Assignment 4: Local Search

Brian Jacobel

Artificial Intelligence, Fall 2013

1 Introduction

In this lab, I implemented the local search algorithm using the MIN-CONFLICTS heuristic in a program designed to efficiently solve the "n-queens" problem. This constraint satisfaction problem describes the task of placing n queens on an n-by-n chessboard, and iteratively arranging them so that no queen is in a position of being able to capture another queen within a single move. In this problem, there are $6n-2$ constraints: n constraints specify that each column may have only one queen, n constraints specify the same for each row, $2n-1$ constraints specify that there may be only one queen in each upwards-sloping diagonal, and $2n-1$ constraints say the same for each downwards-sloping diagonal. The column constraints can be thought of as "freebies" – that is, queens can be obviously distributed at the start such that each is in a separate column, and this can be maintained throughout the problem easily. Thus, the variables that form the search space for this problem are simply the row locations for each queen.

2 Testing methodology

My algorithm is contained in the Python module `nqueens.py`, which can be run interactively via the command line if the user wishes. For testing, this code is imported into the script `test.py`, which runs the code a number of times and averages statistics. For each run of `test.py`, the n-queens problem is solved ten times each with with n starting at 5 and incrementing by 5 up to a maximum of 60. At each value of n, the average run time for the problem to be solved and the average number of moves required to find that solution is displayed. If one of the ten trials should fail, only successful trials are included in these values. However, in my tests, my program seldom failed to find a solution to the problem for a value of n less than 100.

I ran `test.py` six times: one for each of the variants of the algorithms explored in the assignment. In the upcoming section, I will display and discuss the differences in generated results between these algorithm variants.

As a point of comparison between my results and others', my testing was done on a fairly old Intel Xeon E5620 machine (a 2010 Mac Pro) with eight cores clocked to 2.40GHz. This CPU is somewhere in the neighborhood of 60% as fast as the newer i7-2600 CPUs in Searles 224's iMacs, according to benchmarks I found online.

3 Algorithm Variants

3.1 Basic implementation

For the basic local search algorithm with MIN-CONFLICTS as a heuristic, I obtained the results shown in *Fig. 1.* The algorithm's performance as measured by moves shows, after a large initial jump, a roughly linear growth from 60 to 120. On the other hand, the algorithm's performance as measured by time shows an exponential increase as the size of the problem increases. A trendline plotted through these data points conformed to the equation $4.17x^{3.23}$, meaning the running time of this algorithm is roughly bound by $O(n) = 4x^4$.

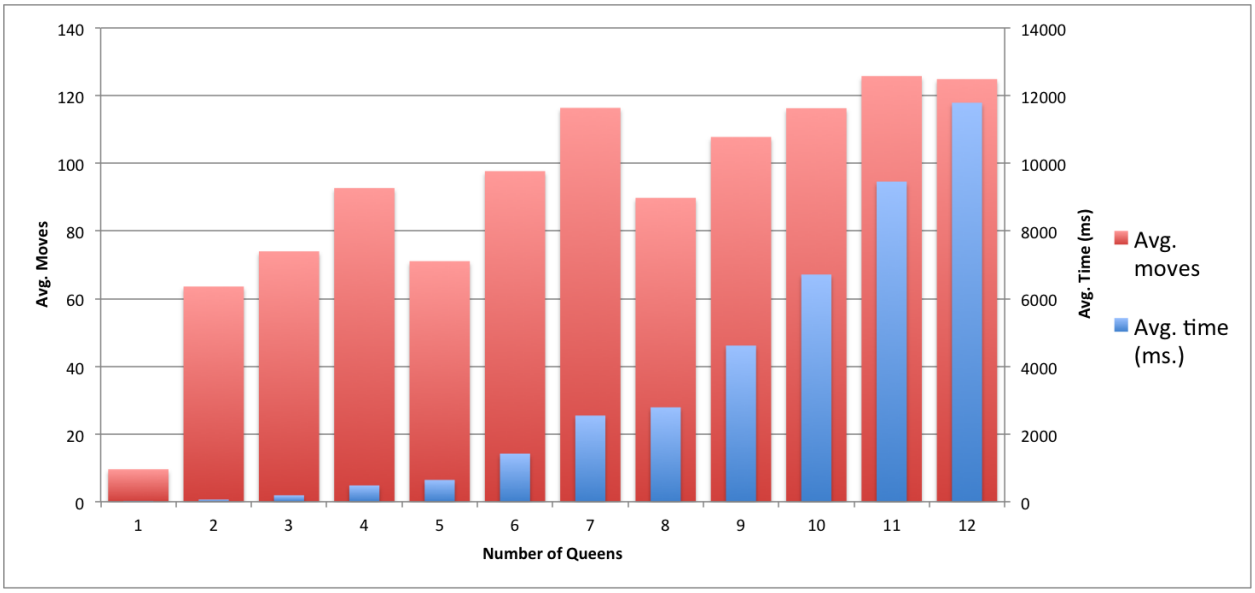

Figure 1: Performance of the basic local search algorithm, in moves and time.

3.2 Greedy implementation

The results of the greedy variant of this algorithm, where the queen to move rather than being picked randomly is the queen with the greatest number of conflicts, are shown in *Fig. 2*. The data conform to the same general patterns shown above: a linear increase in average moves performed an a power increase in the running time.

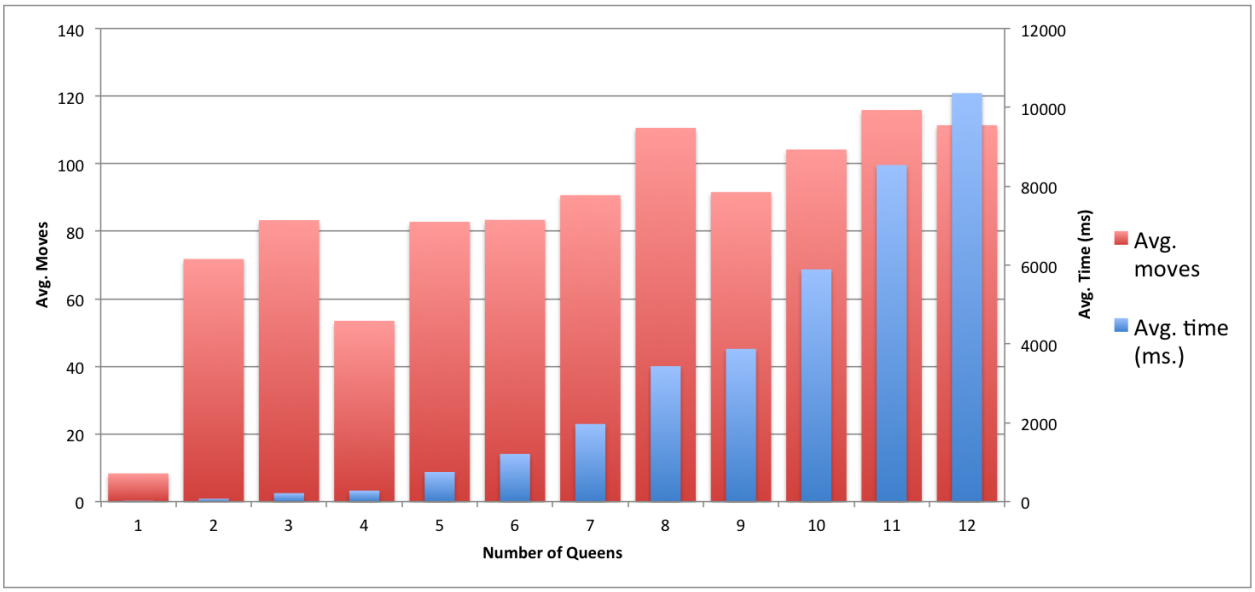

Figure 2: Performance of the greedy variant of the local search algorithm, in moves and time.

3.3 Random implementation

In this algorithm variant, an unsatisfying queen is picked in the same way as in the basic algorithm. At this point, with some probability, instead of using the standard MIN-CONFLICTS heuristic to find a new location for this queen, this modified algorithm instead randomly picks a location in the queen's column to place it. In my trials, I found the probability 20% a good portion of the time to move randomly rather than heuristically; percentages higher than this resulted in the problem not being solved for a significant portion of the tests run. At 20% random movement, the constraints failed to be satisfied after 500 tries once with 35 queens, three times with 55 queens and once with 60 queens. This represents an overall failure rate of 4.16%, as opposed to the base algorithm and the previous variant's failure rate of 0%. Results from this variant are shown in *Fig. 3*.

Of particular note here is that both the number of moves and the time are significantly larger than in the previous two implementations of the algorithm. Comparison between the variants will be discussed further in the next section.

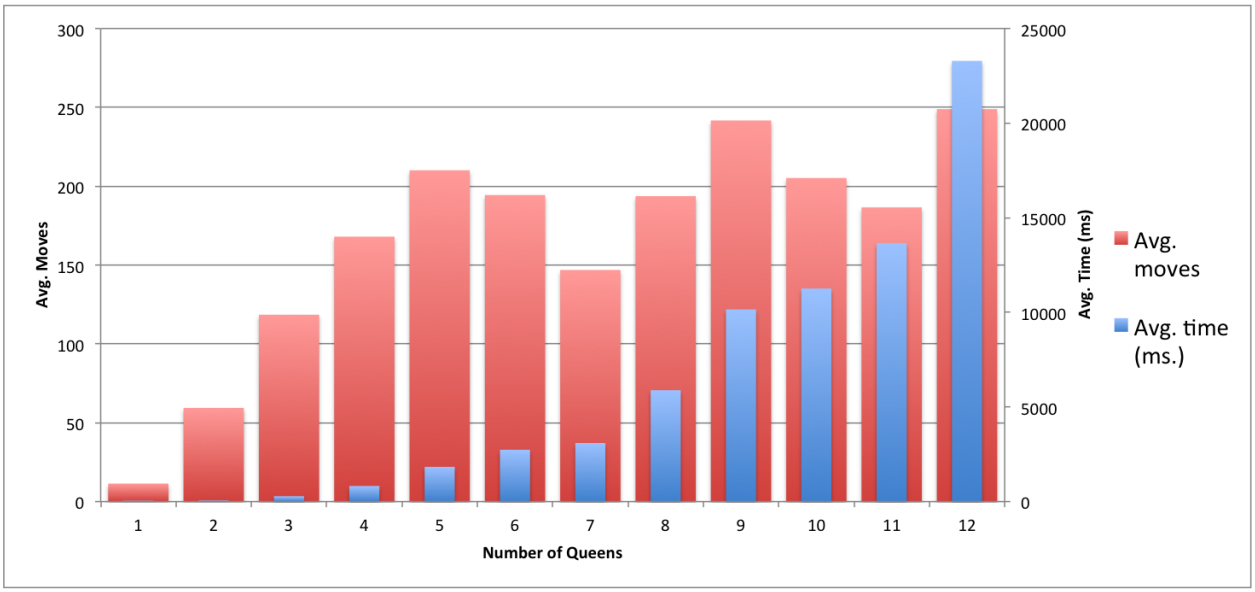

Figure 3: Performance of the local search algorithm with enhanced randomness, in moves and time.

3.4 Comparison of above variants

As can be seen in *Fig. 4*, the data for all three algorithm implementations considered thus far follow the same basic pattern: an average number of moves to the solution that is bounded linearly, and an average time to the solution that is bounded by a power function (likely x^4). However, it is obvious that one of the variants was not an improvement. At its worst ($n = 45$), search with increased randomness takes more than 2.6 times as long to reach an answer as greedy search. This, combined with the fact that search with randomness was causing the failure of about 4% of test cases (as opposed to no test cases failed in the other two variants) lead us to discard increasing randomness as an option for improving local search for this problem. This is not surprising, as this variant essentially says to do the same thing as the base algorithm would have you do, except 20% of the time, when you should take an action that is $\frac{n-1}{n}\%$ likely to be the wrong one. Randomness may help avoid plateaus and "getting stuck," but that wasn't a problem that we were having in solving n-queens (partly because of the strength of our heuristic).

Of the two remaining algorithm variants, greedy search appears to have the slight advantage in running time (and also moves, which I have not graphed because it is less interesting). In the three cases I will discuss next, I have used the greedier variant of local search as a base for improvement.

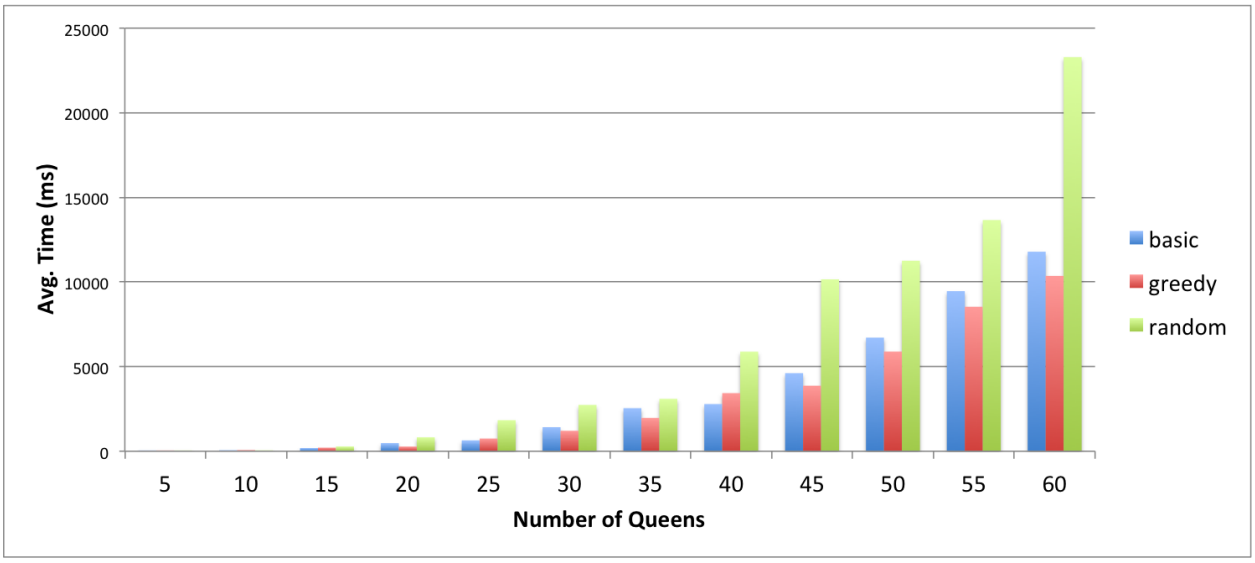

Figure 4: Comparison of the first three local search algorithm variants. Search with enhanced randomness is clearly inferior.

4 Further variation

4.1 Smarter initial placement

The next most obvious place to make improvements in this algorithm is choosing better initial positions. I had tried during my development phase to use both random initial rows, or to place all queens in row 0 – both unscientifically seemed to have about the same efficacy. One solution offered was to place each queen on the board in the position of fewest conflicts at that time. However, this seemed foolish to me for two reasons. First, from an abstract level: isn't this just the same as running the MIN-CONFLICTS heuristic an additional n times at the beginning of the program? Second, from a pragmatic level: My program's method for finding the number of conflicts depends on there being as many entries in the list of queens as there are rows and columns on the board. Implementing this suggestion, I though, would not provide the benefits desired when the time to refactor a large part of my program was considered.

I chose to implement a different approach, which does not require determining the number of conflicts for queens on an incomplete board. For each queen, I determine if the column it is placed in is even or odd. If odd, I place it in the first row; if even, in the last row. For each of the next even and odd queen, I place it in the second row, and the second-to-last row. In this way, there ought to be no initial conflicts between rows. There also ought to be very few conflicts between diagonals. An illustration of the initial step of this algorithm for $n = 8$ is below.

Q0							
		Q2					
				Q4			
						Q6	
							Q7
					Q5		
			Q3				
	Q1						

As can be seen, there are only three conflicts in this initial placement: Q6 with Q7, Q6 with Q3, and Q5 with Q0. Moreover, this placement algorithm is actually more efficient than the one suggested in the assignment write-up, because it does not require running the validator on each new placement to determine the number of conflicts. It is very fast, and very good. Results using this initial placement model are shown in *Fig. 5* and, in *Fig. 6*, compared to results using static queen placement at row 0 for every queen. The improvements are slight, but there is a definite favor to the new initial placement method overall.

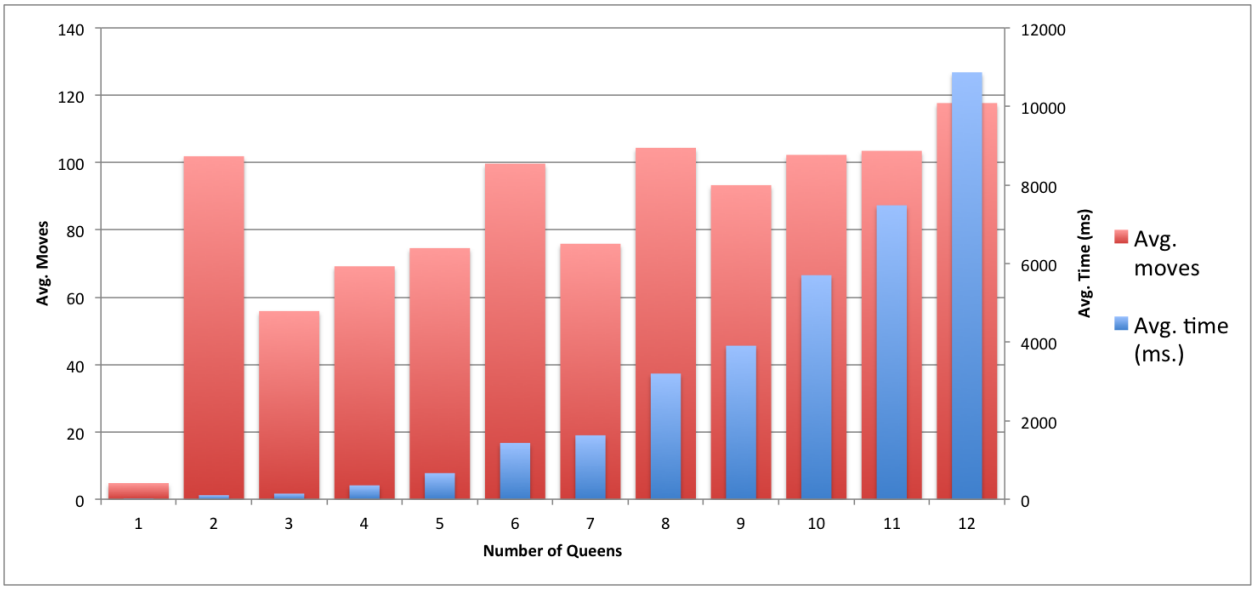

Figure 5: Performance of the local search algorithm with greediness and enhanced initial placement, in moves and time.

4.2 With restarts

Because by the upper end of the problem sizes I am running in my test set, I know that the average number of trials should not exceed 125, it might be a good idea to instead of running trials all the way out to

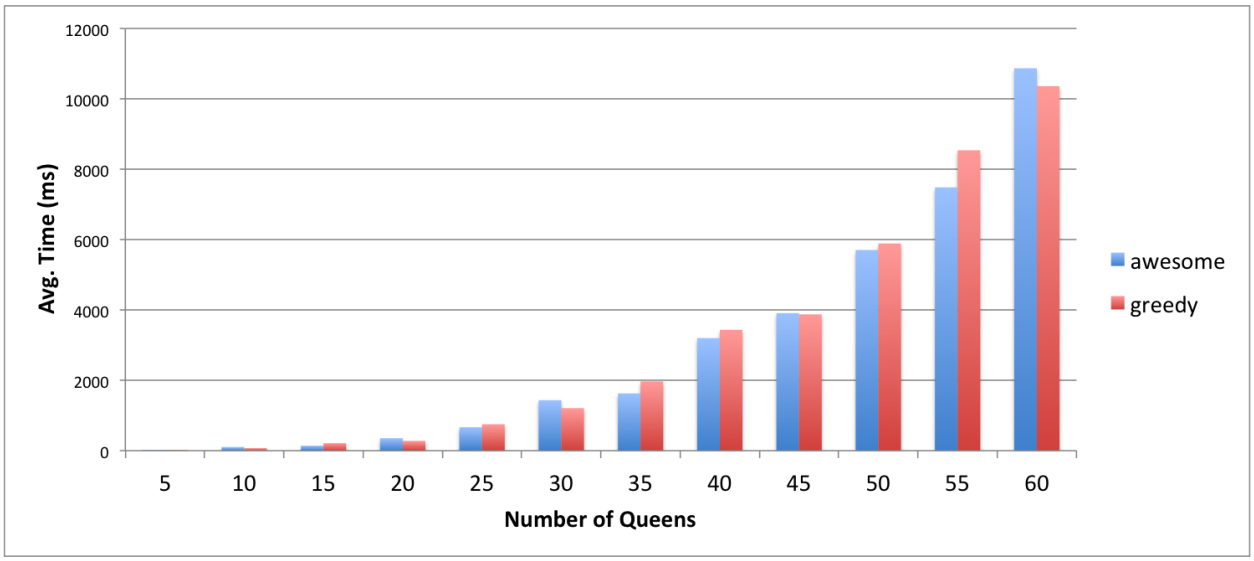

Figure 6: Performance of the local search algorithm with greediness with and without "awesome" initial placement. Improvements are slight but but definite.

500, to stop at 125 if a solution has not been found and do a restart. In theory this is a good idea, because restarting allows one to avoid the "getting stuck" pitfalls: plateaus, local maxima, etc. In practice, I don't think this will actually help, because I don't actually do 500 iterations – I do about 125 iterations, and then I quit because I found the solution. Restarting at 125 won't really help me find the solution faster, because I know the solution is somewhere near 125 moves from where I start, at a maximum. It will just force me to do more iterations. Results are shown in *Fig. 7*

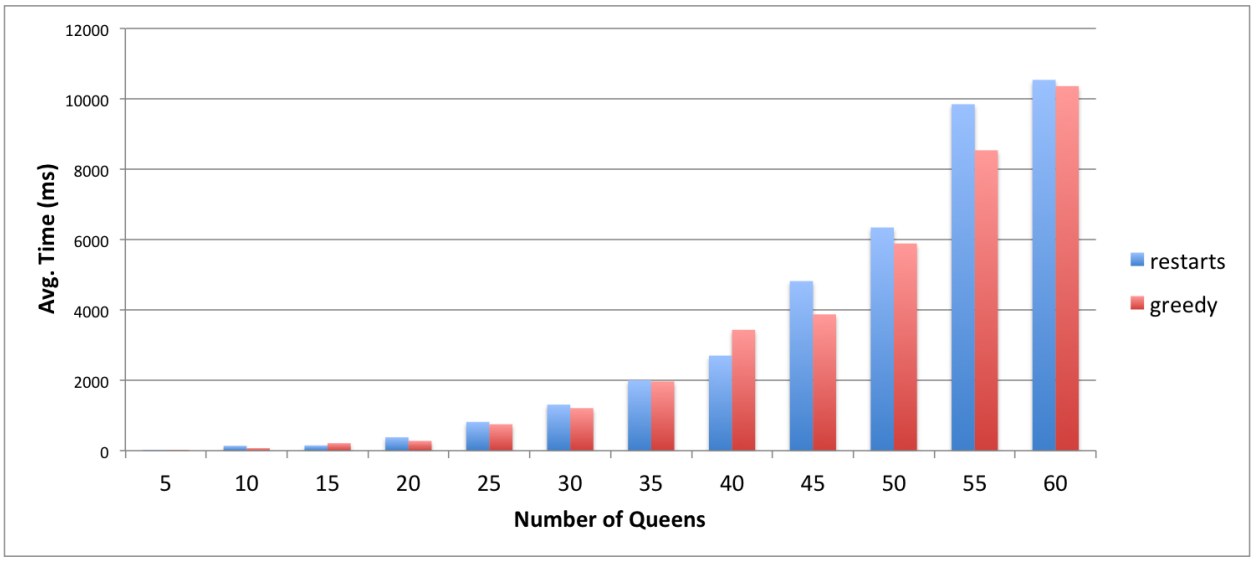

Figure 7: Performance of the local search algorithm with greediness and enhanced initial placement with and without cutoff restarting in moves and time.

As predicted, cutoff restarts have no benefit for the smaller sized problems, because they never reach the cutoff point. For larger problems, cutting them off before they finish simply doubles, triples or quadruples the total running time of individual cases, leading to a higher average time overall.

4.3 Random movements

I have to admit this one confuses me. The assignment describes a algorithm variant that randomly chooses a queen that is an unsatisfier (already implemented), then scans rows until it find any new position better than the current one (in terms of number of conflicts). If it finds one, it moves there and continues with the next iteration. This makes sense; it's a very greedy variant of the algorithm which takes big shortcuts in an attempt to be fast. The next part doesn't make sense to me, though. The description of this variant continues, "If there is no such move, move to the square with the fewest number of conflicts." If there is no move such that the given unsatisfying queen can have less conflicts, isn't it already in the best place for it to be?

I've implemented this algorithm variant without the "If there is no such move..." part, and it basically doesn't run, which isn't surprising. Because this variant only makes the minimum effort to find a better location for a queen in any given iteration, it would have to run for a LOT of iterations to get the same result. I upped the number of iterations to 1500 (from 500, a 200% increase) and got the following result:

n	Success rate
5	100%
10	30%
15	10%
20	30%
25	20%
30	10%
40	0%
45	futile to continue

If I am understanding what this variant is calling for correctly, it is fairly foolish. We are doing an incomplete search of the row to find a "better" location rather than doing a complete search to find the "best" location, but the time this saves pales in comparison to the huge amount of time lost by having to iterate the whole algorithm three times as long – and even then getting abysmal success rates.

Since very little averaging of data was possible due to the high number of problems failed to solve, I did not think the data I gathered here was accurate enough to merit graphing.

5 Large values

My implementation of the MIN-CONFLICTS algorithm successfully solved the n-queens problem for an n of up to 350. Here are the n-values, moves and run times for various large values of n I tried. All tests below were done with the initial, unmodified implementation of the algorithm. Because of the times involved in obtaining the results, each was only run once.

I was slightly disappointed that my implementation of the algorithm could not get anywhere near solving even 1,000 queens in a reasonable amount of time. I do not know if this was a realistic expectation. I have declined to graph these values because of their uneven distribution, incompleteness, and lack of multiple trials to back them up – a graph would not reveal anything significant other than that this problem gets quite hard to solve with size.

n-value	moves to solution	time to solution(sec)
25	28	2.456
50	118	6.715
75	121	20.935
100	188	74.017
150	224	281.348
200	295	850.566
250	298	1637.862
350	407	8119.451 (2.255 hours)
500	N/A (hit max with 26 conflicts unfixed)	N/A
1000	never finished	More than 18 hours
1000000	never finished	More than 18 hours

6 Conclusion

The local search algorithm has several variants which can be used to augment its already fairly good performance at solving constraint problems like n-queens. However, several of the suggested variations did not improve the performance of the algorithm, or negatively impacted it severely. I have left the improvements that were worthwhile in my code, which is now quite successful at solving moderately-sized n-queens problems, although problems of significant size remain out of its reach.