WRITE UP
Zhiming Zeng (Kate)
CSC 173 Scheme Week 3 - 4

1. OVERVIEW OF PROJECT

I made three programs to solve the N-Queen problem, that is, place N queens into a N X N board such that no queens clash with others by row, column and diagonal. The two mains methods I used are: backtracking and minimum conflict. I made several variations of minimum conflict by altering the parameters for column selection, chessboard configuration and calculating minimum conflict.

I will first discuss the bonus, followed by a review of my definition, design and implementation and a discussion of the results (alternatives I have tested and the ones which are more efficient).

2. BONUS

- I tried various configurations before finally arriving at the final three programs. I will discuss them more in the results section.
- One of my programs (nq-mc-config02) found the **solution with 0 step!** (for problem size of 5^k I believe)

3. DEFINITIONS

Step

1. In **nq-bt**, a step is defined for each assignment of the queen on a column (be it safe or unsafe) in “place_queen”. When none of the rows are safe for that column, backtracking is called and backtracking will call “place_queen” to put the previous queen on the next row. Hence, backtracking takes one step.
2. In **nq-mc** (all versions), a step occurs every time a queen moves to the row with the minimum conflict.

3. DESIGN AND IMPLEMENTATION

3. 1 Backtracking

Brief description of backtracking

- Program starts with an empty board and places queens on a safe row, backing up (thus taking them off the board) when there are no more safe rows in a column to put the next queen.

Queen Position Representation: List

- I chose to represent the queen positions with a list. The ith element represents the ith column (from 1 to N) and the number of the element represents the row number (from 1 to N). The list contains the current set of queens with legal configurations. An empty list represents an empty board

There are a few reasons I chose the list form:

- If I do a vector, I will need to keep track of which columns are legally assigned using iteration, which is messier to implement on scheme.
- With a list however, I can just recurse the function by called the function on the cdr of the list until the list is null '(). This is much simpler and elegant.
- Accessing each element is actually constant time. When I need to backtrack, I simply call the function on the cdr of the list. Since my columns are selected linearly, all backtracks can be done by a single cdr of the list.

There are limitations though:

- Using a list, it is hard to select unassigned columns in a flexible manner. Currently, I can only select unassigned columns linearly (i.e. from N to 1).
- However, for a simple backtracking, it is sufficient to select unassigned columns linearly. Not the most effective backtracking, but it can derive the solution correctly and represents backtracking at its most basic.

<u>Explanation of functions:</u>

- **full-solution?** checks if the list of legally assigned queens has length N.
- **get-solution** passes the solution list if it is a full-solution to the top of the program. This is where **full-solution?** is called to check if the solution is full. Otherwise, it will suggest a row to place an unassigned queen on the next column, that is, it will indirectly expand the list. It will keep expanding the list via **place-queen** until the length of the list is N.
- **place-queen** checks if the suggested row is safe. If it is, it will expand the list and call **get-solution** to check if it is a full solution. Otherwise, it will increment the suggested row and call itself on that row. This recursion stops when a row is safe or when there is no row to place the queen on. when there is no available safe row to put the queen, **backtracking** is called.
- **safe?** checks if there are violations on a certain row in a certain column. It consists of **clash-row?**, **clash-diag-right-up?**, **and clash-diag-right-down?**. These three functions basically check each element in the existing list for 3*size of list times exactly. Later on, I realized there is a slightly faster way to check the list and I used that method in the min-conflict program.

<u>*3. 2 Minimum-Conflict*</u>:

<u>Brief description of Minimum Conflict</u>

- Program starts with a board with all N queens placed, one in each column. The program then moves the queen to the row with the least number of conflicts in that column.

<u>Queen Position Representation: Vector</u>

- Similar to the list, the ith element represents the ith column (from 1 to N) and the number of the element represents the row number (from 1 to N). Each element starts from index 1 so the length of the vector is N + 1. The 0 index is unused.

- The item in the zero index is not used. This is to make it easier to reference column numbers.

Why I chose vector?

- In minimum conflict, I want to have more flexibility in selecting a column. Vectors allow me to access the ith element in constant time for all elements, unlike lists.
- With a list, it is harder to access columns that are not at the top of the list by constant time.

Selecting row

- Binary heap
    - I chose a binary heap to insert the number of conflict for each row and find the row with least conflict because binary heap has one of the fastest runtime for insert and delete-min among all comparison-based sorting algorithms.

- Breaking ties
    - If there are two rows with the minimum conflict, the program will pick the one that the queen is not already occupying.

Max steps: 30,000

- Interesting observation: Backtracking is able to self-terminate for problem sizes without solutions (e.g. 2, 3). However, for minimum-conflict, it is necessary to set a maximum amount of steps, as the program does not test all possible solutions, unlike backtracking.

Variations

Starting configuration of chessboard

- "nq-mc-config01": Diagonal e.g. #(0 1 2 3 4 5)
- "nq-mc-config02": Staggered by 2 rows e.g. #(0 2 4 1 3 5)
- "nq-mc-config03": All items occupy row 1 e.g #(0 1 1 1 1 1)

Column selection

- "nq-mc-config01", "nq-mc-config02", "nq-mc-config03": Select columns from N to 1
- "nq-mc-random": Select columns randomly

Calculation of minimum conflict

- "MC-config02-avoidcorner": I observed that in many of the full solutions I have seen for various sizes, it is possible to not have a queen in any of the four corners. So for all four corners, I set the conflict score to be at least a 100 so that the program will avoid putting queens in the corner.

Optimising the safe? function

- Initially, I made the safe? traverse the list three times (with just 1 comparison for each traversal) to check if the other queens occupy the same row for the first traversal, and the diagonals for the other traversals.
- However, I realized I can do the check with just one traversal of the list with 3 or less comparisons for each traversal. I used an OR statement to check if the queen occupied the

same row or diagonals. If the first statement is false, there is no need to check the other two conditions.

3.3 Other functions found in all programs

Tracking the counter

- For all programs, A counter variable is created globally. Every time place-queen is called, **counter-bump** is called to increment the counter variable via set!.
- Whenever the main program is called, the counter is set to 0.

Formatting the solutions

- For all programs, **matrix-format** passes its parameters to **format-helper** to print out the list or vector into a matrix form.

4.4 DISCUSSION OF RESULTS

4.4.1 Maximum possible problem size N within a minute on various problem size

	Backtrack	MC-config01	MC-config02	MC-config03	MC-config02-avoidcorner	MC-config02-random
Max. N possible (within 1 min)	26	100	3125*	70	3125	100
Time taken (s)	21.9	1.8	12.4	0.9	24.3	42.9
Steps	10339849	269	0	268	2	281

**I believe MC-config02 will take 0 step for any boardsize of N = 5^k. Much of the time is spent checking if all queens are safe and printing out the solution. For problem size 500 (not the power of 5), it takes 260 steps and roughly 41.7 seconds.*

The minimum conflict strategy which sets the chessboard with queens on the (i + 2)th row gives the best result, both in terms of number of steps and time taken to run the program. Even if the initial configuration is not the perfect solution, it can still find the solution of problem size 500 under a minute. For the others, going beyond the max possible N will exceed 1 minute.

4.4.2 Mean, Median, Min, Max, and Standard Deviation on Various Problem Size

Backtrack

Prob. Size	Val 1	Val 2	Val 3	Mean	Median	Min	Max	Standard deviation
5	15	15	15	15	15	15	15	0
10	975	975	975	975	975	975	975	0
15	20,280	20,280	20,280	20,280	20,280	20,280	20,280	0
20	3,992,510	3,992,510	3,992,510	3,992,510	3,992,510	3,992,510	3,992,510	0
25	1,216,775	1,216,775	1,216,775	1,216,775	1,216,775	1,216,775	1,216,775	0

The number of steps increases exponentially for backtracking. Since I select the order in a fixed way from N to 1, and I test the row linearly from 1 to N, it is not surprising that the same function call on the same problem size gives the same result and number of steps every time, thus giving a standard deviation of 0.

Min-conflict-config01: Diagonal

Prob. Size	Val 1	Val 2	Val 3	Mean	Median	Min	Max	Standard deviation
20	51	51	51	51	51	51	51	0
40	119	119	119	119	119	119	119	0
60	183	183	183	183	183	183	183	0
80	256	256	256	256	256	256	256	0
100	269	269	269	269	269	269	269	0

The number of steps increases somewhat linearly. Same result is given for a fixed parameter. This is a huge improvement from backtracking.

Min-conflict-config02: Stagger by 2

Prob. Size	Val 1	Val 2	Val 3	Mean	Median	Min	Max	Standard deviation
20	63	63	63	63	63	63	63	0
40	21	21	21	21	21	21	21	0
60	48	48	48	48	48	48	48	0
80	50	50	50	50	50	50	50	0
100	56	56	56	56	56	56	56	0
125	0	0	0	0	0	0	0	0
140	82	82	82	82	82	82	82	0
155	0	0	0	0	0	0	0	0
180	100	100	100	100	100	100	100	0
500	260	260	260	260	260	260	260	0

The best strategy among all strategies I have! The number of steps generally increases Iin a non-regular manner.

Min-conflict-config02-avoidcorners

Prob. Size	Val 1	Val 2	Val 3	Mean	Median	Min	Max	Standard deviation
20	63	63	63	63	63	63	63	0
40	21	21	21	21	21	21	21	0
60	48	48	48	48	48	48	48	0
80	50	50	50	50	50	50	50	0
100	56	56	56	56	56	56	56	0
125	2	2	2	2	2	2	2	0

This shows that prioritizing non-corner cells does not make the program faster. It gives the same result as the strategy before, and it performs slightly worse on problem sizes that take 0 steps for the previous program to work.

Min-conflict-config03: Same row 1

Prob. Size	Val 1	Val 2	Val 3	Mean	Median	Min	Max	Standard deviation
20	59	59	59	59	59	59	59	0
40	126	126	126	126	126	126	126	0
60	179	179	179	179	179	179	179	0

This is one of the worse min-conflict algorithms. The number of steps increases somewhat linearly. However, it takes a long time to compute problem size above 60.

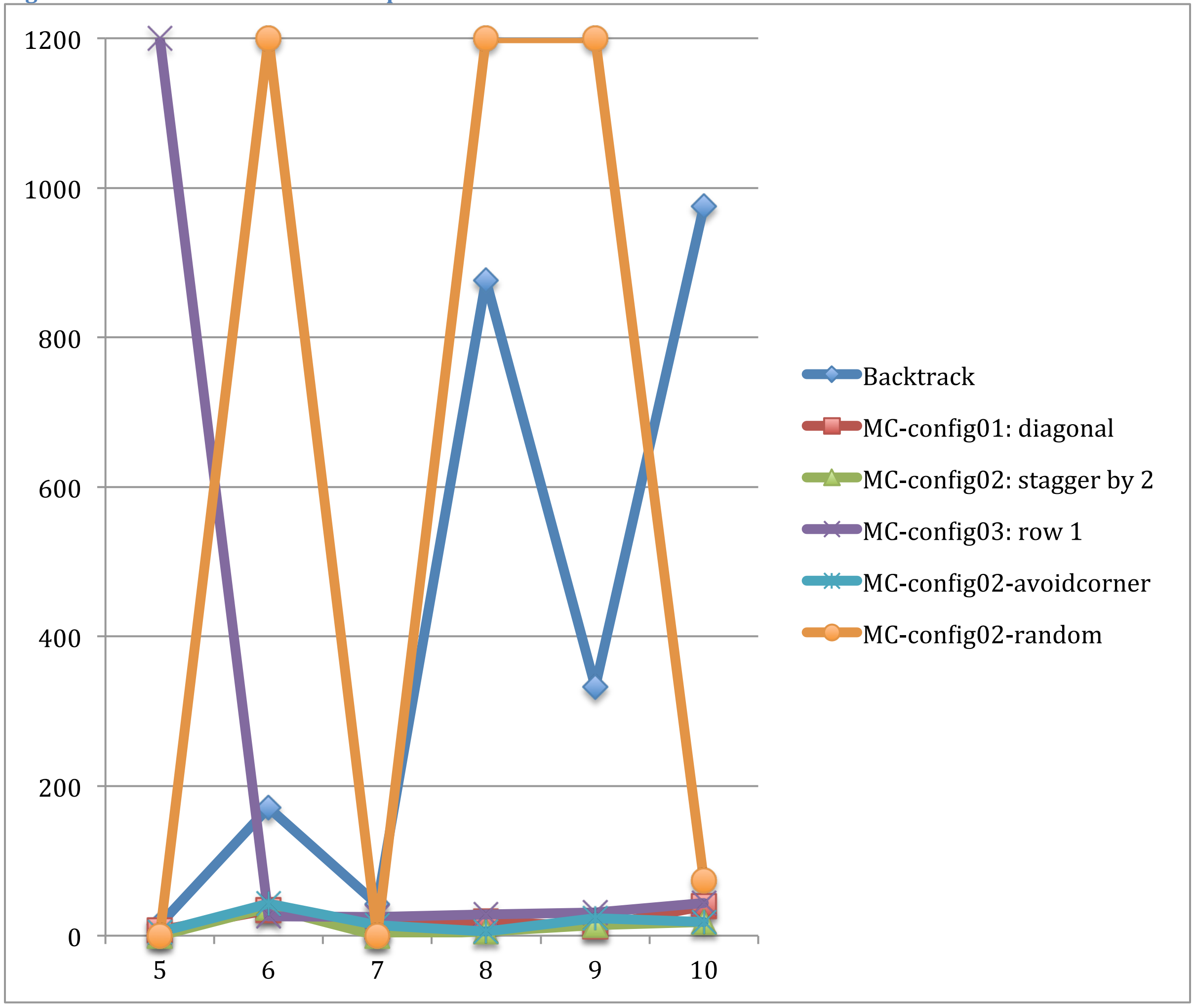
Min-conflict-config02-random: Random

Prob. Size	Val 1	Val 2	Val 3	Mean	Median	Min	Max	Standard deviation
20	37	<30,000	26	10021	37	26	<30,000	17302
40	313	66	214	198	214	66	313	124
60	91	? (<1 min)	139	?	?	?	?	?

Obviously, the randomized strategy gives a huge range of steps. It is either slightly more effective than the non-randomized version, or greatly worse than it.

<u>4.4.3 Graph of Number of Steps Taken as Problem Size Increases for Each Strategy</u>

Figure 1: Problem size vs. No. of steps



Note: 1200 means no solution found within a maximum step of 30,000.

The solutions with smaller variance in the given range of problem size are: MC-diagonal, MC-config02, MC-config02-avoidcorner. They are also more effective even as problem size increases.

Dataset for Figure 1

Problem Size	Backtrack	MC-config01	MC-config02	MC-config03	MC-config02-avoidcorner	MC-config02-random
5	15	7	0	1200	5	0
6	171	34	38	26	43	1200
7	42	12	0	25	14	0
8	876	19	6	28	6	1200
9	333	12	14	31	23	1200
10	975	39	18	44	18	74