Vishnu Chittari

CSC 173

N Queens Scheme Project

Synopsis

Backtracking

Backtracking is a greedy algorithm to solve the N-Queens problem. We start of the problem by initializing an empty board at the first and then it gets filled with queens column by column by column. The queens are placed in a row that does not have conflicts with the filled out part of the board. If there is not a single row in a current column that doesn't cause conflicts, the algorithm will backtrack to a previous column and will try to select a different queen placement. This process will keep going until there is no such empty column.

Min-Conflict

Another approach to solving the N-Queens problem was the Min-Conflicts one. This is much more efficient than that of the Backtracking version. Basically Min-Conflict works by placing random queens all over the "n" chessboard and then finds a random column with a conflict and replaces it in a row with the minimum conflicts for such said column. I had my idea from CB's pseudocode and it helped me gain some much needed inspiration to solve the problem.

Description of the Function:

Backtracking

This is a greedy algorithm to solve the N Queens problem that works by initializing an empty board and later gets filled with queens column by column. My backtracking solution uses implicit recursion to find the solution to N-Queens. I used implicit recursion because my ideal solution wasn't working properly with the explicit resolution as there were some confusing stack errors, which caused some illogical solutions to come up for certain values of N such as 7 and 9. The Implicit code works by iterating throughout the board possibilities by placing each queen one at a time and moving them when there is a conflict (no solution). If we can place a queen in a spot without a conflict you can call the function on the next column at row 0 in an if statement, and if that call returns true and you return true and it is done. Also, if you can place a queen and you are on the last column you are done, and this is how the true "bubbles" up to the top of the stack. This is implicit recursion since the method calls move back and forth based on recursive iteration, (F(x+1)) which is declared true once the next call is made till the next made call is the last call which returns true at the base case.

In addition to the backtracking algorithm I also used true and false values to provide two forms of the answer for the question. The truth value prints out the board, vectors, and the number of steps. The false value gives us the vector and the number of steps.

Backtraking method uses implicit recursion to iterate back and forth behind the scenes until the #t is carried to the top of the recursion stack.

Place_Q function tries to place a queen based on parameters if there is a conflict it removes the queen and returns false, otherwise it returns true.

Sample Output

True:

```
> (nq-bt #t 14)
"Backtracker:"
1 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 1 0 0 0
0 1 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 1 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 1 0 0
0 0 0 1 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 0 0 1 0 0 0 0
0 0 0 0 0 1 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 1
0 0 0 0 1 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 1 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 1 0 0 0 0 0
Queens:#(0 2 4 6 11 9 12 3 13 8 1 5 7 10)
Steps:28380
```

False:

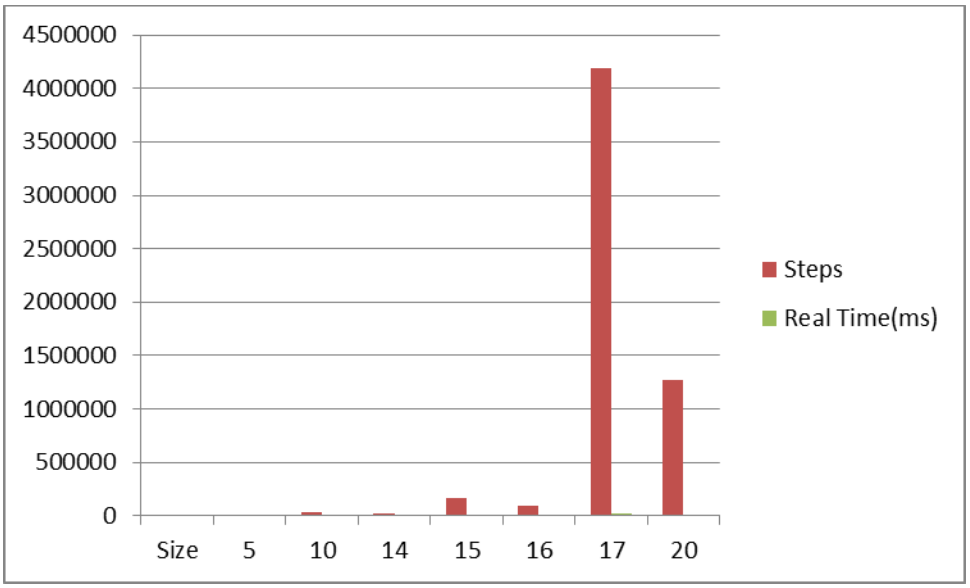
```
Backtracker:
Queens:#(0 2 4 6 11 9 12 3 13 8 1 5 7 10)
Steps:28380
```

Time Taken and The Number of Queen Placements

Backtracking (Boolean Truth Value)

Size	Steps	Real Time(ms)
5	15	2.000976563
10	1067	6.002929688
14	28380	100.0664063
15	21624	116.0771484
16	170748	725.2788086

17	96579	422.2817383
20	4192125	21724.22998



In this chart we compare two different variables (Steps and Real Time) vs Size of the board. We can see that some of the boards take longer than that of the other even if they are bigger than the other board sizes. I think this is mainly due to the fact that some board sizes require more backtracking than the other board sizes. For example when you do the board size of 5 it is done rather instantaneously compared to 10 because it doesn't have to backtrack at all.

Description of the Functions

Min-Conflicts / Hill Climbing Algorithm

My hill-climbing algorithm is pretty simple. I place a queen on each column on a random row; store all the conflicts in a parallel conflict vector. Conf [0] represents the conflicts for the queen in the 0 column). My algorithm first finds the highest value in the conflict vector. If that value is 0, then you have found your solution and you're done. Otherwise, we now loop through the rows of the column that have had the highest conflict value (excluding the row that this column's queen was on at the start of this algorithm iteration). The queen in that column is then moved to the row and the algorithm is called again. This solution works really well at given times, but can fall into infinite loops. I first used the random function to put have a random board of queens to work out my solution, and this in the end did not work properly. So I ended up using another function called initialize which helped me run through the algorithm well. Another thing I noticed was the infinite loop issue. The problem with the infinite loop occurs when 2 or more moves are repeated in a circle, being considered the "best" move by the algorithm each time. So, on an 8x8 board it is possible to have a 3 move repeating pattern that the algorithm cannot break of. I tried to fix this issue the best I possibly could by writing a new initialize function, which I

think should solve the problem, but if it doesn't then the program might need to be stopped and run again. I also used the same Boolean true and false values as I used above in the backtracking. True returns the board, vectors and steps and false returns and false returns the vector list and the number of steps.

The MinConf method finds the column with the highest conflict loops through its rows and finds the row with the least conflicts (not including the row with the least conflicts). If the highest conflict is a 0 then you have a solution.

Get-Conf takes in the vector and a col and returns the number of conflicts.

SetConf is the helper method to quickly reset all the conflicts on the board after moving a queen

FindMax finds the max value in a vector and it is used to find the highest conflict in my Conf vector.

Sample Output:

Truth Value:

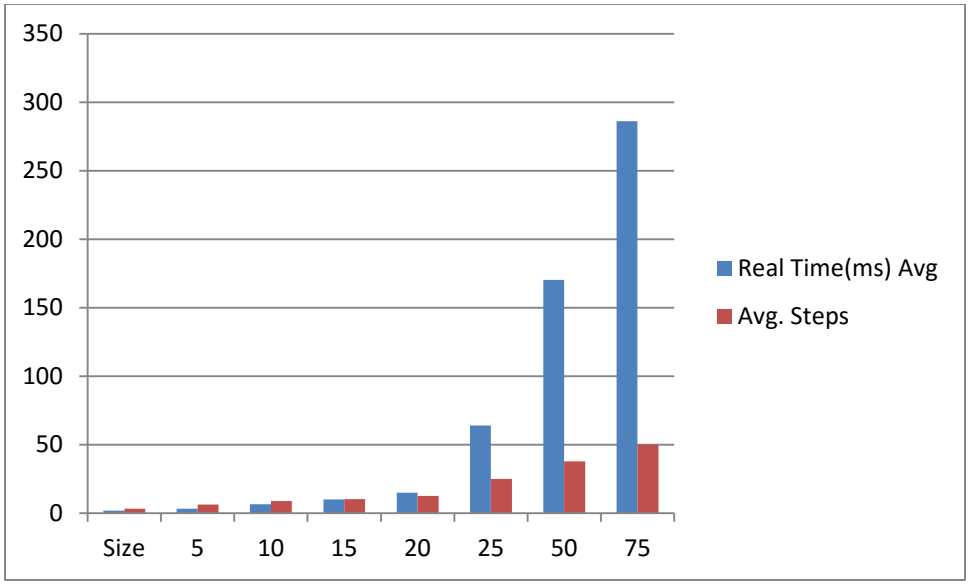
> (nq-mc #t 25)
"Min-Conflict Board:"
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0
0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0
0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0
0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0
0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0
0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0
0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1

0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0
Queens:#(0 2 4 6 8 10 12 14 16 18 20 22 24 1 3 5 7 9 11 13 15 17 19 21 23)
Steps:1
False Value:

> (nq-mc #f 25)
Queens:#(0 2 4 6 8 10 12 14 16 18 20 22 24 1 3 5 7 9 11 13 15 17 19 21 23)
Steps:2

Time Spent and Queen Placements

Size	Real Time(ms) 1st	Real Time (ms) 2nd	Real Time(ms) 3rd	Real Time(ms) Avg
5	2.001220703	2.000976563	2.000244141	2.000813802
10	3.001708984	4.004394531	3.002929688	3.336344401
15	7.004394531	6.003173828	7.004882813	6.670817057
20	11.00952148	9.005615234	10.00634766	10.00716146
25	18.01074219	14.00732422	13.00732422	15.00846354
50	56.03637695	60.0390625	76.04858398	64.04134115
75	164.6032715	172.9143066	173.1135254	170.2103678
100	282.1882324	291.192627	284.7131348	286.0313314



In this chart above we compare 2 variables of size and the time it takes for the algorithm to finish through the queen board. As the number of queens go up so does the amount of time that it takes for the board to complete. I found this to be a reasonable graph to make because I wanted to compare the times between the backtracking algorithm and the minimum conflict algorithm to show, with definitive evidence, which is faster. We also compare the size of the nxn chessboard

and the number of steps it takes for the program to finish the said chessboard. As the size of the chessboard goes up so do the number of steps for the program to finish it. I thought these points would be useful in comparing the efficiency of backtracking vs minimum conflict.

Discussion:

Both of these algorithms are used to solve our N-Queens problem. The backtracking algorithm is slower than that of the Hill Climbing algorithm as provided by the charts above. The time comparison between the backtracking and min-conflicts is seen as such. As we can tell from the charts and the graphs we can tell that Minimum Conflict algorithm takes a lot less time than that of the Backtracking algorithm. The number of steps that the Backtracking takes is a lot more than that of the Minimum Conflict Algorithm ones, and this is mostly due to the nature of how such the algorithm works. I couldn't get the steps to work correctly for the Minimum Conflict algorithm but based on the timings it reasonable to assume that the number of steps is less than that of the Backtracking algorithm.