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

Counting the number of tiles out of place, including the blank as a tile, is an admissible heuristic for the fifteen puzzle. This is because each tile that is not in its correct position, including the blank, must be moved at least once to reach the goal state. This heuristic is optimistic, as it assumes that each tile can be moved to its correct location without extra moves, and it never overestimates the number of moves required. Thus, it adheres to the criteria of an admissible heuristic by providing a lower bound on the number of moves needed to solve the puzzle.

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

In [1]:
```
"""Use A* to solve fifteen puzzle instances.

The "main" of this code is solve_and_print, at the end.  We'll try two diffe
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, blar

    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

    Attributes:
        tiles (numpy array): 2D array of ints for tiles.
        blank_r (int):  Row of the blank, for easy identification of neighbo
        blank_c (int):  Column of blank, same reason
        parent (NumberPuzzle):  Reference to previous puzzle, for backtracki
        dist_from_start (int):  Steps taken from start of puzzle to here
        key (int or float):  Key for priority queue to determine which puzzl
    """
```

```
    def __init__(self):
        """ Just return zeros for everything and fill in the tile array late
        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 alt
        since we're not backtracking so much as jumping around the search tr
        Also, if A and B are numpy arrays, "A = B" only passes a reference t
        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_from_start and child.parent
        child.dist_from_start = self.dist_from_start  # Copy the distance
        child.parent = self  # Set the parent to this puzzle
        return child

    def __eq__(self, other):
        """Governs behavior of ==.

        Overrides == for this object so that we can compare by tile arrangem
        instead of reference.  This is going to be pretty common, so we'll s
        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 improv

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

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

        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 st
        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

        Returns:
            path (list of NumberPuzzle or None) - path from start state to f
            explored - total number of nodes pulled from the priority queue
        """
        # TODO
        open_set = PriorityQueue()
        closed_set = set()
        self.key = self.heuristic(better_h)
        open_set.put((self.key, self))

        while not open_set.empty():
            _, current = open_set.get()
            if current.solved():
                return current.path_to_here(), len(closed_set)

            closed_set.add(current)

            for neighbor in current.legal_moves():
                if neighbor in closed_set:
                    continue

                tentative_g_score = current.dist_from_start + 1

                if tentative_g_score < neighbor.dist_from_start or neighbor
                    neighbor.parent = current
                    neighbor.dist_from_start = tentative_g_score
                    neighbor.key = tentative_g_score + neighbor.heuristic(be

                    open_set.put((neighbor.key, neighbor))

        return None, len(closed_set)

    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

        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
        for i in range(PUZZLE_WIDTH):
            for j in range(PUZZLE_WIDTH):
                # Ignore the blank tile for mismatch count
                if self.tiles[i][j] != 0 and self.tiles[i][j] != i * PUZZLE_
                    mismatch_count += 1
        return mismatch_count

    def manhattan_heuristic(self):
        """Returns total Manhattan (city block) distance from destination ov
        # TODO
        total_manhattan = 0
        for i in range(PUZZLE_WIDTH):
            for j in range(PUZZLE_WIDTH):
                tile = self.tiles[i][j]
                if tile != BLANK:
                    goal_row, goal_col = divmod(tile - 1, PUZZLE_WIDTH)
                    total_manhattan += abs(goal_row - i) + abs(goal_col - j)
        return total_manhattan


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

        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 fir
    """
    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
    """
    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_from_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_from_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.

In [2]:
```
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"""
```

In [3]:
```
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 -
```

In [4]:
```
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 -
```

In [5]:
```
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.

In [6]:
```
%time solve_and_print(sixteen_moves, False)
```

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

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

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

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

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

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

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
13 14 11 15

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

CPU times: user 230 ms, sys: 8.69 ms, total: 239 ms
Wall time: 237 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

The sum of Manhattan distances of all tiles from their final locations, with or without counting the blank as a tile, is an admissible heuristic for the fifteen puzzle. This heuristic is optimistic and never overestimates the actual number of moves required to solve the puzzle because:

Without Counting the Blank: Each tile's Manhattan distance represents the minimum moves needed to reach its correct position, assuming direct movement without obstacles. Since the heuristic sums these minimum distances, it provides a lower bound on the total moves needed, making it admissible.

With Counting the Blank: Including the blank space in the calculation does not change the admissibility. Moving the blank is necessary for puzzle resolution, akin to moving any other tile. The heuristic still underestimates or exactly estimates the moves required, never overestimating.

In both cases, the sum of Manhattan distances is a lower bound on the actual cost to solve the puzzle, hence admissible.

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.

In [7]:
```
%time solve_and_print(sixteen_moves, True)
```

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

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

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

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

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

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

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
13 14 11 15

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

CPU times: user 15.9 ms, sys: 2.58 ms, total: 18.5 ms
Wall time: 16.6 ms
```

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

In [8]:
```
%time solve_and_print(forty_moves, True)
```

```
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
File <timed eval>:1

Cell In[1], line 283, in solve_and_print(puzzle_string, better_h)
    276 """ "Main" - prints series of moves necessary to solve puzzle.
    277 
    278 Args:
    279   puzzle_string (string):  The puzzle to solve.
    280   better_h (boolean):  True if Manhattan distance heuristic, false 
if tile count
    281 """
    282 my_puzzle = read_puzzle_string(puzzle_string)
--> 283 solution_steps, explored = my_puzzle.solve(better_h)
    284 print("{} nodes explored".format(explored))
    285 print_steps(solution_steps)

Cell In[1], line 194, in NumberPuzzle.solve(self, better_h)
    190     continue
    192 tentative_g_score = current.dist_from_start + 1
--> 194 if tentative_g_score < neighbor.dist_from_start or neighbor not in 
[item[1] for item in open_set.queue]:
    195     neighbor.parent = current
    196     neighbor.dist_from_start = tentative_g_score

Cell In[1], line 99, in NumberPuzzle.__eq__(self, other)
     93 def __eq__(self, other):
     94     """Governs behavior of ==.
     95 
     96     Overrides == for this object so that we can compare by tile arr
angement
     97     instead of reference.  This is going to be pretty common, so w
e'll skip
     98     a type check on "other" for a modest speed increase"""
---> 99     return np.array_equal(self.tiles, other.tiles)

File <__array_function__ internals>:180, in array_equal(*args, **kwargs)

File /Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11/site
-packages/numpy/core/numeric.py:2403, in _array_equal_dispatcher(a1, a2, eq
ual_nan)
   2398             cond[both_nan] = both_nan[both_nan]
   2400         return cond[()]  # Flatten 0d arrays to scalars
-> 2403 def _array_equal_dispatcher(a1, a2, equal_nan=None):
   2404     return (a1, a2)
   2407 @array_function_dispatch(_array_equal_dispatcher)
   2408 def array_equal(a1, a2, equal_nan=False):

KeyboardInterrupt: 
```

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 Euclidean distance heuristic runs faster than the tile displacement heuristic but slower than the Manhattan distance heuristic due to its level of estimation accuracy regarding the actual moves needed in the fifteen puzzle. The Manhattan distance directly correlates with the puzzle's movement rules, accurately reflecting the minimum moves needed since only vertical or horizontal movements are allowed. In contrast, the Euclidean distance, which measures the straight-line distance, implies the possibility of diagonal movement. This makes it a less accurate estimate for the puzzle because it slightly underestimates the true cost compared to Manhattan distance, leading to a broader exploration of the state space. Thus, while Euclidean distance provides a better estimate than simply counting misplaced tiles, it doesn't match the efficiency of the Manhattan distance, which more tightly conforms to the puzzle's constraints.

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

Yes, the heuristic that calculates only the number of rows away for each tile, ignoring columns, would still return an optimal solution every time, though it may not do so as efficiently as the correct Manhattan distance. This is because the heuristic remains admissible—it never overestimates the actual minimum cost to reach the goal. Since it considers only the row differences, it underestimates or exactly estimates the distance a tile needs to travel, without ever suggesting a lower-than-actual cost to get all tiles to their target positions.

Admissibility ensures that A* search will find an optimal solution because the search algorithm will continue to explore paths until it confirms that the shortest path to the goal has been found. However, because this heuristic provides a looser estimate of the true cost (by ignoring column distances), it might lead to a broader search space than necessary, as it does not guide the search as effectively towards the goal state as the full Manhattan distance would. The search might explore more states than it would with a tighter heuristic because it does not fully account for the actual costs associated with moving tiles horizontally.

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

With the rule change allowing multiple tiles to slide into the blank space in one move, both the tiles displaced and Manhattan distance heuristics become inadmissible. This is because both heuristics can overestimate the number of moves required to solve the puzzle given the new movement capabilities. They do not account for the possibility of moving several tiles closer to their target positions simultaneously, thus violating the admissibility criterion that a heuristic must never overestimate the cost to reach the goal.

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.