assignment1

October 25, 2017

In [ ]:
```
# -*- coding: utf-8 -*-

import heapq
import numpy as np
import re
from copy import deepcopy
from collections import Hashable


class PuzzleNode(object):
    def __init__(self, size, state=[], parent=None,
                 heuristicCost=0, moves=0):
        """
        PuzzleNode class representing the state of tiles as a 2D array.

        @param size: Size of puzzle board
        @keyword state: Starting state of board (default: [])
        @keyword parent: Pointer to parent PuzzleNode (default: None)
        @keyword heuristicCost: Estimated cost of going from current node to
                                goal node (default: 0)
        @keyword moves: Cost of going from initial node to current node
                        (default: 0)

        @ivar board: 2D array representing state of tiles
        @ivar pathCost: Cost of going from initial node to current node, plus
                        estimated cost of going from current node to goal node
        """
        self.board = np.array(state or [[0]*size for count in range(size)]) \
            if type(state) != np.ndarray else state
        self.parent = parent
        self.pathCost = moves + heuristicCost
        self.moves = moves

        if not len(state):
            self.__populateSelf()

    def __populateSelf(self):
```

```
        boardSize = len(self.board)
        cellNumber = 0
        for i in range(boardSize):
            for j in range(boardSize):
                self.board[i][j] = cellNumber
                cellNumber += 1

    def __hash__(self):
        return int(''.join(
            str(_) for _ in self.board.flatten()
        ))

    def __eq__(self, other):
        if self.__class__ != other.__class__:
            return False
        else:
            return np.array_equal(self.board, other.board)

    def __lt__(self, other):
        if self.__class__ != other.__class__:
            return False
        else:
            return self.pathCost < other.pathCost

    def __str__(self):
        """
        Prints 8-puzzle in following format:
        +-----------+
        | - | - | - |
        | - | - | - |
        | - | - | - |
        +-----------+
        Square puzzles of other sizes are printed analogously.

        @return: Puzzle board formatted as described above.
        """
        boardSize = len(self.board)
        maxCellSize = len(str(boardSize**2 - 1))

        # Fun thing I learned: I've separated the bits which use re.sub and
        # those which use str.replace because re.sub tends to be slower.
        # Also, just because I don't need it if I'm not actually using a
        # non-static expression.
        # More fun discussion here:
        # https://stackoverflow.com/questions/452104/is-it-worth-using-pythons-re-comp
        currentCellSize = 1
        cellsWithSpaces = str(self.board.tolist())[1:-1]
        for i in range(maxCellSize, 0, -1):
```

```
            digitRegex = re.compile(r'(\b\d{{{}}}\b)'.format(currentCellSize))
            cellsWithSpaces = digitRegex.sub(r'\1 |'.rjust(i+4),
                                             cellsWithSpaces)
            currentCellSize += 1
        withDashes = re.sub(r'\b0\b', '-', cellsWithSpaces)

        return '+' + '-' * (boardSize * (maxCellSize + 3) - 1) + '+\n' + \
            withDashes.replace('[', '|') \
                      .replace(']', '\n') \
                      .replace(', ', '') + \
            '+' + '-' * (boardSize * (maxCellSize + 3) - 1) + '+'


class InvalidBoardFormatError(TypeError):
    def __init__(self, board):
        self.message = "Board is not an instance of {} or {}: {}" \
            .format(list, np.ndarray, board)


class IncorrectBoardSizeError(ValueError):
    def __init__(self, board, size):
        self.message = 'Size {0} does not match dimensions of board {1}x{1}' \
            .format(size, board)


class InvalidTilesError(ValueError):
    def __init__(self, actualOccurrences, tile):
        self.message = 'Board contains {} occurrences of tile <{}> when it' \
                       ' should contain 1'.format(actualOccurrences, tile)


class UnsolvableBoardError(ValueError):
    def __init__(self, board):
        self.message = 'Board {} is unsolvable'.format(board)


def memoize(func):
    class memoized(dict):
        def __init__(self, func):
            self.func = func

        def __call__(self, arg):
            # Can we actually memoize this?
            if not isinstance(arg, Hashable):
                return self.func(arg)

            return self[arg]
```

```python
        def __missing__(self, key):
            value = self[key] = self.func(key)
            return value

        def __repr__(self):
            return repr(self.func)

    return memoized(func)


def validatePuzzle(n, state, makeSolvable=False):
    """
    Validates given n-puzzle.

    @param n: Size of puzzle board
    @param state: State of puzzle
    @keyword makeSolvable: If state is unsolvable and makeSolvable is True,
                           swaps two adjacent tiles such that the parity of the
                           board is reversed. If False or state is solvable,
                           nothing is swapped. (default: False)

    @return err: Error code 0 if n and state are valid; else
                 -1 if IncorrectBoardSizeError, InvalidBoardFormatError, or
                 InvalidTilesError are raised; else
                 -2 if UnsolvableBoardError is raised
    @return solvableBoard: If state is unsolvable and makeSolvable is True,
                           a solvable board. If either state is solvable or
                           makeSolvable is False, None.

    @raise IncorrectBoardSizeError: Raised if size of both state and all
                                    sublists of state doesn't equal n
    @raise InvalidBoardFormatError: Raised if state is formatted incorrectly
    @raise InvalidTilesError: Raised if state doesn't contain every number
                              from 0 to n^2-1 exactly once
    @raise UnsolvableBoardError: Raised if state cannot be solved to goal state
                                 (i.e. board parity is not even for odd n;
                                 board parity is not even for even n with 0 on
                                 odd row; or board parity is not odd for even
                                 n with 0 on even row, counting from top
                                 starting from 1  modified from Johnson &
                                 Story's (1879) use of the Manhattan distance
                                 to fit our different goal state).
    """

    err = 0
    solvableBoard = None

    try:
```

```python
if type(state) not in (np.ndarray, list):
    raise InvalidBoardFormatError(state)

if len(state) != n:
    raise IncorrectBoardSizeError(n, state)

for i in range(n):
    if len(state[i]) != n:
        raise IncorrectBoardSizeError(n, state)

expectedTile = 0
for j in range(n**2):
    actualOccurrences = sum(_.count(expectedTile) for _ in state)
    if actualOccurrences != 1:
        raise InvalidTilesError(actualOccurrences, expectedTile)
    expectedTile += 1

solvable = True

# Parity of board permutations
numTiles = 0
for k in range(n):
    for l in range(n):
        currentTile = state[k][l]

        if currentTile not in (0, 1):
            flattenedBoard = np.array(state).flatten().tolist()
            currentTileIdx = flattenedBoard.index(currentTile)
            numTiles += len(list(filter(
                lambda tile: tile < currentTile and tile != 0,
                flattenedBoard[currentTileIdx+1:]
            )))
permutationsParity = numTiles % 2

if n % 2 != 0:
    if permutationsParity != 0:
        solvable = False
else:
    # Parity of row with empty square
    emptyI, emptyJ = list(zip(*np.where(np.array(state) == 0)))[0]
    emptySquareRowParity = emptyI % 2

    if permutationsParity != emptySquareRowParity:
        solvable = False

if not solvable:
    if makeSolvable:
        # Unsolvable puzzles can be made solvable by swapping the first
```

```
                # two non-zero tiles.
                for x in range(1, n):
                    for y in range(1, n):
                        if state[x-1][y-1] != 0:
                            state[x][y], state[x-1][y-1] = \
                                state[x-1][y-1], state[x][y]

                            return err, solvableBoard
            else:
                raise UnsolvableBoardError(state)

    except (
        InvalidBoardFormatError,
        IncorrectBoardSizeError,
        InvalidTilesError
    ):
        err = -1
    except UnsolvableBoardError:
        err = -2

    return err, solvableBoard


@memoize
def misplacedTiles(state):
    """
    Heuristic function for number of misplaced tiles in given state.

    @param state: State of n-puzzle board
    @return misplacedTiles: Number of misplaced tiles in state
    """
    boardSize = len(state)
    goal = PuzzleNode(boardSize)

    misplacedTiles = 0

    for i in range(boardSize):
        for j in range(boardSize):
            if (state[i][j] != goal.board[i][j]):
                misplacedTiles += 1

    return misplacedTiles


@memoize
def manhattanDistance(state):
    """
    Heuristic function for how far away each tile is from its goal position,
```

```
    using sum of individual Manhattan distances as the distance metric.

    @param state: State of puzzle board
    @return distance: Sum of how far away each tile is from its goal position
    """
    boardSize = len(state)
    goalCoords = [(i, j) for i in range(boardSize) for j in range(boardSize)]
    goalCoordsDict = {tile: coords for tile, coords in enumerate(goalCoords)}
    distance = 0

    for x in range(boardSize):
        for y in range(boardSize):
            tile = state[x][y]

            if tile != 0:
                goalX, goalY = goalCoordsDict[tile]
                distance += abs(x - goalX) + abs(y - goalY)

    return distance


def getChildStates(state):
    """
    Generates child states by looking at state to determine which tiles
    can move into the empty space, then returning all swaps of the empty
    space with those tiles.

    @param state: Current PuzzleNode
    @return possibleChildren: Possible child states
    """
    # First find which tiles can move into empty space
    emptyI, emptyJ = list(zip(*np.where(state.board == 0)))[0]
    canMoveIntoEmpty = set()

    for i in (1, -1):
        if (emptyI + i < len(state.board)) and (emptyI + i >= 0):
            canMoveIntoEmpty.add((emptyI + i, emptyJ))

        if (emptyJ + i < len(state.board)) and (emptyJ + i >= 0):
            canMoveIntoEmpty.add((emptyI, emptyJ + i))

    # Swap tiles to generate each possible child state
    possibleChildren = []

    for possibleMove in canMoveIntoEmpty:
        i, j = possibleMove
        possibleChild = deepcopy(state.board)
        possibleChild[i][j], possibleChild[emptyI][emptyJ] = \
```

```
            possibleChild[emptyI][emptyJ], possibleChild[i][j]
        possibleChildren.append(possibleChild)

    return possibleChildren


def solvePuzzle(n, state, heuristic, printed=True):
    """
    n-puzzle solver using A*, modified from R. Shekhar's A* code from
    lesson 3.1.

    @param n: Size of puzzle board
    @param state: Starting state of puzzle
    @param heuristic: Heuristic function handler
    @keyword printed: If True, solvePuzzle prints all states of the board (as
                      lists of lists) from initial state to goal, as well as
                      the number of moves to reach the goal and the maximum
                      frontier size. If False, nothing is printed. (default:
                      True)

    @var frontier: A priority queue ordered by path cost
    @var explored: A dictionary of previously seen states mapped to the
                   PuzzleNode (and thus pathCost, depth) at which we saw them.
                   This prevents repeated states, including those which arise
                   from simply undoing the previous move.

    @return totalMoves: Number of moves required to reach goal state from
                        initial state
    @return maxFrontierSize: Maximum size of frontier during the search
    @return err: -1 if n or state is invalid, -2 if board can't be solved,
                 else 0. If err is non-zero, moves and frontierSize are 0.
    """

    moves = 0
    totalMoves = 0
    maxFrontierSize = 0
    err, solvableBoard = validatePuzzle(n, state)

    if not err:
        state = solvableBoard or state

        initialNode = PuzzleNode(
            size=n,
            state=state,
            heuristicCost=heuristic(state)
        )
        goalNode = PuzzleNode(size=n)
```

```python
explored = {}
frontier = [initialNode]
maxFrontierSize += 1

while frontier:
    currentNode = heapq.heappop(frontier)

    if currentNode == goalNode:
        totalMoves = currentNode.moves

        if printed:
            solutionPath = [str(currentNode.board.tolist())]
            while currentNode.parent:
                solutionPath.append(
                    str(currentNode.parent.board.tolist())
                )
                currentNode = currentNode.parent

            print('\n'.join(solutionPath[::-1]))
            print(totalMoves)
            print(maxFrontierSize)

        return totalMoves, maxFrontierSize, err

    explored[currentNode] = currentNode

    childStates = getChildStates(currentNode)

    for childState in childStates:
        childNode = PuzzleNode(
            size=n,
            state=childState,
            parent=currentNode,
            heuristicCost=heuristic(childState),
            moves=currentNode.moves + 1
        )

        exploredNode = explored.get(childNode)

        # If we: a) haven't yet explored this node, or
        # b) explored this node when it had a higher pathCost,
        # then we should put it into the frontier so that
        # we can explore it.
        if (
            exploredNode is None or
            childNode.pathCost < exploredNode.pathCost
        ):
            heappush(frontier, childNode)
```

```python
                # If we're already planning to explore this node later on
                # (i.e. it's in the frontier), AND the version of the node
                # we're currently looking at has a lower pathCost than the
                # one in our frontier, then we should replace the node in
                # our frontier with the one we're currently looking at.
                elif (
                    exploredNode is not None and
                    exploredNode in frontier and
                    childNode.pathCost < exploredNode.pathCost
                ):
                    frontier.remove(exploredNode)
                    heapq.heappush(frontier, childNode)

            moves += 1
            if len(frontier) > maxFrontierSize:
                maxFrontierSize = len(frontier)

    return totalMoves, maxFrontierSize, err


def testSolvePuzzle(boards, heuristics):
    """
    Tests solvePuzzle, printing out number of moves and maximum frontier size
    for each given initial condition.

    @param boards: Scrambled boards to use as initial states in tests
    @param heuristics: Available heuristics for A* algorithm
    """

    for board in boards:
        for heuristic in heuristics:
            n = len(board)
            print('Current board: {}'.format(board))
            print('Current heuristic: {}'.format(heuristic))
            moves, frontierSize, err = solvePuzzle(n=n,
                                                   state=board,
                                                   heuristic=heuristic,
                                                   printed=False)

            print('Moves to solve current board: {}'.format(moves))
            print('Current board\'s max frontier size: {}\n'.format(
                  frontierSize))


heuristics = [misplacedTiles, manhattanDistance]
testBoards = [
    [[5, 7, 6], [2, 4, 3], [8, 1, 0]],
```

```
    [[7, 0, 8], [4, 6, 1], [5, 3, 2]],
    [[2, 3, 7], [1, 8, 0], [6, 5, 4]]
]
testSolvePuzzle(testBoards, heuristics)
```

initial state	heuristic	number of moves	max frontier size
[[5, 7, 6], [2, 4, 3], [8, 1, 0]]	Misplaced tiles	28	28547
	Manhattan distance	28	937
[[7, 0, 8], [4, 6, 1], [5, 3, 2]]	Misplaced tiles	25	15444
	Manhattan distance	25	983
[[2, 3, 7], [1, 8, 0], [6, 5, 4]]	Misplaced tiles	17	452
	Manhattan distance	17	68

From the above table, it's clear that the Manhattan distance heuristic far outperforms the misplaced tiles heuristic in terms of the number of possible positions it generates (i.e., Manhattan distance usually has a frontier size smaller than misplaced tiles, by 1 order of magnitude).

```
In [22]: %%html
         <!-- This styling bit just helps me style my table, since Jupyter/IPython doesnt
         allow that like in true Github Markdown -->
         <style>
         table, th, td {
             border: 1px solid black !important;
             border-collapse: collapse;
         }
         td:last-child, th:last-child, td:nth-child(3), th:nth-child(3) {
             text-align: center !important;
         }
         </style>
```

```
<IPython.core.display.HTML object>
```