8puzzle

Project 1: the 8 puzzle game.

Description

This program uses A-star search on the classic 8-puzzle game to find a solution. We programmed this project using Python due to it's syntactical simplicity.

You pass in a document with the initial and goal states and it returns the shallowest depth, total number of nodes generated, the sequence of moves, and the f(n) values for each node in the solution tree.

A big topic of discussion was how to dequeue when we get the same f(n) values. We decided to dequeue the most recent node first because the best case is that it was at a deeper depth, giving it a lower h(n) value. This would indicate that it's closer to the goal node. The worst case is that it's just a sister node, which is at the same depth, so, it's not a big sacrifice.

Resources Used

We had to look at Python documentation a whole lot, and occasionally email the professor, but no other outside resources were used. We also used the slides on A-star search and the textbook.

We had a lot of questions about implementing Nilsson Sequence Score, so we spoke to the professor. He told us to skip the blank tile when going clockwise and count the centre tile of the goal node. We account for this by stating that the successor of the centre tile is itself.

Running the Project

It is recommended that you use a computer running Linux or macOS to run this code. Also ensure that the computer has a clean copy of `Python >3.8` installed. This project has been tested with all releases of Python till the latest one, `Python 3.10`.

This code has been tested on Ubuntu, Arch, and macOS.

Keep in mind that you need to have Python installed.

You need to invoke this file from the terminal using two command line parameters:
1. Path to input file
2. Heuristic: 1 for Manhattan distance or 2 for Nilsson's Sequence score.

```
python3 puzzle8.py path/to/input/file.txt heuristic
```

For example, if you wanted to call this file on `Input1.txt` with Manhattan distance, and both files were in the same directory, you would invoke:

```
python3 puzzle8.py Input1.txt 1
```

If either argument is ignored, the file will throw an error.

Files Submitted

As stated above, this project has been categorised into multiple module files that add layers of abstraction to better keep track of what is going on.

File	Purpose
board.py	Holds the entire Board and Node modules, including the heuristic functions.It also contains an equality operator definition to compare nodes.
algorithm.py	The algorithm code. Holds the A* search algorithm, node expansion, and priority queue handling
puzzle8.py	The main file that you should run. This takes the path of the input file as an input and returns the final output as a file.

Miscellaneous

Everything has been commented in Google Docstings.

There are additional comments in the files to explain the obscure blocks of code, to help grade.

Overall, this project was quite interesting to finish. We had fun.

Authors

Juan Jose Castano Moreno
jc10536@nyu.edu
@juanjomoreno11

Rishyak Panchal
rishyak@nyu.edu
@rishyak

Course

NYU Tandon's CS-UY 4613 Intro to Artificial Intelligence.

Solutions Generated

These are the solutions generated for all files.

output1h1.txt: Input 1, H1

```
 4 1 6
8 3 5
2 0 7

8 4 6
0 1 5
2 3 7

4
10
U U L D
4 4 4 4 4
```

output1h2.txt: Input 1, H2

```
 4 1 6
8 3 5
2 0 7

8 4 6
0 1 5
2 3 7

4
10
U U L D
31 25 16 16 4
```

output2h1.txt: Input 2, H1

```
 2 6 0
1 3 7
4 5 8

1 2 0
7 5 3
4 8 6

12
40
L D R U L L D R D R U U
10 10 12 12 12 12 12 12 12 12 12 12 12
```

output2h2.txt: Input 2, H2

```
 2 6 0
1 3 7
4 5 8

1 2 0
7 5 3
4 8 6

12
40
L D R U L L D R D R U U
49 43 51 45 45 33 45 39 33 24 24 24 12
```

output3h1.txt: Input 3, H1

```
 8 6 3
0 4 5
7 2 1

1 2 3
4 0 7
6 5 8

25
914
U R D D R U L D L U U R D R D L L U U R D R D L U
19 19 19 21 21 21 21 23 23 23 23 23 23 23 23 25 25 25 25 25 25 25 25 25 25 25
```

output3h2.txt: Input 3, H2

```
 8 6 3
0 4 5
7 2 1

1 2 3
4 0 7
6 5 8

25
85
U R D R D L L U R R D L L U U R D R D L L U U R D
58 52 52 51 48 54 50 56 50 43 46 46 52 52 52 52 46 43 40 40 40 40 40 40 34 25
```

Source Code

This is all the source code of our program.

board.py

```
#!/usr/bin/python3

""" Object file for the Board class and Node class.
    This provides the objects and heuristic functionality
    for the puzzle. This is imported to algorithm.py and
    Node is imported to puzzle8.py.

Authors:
    Juan Jose Castano Moreno
    jc10536@nyu.edu

    Rishyak Panchal
    rishyak@nyu.edu
"""

class Board:
    """ Board Class

    Attributes:
        board (list[int]): List symbolising state of the puzzle.
        goal (list[int]): List symbolising goal state of the puzzle.
        successors (dict[int, int]): Dictionary with the successors of the goal state.
    """
    def __init__(self, board : list[int], goal : list[int]) -> None:
        self.board = board
        self.goal = goal


        self.successors = {}
        # Getting the successors from the goal state
        # Note: Yes it's ugly, but it's better than a loop.
        self.successors[goal[0]] = goal[1]
        self.successors[goal[1]] = goal[2]
        self.successors[goal[2]] = goal[5]
        self.successors[goal[3]] = goal[0]
        # Since the centre tile does not have a
        # successor, we just say that it is its
        # own successor.
        self.successors[goal[4]] = goal[4]
        self.successors[goal[5]] = goal[8]
        self.successors[goal[6]] = goal[3]
        self.successors[goal[7]] = goal[6]
```

```
        self.successors[goal[7]] = goal[6]
        self.successors[goal[8]] = goal[7]


    def __str__(self):
        """ Print method mainly for debugging purposes.

        Returns:
            str : String representation of the board.
        """
        return f"Current State = \n{self.board}\nGoal State = \n{self.goal}\n"


    @staticmethod
    def getRow(index : int) -> int:
        """ Returns the row of the index.

        Args:
            index (int): Index of the board.

        Returns:
            int : Row of the index.
        """
        return index // 3


    @staticmethod
    def getCol(index : int) -> int:
        """ Returns the column of the index.

        Args:
            index (int): Index of the board.

        Returns:
            int : Column of the index.
        """
        return index % 3


    @staticmethod
    def distance(current : int, goal : int) -> int:
        """ Calculates the expected distance
            from a tile's current position
            to its goal position.

        Args:
            current (int): Index of current tile.
            goal (int): Index of goal tile.

        Returns:
            int : The expected distance.
```

```
        """
        # Position of current
        currRow = Board.getRow(current)
        currCol = Board.getCol(current)

        # Goal position of goal
        goalRow = Board.getRow(goal)
        goalCol = Board.getCol(goal)

        # Distance between goal and current
        horDist = abs(goalRow - currRow)
        verDist = abs(currCol - goalCol)

        # Distance
        distance = horDist + verDist

        return distance


    def manhattanDistance(self) -> int:
        """ Calculates the expected Manhattan
            Distance from board to goal.

        Returns:
            int : The Manhattan Distance.
        """
        distance = 0
        # Get distance of each item, adding it to accumulator
        for index, item in enumerate(self.board):
            if item == 0:
                continue

            distance += self.distance(index, self.goal.index(item))

        return distance


    def nilssonScore(self) -> int:
        """ Calculates the expected Nilsson
            Sequence Score from board to goal.

        Returns:
            int : The Nilsson Sequence Score.
        """
        # Get Manhattan distance
        pn = self.manhattanDistance()
        sn = 0

        # Successors
        successors = self.successors
```

```
        # Clockwise order of access based on our structure
        indices = [0, 1, 2, 5, 8, 7, 6, 3]
        # Nilsson algorithm
        for index, val in enumerate(indices):
            curr = self.board[val]

            # Skip blank tile
            if curr == 0:
                continue

            # Get the successor of the current tile and the next tile
            succ = successors[curr]
            next = self.board[0] if val == 3 else self.board[indices[index + 1]]

            if succ != next:
                sn += 2

        # Checking the centre tile
        if self.board[4] != self.goal[4]:
            sn += 1

        return pn + 3 * sn


class Node:
    """ Node Class

    Attributes:
        board (list[int]): List symbolising state of the puzzle.
        goal (list[int]): List symbolising goal state of the puzzle.
        heuristic (int): The heuristic value of the current node.
            1 for Manhattan Distance
            2 for Nilsson Sequence Score
        parent (list[int]): Pointer to the parent node. Defaults to None.
        depth (Node): Depth of the current node in the tree. Defaults to 0.
        move (str): The movement of the blank square. One of [U, D, L, R]. Defaults to N
    """
    def __init__(self, board, goal, heuristic, parent = None, depth = 0, move = None):
        self.state = Board(board, goal)
        self.parent = parent
        self.depth = depth
        self.move = move
        self.heuristic = heuristic

        # Get the heuristic value for each node
        if heuristic == 1:
            self.hn = self.state.manhattanDistance()
        elif heuristic == 2:
            self.hn = self.state.nilssonScore()
        else:
            raise ValueError("Invalid Heuristic")
```

```
            raise ValueError('Invalid Heuristic')

        self.pathcost = self.hn + self.depth

    def __eq__(self, other) -> bool:
        """ Checks if two nodes are equal.

        Args:
            other (Node): Node to compare to.

        Returns:
            bool: True if the nodes are equal, False otherwise.
        """
        if isinstance(other, Node):
            return self.state == other.state
        else:
            return False
```

algorithm.py

```
#!/usr/bin/python3

""" Object file for the Algorithm class.
    This contains the entire algorithm
    and all execution is done here.
    It is invoked from puzzle8.py.

Authors:
    Juan Jose Castano Moreno
    jc10536@nyu.edu

    Rishyak Panchal
    rishyak@nyu.edu
"""

from board import Board, Node
distance = Board.distance
getRow = Board.getRow
getCol = Board.getCol

class Algorithm:
    """ Algorithm Class

    Attributes:
        active (list[Node]): List with the current states of the puzzle to be expanded
        visited (list[Node]) : List with the Nodes that were visited already
        visitedArrays (list[list[int]]): List with the arrays of the visited states
        totalNodes(Int) : Total amount of nodes generated by the algorithm
    """
```

```
    """
    def __init__(self, initialNode):
        self.active = []
        self.insertActive(initialNode)

        self.visited = []
        self.visitedArrays = []
        self.totalNodes = 1


    def insertActive(self, stateNode : Node) -> None:
        """ Properly inserts a node in the list of active nodes.

            Args:
                stateNode (Node): Node that we want to insert into active

            Returns:
                None.
                Function modifies the activeNodes and the visitedStates lists.
        """
        self.active.append(stateNode)
        if(len(self.active) == 1):
            return
        # Move stateNode to the left so that list is organized from biggest to smallest
        for i in reversed(range(1, len(self.active))):
            if(self.active[i].pathcost > self.active[i-1].pathcost):
                self.active[i], self.active[i-1] = self.active[i-1], self.active[i]


    def expand(self, expandNode : Node) -> None:
        """ Creates the current node's children.

        Args:
            expandNode (Node): Node that we want to expand.

        Returns:
            None.
            Function modifies the activeNodes and the visitedStates lists.
        """
        emptyIndex = expandNode.state.board.index(0)
        emptyRow = getRow(emptyIndex)
        emptyCol = getCol(emptyIndex)

        # Going over all values of the current state that we need to expand
        for index in range(len(expandNode.state.board)):

            currRow = getRow(index)
            currCol = getCol(index)

            # If the distance between the current tile and the empty tile is 1,
            # that means we can obtain new states from there
```

```
            # that means we can obtain new states from there
            if (distance(index, emptyIndex) == 1):
                newState = expandNode.state.board[:]
                newState[index], newState[emptyIndex] = newState[emptyIndex], newState[i
                verticalMove = currRow - emptyRow
                horizontalMove = currCol - emptyCol

                if(verticalMove < 0):
                    move = "U"
                elif(verticalMove > 0):
                    move = "D"
                elif(horizontalMove < 0):
                    move = "L"
                elif(horizontalMove > 0):
                    move = "R"

                newNode = Node(newState, expandNode.state.goal, expandNode.heuristic, ex

                # If the new node created hasn't been visited yet,
                # then add it to the list of nodes to expand
                if(newNode.state.board not in self.visitedArrays):
                    self.insertActive(newNode)
                    self.totalNodes += 1

        self.visited.append(expandNode)
        self.visitedArrays.append(expandNode.state.board)


    def aStarSearch(self) -> list:
        """ Calculates shortest path to goal state

        Returns:
            A list with the shortest path from the initial state to goal state.
        """
        # While there are active states in the tree
        while(len(self.active) > 0):

            # This will give us the active node with the smallest f(n)
            nodeToExpand = self.active.pop()
            shallowestD = nodeToExpand.depth

            # If the current node is a goal node
            if(nodeToExpand.state.board == nodeToExpand.state.goal):
                parent = nodeToExpand.parent
                moves = []
                pathcosts = []

                # This loop goes over the solution path to gather the data
                # that the algorithm should return
                while(True):
                    moves.insert(0, nodeToExpand.move)
```

```
                    pathcosts.insert(0, nodeToExpand.pathcost)
                    nodeToExpand = nodeToExpand.parent
                    parent = nodeToExpand.parent
                    if parent == None:
                        pathcosts.insert(0, nodeToExpand.pathcost)
                        nodeToExpand = nodeToExpand.parent
                        break

                return [shallowestD, self.totalNodes, moves, pathcosts]

            self.expand(nodeToExpand)

        return 0
```

puzzle8.py

```
#!/usr/bin/python3

""" Driver code for the program.
    This is the file that you will invoke from
    the command line like so:
    $ python3 puzzle8.py <filename> <heuristic>

Authors:
    Juan Jose Castano Moreno
    jc10536@nyu.edu

    Rishyak Panchal
    rishyak@nyu.edu
"""

import argparse

from algorithm import Algorithm
from board import Node

def makeFile(filename : str, heuristic : str, initial : list[int], goal : list[int], sol
    """ Creates a file with the solution.

    Args:
        filename (str): Name of the file to be created.
        heuristic (str): Heuristic used.
        initial (list[int]): Initial state.
        goal (list[int]): Goal state.
        solution (list[Node]): List with the solution to the puzzle.
            Contains:
                1. Shallowest depth of goal node.
                2. Total number of nodes generated.
```

```
            3. Sequence of actions to get to the goal state.
            4. f(n) values for each node in the solution path.

    Returns:
        None.
        Just creates a file with the solution.
    """
    # Format filename properly (remove ".txt")
    filename = filename[:-4]
    filename = f"{filename}_{heuristic}_output.txt"

    depth = solution[0]
    totalGen = solution[1]
    sequence = solution[2]
    fnValues = solution[3]

    # Write to file
    with open(filename, 'w') as f:
        # Counter for the formatting
        j = 0
        # Write initial board
        for i in initial:
            j += 1
            f.write(str(i) + ' ')
            if(j == 3):
                f.write('\n')
                j = 0
        f.write('\n')

        # Write goal board
        for i in goal:
            j += 1
            f.write(str(i) + ' ')
            if(j == 3):
                f.write('\n')
                j = 0
        f.write('\n')

        # Write depth
        f.write(str(depth) + '\n')
        # Write total number of generated nodes
        f.write(str(totalGen) + '\n')

        # Write sequence of moves of black tile
        for i in sequence:
            f.write(str(i) + ' ')
        f.write('\n')

        # Write f(n) values
        for i in fnValues:
```

```
            f.write(str(i) + ' ')


def main() -> None:
    """ Main function.

    Returns:
        None.
    """
    # Parse arguments
    parser = argparse.ArgumentParser(description='Solve the 8-puzzle problem using A* se
    parser.add_argument('filename', help='The txt file containing the initial and goal s
    parser.add_argument('heuristic', help='The heuristic to be used. 1 for Manhattan Dis
    cmdline = parser.parse_args()

    # Read input file
    with open(cmdline.filename, 'r') as f:
        input = []
        for line in f:
            line = line.strip().split()
            input = input + line

    # Create initial and goal states
    input = list(map(int, input))
    init, goal = input[:9], input[9:]

    # Find heuristic from command line
    if cmdline.heuristic == '1':
        heuristic = 1
        heur = "h1"
    elif cmdline.heuristic == '2':
        heuristic = 2
        heur = "h2"
    else:
        raise(ValueError('Invalid heuristic.'))

    # Create initial node
    initialNode = Node(init, goal, heuristic)
    # Call algorithm
    aStar = Algorithm(initialNode)

    # Get solution
    solution = aStar.aStarSearch()
    # Make the file
    makeFile(cmdline.filename, heur, init, goal, solution)


if __name__ == '__main__':
    main()
```