Tree and Node classes

Here, I define my tree and node classes, I decided to define scoring and cost logic all within the Node class and attached to node objects. This seemed like the easiest way to manage the data to me. I calculate and add all the children of any single node all using functions defined within the Node class.

In [157…

```
class Tree:
    def __init__(self, root):
        self.root = root
class Node:
    children = []
    def __init__(self, parent, board, scorer, cost, goal):
        self.parent = parent
        # score produced by your heuristic of choice
        self.heuristic_score = scorer(board, goal)
        # the game state or board associated with this Node
        self.board = board
        self.scorer = scorer
        # the cost to arrive at this node defined as the distance from the root
        self.cost = cost
        # the goal state
        self.goal = goal
    # this function adds children to a given node. This is not done automaticall
    # called directly in the search method.
    def addChildren(self, childs):
        self.children.extend(childs)
    # this method uses the expandMoves method defined in the Board class to gene
    # them as chilren to the current node, while each successor board being a ne
    def expandChildren(self):
        children = []
        kids = self.board.expandMoves()
        for e in kids:
            if e == -1:
                continue
            children.append(Node(self,e,self.scorer,self.cost+1,self.goal))
        return children
```

Heurisitics and random functions

Here, I define all my heuristics and any other random functions I found useful to have.

In [261…

```
def d2d(i):
    x = i//3
    y = i%3
    return (x,y)
# my implementation of the manhattan distance heuristic
def manhattan(board: Board, list_goal):
    score = 0
```

```
    for i in range(9):
        xd = abs(d2d(i)[0] - d2d(list_goal.index(board.pieces[i]))[0])
        yd = abs(d2d(i)[1] - d2d(list_goal.index(board.pieces[i]))[1])
        score += (xd+yd)
    return score
# my implementation of hamming distance heuristic
def hamming(board: Board, list_goal):
    score = 0
    for i in range(9):
        if list_goal[i] != board.pieces[i]:
            score += 1
    return score
# my implementation of permutation inversions heurisitic
def permutation_inversions(board: Board, list_goal):
    sscore = 0
    for i in range(9):
        if board.pieces[i] == 'B':
            continue
        score = 0
        for j in range(i+1,9):
            # I compare the indexes of these elements in the goal state
            if list_goal.index(board.pieces[j]) < list_goal.index(board.pieces[i
                score += 1
        sscore += score
    return sscore
# This is my non-admissible heuristic, which scores a state that does not have t
# square in the right spot as 5, an incorrect state with the empty square in the
# and the correct or goal state as 0.
def nonAdmissible(board, goal):
    if board.pieces == goal:
        return 0
    elif board.pieces.index('B') == goal.index('B'):
        return 2
    else:
        return 5

# a method to display a board if it is not in the form of a board object, and ju
def displayBoard(board):
        for i in range(9):
            print(board[i], end = '')
            if (i+1)%3 == 0:
                print('')
```

Board Class

Here is the Board class, which is my extension on the list representation of the state of the game. I include all funtions for making moves, moveUp, moveLeft etc.., and also a function that generates all possible moves that can be made given a board, and what boards these moves produce. I use this function in my Node class within my expandChildren() function to expand children at a given node. I also keep track of the position of the empty square separately to allow for quick elimination of impossible moves. Within this class you will find the logic regarding the generation of successor states.

In [262…

```
class Board:
```

```
    def __init__(self, pieces, index_empty):
        self.pieces = pieces
        self.index_empty = index_empty
    def moveLeft(self):
        new = Board(list(self.pieces), self.index_empty)
        if self.index_empty in [0, 3, 6]:
            return -1
        new.pieces[self.index_empty] = new.pieces[self.index_empty-1]
        new.pieces[self.index_empty-1] = 'B'
        new.index_empty = self.index_empty - 1
        return new
    def moveRight(self):
        new = Board(list(self.pieces), self.index_empty)
        if self.index_empty in [2, 5, 8]:
            return -1
        new.pieces[self.index_empty] = new.pieces[self.index_empty+1]
        new.pieces[self.index_empty+1] = 'B'
        new.index_empty = self.index_empty + 1
        return new
    def moveDown(self):
        new = Board(list(self.pieces), self.index_empty)
        if self.index_empty in [6, 7, 8]:
            return -1
        new.pieces[self.index_empty] = new.pieces[self.index_empty+3]
        new.pieces[self.index_empty+3] = 'B'
        new.index_empty = self.index_empty + 3
        return new
    def moveUp(self):
        new = Board(list(self.pieces), self.index_empty)
        if self.index_empty in [0, 1, 2]:
            return -1
        new.pieces[self.index_empty] = new.pieces[self.index_empty-3]
        new.pieces[self.index_empty-3] = 'B'
        new.index_empty = self.index_empty - 3
        return new
    def display(self):
        for i in range(9):
            print(self.pieces[i], end = '')
            if (i+1)%3 == 0:
                print('')

    def expandMoves(self):
            kids = []
            kids.append(self.moveRight())
            kids.append(self.moveLeft())
            kids.append(self.moveUp())
            kids.append(self.moveDown())
            return kids
```

Search Function

I created a search function that accepts as parameters all relevant conditions of the search. The initial state, the goal state, the heuristic we will be using to estimate the distance from the solution, and the search style. The heuristic paramater accepts a function that takes as

parameters a Board object and goal state in the form of a list, and produces as output a score that is a number. The search style parameter takes a string. This string can be one of four options. 'BestFS', 'BFS', 'DFS', and 'A'. *These four options represent Best First Search, Breadth First Search, Depth First Search, and A* Search, respectively. How this is implemented and achived is explained below using inline comments. I also added parameters for what you want to print while it is searching. There is the boolean parameter print_current, which when set to True will make it so the search function prints the current node it is visiting at every iteration, there is print_lists, which when set to true makes it so that the search function prints the open and closed lists during every iteration of the search, and finally there is print_sol, which allows you to decide whether or not you would like to see the solution path printed when the search is over. These are all True by default. If the function finds a solution it will return both how long the search path was and how long the solution path is.

In [263…

```
# Here is the Search function that prints the open and closed lists every iterat
def search(input_state, goal_state, heuristic, search_style, print_current=True,
    # Here I initialize the root node
    root = Node(None, Board(input_state,input_state.index('B')), heuristic,0, go
    current = root
    open_list = []
    closed_list = []
    search_path_length = 0
    # a check to ensure we do not pointlessly pursue impossible goals
    while search_path_length < 100000:
        search_path_length += 1
        # this checks if we have found the solution and if so, proceeds to print
        # root node to the solution
        if current.heuristic_score == 0:
            if print_sol:
                print('---------------')
                print('you have found a solution! Here is the solution path star
                print('---------------')
            solution_list = []
            solution_list.append(current)
            # using a while loop, I go all the way up from the solution node to
            # the visitied node into the front of solution_list during every ite
            while current != root:
                solution_list.insert(0,current.parent)
                current = current.parent
            # Now I simply iterate through solution_list, displaying each board
            if print_sol == True:
                for e in solution_list:
                    e.board.display()
                    if e == solution_list[len(solution_list) - 1]:
                        break
                    print('------next move-------')
            return (search_path_length, len(solution_list))
            break
        kids = current.expandChildren()
        # Here I check to see if any child has already been visited, if so, it i
        # current node, but will not be added to the open list.
        for e in kids:
            if e.board.pieces in closed_list:
                continue
            # if it is DFS, I add children to the front of the open list
            if search_style =='DFS':
```

```
                open_list.insert(0,e)
            # if it is BFS, I add children to the back of the open list
            else:
                open_list.append(e)
        current.addChildren(kids)
        # if it is BestFS, I sort the open list using the values of the chosen h
        if(search_style == 'BestFS'):
            open_list = sorted(open_list, key = lambda a: a.heuristic_score)
        # if it is A*, I sort the open list using the values of the chosen heuri
        elif(search_style =='A*'):
            open_list = sorted(open_list, key = lambda a: a.heuristic_score + a.
        # if it is not A* or BestFS, I do not sort the open_list
        closed_list.append(current.board.pieces)
        if print_current:
            current.board.display()
            print('__')
            print('Heuristic Score : '+ str(heuristic(current.board, goal_state)
            print('__')
        if print_lists:
            print('------OPEN LIST------')
            for e in open_list:
                e.board.display()
                print('__')
                print('Heuristic Score : '+ str(heuristic(e.board, goal_state)))
                print('__')
            print('-----ClOSED LIST------')
            for e in closed_list:
                displayBoard(e)
                print('__')
            print('_________________________')
        # Here I set the new 'current' node to the node that is currently at the
        # and then remove the element from the open list
        current = open_list[0]
        open_list.pop(0)


input_state = [2,8,3,1,6,4,7,'B',5]
goal_state = [1,2,3,8,'B',4,7,6,5]
```

Best First Search Comparisons

Here, I compare all heuristics using best first search.

In [265…

```
input_state = [2,8,3,1,6,4,7,'B',5]
goal_state = [1,2,3,8,'B',4,7,6,5]
print("Manhattan Distance")
x, y = search(input_state, goal_state, manhattan, 'BestFS', False, False, False)
print('Solution Path Length: '+ str(x), 'Search Path Length: '+str(y))
print('_____________________________________________________')
print("Hamming Distance")
x1, y1 = search(input_state, goal_state, hamming, 'BestFS', False, False, False)
print('Solution Path Length: '+ str(x1), 'Search Path Length: '+str(y1))
print('_____________________________________________________')
print("Permutation Inversions")
x2, y2 = search(input_state, goal_state, permutation_inversions, 'BestFS', False
print('Solution Path Length: '+ str(x2), 'Search Path Length: '+str(y2))
```

```
print('_____________________________________________________________')
print("Non-Admissible Heuristic")
x3, y3 = search(input_state, goal_state, nonAdmissible, 'BestFS', False, False,
print('Solution Path Length: '+ str(x3), 'Search Path Length: '+str(y3))
```

```
Manhattan Distance
Solution Path Length: 6 Search Path Length: 6

_____________________________________________________________
Hamming Distance
Solution Path Length: 7 Search Path Length: 6

_____________________________________________________________
Permutation Inversions
Solution Path Length: 6 Search Path Length: 5

_____________________________________________________________
Non-Admissible Heuristic
Solution Path Length: 37 Search Path Length: 6
```

A* Comparisons

Here I compare all 4 heuristics using A* search

In [266…

```
input_state = [2,8,3,1,6,4,7,'B',5]
goal_state = [1,2,3,8,'B',4,7,6,5]
print("Manhattan Distance")
x, y = search(input_state, goal_state, manhattan, 'A*', False, False, False)
print('Solution Path Length: '+ str(x), 'Search Path Length: '+str(y))
print('_____________________________________________________________')
print("Hamming Distance")
x1, y1 = search(input_state, goal_state, hamming, 'A*', False, False, False)
print('Solution Path Length: '+ str(x1), 'Search Path Length: '+str(y1))
print('_____________________________________________________________')
print("Permutation Inversions")
x2, y2 = search(input_state, goal_state, permutation_inversions, 'A*', False, Fa
print('Solution Path Length: '+ str(x2), 'Search Path Length: '+str(y2))
print('_____________________________________________________________')
print("Non-Admissible Heuristic")
x3, y3 = search(input_state, goal_state, nonAdmissible, 'A*', False, False, Fals
print('Solution Path Length: '+ str(x3), 'Search Path Length: '+str(y3))
```

```
Manhattan Distance
Solution Path Length: 6 Search Path Length: 6

_____________________________________________________________
Hamming Distance
Solution Path Length: 7 Search Path Length: 6

_____________________________________________________________
Permutation Inversions
Solution Path Length: 6 Search Path Length: 5

_____________________________________________________________
Non-Admissible Heuristic
Solution Path Length: 37 Search Path Length: 6
```

Harder Initial State Performance

Being that this current initial state makes the goal quite easy to reach, I feel as though we may not have tested the search well enough to confirm or deny any expectations of behavior.

Because of this, all the tests ran above were ran again but with the more difficult initial state

Best First Search

In [267…

```
input_state = [5,1,4,7,'B',6,3,8,2]
goal_state = [1,2,3,8,'B',4,7,6,5]
print("Manhattan Distance")
x, y = search(input_state, goal_state, manhattan, 'BestFS', False, False, False)
print('Solution Path Length: '+ str(y), 'Search Path Length: '+str(x))
print('_____________________________________________________________')
print("Hamming Distance")
x1, y1 = search(input_state, goal_state, hamming, 'BestFS', False, False, False)
print('Solution Path Length: '+ str(y1), 'Search Path Length: '+str(x1))
print('_____________________________________________________________')
print("Permutation Inversions")
x2, y2 = search(input_state, goal_state, permutation_inversions, 'BestFS', False
print('Solution Path Length: '+ str(y2), 'Search Path Length: '+str(x2))
print('_____________________________________________________________')
print("Non-Admissible Heuristic")
x3, y3 = search(input_state, goal_state, nonAdmissible, 'BestFS', False, False,
print('Solution Path Length: '+ str(y3), 'Search Path Length: '+str(x3))
```

```
Manhattan Distance
Solution Path Length: 33 Search Path Length: 118

_____________________________________________________________
Hamming Distance
Solution Path Length: 53 Search Path Length: 1128

_____________________________________________________________
Permutation Inversions
Solution Path Length: 88 Search Path Length: 1178

_____________________________________________________________
Non-Admissible Heuristic
Solution Path Length: 21 Search Path Length: 33072
```

A*

In [269…

```
print("Manhattan Distance")
ya, xa = search(input_state, goal_state, manhattan, 'A*', False, False, False)
print('Solution Path Length: '+ str(xa), 'Search Path Length: '+str(ya))
print('_____________________________________________________________')
print("Hamming Distance")
y1a, x1a = search(input_state, goal_state, hamming, 'A*', False, False, False)
print('Solution Path Length: '+ str(x1a), 'Search Path Length: '+str(y1a))
print('_____________________________________________________________')
print("Permutation Inversions")
y2a, x2a = search(input_state, goal_state, permutation_inversions, 'A*', False,
print('Solution Path Length: '+ str(x2a), 'Search Path Length: '+str(y2a))
print('_____________________________________________________________')
print("Non-Admissible Heuristic")
y3a, x3a = search(input_state, goal_state, nonAdmissible, 'A*', False, False, Fa
print('Solution Path Length: '+ str(x3a), 'Search Path Length: '+str(y3a))
```

```
Manhattan Distance
Solution Path Length: 21 Search Path Length: 154

_____________________________________________________________
Hamming Distance
```

```
Solution Path Length: 21 Search Path Length: 4429

_____________________________________________________________
Permutation Inversions
Solution Path Length: 21 Search Path Length: 520

_____________________________________________________________
Non-Admissible Heuristic
Solution Path Length: 21 Search Path Length: 45425
```

Tabular presentation of data and analysis

Since the harder seemed to highlight the differences between the different search algorithms and heuristics better, I used it to collect the data for this table.

Heuristic	Best First Search	A*
Manhattan (Solution Path Length, Search Path Length)	33, 118	21, 154
Hamming (Solution Path Length, Search Path Length)	53, 1128	21, 4429
Permutation Inversions (Solution Path Length, Search Path Length)	88, 1178	21, 520
Non-Admissible (Solution Path Length, Search Path Length)	21, 33072	21,45425

Analysis betweem heuristics

Everything performed close to what was expected, except for the permutation inversions heuristic. Manhattan outperformed Hamming distance when using both A* and BestFS, both in terms of the effort required to find the solution and the quality of the solution when found. The quality of the solution was marginally better, but how quickly we were able to find the solution, which is arguably the metric that a good heuristic most contributes to was signficantly improved when using manhattan. My non-admissble heuristic performed terribly, as expected, but did actually find the optimal solution even when using BestFS. This was not expected, but I think it is safe to say that taking 30x the time to find a solution that is 30 percent better is not optimal. The dissapointment here was the Permutation Inversions heuristic, which performed similarly to hamming in the BestFS, and much better than hamming when using A*. However, even though it was said to be superior to manhattan during the course lectures, performed signigicantly worse than manhattan in every configuration, in regards to both effeciency and effectivness.

Analysis between searches

The difference between the searches is even more clear, as the A* was able to find the lowest cost solution path everytime, with every heuristic. Weirdly enough, that was the case even with the non-admissible heuristic. A* took consistently longer searching with every single heuristic, except for permutation inversions, where it took about half the time. Overall, especially in the case of manhattan, A* was far superior to BestFS in that for a slightly slower search time, the lowest cost solution path was guaranteed, which mean more than a 50 percent reduction for Hamming and a more than 75 percent reduction for Permutation Inversions.

In [ ]: