# (1) Program Flow Overview

This section provides a written description of the program structure to give a high level understanding of how this program has been designed. It has been split into three sections: Managing Input and Creating Initial Configuration, Managing Core Logic of Min Conflicts Function and Managing Output and Re-Try Logic. In the designing this program, many helper functions were used to ensure that the code is readable and maintainable. This section highlights the decision decision to use modular code by mentioning how helper functions have been used within the program. Note that in this section the italicized words correspond to function names in the program.

## Managing Input and Creating Initial Configuration

The program begins by calling the *manage_input* function. The *manage_input* function is responsible for reading input from the file 'nqueens.txt'. It then creates a scrambled list for each number in the input file as the initial configuration of queens on the board. The scrambled list is created by calling the *create_configurations* function. The *create_configurations* function first creates an array of the size number that is passed into the function and then calls the *create_scrambled_list* function. The *create_scrambled_list* function is responsible for creating the list where each queen is placed in it's own row and column. The scrambled list is then used to call the *run_nqueens* function.

## Managing Core Logic of Min Conflicts Function

The *run_nqueens* function calls the *min_conflicts* function and evaluates the result of this function call to check if it is None. The *min_conflicts* function contains the core logic for sequentially moving the queens until a configuration is found where the sum of the conflict values are zero. This function uses the min conflict heuristic to calculate conflicts. The min_conflicts function has two helper functions called *find_conflicts* and *calc_conflicts* which assist the *min_conflict* function in calculating conflicts. The *min_conflicts* function uses the helper function *find_conflicts* to calculate positions with many conflicts by calling *determine_max_conflicts* and randomly selects a single value from this list. It then uses the helper function *calc_conflicts* to call the *determine_min_conflicts* and randomly selects a single value from this list. The list of solutions is then updated to this value that minimizes the conflicts.

This process is repeated in a for loop unless it stops early due to reaching a solution where the sum of the conflict values of the placement of queens is zero. In the event that the *min_conflicts* function is unsuccessful in reaching a solution within the number of max steps, it returns None.

### Managing Output and Re-Try Logic

If the result of the call to *min_conflicts* is successful within the function *run_nqueens*, then *run_nqueens* will call the function *convert_results_list* to convert the results into the correct output format. The result of *run_nqueens* is passed into the function *manage_nqueens.* The function *manage_nqueens* is responsible for determining whether the *run_nqueens* function was successful by evaluating the result of *run_nqueens* to see if it is None. In the case that it is None, it means that *run_nqueens* was unsuccessful. The function *manage_nqueens* will then re-run the *run_nqueens* function with a new configuration initial configuration of the queens. It will continue to re-run this process of resetting the initial configuration and re-running *run_nqueens* until a solution is found and the number of tries is less than 100. After the number of tries reaches the point where it is greater than 100, the program stops attempting and returns None. In the case that *manage_nqueens* is successful, in other words it receives a valid list of results as a parameter, it then calls the *manage_output* function. The manage_output function writes the results to the file ‘nqueens_out.txt’.

## (2) Algorithm Overview

This section discusses the min conflict heuristic algorithm as well as the benefits of using a min conflict heuristic to solve the nqueens problem over other methods. This project uses an iterative repair algorithm to solve the n-queens problem. It does so by initially placing each queen in it’s own column and row. For each queen, the number of conflicts with the other queens are then counted. Based on the number of conflicts, a min conflicts heuristic is used to improve the placement of the queens on the board so that no queens conflicts with another vertically, horizontally, or diagonally. Since the min conflict method may get stuck at local minima, the program is designed to restart the program with a new initial placement of the queens to find a solution.

To conclude, the benefit of using a min conflicts heuristic to solve the nqueens problem is it’s effectiveness. It requires fewer steps of iteration in order to find a solution than other

approaches. An additional benefit of this solution is that it can solve the nqueens problem in nearly constant time with high probability. However, since it is a local search method, it may get stuck at local optima. Thus, the program must account for this by including retry logic.

## (3) Algorithm Outline

This section gives an overview of how the conceptual algorithm was translated into code in the program. The following outlines the official min conflict algorithm step by step:

```
function min_conflicts(csp, max_steps):
(A)     current <— an initial complete assignment for csp
(B)     for i = 1 to max_steps do:
                if current is a solution for csp then return current
                var <— a randomly chosen, conflicted variable form variables[csp]
                value <— the value v for var that minimizes conflicts(var, v, current, csp)
                set var = value in current
        return failure
```

Based on the official algorithm above that was supplied with the assignment, I have implemented the algorithm using two steps. **Part A** is responsible for creating the initial configuration of queens where each queen is placed in it's own row and column. **Part B** contains the core logic of the min_conflicts function.

**Part (A): Create Initial Configuration of Queens**

```
def create_scrambled_list(initial_list):
    for i in range(0, len(initial_list) - 1):
        rand_num = random.randint(0, len(initial_list) - 1)
        swap_row = initial_list[i]
        initial_list[i] = initial_list[rand_num]
        initial_list[rand_num] = swap_row
    return initial_list
```

**Part (B): Core Logic Min Conflicts**

```
def min_conflicts(solution, numb, max_steps=100):
    for i in range(max_steps):
        conflicts = find_conflicts(solution, numb)
        if sum(conflicts) == 0:
            return solution
        col_list = determine_max_conflicts(conflicts, numb)
        col = random_position(col_list)
        conflicts_list = [calc_conflicts(solution, numb, col, row) for row in range(numb)]
        min_conflicts_list = determine_min_conflicts(conflicts_list, numb)
        solution[col] = random_position(min_conflicts_list)
    return None
```