Grace Handler, Clara Hunnewell, Kote Mushegian
Professor Majercik
Artificial Intelligence
November 14, 2016

Assignment 4: N-Queens Technical Report

N-Queens Problem Description:

The N-queens problem tests the problem of placing "*n* non-attacking queens on an *nxn* chessboard, for which solutions exist for all natural numbers *n* with the exception of n=2 and n=3." Essentially, the goal is to place *n* queens on an *nxn* chessboard such that no queen can attack any other queen. A queen can attack another queen when it is in the same column, row, and/or diagonal as the other. The textbook writes, "Roughly speaking, *n-queens* is easy for local search because solutions are densely distributed throughout the state space" (pg. 221). For the sake of this project, we will be writing a program that tests various ways of using local search to solve this problem for at least up to 8 queens.

The Local Search Approach:

In local search, one begins with a randomly generated complete-state of the problem and then iteratively improves it with the goal arriving at a solution to the problem. Instead of searching through the state space, one is searching through the solution space of one's problem. Local search thus involves generating a complete-state of the problem, evaluating that complete-state, if it is not a solution, evaluating the neighbors of that complete-state (the neighbors are other complete-states that differ by a small amount from the current complete-state), and then moving to a "better" complete-state (whether or not a neighbor is "better" is determined by the "fitness" of the neighbor compared to the "fitness" of the current complete-state, where fitness is defined based on the needs of the problem). Local search ends when a solution is found or when one runs out of time or memory. Quoting the textbook again:

> "local search algorithms turn out to be effective in solving many CSPs. They use a complete-state formulation: the initial state assigns a value to every variable, and the search changes the value of one variable at a time...The point of local search is to eliminate the violated constraints. In choosing a new value for a variable, the most obvious heuristic is to select the value that results in the minimum number of conflicts with other variables - the min-conflicts heuristic."

For this project, we are testing the claim that the min-conflicts heuristic approach can solve the N-queens problems in nearly constant time.

To do local search for n-queens, the program starts with a complete state of the N-queen problem, in this case an nxn board with n queens on it (one in each row), and determines whether or not that board is a solution. In other words, the program evaluates whether there are conflicts among the n queens. The nxn "board" in our approach is represented as a one-dimensional vector of size n, which holds the column locations of each queen. So, the column of a queen in row `i` is at `queenLocations[0]`.

If there are no conflicts, then the given state is a solution to the N-queens problem, and that solution will be returned. Otherwise, if the given complete-state is not a solution, the program changes the value of one of the variables (moves one of the queens), generating a new and different complete-state of the problem. Which variable to change and to what value determines how quickly one arrives at a solution state. The purpose of a heuristic is to guide the choice of variable and value. In this project, the min-conflicts heuristic is used. This heuristic calculates the number of conflicts for a given queen with the assumption that reducing the conflicts of a given queen will be moving toward a solution (when all queens have no conflicts). In our approach, there is always one queen in each row and then the min-conflicts heuristic is used to determine in which columns we should place each queen so as to result in zero conflicts across the board.

For the min-conflicts heuristic, we implemented "basic," "greedy," and "random" variations. The "basic" variation implemented the min-conflicts heuristic. Given a complete-state, it chooses a random queen to move (row to manipulate) and then looks at each column in which that queen can be placed. For each potential column for the given queen, it calculates the number of conflicts that placing the queen in that column will cause. It then moves the queen to the column that minimizes the number of resulting conflicts. If two or more columns result in the same minimized number of conflicts, the tie is broken by randomly choosing one of them for the queen's new position. At the end of this process, there is a new complete-state and, if it is not a solution, the process is repeated for STEPS amount of time, where the user can specify the value of STEPS as a command-line parameter.

With the "greedy" variation, instead of choosing a random queen to move each time, the min-conflicts heuristic is calculated for every queen, then the move that results in the smallest number of conflicts is executed. Like in the basic approach, if two or more moves result in the same minimized number of conflicts, the tie is broken by randomly choosing among them. The "greedy" variation calculates the heuristic more times with the hope that it can reduce the number of steps required to arrive at the solution state in order to compensate for the increased amount of time spent calculating the heuristic for each possible move. At each step, "greedy" calculates the heuristic n^2 times for each spot on the board whereas "basic" only calculates the heuristic n times, for each spot in the randomly chosen row. With "greedy" comes a reduction (though not a total loss) of randomness, since one is no longer choosing a queen randomly. There is still an element of randomness given that, if more than one move result in the same number of conflicts, the tie is randomly broken. Randomness is useful in local search because it can keep the problem from getting stuck rotating between a couple complete-states and never

arriving at the solution state. Finding the right balance between greediness and randomness is key to local search.

In the “random” variation, even more randomness is introduced into the problem. In this variation, at each step, a random queen is chosen to be moved. Then, with probability 0.4 the queen is moved to a randomly selected column. Otherwise, the queen is placed into an appropriate location using the MIN-CONFLICTS heuristic, i.e. the queen is moved to the column that results in the last number of conflicts, similarly to in the BASIC variation.

The best of the above variations in both runtime and number of steps required to arrive at a solution was the BASIC algorithm. To this, we added two tweaks: “smart-start,” and “first-better.” “Smart-start” is mindful of the initial placement of the queens. It places the queens such that, when placed, they are attacked by as few queens already on the board as possible. If there are ties among locations, it chooses the first one found. “First-better” calculates the number of conflicts in which the chosen queen is involved and then moves the queen to the first column it finds that results in fewer conflicts. If no better column is found, the program will randomly choose among the columns that result in the minimized number of conflicts.

Testing the overall claim that the N-queens problem can be solved in nearly constant time, here were our initial hypotheses of the algorithms’ performance in relation to one another and in relation to the claim:

1) 1BASIC -- We predict this will have a slower run-time and require a greater number of steps relative to RANDOM and GREEDY and will not run in nearly constant time.
2) GREEDY -- We predict this will have a faster run-time and require less number of steps relative to BASIC (since it is choosing moves in a more targeted fashion) but a slower run-time and require more steps relative to RANDOM and will not run in nearly constant time.
3) RANDOM -- We predict this will have the fastest run-time and the least number of steps relative to BASIC and GREEDY and will not run in nearly constant time.
4) SMART-START -- Assuming this algorithm will use RANDOM, we predict this algorithm will be the best algorithm overall both in terms of fastest run time and least number of steps and will run in nearly constant time.
5) RANDOM -- Assuming this algorithm will use RANDOM, we predict this algorithm to have a slower run time than SMART-START but faster than GREEDY, RANDOM, and BASIC and will not run in nearly constant time.

Description of Experiments and Data:

Our data can be found here: goo.gl/YEo5YI

→ “**Table 1:** 5-Algorithm Comparison” (the first data sheet): Problem was run with 500 steps and 8 queens 10 times for each algorithm (so a total of 3x10 = 30 times). The number of steps and run-time required to arrive at a solution was recorded and then both were averaged across the ten runs for each algorithm to determine which algorithm was

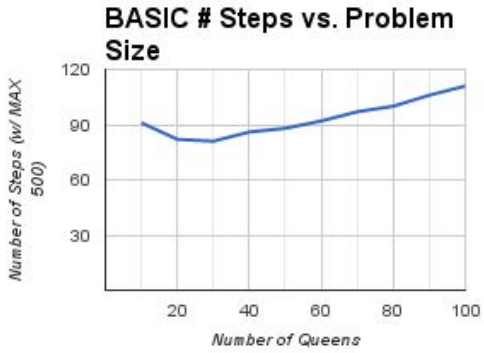
the best (our definition of best is defined in the results section). The best of the first three was BASIC; RANDOM and SMART-START were then run on BASIC with 500 steps and 8 queens. *This data set was used to determine on which algorithm to implement the SMART-START and RANDOM tweaks.*

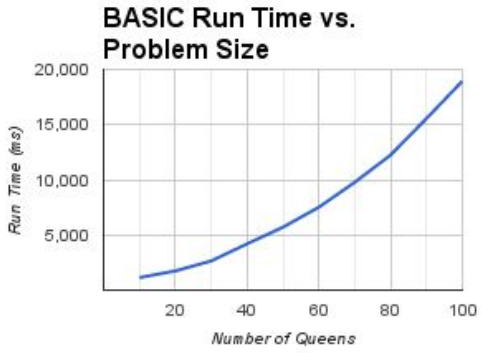
→ "**Table 2:** N-Queen Problem Size Comparison" (the second data sheet): Problem was run with 500 steps and an increasing number of queens for each of the five variations from above, 10 times each, beginning with 10 queens (and incrementing by 10 queens thereafter up to 100). This data sheet records the average at each number of queens, and it is graphed below twice for each algorithm, one with # Steps vs. Problem Size and the other with Run Time vs. Problem size. After those graphs, the algorithms are all graphed on one line graph for the same metrics in two graphs. In addition, the log of the run times is graphed for all algorithms to better represent the data. *This data set was used to create figures 1-13 (below).*

Table 3: Maximum problem size (# queens) that can be "reliably" solved in 500 steps:

BASIC	475
GREEDY	650
RANDOM	20
SMART-START	800
FIRST-BETTER	370

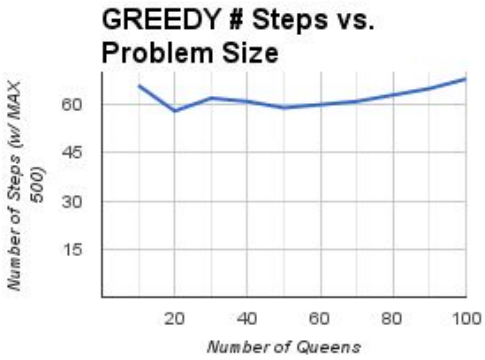
Below are the ten graphs, two for each algorithm. Note that for each the y-axis has a different scale:

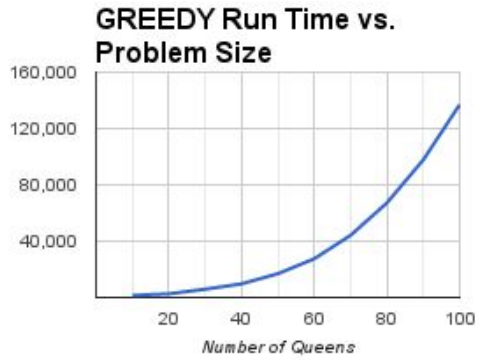




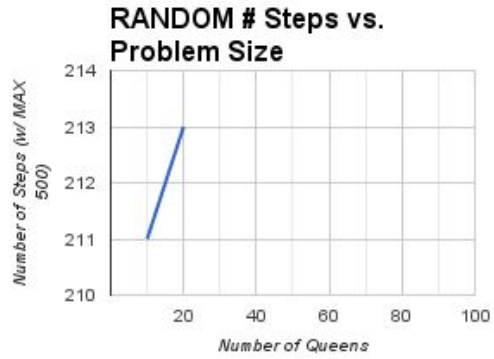
Figures 1 and 2: Figure 1 shows the number of steps required to arrive at a solution for different numbers of queens using the BASIC variation and a maximum number of steps

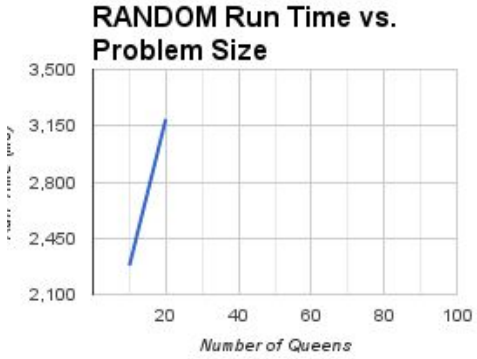
allowed of 500. Figure 2 shows the runtime required to arrive at a solution for different numbers of queens using the BASIC variation and a maximum number of steps allowed of 500.



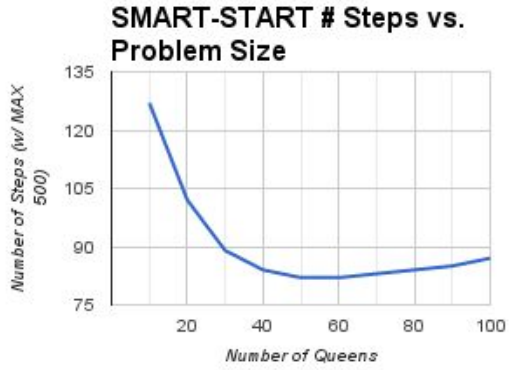


Figures 3 and 4: Figure 3 shows the number of steps required to arrive at a solution for different numbers of queens using the GREEDY variation and a maximum number of steps allowed of 500. Figure 4 shows the runtime required to arrive at a solution for different numbers of queens using the GREEDY variation and a maximum number of steps allowed of 500.

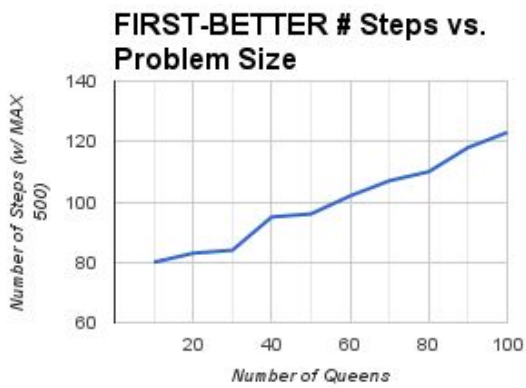


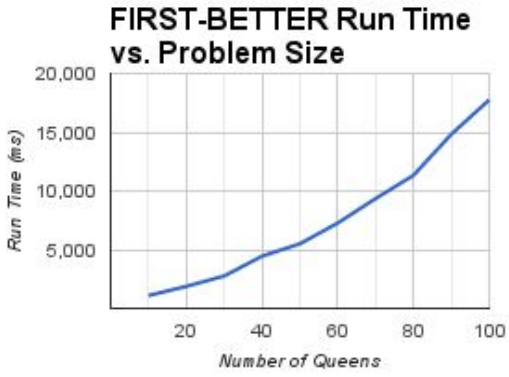


Figures 5 and 6: Figure 5 shows the number of steps required to arrive at a solution for different numbers of queens using the RANDOM variation and a maximum number of steps allowed of 500. Figure 6 shows the runtime required to arrive at a solution for different numbers of queens using the RANDOM variation and a maximum number of steps allowed of 500. Note that in both figures 5 and 6, for problem sizes of 20 and higher, the RANDOM algorithm had less than a 90% success rate for solving this problem. These points were not plotted.



Figures 7 and 8: Figure 7 shows the number of steps required to arrive at a solution for different numbers of queens using the BASIC variation with the RANDOM tweak and a maximum number of steps allowed of 500. Figure 8 shows the runtime required to arrive at a solution for different numbers of queens using the BASIC variation with the RANDOM tweak and a maximum number of steps allowed of 500.





Figures 9 and 10: Figure 9 shows the number of steps required to arrive at a solution for different numbers of queens using the BASIC variation with the FIRST-BETTER tweak and a maximum number of steps allowed of 500. Figure 10 shows the runtime required to arrive at a solution for different numbers of queens using the BASIC variation with the FIRST-BETTER tweak and a maximum number of steps allowed of 500.

Below are the two graphs summarizing the relationship between the five:

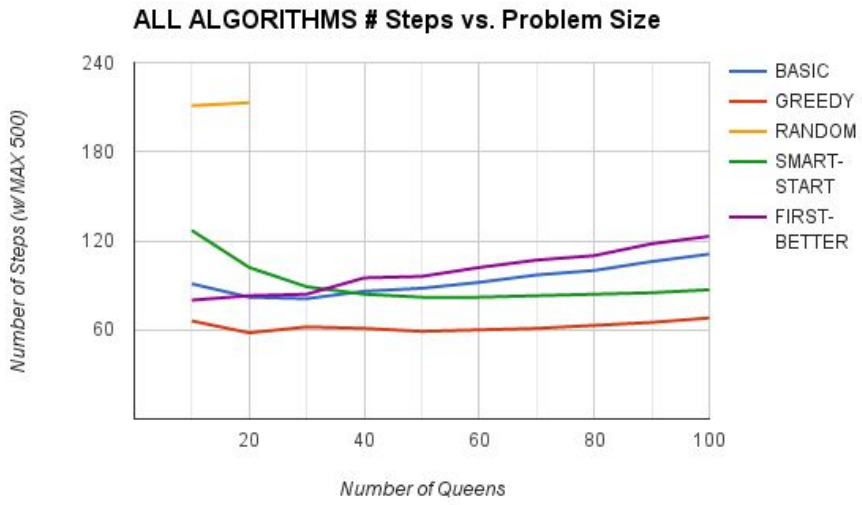


Figure 11: Figure 11 compares the number of steps to arrive at a solution for the five different algorithms as the problem size (number of queens increases). This is the same data from figures 1, 3, 5, and 7 plotted onto one graph so that they can be compared. Note that missing data points for the FIRST-BETTER algorithm represent the problem sizes for which the given algorithm did not reliably find a solution in 500 steps.

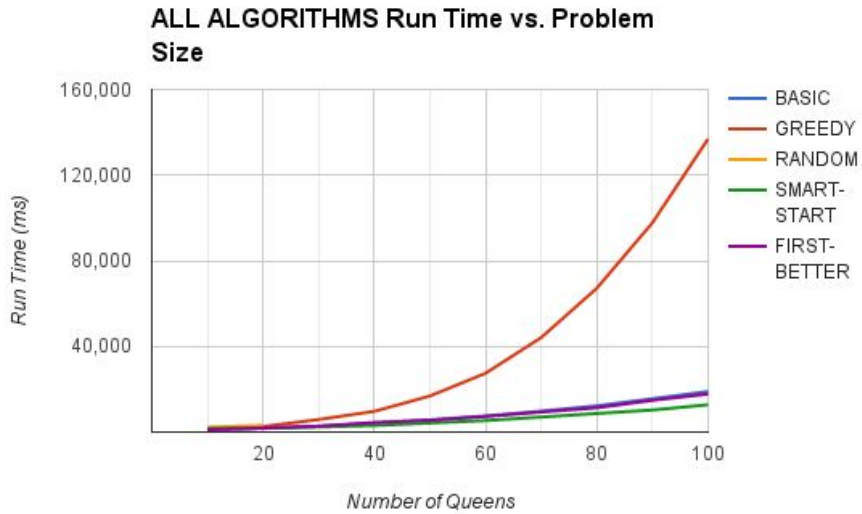


Figure 12: Figure 12 compares the runtime to arrive at a solution for the five different algorithms as the problem size (number of queens increases). This is the same data from figures 2, 4, 6, and 8, plotted onto one graph so that they can be compared. The log of the runtime is presented in Figure 13 to ease the differentiation between the runtimes of the respective algorithms.

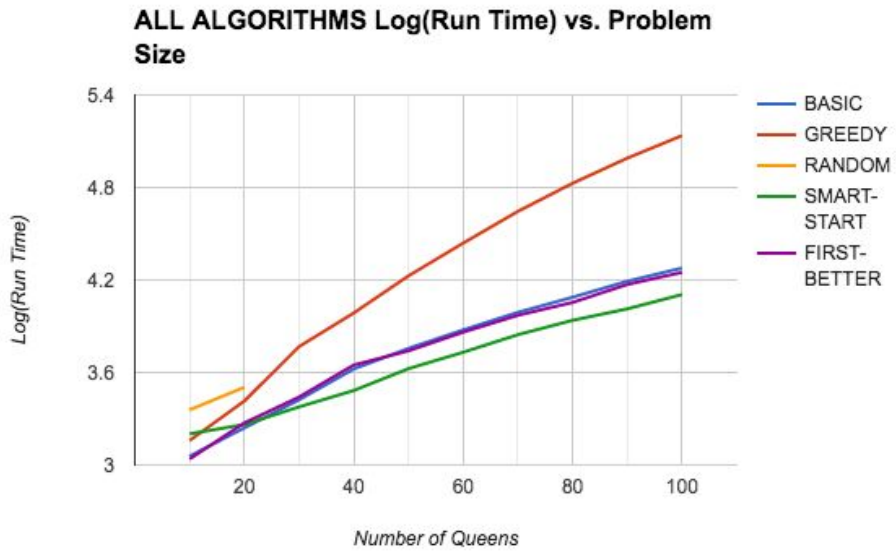


Figure 13: Log of runtime in figure 12. Note that missing data points represent the problem sizes for which the given algorithm did not reliably find a solution in 500 steps.

Discussion of Results:

The purpose of the “Table 1: 5-Algorithm Comparison” data sheet was to determine which algorithm of BASIC, RANDOM, and GREEDY was the best (so that the FIRST-BETTER and RANDOM tweaks could be applied to the best algorithm). For the purpose of this project, we defined “best” as being able to find a solution to the n-queens problem in the fewest steps possible while maintaining a reasonable runtime. The results of the “Table 1: 5-Algorithm Comparison,” whose data is presented in Table 1, showed that the BASIC algorithm was the best. The BASIC algorithm resulted in an average number of steps of 59 (across 10 runs) compared to 67 for GREEDY and 139 for RANDOM. Thus, the BASIC algorithm resulted in a reduction in steps of about 13% compared to GREEDY and a reduction in steps of about 58% relative to RANDOM. The runtime for the BASIC was 405 milliseconds compared to 674 for GREEDY and 890 for RANDOM. Thus, the BASIC algorithm resulted in a reduction in runtime of about 40% compared to GREEDY and a reduction in steps of about 54% relative to RANDOM. As a result, since BASIC was better both in terms of a faster runtime and a lesser number of steps for 8 queens, we determined that BASIC was the “best” and applied FIRST-BETTER and SMART-START to BASIC.

Examining the Figures 2, 4, 6, 8, and 10 and their corresponding data shown in Table 2, there is a consistent relationship between runtime and problem size. Most of the five algorithms experienced an increase in runtime at an exponential rate with respect to the size of the problem (Figure 4 is the most poignant example of this), although there were some of them had minor fluctuations (RANDOM, depicted in Figure 6 looks linear because we only had two data points for this graph, given that RANDOM did not reliably solve the problem 90% of the time for problem sizes higher than 20 queens). The general trend is, however, that the larger the problem size the longer the runtime.

On the other hand, the relationship between number of steps necessary to arrive at a solution and problem size has a slightly more linear, though still positive, relationship, though not “nearly constant” as the textbook had asserted, as runtime increases as problem size increases (Figures 1, 3, 5, 7, and 9). A noticeable exception to the generally positive relationship between number of steps and problem size is figure 7 which compares number of steps to problem size for the SMART-START algorithm. Here, as queens are increased up to 50 queens, the number of steps to solve the problem is decreasing. After problem size of 50 queens it begins to increase again. We believe that this could be attributed to our hypothesis that, for smaller boards, it is easier to get stuck since there is less variety for our random functions (there is a greater possibility that the same move will be made repeatedly). Additionally, our SMART-START function, when setting up the board, places each queen in the first place it finds that results in minimizing the number of conflicts (ie there will always be a queen at (0,0)) instead of randomly choosing among the locations that are tied for minimizing conflicts. This results in the same board for each of the ten runs for a given problem size of N. These boards could be particularly prone to getting “stuck” at small problem sizes and are never varied.

Comparing all five algorithms (see Figures 11-13), several metrics have been introduced to measure algorithm performance. In terms of runtime versus the increasing problem size, SMART-START turned out to be the best algorithm for the problem size of more than 20 queens; SMART-START exhibits consistent superiority over other approaches, as seen in Figure 13. BASIC and FIRST-BETTER performed similarly in terms of run-time and GREEDY performed the worst by far, increasing exponentially faster than the other algorithms (Figure 12). However, in terms of the number of steps versus the increasing problem size metric, the GREEDY algorithm turned out to be the best, with an average of around 60 steps to solve a problem compared to the second best algorithm in the context of this metric, SMART-START, which did it in 80-90 steps.

From Table 3, we can observe that the algorithms solved in 500 or less steps problems of different sizes with varying degrees of success. RANDOM was by far the worst, only reliably solving problems up to 20 queens successfully. On the other hand, SMART-START was the best, solving problems of up to 800 queens in under 500 steps, an increase of 3900% over RANDOM. Over a problem size of 800, our computer did not have enough memory to complete the problem. GREEDY performed better compared to BASIC, solving problems of size 650 though took a much longer time, as mentioned above.

Per the above discussion, here is how our results compared to our initial predictions, working off of Figures 11-13:

1) BASIC -- After testing, BASIC ran in the fewer number of steps over different problem sizes at reasonable run times and thus outperformed GREEDY and RANDOM, contrary to our hypothesis.
    a) ***Our Prediction:*** *We predicted this will have a slower run-time and require a greater number of steps relative to RANDOM and GREEDY and will not run in nearly constant time.*
2) GREEDY -- After testing across different problem sizes (data from Table 2), GREEDY had the slowest overall run-times, which makes sense because it is calculating the min-conflicts heuristic more times. We had assumed that its more thorough look ahead would solve the problem in significantly less steps so as to compensate for the increased time needed to complete the look ahead and thus thus not having a significant impact on run-time. This ended up being wrong. GREEDY also solved problems with fewer steps than BASIC, which aligned with our hypothesis, but also fewer steps than RANDOM, which we did not predict. Calculating the min-conflicts heuristic for every move achieved finding the solution in fewer steps than the other algorithms, but yielded significantly larger run-times.
    a) ***Our Prediction:*** *We predict this will have a faster run-time and require less number of steps relative to BASIC (since it is choosing moves in a more targeted fashion) but a slower run-time and require more steps relative to RANDOM and will not run in nearly constant time.*
3) RANDOM -- After testing, RANDOM had run-times greater than BASIC and GREEDY for the queen sizes that it did solve but solved the problem in many more steps. This makes sense because RANDOM is moving toward a solution in a less efficient/directed way than, BASIC or GREEDY. RANDOM did not perform reliably on problems larger than 20 queens.
    a) ***Our Prediction:*** *We predict this will have the fastest run-time and the least number of steps relative to BASIC and GREEDY and will not run in nearly constant time.*
4) SMART-START -- Of course, our assumption of running on RANDOM was incorrect, and we instead ran on our “best” of the first three algorithms, which is BASIC. After testing, SMART-START was not the “best” algorithm overall in terms of steps. However, it did have the best run-time of the algorithms and was able to reliably solve the biggest board. As the problem size increased, the other algorithms found the solution overall in more steps than smaller problems. SMART-START, however, had a slight decrease in the number of steps it took as the problem size increased. This makes sense, because we are placing more queens “smartly” as N increases.
    a) ***Our Prediction:*** *Assuming this algorithm will use RANDOM, we predict this algorithm will be the best algorithm overall both in terms of fastest run time and least number of steps and will run in nearly constant time.*
5) FIRST-BETTER -- Of course, our assumption of running on RANDOM was incorrect, and we instead ran on our “best” of the first three algorithms, which is BASIC.

FIRST-BETTER ended up having very similar number of steps and respective run-times to the BASIC algorithm's results. BASIC slightly, but not significantly, out-performed FIRST-BETTER for run-time.

a) ***Our Prediction:*** *Assuming this algorithm will use RANDOM, we predict this algorithm to have a slower run time than SMART-START but faster than GREEDY, RANDOM, and BASIC and will not run in nearly constant time.*

Conclusion:

Per our in-class lecture, striking a good balance between greediness/exploitation and randomness/exploration is important in local search. Looking back at our hypotheses, we had predicted RANDOM to be the "best" of the first three, which was proven false, because BASIC ran in the fewest steps with reasonable run-times overall; it was clearly a good mix of randomness and exploitation. We also said that overall, SMART-START would have the fastest runtime and run in nearly constant time. Our hypothesis was right in terms of running time, but the actual results did not satisfy the "nearly constant" runtime assumption. None of the algorithms ran in "nearly constant" run-time. GREEDY, however, solved the problem in "nearly constant" number of steps.

For all the algorithms except GREEDY, runtime and number of steps revealed similar conclusions and these held a consistent relationship. For GREEDY, the run-time increased exponentially at a much higher rate than the others but maintained low number of steps to find the solution. Our GREEDY algorithm fell too far on the greediness side of the spectrum, with very high run-times, and our RANDOM algorithm fell too far on the randomness side of the spectrum, with no solutions for bigger problem sizes and high number of steps overall. Because of this, BASIC, and SMART-START and FIRST-BETTER, which are iterations on BASIC were our best performers, because they stayed in the middle of the spectrum randomness and greediness.

Sources:

https://en.wikipedia.org/wiki/Eight_queens_puzzle
Artificial Intelligence: A Modern Approach, Stuart J. Russell and Peter Norvig
In-Class Lecture for CSCI 2400: Artificial Intelligence @ Bowdoin College, Professor Majercik,Fall 2016.