```
# class Anagram:
#     num_iterations = 0

#     def anagram_expand(self, state, goal):
#         node_list = []

#         for pos in range(1, len(state)):  # Create each possible state that can be created from the current one in a single step
#             new_state = state[1:pos + 1] + state[0] + state[pos + 1:]

#             # TO DO: c. Very simple h' function - please improve!
#             if new_state == goal:
#                 score = 0
#             else:
#                 score = 1

#             node_list.append((new_state, score))

#         return node_list



class Anagram:
    num_iterations = 0

    def anagram_expand(self, state, goal):
        node_list = []

        for pos in range(1, len(state)):
            new_state = state[1:pos + 1] + state[0] + state[pos + 1:]

            # Calculate the h-score by counting the number of misplaced letters
            h_score = sum(1 for a, b in zip(new_state, goal) if a != b)

            node_list.append((new_state, h_score))

        return node_list

    # Rest of your code...


    # TO DO: b. Return either the solution as a list of states from start to goal or [] if there is no solution.
    def a_star(self, start, goal, expand):
        open_list = [(start, 0)]  # Priority queue with the initial state and g-score

        g_scores = {start: 0}  # g-scores for all states
        f_scores = {start: 0}  # f-scores for all states

        came_from = {}  # Dictionary to store the previous state for each state

        while open_list:
            current, g_score = open_list.pop(0)  # Get the state with the lowest f-score

            if current == goal:
                path = [current]
                while current in came_from:
                    current = came_from[current]
                    path.append(current)
                path.reverse()
                return path

            for neighbor, h_score in expand(current, goal):
                tentative_g_score = g_score + 1  # Assuming a cost of 1 for each step

                if neighbor not in g_scores or tentative_g_score < g_scores[neighbor]:
                    came_from[neighbor] = current
                    g_scores[neighbor] = tentative_g_score
                    f_scores[neighbor] = tentative_g_score + h_score
                    open_list.append((neighbor, f_scores[neighbor]))

            open_list.sort(key=lambda x: x[1])  # Sort the open_list by f-score
            self.num_iterations += 1

        return []

    # Finds a solution, i.e., the set of steps from one word to its anagram
    def solve(self, start, goal):
        self.num_iterations = 0

        # TO DO: a. Add code below to check in advance whether the problem is solvable
        if sorted(start) != sorted(goal):
            print('This is impossible to solve')
            return "IMPOSSIBLE"

        self.solution = self.a_star(start, goal, self.anagram_expand)

        if not self.solution:
            print('No solution found')
            return "NONE"

        print(str(len(self.solution) - 1) + ' steps from start to goal:')

        for step in self.solution:
            print(step)

        print(str(self.num_iterations) + ' A* iterations were performed to find this solution.')

        return str(self.num_iterations)


if __name__ == '__main__':
    anagram = Anagram()
    anagram.solve('TEARDROP', 'PREDATOR')
```