Solving the N Queens Problem using Min-Conflict

Rachel Conforti
Computer Science Department
University of North Carolina at Charlotte
Charlotte, United States of America
rconfort@uncc.edu

***Abstract*— Min conflict is a local search algorithm that can be used for n queens. This algorithm was also used for the Hubble Space Telescope to reduce the time taken to schedule a week of observation from three weeks to around 10 mins. It is one of many local search algorithms however min-conflicts is an effective way to solve constraint satisfaction problems (CSP). The assignment is to solve the n queens problem using a local search algorithm. I will overview the n queen problem, the min-conflict algorithm, my code to accomplish this, and my results.**

Keywords— Min-conflict algorithm, n queens, constraint satisfaction problems (CSP), local search algorithm

I. Introduction

The n queens and the min-conflict algorithm are a classic combination. There are many ways to solve the n queen problem such as using simulated annealing, random-restarting hill climbing, tree-CSP-solver, or the min-conflict algorithm. The goal of this project is to learn about that n queens problem, the min-conflict algorithm, and produce a practical code solution of the algorithm solving the n queens problem. In an attempt to prevent similar mistakes taken in the previous project, I tried to minimize the amount of variables used and use more efficient data structures. This was an interesting project to see how quick and efficient this algorithm can be.

II. Overview of the 8 queens and N Queens Problem

In this project we are solving the n queens problem which is a step above the eight queen problem. The eight queen problem consists of placing eight queens on a eight by eight board. The goal of the problem is to produce a solution that has no two queens in attacking range of one another. What it means to have a queen in attacking range is if two queens are in the same row, column, or diagonal. An example is shown below with the red lines showing conflict:

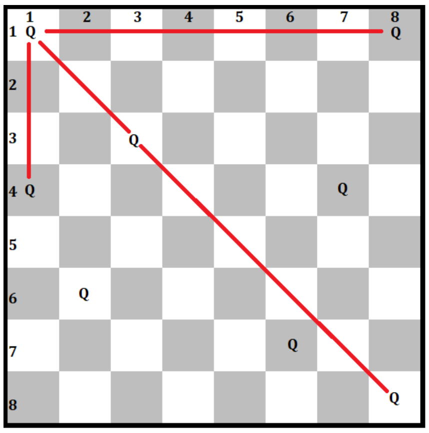


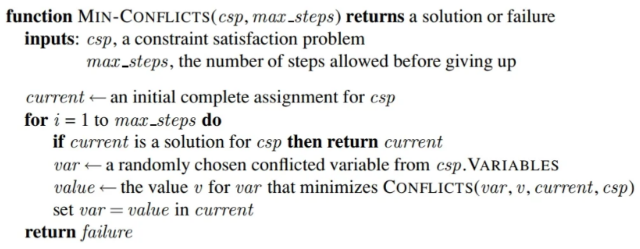
Figure 1: Example a queens in conflict

As it is shown above the queen is in conflict at location (1,1) with the queen at (4, 1) because of column conflicts. It also conflicts with another queen at (1, 8) due to row conflicts. Lastly, it conflicts with queens (3, 3) and (8, 8) due to diagonal conflicts. For the eight-queen problem there are 92 unique solutions for the board.

For this project, we are solving the n queens problem which is the eight queen problem however we can ask the program to solve any number (n) of queens on a (n) sized board. In this project our initial n sized board will be randomized with all eight queens in random order and will try to place the all eight queens in a way that none of them are in an attacking position using the min-conflict algorithm.

III. Overview of the Min-conflict algorithm

The beginning of the min-conflict algorithm began in the 1990s with Mark Johnston of the Space Telescope Science Institute's idea to help create a schedule for the Hubble Space Telescope. Steven Minton and Andy Philips then analyzed Mr. Johnston's work and split it into two different areas: greedy algorithm and min conflict. Below is the basic pseudocode for the min-conflicts algorithm:



```
function MIN-CONFLICTS(csp, max_steps) returns a solution or failure
  inputs: csp, a constraint satisfaction problem
          max_steps, the number of steps allowed before giving up

  current ← an initial complete assignment for csp
  for i = 1 to max_steps do
    if current is a solution for csp then return current
    var ← a randomly chosen conflicted variable from csp.VARIABLES
    value ← the value v for var that minimizes CONFLICTS(var, v, current, csp)
    set var = value in current
  return failure
```

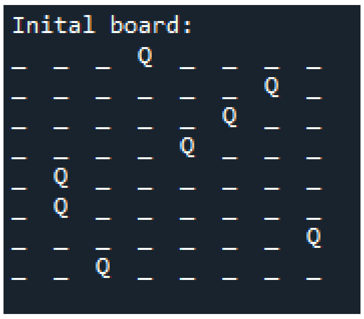
The way the min-conflict algorithm works is it will randomly select a column from the board to move. This selection will calculate the number of conflicts that will result because of this new move. Once the algorithm has calculated the conflicts for every possible move it will choose the outcome with the smallest amount of conflicts. If there is a tie between options for the minimum amount of conflicts, a new move will be chosen at random. And if the new option has greater conflicts than the current position it will not make that move. The algorithm will continue to iterate through finding the minimum conflict, changing the board to that position, and then finding the new minimum conflict for the new move, over and over and over again until it either finds the solution or it hits the max steps resulting in the algorithm giving up.

IV. Code Overview

In this section I will break down how my code is working. I broke my code into sections that I thought would make it easier to understand my logic.

A. *N - Queen Class Initialization*

The way I decided to handle this project was to create everything within the NQueens class. Inside of the object initialization, it will take in the desired number of queens, automatically create a board of n * n size and randomly place the queens on the board. An example of this from my code printout is below:



```
Inital board:
_ _ _ Q _ _ _ _
_ _ _ _ _ _ Q _
_ _ _ _ _ Q _ _
_ _ _ _ Q _ _ _
_ Q _ _ _ _ _ _
_ Q _ _ _ _ _ _
_ _ _ _ _ _ _ Q
_ _ Q _ _ _ _ _
```

Figure 3: Initial randomized board

The passed in size is saved to self.size and creates an empty board called self.board. Using two for loops for the length of n it will add '_' to indicate empty spaces and 'Q"s for queens. This creates a two-dimensional list of a populated board.

B. *Board Methods*

I utilized two board methods: one to print the boards and one to swap the boards.

The first being my print_board() method takes in two parameters: self and option. The option is to indicate if it should print "Initial Board" or "Final Board". It will loop through self.size and print out the values at each location. This method is utilized in the initialization of the NQueen object to print the initial board and is used at the end of the min-conflict method to print the final board with the title "Final board" above it.

The second method that I use is a simplistic one called swap_positions() it is used within the min-conflict method. The purpose of this method is after we have found a better solution it will swap the random conflict with the better option. This method takes in the following five parameters: self and four integer variables. The first two integer variables represent the x and y values of a random conflict that the algorithm picked to solve at random. The last two integer values represent the x and y values for the minimum conflict solution. This algorithm will replace the first integer values with '_' which indicates an empty space and the last two integer values get replaced with a 'Q' effectively swapping their two locations.

C. *Number of Conflicts Method*

This method is responsible for figuring out how many conflicts are occurring with the current queen. This method is called num_of_conflicts() which takes in two parameters: self and an integer parameter. The integer parameter is going to be in the format of (x, x) which is then saved to separate variables. I have a directions variable that is a list that holds the following: [(1,1), (1, -1), (-1, 1), (-1, -1), (1,0), (-1, 0), (0, -1), (0,1)]. This is representing the multiple directions (row, column, or diagonal) in which there could be a conflict. The first four objects in the list (1,1), (1, -1), (-1, 1), (-1, -1), represent the diagonal check. The next two in the list, (1,0), (-1, 0) are checking the column for conflicts. Lastly, (0, -1), (0,1) is checking the row for queen conflicts.

For each object in the direction list, the method will take the passed inboard position and add the current object in the direction list to move it in either an up, down, or diagonal direction. First, it will check if this new move is within the bounds of the board. If it is not within the bounds of the board it will break and continue to the next item in the list. If it is a valid move on the board it will then check if there is a queen ('Q') at that position. If there is a queen then it is a conflict and it will add one to the total number of conflicts. If there is no queen at this position even if it is a viable move it will continue to the next item.

D. *Min_conflict Method*

The min_conflict method is the portion of the program that everything ties into. This method's job is to solve the board by enacting the min-conflict algorithm. The min_conflict() method takes in two parameters: self and max_step which I have equal to 100000. This is an extreme amount of steps allowed for the minimum of the project which is eight queens however, to leave room for larger boards I left it at this size. The algorithm will continue until it finds a solution or hits the max steps limit. The first thing that this method will do is iterate through the board and call the num_of_conflicts() method to check if there is a queen conflict, if so it will add it to the dictionary called conflicted. The key in the dictionary will be the position of the conflict (x, x) and the value for each location which is hardcoded to be a zero. A zero value is used here because the value has no meaning here, we are just trying to find the location of conflicts as of right now.

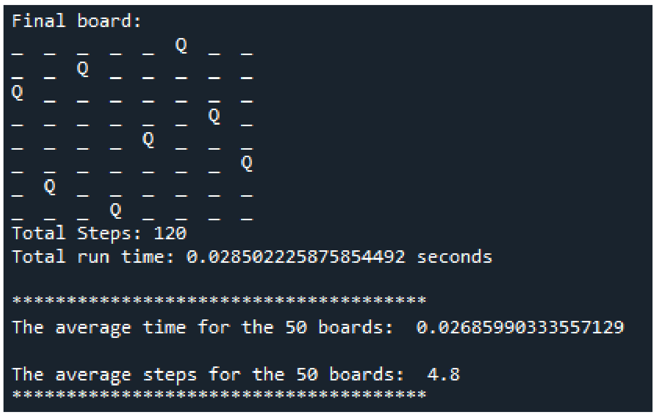
Next the algorithm will check to see if the conflicted dictionary is equal to zero to see if we found a solution. If we did not find the solution, it would randomly choose a conflict in the dictionary using random.choice and save it to var_x and var_y. From here the algorithm will clear the dictionary and find all the conflicts for this newly generated random conflict so we can find the least conflicting movement from this new position. Now that we have a dictionary full of key values of conflict locations and the value being the number of conflicts, we will sort the conflict dictionary from minimum value to maximum value. Doing so allows our next step to quickly find the least amount of conflicts..

From here, each object in the conflicted dictionary will be checked to see if any other key, value pairs have the same dictionary value. If they do match the lowest min-conflict value they will be added to the new_pos list. The purpose of the new_pos list is to hold all the locations of the lowest min-conflict values so we can do a random tie breaker. Another thing that is happening during this loop is creating a chance for a random queen location to be added to the new_pos list to help get out of local minimums. By doing a simple random.random() method call, if this results in a number less than 0.1 (an arbitrary number I randomly picked) that key will be added to the new_pos list. Lastly, to do the random tie breaker it will call the random.choice() method on the new_pos list.

To finish out the method, it will call the swap_positions method() and add one to the step total. Then this continues until either the max steps are reached or the solution is found. Then once it has broken out of the main loop, the method will print out the board and total step count if it found a solution. If no solution has been found it will return “unable to solve the problem is x amount of steps”. The method itself returns the total number of steps so I can calculate the average amount of steps for all my boards called.

E. *Main*

The main method is set up to automatically run 50 boards solving for the 8 queens. For each board, it will produce its initial board, final board, total steps, and run time for each individual board. After all 50 boards have run it will calculate the average run time for 50 amount of boards, the average step cost of 50 amount of boards. An example is below:



```
Final board:
_ _ _ _ _ Q _ _
_ _ Q _ _ _ _ _
Q _ _ _ _ _ _ _
_ _ _ _ _ _ Q _
_ _ _ _ Q _ _ _
_ _ _ _ _ _ _ Q
_ Q _ _ _ _ _ _
_ _ _ Q _ _ _ _
Total Steps: 120
Total run time: 0.028502225875854492 seconds

**************************************
The average time for the 50 boards:  0.02685990333557129

The average steps for the 50 boards:  4.8
**************************************
```

Figure 4: Example printout of a final board and average time and steps for 50 boards for 8 queens

F. *What I would do differently*

Within my code, I believe that there are too many steps being taken to solve the n queens problem. I would try to find a better solution to lower the number of steps required to find the solutions. I believe I could do that by creating a better and more efficient way to search for and find the conflicts with the current queen.

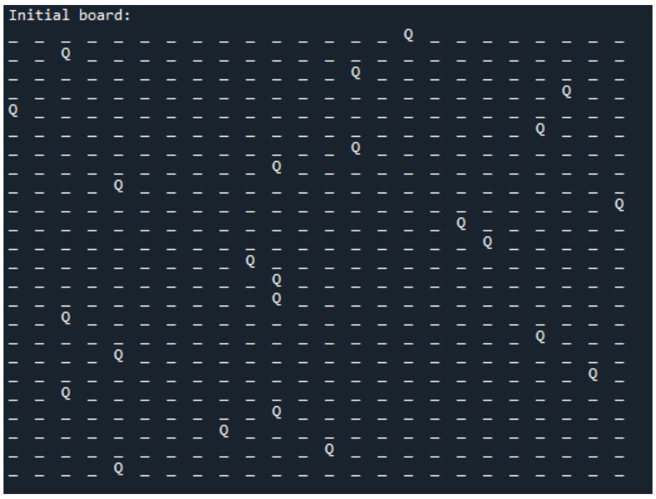
V. Results

The results of my program is as follows:

Program Results Per Board		
Number of queens	**Average time for 50 boards in seconds**	**Average steps for 50 boards**
8	0.02840001106262207	5.36
10	0.11536005020141601	12.84
12	0.3327299499511719	107.12
14	0.6321102476119995	258.08
16	1.747939863204956	13.96
18	4.59649007320404	475.16
20	8.099520001411438	87.04

For my curiosity and to show the program's ability to solve more than just the 8 eights problem I recorded the average steps and run times for various sized boards. As it can be seen above the time steadily increases with each increasingly larger board. The 16 queen board is showing a 60.5% increase in average run time for the boards. This is significant because logically it is double the size, it should run only 50% longer than a board half its size. So, some efficiency issues will surface as the boards get bigger. I stopped at twenty boards for the 50 board printout because it takes a very long time to go through 50 boards to compare the results.

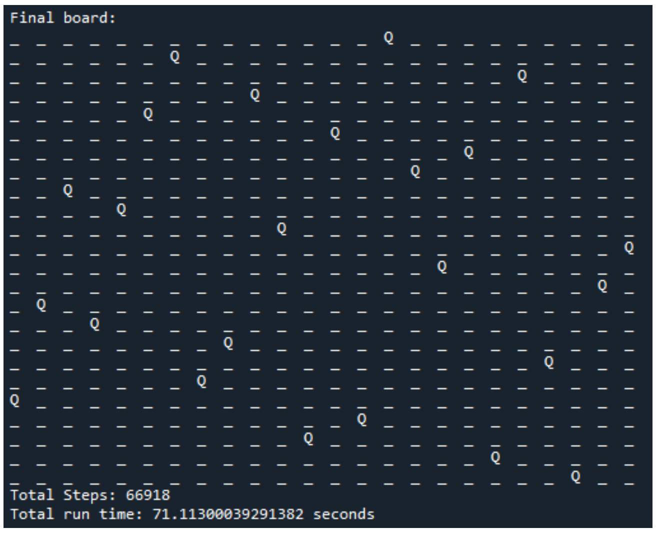
However, for a board of 24 queen size, my program was able to find solutions for the boards. An example printout is below:



```
Initial board:
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ Q _ _ _ _ _ _ _ _
_ _ Q _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ _ _ _ _ _ Q _ _ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ Q _ _
Q _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ Q _ _ _
_ _ _ _ _ _ _ _ _ _ _ _ _ Q _ _ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ _ _ Q _ _ _ _ _ _ _ _ _ _ _ _ _
_ _ _ _ Q _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ Q
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ Q _ _ _ _ _ _
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ Q _ _ _ _ _
_ _ _ _ _ _ _ _ _ Q _ _ _ _ _ _ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ _ _ Q _ _ _ _ _ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ _ _ Q _ _ _ _ _ _ _ _ _ _ _ _ _
_ _ Q _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ Q _ _ _
_ _ _ _ Q _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ Q _
_ _ Q _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ _ _ Q _ _ _ _ _ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ Q _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ _ _ _ _ Q _ _ _ _ _ _ _ _ _ _ _
_ _ _ _ Q _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
```

Figure 5: 24 queen sized board randomized initial board

As it can be seen, there are quite a few queens in multiple conflicts with one another. Below is the final board printout:



```
Final board:
_ _ _ _ _ _ _ _ _ _ _ _ _ _ Q _ _ _ _ _ _ _ _ _
_ _ _ _ _ _ Q _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ Q _ _ _ _
_ _ _ _ _ _ _ _ _ Q _ _ _ _ _ _ _ _ _ _ _ _ _ _
_ _ _ _ _ Q _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ _ _ _ _ Q _ _ _ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ Q _ _ _ _ _ _
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ Q _ _ _ _ _ _ _ _
_ _ Q _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
_ _ _ _ Q _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ _ _ Q _ _ _ _ _ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ Q
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ Q _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ Q _
_ Q _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
_ _ _ Q _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ Q _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ Q _ _ _
_ _ _ _ _ _ _ Q _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
Q _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ _ _ _ _ _ Q _ _ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ _ _ _ Q _ _ _ _ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ Q _ _ _ _ _
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ Q _ _
Total Steps: 66918
Total run time: 71.11300039291382 seconds
```

Figure 6: Final solution of a 24 queen-sized board

This board as it can be seen takes significantly longer than the 8 queen board. I am proud that it can solve multiple-sized boards quickly. 24 boards were the first time the program was unable to solve the board within the max steps allowed. This is fine because 50 boards is a lot to run at once for larger boards so if you were to only run one 24 queen board you will only have to wait, on average, one minute for a board to be solved.

VI. Conclusion

This assignment was a very interesting project. I have never coded this algorithm before and it was interesting to see how simplistic yet effective and efficient this algorithm is. The n queens have an interesting balance of trying to figure out an efficient program that also is strict on the conflict rules. Because even if it is very fast but there are still conflicts, the fast solution is useless. However, if the solution takes 10 minutes to solve one board that is also a useless algorithm. So, finding a middle ground that was efficient and still produced no conflicts on the board was a fun puzzle to figure out.