
    Class PuzzleNode: Provides a structure for performing A* search for the␣
↪n^2-1 puzzle

    Attributes
    ----------
        state: list of lists
```

```
            The state of the tiles of the puzzle.
        fval: float
            Estimated cost of the cheapest solution through this node. Formula:␣
↪g(n) + h(n).
            The sum of 1) path cost from start to node this node (gval) and 2)␣
↪estimated cost from
            the cheapest path from this node to goal state.
        gval: float
            Path cost from start to this node represented as the number of␣
↪steps taken to get to this state.
        parent: PuzzleNode, optional
            The parent node of this Puzzle Node.

    Methods
    -------
        No special methods. Dunder methods lt and str are overwritten.

    """

    # Class constructor
    def __init__(self, state, fval, gval, parent=None):
        """
        Parameters
        ----------
        state: list of lists
            The state of the tiles of the puzzle.
        fval: float
            Estimated cost of the cheapest solution through this node. Formula:␣
↪g(n) + h(n).
            The sum of 1) path cost from start to node this node (gval) and 2)␣
↪estimated cost from
            the cheapest path from this node to goal state.
        gval: float
            Path cost from start to this node represented as the number of␣
↪steps taken to get to this state.
        parent: PuzzleNode, optional
            The parent node of this Puzzle Node.
        """

        self.state = state
        self.fval = fval
        self.gval = gval
        self.parent = parent
        self.pruned = False
    # Comparison function based on f cost
    def __lt__(self,other):
        return self.fval < other.fval


    # Convert to string
    def __str__(self):
        # Return every row of the state list on a new line.
        output = ('\n'.join(map(str, self.state)))
        return output
```

[2]:
```
# Test whether the class works correctly.
classic = PuzzleNode([[0,1,2], [3,4,5], [6,7,8]], 0, 0)
print(classic)
```

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

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.

[3]:
```
# No heuristic, models uninformed search.
# I used this heuristic for debugging.
def h0(state):
    """
    This function returns 0 to model an uninformed (uniform-cost) search.

    Parameters
    ----------
        state: the board state as a list of lists
    Returns
    -------
        0
    """

    return 0

# Misplaced tiles heuristic
def h1(state):
    """
    This function returns the number of misplaced tiles, given the board state
    Parameters
    ----------
        state: list of lists
            The board state as a list of lists.
    Returns
    -------
        misplaced_tiles: int
            The number of misplaced tiles

    """


    # Make the current state into a single list.
    flatten_state = flatten_list(state)

    # Create goal state for board.
    dimension = len(state)
    goal_state = list(range(dimension**2))

    # Make a list with 0s and 1s based on whether the elements match and sum it.
    bool_misplaced_tiles =[num_current != num_goal for num_current, num_goal in␣
 ↪zip(flatten_state,goal_state)]

    # Don't count the empty tile.
    misplaced_tiles = sum(bool_misplaced_tiles[1:])


    return misplaced_tiles

def flatten_list(list_of_lists):
    """
    This function joins the sublists of a list of lists into a single list.

    Parameters
    ----------
        list_of_lists: list of lists
            The initial list of lists
    Returns
    -------
        The concatenated list of lists into a single list.

    """
    return [number for row in list_of_lists for number in row]



# Manhattan distance heuristic
```

```
def h2(state):
    """
    This function returns the Manhattan distance from the solved state, given␣
↪the board state.

    The Manhattan distance for a tile is defined as its the distances from its␣
↪goal position.
    The function returns the sum of all such values of the tiles.

    Parameters
    ----------
        state: list of lists
            The board state as a list of lists.
    Returns
    -------
        misplaced_tiles: int
            The Manhattan distance from the solved configuration
    """

    mahnattan_distance = 0
    dimension = len(state)

    # Traverse the current board state.
    for i in range(dimension):
        for j in range(dimension):
            # Add up the Manhattan Distance for each tile.

            current_number = state[i][j]

            # Don't count the empty tile.
            if current_number == 0:
                continue

            current_row = i
            current_column = j

            # Get goal row and column.
            goal_row = current_number // dimension
            goal_column = current_number % dimension

            # Calculate the Manhattan Distance for the tile.
            tile_MD = abs(current_row-goal_row) +␣
↪abs(current_column-goal_column)

            mahnattan_distance += tile_MD

    return mahnattan_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 board state.


    It implements the linear conflicts heuristic. It counts the Manhattan␣
 ↪distance but also takes
    into account any tiles in conflict. This is because tiles can't slide over␣
 ↪each other, meaning
    one of them will have to move out the way and then slide back, resulting in␣
 ↪2 additional moves.
    It dominates the misplaced tiles and Manhattan distance because it accounts␣
 ↪for more information.


    Sources used:
    https://cse.sc.edu/~mgv/csce580sp15/gradPres/HanssonMayerYung1992.pdf
    https://medium.com/swlh/looking-into-k-puzzle-heuristics-6189318eaca2

    Input:
        state: list of lists
            The board state as a list of lists
    Output:
        linear_conflicts*2 + manhattan_distance
            the Heuristic distance of the state from its solved configuration
    """


    dimension = len(state)

    # Store the columns of the state as rows.
    # Used to check for linear conflicts in columns.
    transposed_state = list(map(list, zip(*state)))

    # Initialize variables.
    linear_conflicts = 0
    manhattan_distance = 0

    # Traverse tiles to find conflicts in rows.
    for i in range(dimension):
        for j in range(dimension):
            current_tile = state[i][j]
            current_row = i
            current_column = j

            # Update the manhattan_distance.
            manhattan_distance += tile_manhattan_distance(dimension,␣
↪current_tile, current_row, current_column)

            # Don't count the empty tile.
            if current_tile == 0:
                continue


            # Get goal row and column.
            goal_row_tile = current_tile // dimension
            goal_column_tile = current_tile % dimension


            # If it's not in the correct row and column, no linear conflicts␣
↪exist.
            if goal_row_tile != current_row and goal_column_tile !=␣
↪current_column:
                continue


            # Update linear conflicts for the row and/or column.
            if goal_row_tile == current_row:
                linear_conflicts += tile_linear_conflicts(state, j, i, "row")
            if goal_column_tile == current_column:
                linear_conflicts += tile_linear_conflicts(transposed_state, i,␣
↪j, "column")

    # Mark 2 additional moves per conflict on top of the MD.
    return linear_conflicts*2 + manhattan_distance



def tile_linear_conflicts(state, current_column, current_row, line_type):
    """
    This function finds the linear conflicts of a tile the row or column.


    Input:
        state: list of lists
            The board state as a list of lists
        current_column: int
```

```
            Column position of current tile
        current_row: int
            Row position of current tile.
        line_type: str
            Line type can be either a "row" or "column". Helps with the␣
↪indexing.
    Output:
        linear_conflicts
            The linear conflicts of the tile with the right elements on the␣
↪line.
    """

    # Initialize variables.
    linear_conflicts = 0

    dimension = len(state)

    current_tile = state[current_row][current_column]

    if line_type == "row":
        # Get goal row and column.
        goal_row_tile1 = current_tile // dimension
        goal_column_tile1 = current_tile % dimension
    elif line_type == "column":
        # Get goal row and column.
        goal_row_tile1 = current_tile % dimension
        goal_column_tile1 = current_tile // dimension



    # Compare with elements to the right.
    for k in range(current_column+1, dimension):


        comparison_tile = state[current_row][k]

        # Don't count the empty tile.
        if comparison_tile == 0:
            continue


        if line_type == "row":
            # Get goal row and column.
            goal_row_tile2 = comparison_tile // dimension
            goal_column_tile2 = comparison_tile % dimension
        elif line_type == "column":
            # Get goal row and column.
            goal_row_tile2 = comparison_tile % dimension
            goal_column_tile2 = comparison_tile // dimension


        # If the two tiles aren't supposed to be in the same row, move on.
        if goal_row_tile2 != current_row:
            continue
        # If the first tile is supposed to be on the right side, linear␣
↪conflict exists.
        elif goal_column_tile1 > goal_column_tile2:
            linear_conflicts += 1

    return linear_conflicts

def tile_manhattan_distance(dimension, current_tile, current_row,␣
↪current_column):
    """
    This function finds the Manhattan distance of a tile.


    Input:
        dimension: int
            Size of the board (number of rows/columns)
        current_tile: int
            Current element.
        current_column: int
            Column position of current tile
        current_row: int
            Row position of current tile.
    Output:
        tile_MD
            The Manhattan distance of a tile.
    """

    # Don't count the empty tile.
    if current_tile == 0:
        return 0


    # Get goal row and column.
    goal_row = current_tile // dimension
    goal_column = current_tile % dimension

    # Calculate the Manhattan Distance for the tile.
    tile_MD = abs(current_row-goal_row) + abs(current_column-goal_column)
    return tile_MD

# If you implement more than 3 heuristics, then add any extra heuristic␣
↪functions onto the end of this list.
heuristics = [h1, h2, h3]
```

[4]:
```
# Test h1 to see if it works correctly.

# Goal state
assert (h1([[0,1,2], [3,4,5], [6,7,8]]) == 0)

# Switched positions of two tiles
assert (h1([[0,8,2], [3,4,5], [6,7,1]]) == 2)

# All wrong
assert (h1([[1,2,3], [4,5,6], [7,8,0]]) == 8)

# Goal state for a bigger dimension 4x4 (n=4)
assert (h1([[0,1,2,3], [4,5, 6, 7], [8,9,10,11], [12,13,14,15]]) == 0)

# Mistake for a bigger dimension 4x4 (n=4)
assert (h1([[0,15,2,3], [4,5, 6, 7], [8,9,10,11], [12,13,14,1]]) == 2)

# All wrong
assert (h1([[1,2,3, 4], [5, 6, 7, 8], [9,10,11, 12], [13,14,15,0]]) == 15)
```

[5]:
```
# Test h2

# Goal state
assert (h2([[0,1,2], [3,4,5], [6,7,8]]) == 0)

# Switched adjacent tiles
assert (h2([[0,2,1], [3,4,5], [6,7,8]]) == 2)

# Goal state for a bigger dimension 4x4 (n=4)
assert (h2([[0,1,2,3], [4,5, 6, 7], [8,9,10,11], [12,13,14,15]]) == 0)

# Switched the last and first tiles
assert (h2([[15,1,2,3], [4,5, 6, 7], [8,9,10,11], [12,13,14,0]]) == 6)

# More misplaced tiles
assert (h2([[15,14,13,3], [4,5, 6, 7], [8,9,10,11], [12,2,1,0]]) == 22)
```

[8]:
```
# Test reversed row.
assert (h3([[0,1,2], [5,4,3], [6,7,8]]) == 10)

# Don't count empty tile
```

```
assert (h3([[2,1,0], [3,4,5], [6,7,8]]) == 4)

# Test reversed column
assert (h3([[0,7,2], [3,4,5], [6,1,8]]) == 10)

# Reversed column and row
assert (h3([[0,7,2], [5,4,3], [6,1,8]]) == 20)
```

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.

[9]:
```
# Main solvePuzzle function.
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:
        -state: The initial state of the puzzle as a list of lists
        -heuristic: a handle to a heuristic function.  Will be one of those␣
 ↪defined in Question 2.
    Outputs:
        -steps: The number of steps to optimally solve the puzzle (excluding␣
 ↪the initial state)
        -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 as a list of list of lists.  That is,␣
 ↪opt_path[:,:,i] should give a list of lists
                    that represents the state of the board at the ith step of␣
 ↪the solution.
        -err: An error code.  If state is not of the appropriate size and␣
 ↪dimension, return -1.  For the extention task,
          if the state is not solvable, then return -2
    """

    # Verify input state is of the appropriate size and dimension.
    err = verify_input(state)


    if err == -1:
        return None, None, None, None, err


    # Extension 1.
    # Make sure start state is solvable.
    err = is_puzzle_solvable(state)

    if err == -2:
        return None, None, None, None, err


    # Dimension of the puzzle based on the number of rows.
    dimension = len(state)

    #______________________________A*␣
↪search______________________________________


    max_frontier = 0

    # Start node
    start_node = PuzzleNode(state,heuristic(state),0)

    # Goal state
    goal = get_goal_state(dimension)

    # Dictionary with current cost to reach all visited nodes
    costs_db = {str(start_node.state):start_node} #lists are unhashable, but␣
↪strings are

    ### Extension 3.

    heuristics_db = {str(start_node.state):heuristic(start_node.state)}
    called_heuristic = 0

    ### Extension 3.


    # Frontier, stored as a Priority Queue to maintain ordering
    from queue import PriorityQueue

    # Get node from the top of the queue.
    frontier = PriorityQueue()
    frontier.put(start_node)

    # Next moves to move empty tile: left, right, up, down.
    moves_orth = ((0,-1), (0,1), (-1,0), (1,0))


    # Begin A* Tree Search
    step_counter = 0
    # While there are nodes we can explore.
    while not frontier.empty():

        # Take the next available node from the priority queue
        cur_node = frontier.get()

        if cur_node.pruned:
            continue # Skip if this node has been marked for removal

        # Check if we are at the goal
        if cur_node.state == goal: break


        # Expand the node according to the moves that generate new possible␣
↪states.
        for m in moves_orth:
            next_state = get_next_state(cur_node.state, m)

            # Can only move the empty tile there if it's in the boundary of the␣
↪board.
            if next_state != None:
                step_counter += 1 # Each valid child node generated is another␣
↪step
                gval = cur_node.gval + 1 # Tentative cost value for child

                # If the child node is already in the cost database (i.e.␣
↪explored) then see if
                # we need to update the path.  In a graph search, we wouldn't␣
↪even bother exploring it again.
                if str(next_state) in costs_db:
                    if costs_db[str(next_state)].gval > gval:
                        costs_db[str(next_state)].pruned = True # Mark existing␣
↪value for deletion from frontier
                    else:
                        continue # ignore this child, since a better path has␣
↪already been found previously.

                ### Extension 3

                # If the hval for this state has already been computed, get it␣
↪from the databse.
                if str(next_state) in heuristics_db:
                    hval = heuristics_db[str(next_state)]

                # Otherwise, find the hval by calling the heuristic.
                else:
                    hval = heuristic(next_state) # Heuristic cost from next␣
↪node to goal
                    heuristics_db[str(next_state)] = hval
                    called_heuristic +=1

                ### Extension 3 End

                # Create the next node, add to frontier, and dictionary.
                next_node = PuzzleNode(next_state,gval+hval,gval,cur_node) #␣
↪Create new node for child
                frontier.put(next_node)
                costs_db[str(next_state)] = next_node #Mark the node as explored

                # Get the max frontier size so far.
                max_frontier = max(max_frontier,frontier.qsize())



    # Reconstruct the optimal path using the states.
    opt_path = [cur_node.state]

    # Until the root (start) node is reached, get the parents.
    while cur_node.parent:
        opt_path.append((cur_node.parent).state)
        cur_node = cur_node.parent

    # Reverse the order to start with the root node.
    opt_path = opt_path[::-1]

    # Use these for a nice output and debugging. Commented out for the tests.
#     print(f"A* search completed in {step_counter} steps\n")
#     print(f"A* path length: {len(opt_path)-1} steps\n")
#     print(f"A* path to goal:\n")
#     print(opt_path)

    steps = len(opt_path)-1
    exp = len(costs_db)

    return steps, exp, max_frontier, opt_path, err


# Subfunctions for solvePuzzle().


def verify_input(state):
    """
    This function verifies that the input state is of the appropriate size and␣
↪dimension and that
    it contains correct elements without duplicates.

    Parameters
    ----------
        state: list of lists
            The board state as a list of lists.
    Returns
    -------
        error_value: int
            An error code. If state is not of the appropriate size and␣
↪dimension, return -1.
            Otherwise, return 0.
    """
    # 1) Check for correct size and dimension.

    # Get the number of rows.
    dimension = len(state)

    # Check if all rows have the same dimension.
    # If False, erro = -1. Else, error = 0.
    error_dimension = all(len(row) == dimension for row in state)

    # 2) Check for correct elements without duplicates.
    error_elements = sorted(flatten_list(state)) == list(range(0,␣
↪dimension*dimension))

    # If either condition isn't met, return -1 error. Otherwise, no error.
    if not error_dimension or not error_elements:
        error_value = -1
    else:
        error_value = None

    return error_value


def get_goal_state(dimension):
    """
    This function generates the goal state for a puzzle based on the puzzle's␣
↪dimension.

    Parameters
    ----------
        dimension: int
            The dimension of the puzzle board.
    Returns
    -------
        goal_state: list of lists
            The goal state of the puzzle.
    """
    goal_state = []

    # Traverse each row.
    for i in range(dimension):
        # Add a list for the row.
        goal_state.append([])
        # Traverse column.
        for j in range(dimension):
            # Add each element.
            goal_state[i].append(i*dimension+j)

    return goal_state


def get_next_state(state, m):

    """
    This function generates the next state of the puzzle board based on the␣
 ↪current state and move.

    Parameters
    ----------
        state: list of lists
            The board state as a list of lists.
        move: tuple
            The move for the empty tile: left (0,-1), right (0,1), up (-1,0),␣
 ↪down (1,0).
    Returns
    -------
        next_state: list of lists
            Next state of the board after the empty tile is moved.
    """

    current_state = state
    dimension = len(current_state)

    current_index0 = None

    # Traverse current state to find position of empty tile.
    for row_index, row in enumerate(current_state):
        if 0 in row:
            current_index0 = (row_index, row.index(0))
            break
    # Find the position of the empty tile based on the move.
    next_index0 = tuple(sum(x) for x in zip(current_index0,m))

    # If position is out of bounds, move is not possible.
    if next_index0[0]>=dimension or next_index0[1]>=dimension or␣
↪next_index0[0]<0 or next_index0[1]<0:
        return None # No next state with this move.
    else:

        import copy
        # Swap the empty tile and the adjacent tile.
        new_state = copy.deepcopy(current_state)

        new_state[current_index0[0]][current_index0[1]],␣
↪new_state[next_index0[0]][next_index0[1]] =␣
↪new_state[next_index0[0]][next_index0[1]],␣
↪new_state[current_index0[0]][current_index0[1]]

        return new_state

# EXTENSION 1

def is_puzzle_solvable(state):

    """
    This function determines whether the initial state is solvable: whether␣
↪it's possible to reach the
    goal state from the start state.

    If the sum of all inverse elements is odd then we can't reach the desired␣
↪goal state. This subset of
    states can reach 'another goal state' where the empty tile is in the middle␣
↪and all elements are in order
    rather than having the empty tile in the top left corner. The other subset␣
↪of states that have an even
    number of inverses, can reach our desired goal state.

    To find this method and proof I referenced several websites and then␣
↪implemented it in Python.
    To find the proof:
    https://www.8puzzle.com/8_puzzle_algorithm.html
    https://cpentalk.com/502/
↪puzzle-states-divided-disjoint-reachable-while-reachable
```

```
    First, I implemented this method but it wasn't always correct (for even␣
↪sized boards):
    https://www.cs.princeton.edu/courses/archive/spring21/cos226/assignments/
↪8puzzle/specification.php#:~:
↪text=First%2C%20we'll%20consider%20the,number%20(zero)%20of%20inversions.

    So then I implemented this method:
    https://www.geeksforgeeks.org/check-instance-15-puzzle-solvable/

    Parameters
    ----------
        state: list of lists
            The board state as a list of lists.
    Returns
    -------
        error_value: int
            An error code. If state is not solvable, error value is -2.␣
↪Otherwise, None.
    """
    # Initialize error value.
    error_value = None

    dimension = len(state)

     # Make the current state into a single list.
    flatten_state = flatten_list(state)

    row_of_0 = 0

    # If the dimension of the board is an even number we need to consider the␣
↪row index of the empty tile.
    if dimension % 2 == 0:
        # Traverse current state to find position of empty tile.
        for row_index, row in enumerate(state):
            if 0 in row:
                index_of_0 = (row_index, row.index(0))
                break
        # Get the row number of the empty tile.
        row_of_0 = index_of_0[0] + 1

    inverses = 0
    # Traverse state to get all inverses.
    for i in range(len(flatten_state)):
        tile_inverses = 0
        for j in range(i,len(flatten_state)):

            # 1 is always the smallest.
            if flatten_state[i] == 1:
                continue
            # Otherwise compare with elements on the right.
            elif flatten_state[i] > flatten_state[j] and flatten_state[j] != 0␣
↪and flatten_state[i] != 0:
                tile_inverses+=1

        inverses += tile_inverses

    # If dimension is even.
    if dimension % 2 == 0:
        if row_of_0 % 2 == 0 and inverses % 2 != 1:
            error_value = -2
        elif row_of_0 % 2 == 1 and inverses % 2 != 0:
             error_value = -2
    # If dimension is odd and the number of inverses is odd, unsolvable.
    elif dimension % 2 == 1 and inverses % 2 == 1:
        error_value = -2

    return error_value
```

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.

2. Extra heuristic function:

The Manhattan distance heuristic performs better than the Misplaced Tiles heuristic. So we need to find a heuristic that dominates the Manhattan distance.

Why Manhattan distance dominates the Misplaced Tiles heuristic

The Misplaced Tiles heuristic counts the number of tiles that are not in the correct spot. It does not take into account how far away they are from the correct position so it does not make a difference between having to make a single move to place a tile correctly and haveing to make three moves to get to the correct position. Manhattan distance however takes that into account which is why we see it dominating the misplaced tiles (in the provided tests)if we take into consideration the number of nodes expanded to reach the solution.

Linear Conflicts Heuristic

One possibility for an admissible heuristic that dominates Manhattan Distance is Linear Conflicts. It is a heuristic because it solved a relaxed version of the original problem (the same as Manhattan Distance). In this relaxed version of the problem it is possible to move tiles to adjacets positions horizontally or vertically and to stack them as well.

What the Linear Heuristic does is, it still used the Manhattan distance but also adds additional information to make the estimation. The Manhattan distance does not take into account if there are tiles that are in linear conflict. Two twiles are in linear conflict if they are on the correct row but not in the correct column. That would mean that they will have to 'jump over' each other to go to their spots. In the puzzle game that is not possible, meaning that one of them will have to move out to make space and then move back in, which adds 2 more additional moves which we should take into account. It still never overestimates the number of steps to the solution as it only counts moves that would be absolutely neccessary to solve the puzzle and nothing more.

Source about Linear Conflicts heuristic: https://cse.sc.edu/~mgv/csce580sp15/gradPres/HanssonMayerYung1992. and https://medium.com/swlh/looking-into-k-puzzle-heuristics-6189318eaca2

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.

[10]:
```
## 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)
```

[11]:
```
## 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]], [[5,7,6],[2,4,3],[8,1,0]])

# 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])
```

[12]:
```
## A* Tests for 3 x 3 boards
## This test runs A* with both heuristics and ensures that the same optimal␣
↪number of steps are found
```

```
## 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_puzzle[i], heuristics[0])
    steps_man, expansions_man, _, opt_path_man, _ =␣
↪solvePuzzle(working_initial_states_8_puzzle[i], heuristics[1])
    # 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 every case
    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)
```

[13]:
```
## A* Test for 4 x 4 board
## This test runs A* with both heuristics and ensures that the same optimal␣
↪number of steps are found
## 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[0])
```

```
steps_man, expansions_man, _, _, _ =␣
↪solvePuzzle(working_initial_state_15_puzzle, heuristics[1])
# 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 every case
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.

[14]:
```
## 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)
```

[15]:
```
# Additional Tests for Puzzle solvability

# Unsolvable states for odd dimension board.
unsolvable_initial_states = [ [[8,1,2], [0,4,3],[7,6,5]],
                             [[1,0,3], [2,4,5], [6,7,8]],
                             [[7,0,2], [8,5,3],[6,4,1]] ]

# Solvable states for odd dimension board.
solvable_initial_states = [ [[1,8,2], [0,4,3],[7,6,5]],
                           [[1,0,3], [2,5,4], [6,7,8]],
                           [[1,2,3],[4,5,6],[7,8,0]]]

for i in range(0,3):
    err = is_puzzle_solvable(unsolvable_initial_states[i])
    assert(err == -2)
    err = is_puzzle_solvable(solvable_initial_states[i])
    assert(err == None)

# Solvable of even dimension.
assert (is_puzzle_solvable([[2,1,3,4],[5,6,0,8],[9,10,7,11],[13,14,15,12]]) ==␣
↪None)

# Unsolvable of even dimension.
assert (is_puzzle_solvable([[1,2,3,4],[5,6,0,8],[9,10,7,11],[13,14,15,12]]) ==␣
↪-2)
```

[16]:
```
## Extra heuristic function test.
## This tests that for all initial conditions, the new heuristic dominates over␣
↪the manhattan distance.

dom = 0
for i in range(0,3):
    steps_new, expansions_new, _, _, _ =␣
↪solvePuzzle(working_initial_states_8_puzzle[i], heuristics[2])
    steps_man, expansions_man, _, _, _ =␣
↪solvePuzzle(working_initial_states_8_puzzle[i], heuristics[1])
    # 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 case, by checking
    # the number of nodes expanded
    dom = expansions_man - expansions_new
    assert(dom > 0)
```

[17]:
```
## Memoization test - will be carried out after submission
```

1 Extension 3 Attempt with Decorator

I have encountered decorators only once before in CS162. However, I have not used them so far but I wanted to show my attempt, which I believe is mostly correct. I compared the runtime using the time library of the two approaches. The memoization approach was faster for most (not all) initial states but the difference was minimal. There is perhaps a more significant difference in puzzles of higher dimension. Or my implementation is not entirely correct but on the right track.

In my code above I achieved the same result by having a dictionary as heuristic database in the main function. Here I do the same by decorating the heuristic function so `solvePuzzle()` is not changed.

I referenced this website to learn more about them: https://python-course.eu/advanced-python/memoization-decorators.php

[18]:
```
### CHANGED added decorator function.

# Decorator.
def memoize_heuristic(some_function):
    """
    This function is used as a decorator for the heuristic functions. It stores␣
↪the hvalue in a database and
    access it if it has already been computed for the current state. Otherwise␣
↪it calls the heuristic function
    to calculate and store it.

    Parameters
    ----------
        some_function: function
            Some heuristic function (e.g. h1, h2, h3).
    Returns
    -------
        int
            Returns the helper function that either fetches the h value if␣
 ↪available in the databse
            or calls the function to calculate the value, stores it in a␣
 ↪dictionary, and then returns it.
    """
    # Store all values in a dictionary.
    heuristic_db = {}

    def helper(state):
        # If the state hasn't been encountered before, store its h value.
        if str(state) not in heuristic_db:
            heuristic_db[str(state)] = some_function(state)
        return heuristic_db[str(state)]
    return helper

# Decorate the heuristic
@memoize_heuristic
# Manhattan distance heuristic
def h2(state):
    """
    This function returns the Manhattan distance from the solved state, given␣
 ↪the board state.

    The Manhattan distance for a tile is defined as its the distances from its␣
 ↪goal position.
    The function returns the sum of all such values of the tiles.

    Parameters
    ----------
        state
            The board state as a list of lists.
    Returns
    -------
        misplaced_tiles
            The Manhattan distance from the solved configuration
    """

    mahnattan_distance = 0
    dimension = len(state)

    # Traverse the current board state.
    for i in range(dimension):
        for j in range(dimension):
            # Add up the Manhattan Distance for each tile.

            current_number = state[i][j]

            # Don't count the empty tile.
            if current_number == 0:
                continue

            current_row = i
            current_column = j

            # Get goal row and column.
            goal_row = current_number // dimension
            goal_column = current_number % dimension

            # Calculate the Manhattan Distance for the tile.
            tile_MD = abs(current_row-goal_row) +␣
 ↪abs(current_column-goal_column)

            mahnattan_distance += tile_MD

    return mahnattan_distance

heuristics = [h1, h2]
```

[19]:
```
# This is the same function as defined before but without any code related to␣
 ↪Extension 3.
# Main solvePuzzle function.
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:
        -state: The initial state of the puzzle as a list of lists
        -heuristic: a handle to a heuristic function.  Will be one of those␣
 ↪defined in Question 2.
    Outputs:
        -steps: The number of steps to optimally solve the puzzle (excluding␣
 ↪the initial state)
        -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 as a list of list of lists.  That is,␣
 ↪opt_path[:,:,i] should give a list of lists
                    that represents the state of the board at the ith step of␣
 ↪the solution.
```

```
        -err: An error code.  If state is not of the appropriate size and␣
↪dimension, return -1.  For the extention task,
          if the state is not solvable, then return -2
    """

    # Verify input state is of the appropriate size and dimension.
    err = verify_input(state)


    if err == -1:
        return None, None, None, None, err

    # Make sure start state is solvable. Extension 1.

    err = is_puzzle_solvable(state)

    if err == -2:
        return None, None, None, None, err




    #______________________________A*␣
↪search______________________________________

    called_heuristic = 0

    # Dimension of the puzzle based on the number of rows.
    dimension = len(state)

    max_frontier = 0

    # Start node
    start_node = PuzzleNode(state,heuristic(state),0)

    # Goal state
    goal = get_goal_state(dimension)

    # Dictionary with current cost to reach all visited nodes
    costs_db = {str(start_node.state):start_node} #CHANGED bc lists are␣
↪unhashable


    # Frontier, stored as a Priority Queue to maintain ordering
    from queue import PriorityQueue

    frontier = PriorityQueue()
    frontier.put(start_node)

    # Next moves to move empty tile: left, right, up, down.
    moves_orth = ((0,-1), (0,1), (-1,0), (1,0))


    # Begin A* Tree Search
    step_counter = 0

    while not frontier.empty():
        # Take the next available node from the priority queue
        cur_node = frontier.get()

        if cur_node.pruned:
            continue # Skip if this node has been marked for removal

        # Check if we are at the goal
        if cur_node.state == goal: break


        # Expand the node in the orthogonal and diagonal directions
        for m in moves_orth:
            next_state = get_next_state(cur_node.state, m)

            # Can only move the empty tile there if it's in the boundary of the␣
↪board.
            if next_state != None:
                step_counter += 1 # Each valid child node generated is another␣
↪step
                gval = cur_node.gval + 1 # Tentative cost value for child

                # If the child node is already in the cost database (i.e.␣
↪explored) then see if we need to update the path.  In a graph search, we␣
↪wouldn't even bother exploring it again.
                if str(next_state) in costs_db:
                    if costs_db[str(next_state)].gval > gval:
                        costs_db[str(next_state)].pruned = True # Mark existing␣
↪value for deletion from frontier
                    else:
                        continue # ignore this child, since a better path has␣
↪already been found previously.

                hval = heuristic(next_state) # Heuristic cost from next node to␣
↪goal
                next_node = PuzzleNode(next_state,gval+hval,gval,cur_node) #␣
↪Create new node for child
```

```
                frontier.put(next_node)
                costs_db[str(next_state)] = next_node #Mark the node as explored

                max_frontier = max(max_frontier,frontier.qsize())



    # Reconstruct the optimal path using the states.
    opt_path = [cur_node.state]

    # Until the root (start) node is reached, get the parents.
    while cur_node.parent:
        opt_path.append((cur_node.parent).state)
        cur_node = cur_node.parent

    # Reverse the order to start with the root node.
    opt_path = opt_path[::-1]

    # Use these for a nice output and debugging. Commented out for the tests.
#     print(f"A* search completed in {step_counter} steps\n")
#     print(f"A* path length: {len(opt_path)-1} steps\n")
#     print(f"A* path to goal:\n")
#     print(opt_path)

    steps = len(opt_path)-1
    exp = len(costs_db)


    return steps, exp, max_frontier, opt_path, err
```