CS 2400 – Fall 2013
Assignment 4
Local Search for the N-Queens Problem
Due: Wednesday, 30 October 2013

I have decided to make this assignment an all-programming assignment in which you test various ways of using local search to solve the N-queens problem. I would suggest working in pairs for this assignment. Groups of three are also okay.

Requirements

Programming

Implement the local search algorithm that I described in class for solving the N-queens problem. Use the MIN-CONFLICTS heuristic. I have attached two pages from the text (pp. 220-21) that include pseudocode. Then test the claim that this approach can solve N-queens problems in nearly constant time. Satisfy yourself that your program is working for at least up to 8-queens. To help do this (for you *and* for me), your program should be able to print out a solution in some way. It doesn't have to be fancy; something like this would be fine:

```
---------------------------------
|       |       |       |       |
|       |       |  Q3   |       |
|       |       |       |       |
---------------------------------
|       |       |       |       |
|  Q1   |       |       |       |
|       |       |       |       |
---------------------------------
|       |       |       |       |
|       |       |       |  Q4   |
|       |       |       |       |
---------------------------------
|       |       |       |       |
|       |  Q2   |       |       |
|       |       |       |       |
---------------------------------
```

Of course, we wouldn't be using this for anything much bigger than an 8-queens instance.

Then go for it. According to the textbook, the 1,000,000-queens problem can be solved in about 50 steps using the MIN-CONFLICTS heuristic. Let's be generous—use 500 steps as the maximum number of steps in the local search. Find the maximum size problem you can "reliably" solve in 100 steps. Let's say "reliably" means at least 90%

of the time for at least 10 runs. If 500 steps turns out to be insufficient to get you up to 1,000,000, increase the number of steps.

Of course, I don't really expect you to get up to 1,000,000. At some point, well before you hit 1,000,000, you're going to run out of time and/or space. I dont' know how bad the time constraint is going to be, but the maximum size for a 2-d array of booleans in Java (on my machine, and very roughly) is 11,000 by 11,000, but this was without increasing the heap space available to Java. You can do this by using the command line flag `Xmx`, e.g.

```
java -Xmx2g <java-class-file>
```

would increase the heap size to 2 GBytes. A more efficient implementation would help here (maybe BitStrings?), but you don't need to do that to get full credit.

Now try two variations. As I mentioned in class, striking a good balance between greediness/exploitation and randomness/exploration is important in local search. So:

1. Try increasing the greediness. MIN-CONFLICTS picks a random queen to move and then moves it to the square that has the fewest conflicts. Instead of picking randomly, calculate the heuristic for *every* queen, and pick the queen and move that leads to the smallest number of conflicts.
2. Try increasing the randomness. Pick a random queen. Then, with some probability, move the queen to a random square in that queen's column; otherwise, calculate the MIN-CONFLICTS move and make that move. In a similar approach, which was used to solve satisfiability problems, a probability bewteen 0.3 and 0.5 worked well.

Using the best of these three approaches, try the following things:

1. Be smarter about the initial placement of the queens; a very simple approach would be to try to place the next queen so that it is being attacked by as few queens already on the board as possible. But, you are welcome to think of (or find online) a different approach and use that.
2. Try allowing some number of restarts after a smaller number of iterations, i.e. instead of doing a single run of 500 iterations, try doing 5 runs of 100 iterations, or 10 runs of 50 iterations. If you used more than 500 iterations in your tests, the idea is still the same. Do more than one run, but with fewer iterations, so that the overall total number of iterations is the same.
3. Randomly choose a queen to move and calculate the number of conflicts that queen is involved in. Then look at the squares you could move it to in some order (it doesn't matter what the order is), and move to the first one you find that has fewer conflicts. If there is no such move, move to the square with the fewest number of conflicts. This could help a lot when N gets really large.

Report

So, there are six algorithms to report results from. The basic MIN-CONFLICTS, the two variations, and the last three things to try on the best of the first three algorithms. We're interested in both the number of steps and the time taken to solve the problem, since, of course, fewer steps is not better if the time spent on each step increases too much. So to present your results, graph both the number of steps and the run time as the size of the problem increases. Contruct you graphs in a way that makes it easy to compare the performance of the different algorithms.

The numbers you graph should be averages over 10 runs. However, if you have results for very large problems, but the run time made it impossible to collect data from 10 runs, report those results anyway; just make clear the number of runs that that data is based on.

Extra Credit

Finally, here's a different way to approach the problem that you could do for a substantial amount of extra credit. Every row has to have exactly one queen. So you can look at a placement of queens as a permutation of the numbers from 1 to N, e.g. in a 4-queens problem, `2 4 1 3` would represent the board where Q_1 is in row 2, Q_2 is in row 4, Q_3 is in row 1, and Q_4 is in row 3. Then you can think of moves as a change in the current permutation, e.g. swapping two numbers. I've posted a paper on Blackboard that describes this approach. They were able to solve N-queens problems with over 1,000,000 queens in several hours. And this was in 1990 on a 25 Mhz Motorola 68030!

Grading

Your assignment will be graded approximately as follows:

70-80%: your code (correctness, clarity, etc.), and

20-30%: your report (completeness, clarity, use of tables and graphs, etc.).

Submitting Your Work

When you have completed the assignment, you should:

1. Upload a folder to Blackboard that contains all the files I need to run your program, including a `README` file with clear instructions about how to run your program. The folder should also contain an electronic copy of your report. You can upload it in the `Assignments` section on BlackBoard.
2. Please turn in a hard copy of your report as well.