DS340 Assignment 1: A*, Heuristics, and the Fifteen Puzzle

In this assignment, you'll get some experience abstracting a problem into a search problem, implement the A* search algorithm, and experiment with the effects of using different heuristics for the search. You'll hopefully see how solving a complex multistep problem from first principles is something classic AI is quite good at.

You will only need to submit this completed .ipynb notebook to Blackboard, as well as a PDF version in case the .ipynb has a problem (Print->Save to PDF). Despite the redundancy, please make sure you submit the most recent version of your notebook file.

The goal of a fifteen puzzle is to get all fifteen tiles in order from left to right, top to bottom, like so:

1 2 3 4

5 6 7 8

9 10 11 12

13 14 15 -

The only legal moves are to move a tile adjacent to the blank into the blank space, making the tile's previous space blank. Thus the maximum number of neighbors is 4, but the number of neighbors could be as small as 2 if the blank is in a corner.

1, 4 points) If the blank is not counted as a tile, then tiles displaced is an admissible heuristic, because every tile must take at least one move to get to its final location. Is a count of number of tiles out of place an admissible heuristic if the blank *is* counted as a tile? If the heuristic is admissible, explain how you know, and if it is not, give an example that shows the heuristic is inadmissible.

TODO

If the blank tile *is* considered a tile, then the number of tiles out of place is not an admissible heuristic. The reason for this is because the blank tile being not in the correct location, the heuristic will overestimate the cost to reach a valid solution. One example of this is when there is just one move left to finish the puzzle. If the empty square can be switched with a neighboring tile to solve the puzzle, the heuristic would overestimate the cost to reach the goal since the heuristic would think there are 2 tiles out of place instead of the 1 tile that is out of place.

Here is some provided code to get you started. Notice the functions that are available to you.

```
"""Use A* to solve fifteen puzzle instances.
```

```
The "main" of this code is solve_and_print, at the end.  We'll try two different
heuristics, counting tiles out of place and summing Manhattan distance from
the destination over all tiles (the better heuristic)."""

import sys
import copy
import numpy as np
from queue import PriorityQueue

PUZZLE_WIDTH = 4
BLANK = 0  # Integer comparison tends to be faster than string comparison

def read_puzzle_string(puzzle_string):
    """Read a NumberPuzzle from string representation; space-delimited, blank is "-".

    Args:
      puzzle_string (string):  string representation of the puzzle

    Returns:
      A NumberPuzzle
    """
    new_puzzle = NumberPuzzle()
    row = 0
    for line in puzzle_string.splitlines():
        tokens = line.split()
        for i in range(PUZZLE_WIDTH):
            if tokens[i] == '-':
                new_puzzle.tiles[row][i] = BLANK
                new_puzzle.blank_r = row
                new_puzzle.blank_c = i
            else:
                try:
                    new_puzzle.tiles[row][i] = int(tokens[i])
                except ValueError:
                    sys.exit("Found unexpected non-integer for tile value")
        row += 1
    return new_puzzle

class NumberPuzzle(object):
    """ Class containing the state of the puzzle, as well as A* bookkeeping info.

    Attributes:
        tiles (numpy array): 2D array of ints for tiles.
        blank_r (int):  Row of the blank, for easy identification of neighbors
        blank_c (int):  Column of blank, same reason
```

```
        parent (NumberPuzzle):  Reference to previous puzzle, for backtracking later
        dist_from_start (int):  Steps taken from start of puzzle to here
        key (int or float):  Key for priority queue to determine which puzzle is next
    """

    def __init__(self):
        """ Just return zeros for everything and fill in the tile array later"""
        self.tiles = np.zeros((PUZZLE_WIDTH, PUZZLE_WIDTH))
        self.blank_r = 0
        self.blank_c = 0
        # This next field is for our convenience when generating a solution
        # -- remember which puzzle was the move before
        self.parent = None
        self.dist_from_start = 0
        self.key = 0

    def __str__(self):
        out = ""
        for i in range(PUZZLE_WIDTH):
            for j in range(PUZZLE_WIDTH):
                if j > 0:
                    out += " "
                if self.tiles[i][j] == BLANK:
                    out += "-"
                else:
                    out += str(int(self.tiles[i][j]))
            out += "\n"
        return out

    def copy(self):
        """Copy the puzzle and update the parent field.

        In A* search, we generally want to copy instead of destructively alter,
        since we're not backtracking so much as jumping around the search tree.
        Also, if A and B are numpy arrays, "A = B" only passes a reference to B.
        We'll also use this to tell the child we're its parent."""
        child = NumberPuzzle()
        child.tiles = np.copy(self.tiles)
        child.blank_r = self.blank_r
        child.blank_c = self.blank_c
        # TODO:  set child.dist_to_start and child.parent
        child.parent = self
        child.dist_from_start = self.dist_from_start
        return child

    def __eq__(self, other):
```

```
        """Governs behavior of ==.

        Overrides == for this object so that we can compare by tile arrangement
        instead of reference.  This is going to be pretty common, so we'll skip
        a type check on "other" for a modest speed increase"""
        return np.array_equal(self.tiles, other.tiles)

    def __hash__(self):
        """Generate a code for hash-based data structures.

        Hash function necessary for inclusion in a set -- unique "name"
        for this object -- we'll just hash the bytes of the 2D array"""
        return hash(bytes(self.tiles))

    def __lt__(self, obj):
        """Governs behavior of <, and more importantly, the priority queue.

        Override less-than so that we can put these in a priority queue
        with no problem.  We don't want to recompute the heuristic here,
        though -- that would be too slow to do it every time we need to
        reorganize the priority queue"""
        return self.key < obj.key

    def move(self, tile_row, tile_column):
        """Move from the row, column coordinates given into the blank.

        Also very common, so we will also skip checks for legality to improve speed.

        Args:
            tile_row (int):  Row of the tile to move.
            tile_column (int):  Column of the tile to move.
        """

        self.tiles[self.blank_r][self.blank_c] = self.tiles[tile_row][tile_column]
        self.tiles[tile_row][tile_column] = BLANK
        self.blank_r = tile_row
        self.blank_c = tile_column
        # TODO:  Set self.dist_to_start to the right value, now that we've
        # added a move
        self.dist_from_start += 1

    def legal_moves(self):
        """Return a list of NumberPuzzle states that could result from one move.

        Return a list of NumberPuzzle states that could result from one move
        on the present board.  Use this to keep the order in which
        moves are evaluated the same as our solution.  (Also notice we're still in the
```

```
        moves are evaluated the same as our solution.  (Also notice we're still in the
        methods of NumberPuzzle, hence the lack of arguments.)

        Returns:
            List of NumberPuzzles.
        """
        legal = []
        if self.blank_r > 0:
            down_result = self.copy()
            down_result.move(self.blank_r-1, self.blank_c)
            legal.append(down_result)
        if self.blank_c > 0:
            right_result = self.copy()
            right_result.move(self.blank_r, self.blank_c-1)
            legal.append(right_result)
        if self.blank_r < PUZZLE_WIDTH - 1:
            up_result = self.copy()
            up_result.move(self.blank_r+1, self.blank_c)
            legal.append(up_result)
        if self.blank_c < PUZZLE_WIDTH - 1:
            left_result = self.copy()
            left_result.move(self.blank_r, self.blank_c+1)
            legal.append(left_result)
        return legal

    def solve(self, better_h):
        """Return a list of puzzle states from this state to solved.

        Args:
            better_h (boolean):  True if Manhattan heuristic, false if tile counting

        Returns:
            path (list of NumberPuzzle or None) - path from start state to finish state
            explored - total number of nodes pulled from the priority queue
        """
        # TODO
        # priority queue to store the different board states to visit
        prioQueue = PriorityQueue()
        # set containing the different board states that have been visited before (hashed)
        explored = set()

        # intial setup for the first node
        prioQueue.put(self)

        solution = None

        # while the prioQueue is not empty perform A* search
```

```
        # while the prioQueue is not empty perform A* search
        while prioQueue.not_empty :
            # get the current board state to explore
            current = prioQueue.get()
            # if it hasn't been explored before, then continue (else continue)
            if current in explored:
                continue

            # if current is the goal, then we need to build the path to the solution from the start and return it
            # (we break out of the loop)
            if current.solved():
                solution = []
                trav = current
                while not trav is None:
                    solution.append(trav)
                    trav = trav.parent
                solution.reverse()
                return solution , current.dist_from_start

            # if it hasn't been explored yet, explore the possible states from this board
            explored.add(current)
            # get list of possible moves from this position
            legalMoves = current.legal_moves()

            # iterate over the possible legal moves
            for move in legalMoves:
                # for each legal move, use the heuristic to estimate the cost and add it into the priority queue
                move.key = move.dist_from_start + move.heuristic(False)
                prioQueue.put(move)

        return None, 0

    def solved(self):
        """"Return True iff all tiles in order and blank in bottom right."""
        should_be = 1
        for i in range(PUZZLE_WIDTH):
            for j in range(PUZZLE_WIDTH):
                if self.tiles[i][j] != should_be:
                    return False
                should_be = (should_be + 1) % (PUZZLE_WIDTH ** 2)
        return True

    def heuristic(self, better_h):
        """Wrapper for the two heuristic functions.

        Args:
            better_h (boolean):  True if Manhattan heuristic, false if tile counting
```

```
            better_h (boolean): True if Manhattan heuristic, False if tile counting

        Returns:
            Value of the cost-to-go heuristic (int or float)
        """
        if better_h:
            return self.manhattan_heuristic()
        return self.tile_mismatch_heuristic()

    def tile_mismatch_heuristic(self):
        """Returns count of tiles out of place."""
        # TODO
        mismatch_count = 0
        # iterate over all tiles and check if the number matches the
        for i in range(PUZZLE_WIDTH):
            for j in range(PUZZLE_WIDTH):
                tile = self.tiles[i][j]
                if(tile != BLANK and int(tile) != (i * PUZZLE_WIDTH) + j + 1):
                    mismatch_count += 1
        return mismatch_count

    def manhattan_heuristic(self):
        """Returns total Manhattan (city block) distance from destination over all tiles."""
        # TODO
        total_manhattan = 0
        for row in range(PUZZLE_WIDTH):
            for col in range(PUZZLE_WIDTH):
                tile = self.tiles[row][col]
                if(tile != BLANK and tile != (row * PUZZLE_WIDTH) + col + 1):
                    total_manhattan += abs(row - ((tile - 1) // PUZZLE_WIDTH)) + abs(col - ((tile - 1) % PUZZLE_WIDTH))

        return total_manhattan

    def path_to_here(self):
        """Returns list of NumberPuzzles giving the move sequence to get here.

        Retraces steps to this node through the parent fields."""
        path = []
        current = self
        while not current is None:
            path.insert(0, current)  # push
            current = current.parent
        return path

def print_steps(path):
    """ Print every puzzle in the path.
```

```
    Args:
        path (list of NumberPuzzle): list of puzzle states from start to finish
    """
    if path is None:
        print("No path found")
    else:
        print("{} steps".format(len(path)-1))
        for state in path:
            print(state)


def solve_and_print(puzzle_string : str, better_h : bool) -> None:
  """ "Main" - prints series of moves necessary to solve puzzle.

  Args:
    puzzle_string (string):  The puzzle to solve.
    better_h (boolean):  True if Manhattan distance heuristic, false if tile count
  """
  my_puzzle = read_puzzle_string(puzzle_string)
  solution_steps, explored = my_puzzle.solve(better_h)
  print("{} nodes explored".format(explored))
  print_steps(solution_steps)
```

2, 4 points) Two of the provided functions for generating new board states have been left incomplete for you to finish. copy() needs to set the dist_to_start and parent attributes appropriately - in particular, parent needs to be set to the state being copied so that path_to_here() can later backtrack through move sequences. move() needs update dist_to_start to reflect the fact that a new move has been made. Update both of these functions before proceeding.

3, 22 points) In solve(), implement A*, using a heuristic of “number of tiles in the wrong place” as the optimistic estimate of moves to go. (Treat the blank the way you decided was better in question 1.) Then you will need to make use of two important data structures:

• The queue of puzzle states to explore should be a PriorityQueue, already imported for you at the top. The __lt__() function for NumberPuzzle objects has already been overridden so that it compares the key field to decide what goes first, but that field is currently never initialized.

• Use a set() to efficiently implement a "closed list" of states that have already been explored. (Do not literally use a list, since scanning a list for an item is not efficient.) Sets are hash tables, and the hashing behavior has already been implemented to work in an acceptable way.

solve() should return a list of NumberPuzzles that show the states from the beginning to the end, as well as an integer count of the number of nodes explored (pulled from the front of the priority queue). The latter is to help you debug and help us grade, although there is some "wiggle room" for reasonable differences in implementation.

Note that you may be penalized if you unnecessarily change the provided code. In particular, you must generate neighbors using the provided legal_moves() function, so that your output should match our own if the heuristics are implemented correctly.

When you have an implementation, try your solution on the provided zero_moves, one_move, and six_moves puzzles using solve_and_print(), and check that they are solved in the required number of moves.

```
zero_moves = """1 2 3 4
5 6 7 8
9 10 11 12
13 14 15 -"""

one_move = """1 2 3 4
5 6 7 8
9 10 11 12
13 14 - 15"""

six_moves = """1 2 3 4
5 10 6 8
- 9 7 12
13 14 11 15"""

sixteen_moves = """10 2 4 8
1 5 3 -
9 7 6 12
13 14 11 15"""

forty_moves = """4 3 - 11
2 1 6 8
13 9 7 15
10 14 12 5"""
```

```
solve_and_print(zero_moves, False)
```

```
0 nodes explored
0 steps
1 2 3 4
5 6 7 8
9 10 11 12
13 14 15 -
```

```
solve_and_print(one_move, False)
```

```
1 nodes explored
1 steps
1 2 3 4
5 6 7 8
9 10 11 12
13 14 - 15

1 2 3 4
5 6 7 8
9 10 11 12
13 14 15 -
```

```
solve_and_print(six_moves, False)
```

```
6 nodes explored
6 steps
1 2 3 4
5 10 6 8
- 9 7 12
13 14 11 15

1 2 3 4
5 10 6 8
9 - 7 12
13 14 11 15

1 2 3 4
5 - 6 8
9 10 7 12
13 14 11 15

1 2 3 4
5 6 - 8
9 10 7 12
13 14 11 15

1 2 3 4
5 6 7 8
9 10 - 12
13 14 11 15

1 2 3 4
5 6 7 8
9 10 11 12
13 14 - 15
```

```
1 2 3 4
5 6 7 8
9 10 11 12
13 14 15 -
```

4, 1 point) Now time your implementation on sixteen_moves, using the handy Google Colab syntax demonstrated here.

```
%time solve_and_print(sixteen_moves, False)
```

```
1 10 2 4
5 - 3 8
9 7 6 12
13 14 11 15

1 - 2 4
```

```
9 10 7 12
13 14 11 15

1 2 3 4
5 6 7 8
9 10 - 12
13 14 11 15

1 2 3 4
5 6 7 8
9 10 11 12
13 14 - 15

1 2 3 4
5 6 7 8
9 10 11 12
13 14 15 -

CPU times: user 31.5 ms, sys: 902 µs, total: 32.4 ms
Wall time: 32.1 ms
```

5, 6 points) The Manhattan distance of a tile from its final location is the sum of the difference in rows and the difference in columns. If the blank does not count as a tile, does the sum of Manhattan distances from their final locations over all tiles act as an admissible heuristic? What if the blank does count as a tile? In each case, if the heuristic is admissible, explain how you know, and if it is not, give an example that shows the heuristic is inadmissible.

TODO

If the blank does not count as a tile, the sum of the Manhattan distance from their final locations over all tiles does act as an admissible heuristic. Since tiles can only move by swapping with the blank tile (moving by either one row or one column) and Manhatten distance of a tile from its final location is the sum of the difference in rows and the difference in columns, the Manhattan distance will never overestimate the number of moves it takes to reach a solved puzzle.

If the blank tile does count as a tile, the Manhattan distance of all tiles wouldn't act as an admissible heuristic because it would overestimate the cost to solve the puzzle due to the fact that it will count the distance the blank tile is from the goal. One example of this is in this puzzle state [- 1 2 3 4] (where the puzzle has 4 pieces and is one line). Even though the actual cost to finish the puzzle here is 4 moves (sliding the blank over 4 times), the heuristic counting the Manhattan distance of the blank tile would overestimate the cost to finish the puzzle from this state.

6, 10 points) Now **implement Manhattan distance as a new heuristic** in the same block of code above. Keep your old heuristic, but have the code use the old heuristic if the better_h argument is False, and use the new heuristic if better_h is True. When you are done, **time the new code**

on the sixteen move puzzle.

```
%time solve_and_print(sixteen_moves, True)
```

```
1 10 2 4
5 - 3 8
9 7 6 12
13 14 11 15

1 - 2 4
5 10 3 8
9 7 6 12
13 14 11 15

1 2 - 4
5 10 3 8
9 7 6 12
```

```
1 2 3 4
5 6 7 8
9 10 11 12
13 14 - 15

1 2 3 4
5 6 7 8
9 10 11 12
13 14 15 -

CPU times: user 28.5 ms, sys: 0 ns, total: 28.5 ms
Wall time: 29.6 ms
```

7, 1 point) Your code should now also finish within two minutes for forty_moves. **Run it here** to demonstrate.

```
%time solve_and_print(forty_moves, True)
```

8, 4 points) Suppose you decide to try out Euclidean distance, $\sqrt{r^2 + c^2}$ where r and c are the row and column differences, as a heuristic. **It runs faster than tiles displaced, but slower than Manhattan distance. Why?** (Assume it's not the slowness of square root operations, or anything like that.)

TODO

The reason that the Euclidian distance runs faster than the tiles displaced heuristic but slower than the Manhattan distance heuristic is because it is better at estimating the cost to solve a board state than the number of tiles misplaced, but isn't as accurate as the Manhattan distance. The reason for this is because the Euclidian distance can represent the diagonal distance of tiles from their final position. However, we cannot move the tiles diagonally, so the Euclidian distance underestimates the cost to solve the puzzle much more than the Manhattan distance. This leads to A* exploring more possible states before arriving at the optimal solution, thus taking more time.

9, 4 points) Suppose a bug caused you to calculate the Manhattan distance incorrectly, so that it only returned the number of rows away for each tile, ignoring columns. **Is this heuristic still going to return an optimal solution every time? Why or why not?**

TODO In the case where the Manhattan distance is incorrectly calculated so it only returned the number of rows away for each tile and ignores columns, the heuristic will still return an optimal solution every time. Since the incorrect heuristic underestimates the cost to solve each puzzle, it is an admissible heuristic which guarnatees an optimal solution every time. However, due to how the heuristic dramatically underestimates the cost to solve the puzzle, it will explore many more possibilities before arriving at the correct solution.

10, 4 points) In some fifteen-puzzle implementations, you can slide not just one tile, but all tiles to one side of the blank into the blank space. (For example, if the bottom row were - 13 14 15, one move could cause 13 14 15 -.) **Are the tiles displaced and Manhattan distance heuristics admissible in that case?**

TODO

In the case that you can slide all tiles to one side of the blank into the blank space, the tiles displaced and Manhattan distance heuristics are no longer admissible. The reason for this is because the tiles displaced and Manhattan distance heuristics would overestimate the cost to solve the puzzle in certain cases, leading A* to return a potentially unoptimal solution.

A* is one of the most successful algorithms in the history of AI, a champ at what it does (as long as you can come up with a good heuristic), and is still used extensively in games. It's a classic technique for a reason!

Be sure you've done all other bold text, then "File->Download .ipynb" and upload your .ipynb file to Blackboard, along with a PDF version (File->Print->Save as PDF) of your assignment.

✓ 0s completed at 4:58 PM