CSC 173 Fall 2015

N-Queens:

An Exploration into Backtracking and Min-Conflicts

By: Thomas Pinella

Table of Contents

Introduction

The purpose this paper poses is to provide insight into two algorithms that find solutions for the N-Queens problem: backtracking and min-conflicts. I will proceed to give some explanation of what exactly N-Queens is, the idea behind backtracking and how it works at a basic level, and the results I got from implementing it and using it on boards of various sizes. We will then explore min-conflicts and all the challenges that come with it; we will get to the real meat of this paper in that section when we discuss what I found to be the most optimal way of breaking out of local minima -- the real challenge that min-conflicts, and all hill climbing algorithms for that matter, give us. After an analysis of the results I got from using min-conflicts, we will then conclude this paper, taking a final look at these two CSP solving algorithms and seeing if it's possible to further improve upon them in the future. Hold on tight, the exploration is about to begin.

Problem Space

So what exactly is N-Queens? Well, I'm glad you asked. As you well probably know, (but I'm going to repeat it here anyways) the queen piece in the game of chess is capable of moving any distance in any direction: up, down, left, right, and, of course, all four diagonal directions. A chessboard of dimensions N x N (yes, chess is played on an 8 x 8 board, but in N-Queens, the board can be of any size, N size to be exact), satisfies the conditions when there are N queens on the board that cannot attack each other. That is, there is one queen per column, one per row, and one per diagonal. No queen conflicts with any other queen.

Now, oftentimes people will ask how many solutions to N-Queens are possible for a board of dimensions N x N, and what they all are. Of course, the bigger the board, the more solutions possible. Now, to be clear, the algorithms I'll be referring to in this paper do NOT do this exactly. They do NOT find all possible solutions. They find one. And then return it. That's it. Which brings us to the next section: output.

Output

Although our two algorithms, backtracking and min-conflicts, have vastly different methods of arriving upon their respective answers, they both will be outputting the same thing[1]. The functions will be passed two parameters before computation: a boolean and a number n, signifying the board size. The output will be a vector representation of the solution and the number of steps (each step is defined by the movement or placement of a single queen) required to get there. If the passed in boolean was true, then it will also print out a two dimensional representation of the solution, where 0's represent an empty space and 1's, a space filled by a queen.

To be clear, the vector representation of the solution is where each index refers to a column of the chessboard, and the value stored at each index is the row number at which the queen is placed.

Alright, now that we are clear on what the problem is and what our algorithms will be spitting back at us, let's begin.

Backtracking

The backtracking approach to N-Queens is essentially a depth first search. The vector representation of the board, which I described not too long ago (scroll up an inch or two), is used to do this (it's also used when calculating min-conflicts, but we'll get to that later).

Now, I don't want to get too deep into detail about how the code actually works, see my code and read my comments & README.txt for info on that. Instead, I'll give a quick rundown on how the algorithm works from a higher perspective. I coded my implementation with a bottom-up approach, so I'll explain it in that order. There are, essentially, three levels of work being done here: level I, checks a single square on the board and tells us if there are any conflicts with it, level II, finds the square in a particular column that has zero conflicts, and level III, goes through all the columns, placing queens

[1] By "same thing" I don't mean identical thing. They may very well (and absolutely most likely will) return different N-Queens solutions, though equally valid.

in the correct squares for each one. I'll expound a bit more, but I'll keep it brief. As I said before, see code/comments/README.txt for more detail.

Level I: The Conflict Checker

At the lowest level, this function's only purpose of existence is to tell us whether or not a square has any conflicts with it. It returns a true if there are none, and false otherwise. This is an O(n) operation since we need to loop through all of the indices (which represent columns, remember) of the vector representation up until our current index, or current column, that contains the square we're testing. Actually, it's faster than O(n) since we do not go through the entire vector (it's O(n) worst case when we check the last column).

For each index, we need to check two things: do we have a row conflict and do we have a diagonal conflict? For obvious reasons column conflicts are impossible. This is an easy calculation and you can reference my README.txt for details on how I did that. Long story short, we return true if we get to the end and didn't find any conflicts for that particular square.

Level II: The Queen Placer

In the function that takes care of this level of operation, we are working with a single column and our goal is to find which row to place the queen. Of course, we want to place our queen on the square with zero conflicts. How do we do that? You guessed it, all we have to do is loop through each square in the column, calling our conflict checker function on each one. When we get our first square with a zero, we stop looping and plunk our queen down there. Worst case scenario is we have to look through n squares before finding a square with zero conflicts, making our previously O(n) problem now a O(n^2) problem because of the added dimension.

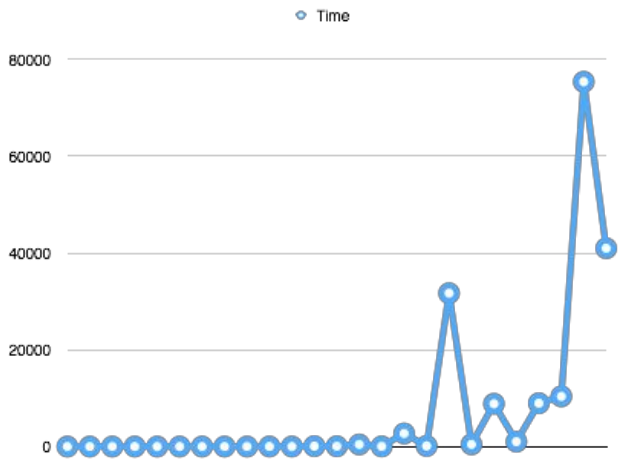
Level III: The Master

Master in the sense that this is the master loop (or tail recursion, as I implemented it) that drives everything. Here we increment columns, looking at each column, from left to right, and calling our place queen function. Here, we also start to see why backtracking is called backtracking. What happens when we're trying to place a queen, and we go through every possible row in that column and they all have conflicts? Then what? Is it over? Do we give up? Fail? No. We backtrack. And it's the master that takes care of this (again, if you want details, please see my code/comments/README.txt. They exist for a reason.) So, when we are unable to place a queen, we go back one column and move that queen to another zero conflict square further down. And if that queen has nowhere to move too, we backtrack yet again. And so goes the process until we can finally start moving forward again.

Now that we have some basic understanding of how the backtracking algorithm works, let's take a look at some results and see how efficient this algorithm really is (spoiler: it's not).

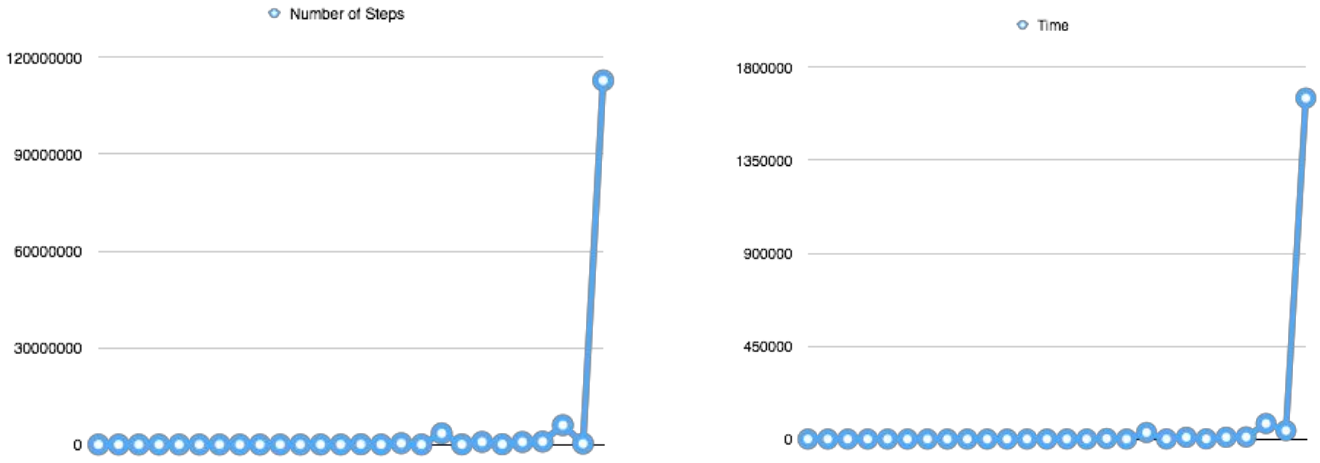
Results

Below, you will see two nearly identical graphs. Rest assured, they are measuring different things. The one on the left is measuring time vs. board size, and the one on the right is measuring the number of steps it took to reach its solution vs. board size[2].



[2] The data used for these graphs can be found in Appendix A

A couple things to keep in mind when looking at these two graphs. The board size, n, ranges from 5 to 29 on the x-axis. It starts at 5, because there is no N-Queens solution on boards any smaller. Now, the two graphs below are the exact same as the ones you just saw except one thing is changed. Can you guess what?



That's right, I did increase the top board size. No, it's not 100. It's 30. The only difference between these two graphs and the two you saw before, is that these include one more dot. Only thing is, is that that dot is high up there. Way high up there, as in stoned. The difference between the number of steps and time it takes to calculate a solution for a board of size 29 versus a board of size 30 is simply remarkable. It skyrockets.

Looking back at the original two graphs we can make a couple other interesting observations other than the obvious fact that it starts increasing exponentially at around board size 22. One thing of note is the spikes on the even numbers. When the board size is 18, we see an obvious increase in time and number of steps over board size of 17. But then something interesting happens. The time it takes to compute a solution and the number of steps goes *down* for a board size of 19. Maybe just a one time thing, right? Okay, let's keep looking. Board size of 20 sees a big spike, as we would expect. But then, looking at board size of 21, it happens *again*. Time decreases, steps decrease. Then 22 happens, which is our first major spike. And then for board size of 23, everything's faster again, as if by magic. It took a mere 447 milliseconds to compute a solution for 23,

whereas it took over 30 seconds to get a solution for 22[3]. Overall, everything is still getting slower, as we see a board size of 29, even though it's odd, still takes about 40 seconds, and of course the even board size of 30 makes us wait an eternity to get our solution.

So, of course, this begs the question of why. Why do board sizes of even numbers compute slower than board sizes of odd numbers? Well, I couldn't tell you. But what I can do is encourage you to look into it, do some research, and present your findings in a well articulated paper, much like this one.

That's pretty much as in depth into backtracking we're going to go. We can conclude that backtracking works very well when we have a small, preferably odd-numbered, board size, less than around 20ish. So can we do better? Does there exist a way, an algorithm, that can compute solutions for N-Queens even faster on bigger chessboards? Well, my friend, yes; there is an algorithm that does exactly that. And it's name is min-conflicts. The remainder of this paper will demystify this hill climbing algorithm for you and show why I chose the specific method of implementation I chose.

Once again, I ask you to prepare yourself; we will be taking a dip into the strange new world of nondeterministic algorithms.

Min-Conflicts

As I did for the backtracking algorithm, I would ask that you take a look at the code/comments/README.txt for more info and particular details about my implementation. I will be focusing on a few aspects that I saw as key. Namely, recognizing local maxima and escaping them, but we'll get into that a bit later. First, let's explore how this algorithm works.

How It Works

Unlike backtracking which begins with an empty board, we start off with n queens on our board, one in each column, and each placed in a random row in its respective column.

[3] For complete data, see Appendix A.

We now have a new goal: instead of creating a solution for N-Queens from scratch (as we did in backtracking), we have something to work with; we have queens already strewn about, and it is now our job to modify, to nudge some queens here and there, until our board satisfies the constraints of the problem. So, how to do that? Randomness is your new best friend.

This is what we do: 1) choose a random column, 2) calculate the number of conflicts in each square of that column from top to bottom, 3) move the queen to whichever square has the least number of conflicts (hence the name, min-conflicts), and 4) repeat this until we have a solution.

At the most fundamental level, the entire thing is fairly basic. We're simply choosing random columns and moving their queens to the best possible place. Rinse and repeat till we have a solution. Things get tricky when you reread my description of the algorithm and realize how vague I was. That's right; I left a lot out. Whether this vagueness is by design or on purpose is beside the matter. What's important is that we ask these questions: how do we know when we're finished and have a solution? And, is it possible to do this in a computationally efficient manner? That brings us to our next section on local maxima and the problems they bring with them.

The Problem of Local Maxima

As I've mentioned before, min-conflicts is a hill climbing algorithm. A hill climbing algorithm is a method of optimization that consistently makes changes that bring it closer to a solution. If the solution to a certain problem is thought to be lying at a mountain peak, then the hill climbing algorithm will try to reach it by making sure it is always moving upward whenever it does work. Now, a problem arises when we reach the peak of lesser mountain that is not our solution summit. From here, all directions lead down, but we can only move up, so we stay put, blind to the possibility that a solution lies on a taller mountain in the distance. So we stop moving, foolishly believing we have reached the highest peak when we are indeed atop no more than a mere anthill.

We need a way to detect when this happens in our min-conflicts algorithm. Because if we don't, then the program will run indefinitely and never arrive upon a correct solution. So, what can we are we to do? (Note: all of the following solutions were tried and tested by yours truly. Unfortunately, I was unable to collect data on all of them for obvious reasons (I don't have infinite time), so you'll have to just take my word for it for now.)

Solution I: Just Check it

As the name of this solution applies, why don't we merely check to see if our board satisfies the conditions of N-Queens with every iteration? And when it works, return it! Problem solved!

Slow down there, not quite. There's a couple issues with this. First, this doesn't actually solve our local maxima issue. If we fall into a local maxima, the program will still run indefinitely, and we'll never arrive upon a solution. We'll be checking incorrect answers for eternity. Okay, so we need a way of shaking things up every now and then, a way of resetting things to make sure if we do get stuck in a local maximum, we can just respawn in the valley and start our trek all over again.

Solution II: Keep checking it, but also reset it every now and then

That's a long solution name, but let's roll with it for a second. Okay, so we're still checking every iteration so when do we have our solution, we'll know and we'll be able to return it. But now we're also resetting after a predetermined number of iterations lest we fall prey to a local maximum. This should work, right?

Well, yes, but it's not all that efficient. We're still doing an entire check every iteration, which, by the way is an O(n^2)[4] operation. That is very computationally draining. Plus, how often should we be resetting? Looks like you avoided answering that in your solution. Long story short, we can do better.

[4] for each index of the vector representation, we are checking all possible row and diagonal conflicts. Very similar to the function that determined whether a square had conflicts or not which I described up in the backtracking section.

Solution IV: Find the most conflicted queen and move her

I like that you're referring to the queen as a "her" here. But okay, this is an interesting idea. This is something of a novel approach and may just make the algorithm faster, as instead of being completely random, it zeroes in on the parts that need the most fixing and fixes them first. Well, at least that's the idea. Let's see how this works in practice.

Nope. Congratulations, instead of solving the local maxima problem and helping us escape them, you just nearly guaranteed them. Since the randomization aspect of the algorithm has been all but cut out, we are essentially guaranteed to fall into a local maximum. It looked like a good idea on paper, but the lack of randomness was its downfall.

Solution V: Find the most conflicted queen and move her, but only 50% of the time

So, as before, we find the queen in the worst position, but now we've sprinkled some randomness back into our algorithm, and only move her 50% of the time. The other 50% of the time we go back to our old method of choosing a completely random column and moving that queen to the best location. And, as before, we check the solution with every iteration and reset after a predetermined number of times.

It works! We certainly will find a solution with this method, and it is indeed faster than the proposed Solution II. But, hold on a second. How do we know we haven't fallen into a local maximum of our own? A meta-maximum, if you will. Perhaps, there are better ways of doing this, more efficient methods. Plus, we never answered the question of how many times we need to reset. So, like our algorithm, let's reset for a moment and take a new approach. (Note: I still have this implementation saved in my code, check code/comments/README.txt for more details).

Solution VI: Only check it before a reset

Forget all the most conflicted queen stuff for a moment. This is a blank slate again. We've reset. What if instead of checking to see if we have a solution with every single iteration, we check only right before we reset? Well, we need to make sure our algorithm is implemented in a specific way before we can do that.

If we choose randomly choose a column that happens to have its queen already placed in a square of no conflicts, then instead of going through the entire O(n^2) operation of searching for the least conflicting square, we simply skip over it and choose another random column. We can quickly check if a queen has no conflicts in O(n) time. Alright, great, time is saved there already.

Now, recall that verifying a board to see if it's a solution or not is an O(n^2) endeavor, since we need to check that every queen has zero conflicts. By doing it only once before our reset as oppose to doing it with every single iteration, we are cutting down on time tremendously. After all, on any given iteration, how likely is it that we've reached our solution? Not too.

Take a look at the two tables below. They are the results of running min-conflicts on a board of size 100 and with a reset after every 10,000[5] iterations. Table A is checking for a solution with every iteration, while Table B is only checking before each reset. Therefore the number of resets plus one is the number of times it's checking; not all that often, you'll notice. I didn't include the total number of iterations in the table, but it's a lot, certainly more than the number of steps (remember, a step is where a queen is moved, so we can have an iteration, where no queen is moved, but not a step).

Table A

Number of Steps	Time (seconds)	Number of Resets
550	5.5	1
490	5.4	1
1.1k	3.4	2
630	6.2	1
110	0.26	0
1.7k	13.5	4

[5] Why 10,000? Well, I'm glad you noticed that I introduced this number out of the blue; it's an important observation. More on this coming up. I'm also glad that you took the time to read this footnote.

Table B

Number of Steps	Time (seconds)	Number of Resets
430	0.88	1
310	0.57	0
1.3k	2.6	4
670	1.5	2
170	0.40	0
480	0.96	1

The steps are roughly the same for both implementations, and this is to be expected, as we aren't changing how columns are chosen. All we're doing is checking less often. And we see that difference expressed most obviously in the time of completion. For Table A, the implementation that checks every iteration, we have an average time of 5.71 seconds. And for Table B, which checks only before each reset, takes the medal with a blazing fast average time of 1.15 seconds.

Our evidence clearly shows that checking only before a reset, as Solution VI proposes is far superior to Solution II, which naively checks every iteration. But is it better than Solution V, which tries to be smart about its column selection by choosing the most conflicted queen, then moving her only 50% of the time in order to escape local maxima. Turns out, yes, Solution II is faster, by longshot. Much like Solution II, Solution V also needs to do a O(n^2) operation every iteration in order to find the column with the most conflicted queen, and the drawbacks of this costly decision outweigh the benefits.

So, there it is. We figured it out. We can finally stop counting solutions in roman numerals. Right? Yes, but an important question remains unanswered. For the keener amongst you, you'll have asked the question of where I got the number 10,000[6] from in

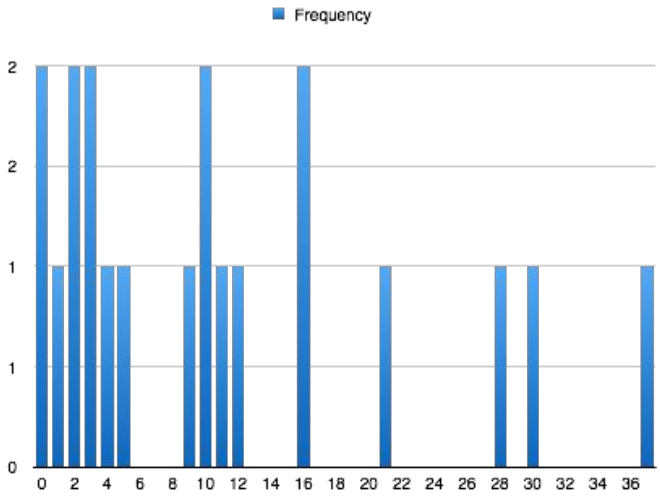
[6] The keenest will have read the previous footnote about it too.

the previous example. The next, and final, section deals with the origin story of this number.

The Art of Resetting

Let's remind ourselves why we reset: to escape local maxima. When the program has gone on too long in an attempt to solve the problem, but just isn't getting anywhere, that's when we need to go back to square one and try again. So, when the number of iterations has exceeded some threshold, our reset value, then we know it's time to reinitialize the board.

Of course, from this, we can immediately infer that the lower this reset value, the more often we'll see resets, and that the higher this reset value, the less often we'll see resets. Additionally, with a lower reset value, each reset will happen faster, whereas with a higher reset, each reset will take longer. As you can see, we have something of a balancing act. We want to reset often enough so that we don't have very long, time-consuming resets, but, at the same time, we don't want to reset so often that we have tons of resets, making solutions scarce to come by and dragging out time[7]. Herein lies the art of resetting.

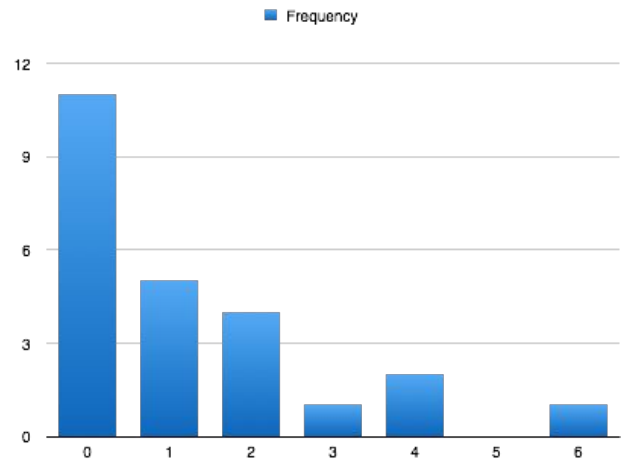


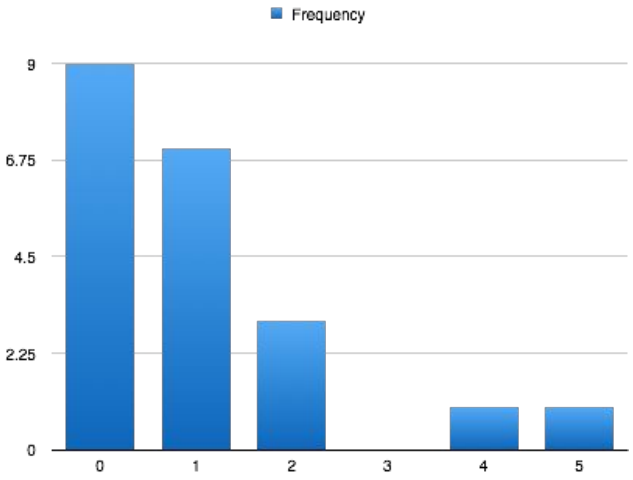
Reset Distributions

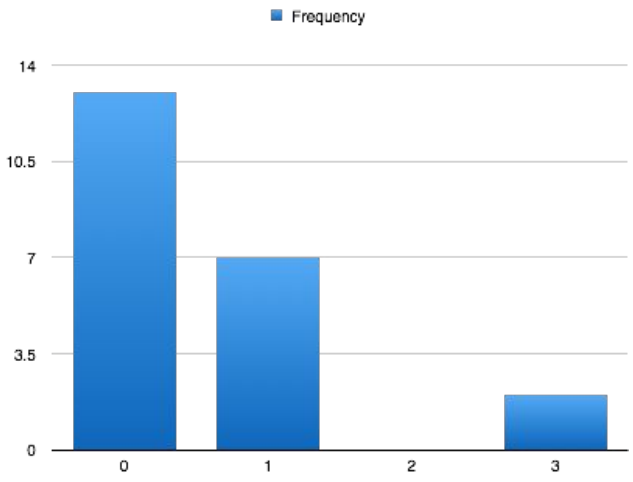
The following charts show the number of resets that occurred when solving a board of size 100[8].

In order from top to bottom, the first graph has a reset value of 1,000, the second has 10,000, the third has 50,000, and the fourth has 100,000.

[7] Ideally, we would like to minimize BOTH number of resets AND the time a reset takes
[8] It is to be assumed from here on out that we are always only checking the solution before each reset, as described in Solution VI, as it is the most optimal way.







Each one shows the distribution of frequencies for each number of possible resets. A sample of about 20 tests was done for each one in order to attain accurate results[9].

Looking at the first chart, where the reset value was set to 1,000, it becomes immediately obvious that there's a wide distribution. There does appear to be a slight right-skew, since there's a slightly higher density of occurrences of solutions appearing in less than four resets than above four.

The rightward skew that we're only starting to make out in the first chart only becomes more and more evident as we continue to go down the line and look at the other charts with higher reset values.

The second chart, with reset value of 10,000, has an obvious right-skew, with the majority of its solutions being found before even two resets. The trend continues, as we see the charts with reset values 50,000 and 100,000 have far more solutions appearing with zero and one resets than the others.

From these charts, we can prove our previously made made inference: the bigger the reset value, the less resets we see. If we could assume that all resets were made equally, then clearly we would always opt for the

[9] The data used to generate these charts can be found in Appendix B

highest possible reset value. However, this is not the case. The second piece of the puzzle is to inspect specific resets.

Measuring Reset Time

We have proved our inference that the higher the reset value, the less resets we will see. Now, let's proceed to prove our inference that the lower the reset value, the faster each reset will be. Below is table[10] composed of averages. The first two rows show the average number of steps and time it took to reach a solution, but only for solutions that were achieved with zero resets. And the second two rows show the same, but for when there was exactly one reset, no more, no less. This is an important concept to grasp. It's as if I'm taking the first two bars from each of the previously seen charts and finding the average time and number of steps it took to get to a solution.

Reset Value:	1,000	10,000	50,000	100,000
Steps (with 0 resets)	87.5	173.64	173.33	172.46
Time (with 0 resets)	0.22	0.40	0.83	1.37
Steps (with 1 reset)	200	474	1.22k	2.3k
Time (with 1 reset)	0.44	0.95	2.6	4.89

Looking at this table, it becomes clear that the lower the reset value, the faster the time. Now, with zero resets, we see that there actually isn't much variation in number of steps (with the first one as an exception), but this is to be expected, as if the problem is solved before the first reset, then the reset value shouldn't have an effect on its number of steps. But, then you ask, shouldn't all the times for zero resets also be similar then too? Not quite. Remember, we only check if we have a solution right before a reset. So although we find a solution within 172.46 steps on average with zero resets with a reset size of 100,000, we still need to iterate all the way up until 100,000 till we confirm this.

[10] The data used to generate this table can be found in Appendix B

But the conclusion from this table is clear; our intuition is proven correct once again; the smaller the reset value, the faster the reset.

Final Results

Now, let's do a straight-up side by side comparison of our different reset value levels and see which one is fastest in computing a board size of size 100. If your intuition is on par, you may be able to guess which reset value is ideal.

Reset Values:	1,000	10,000	50,000	100,000
Time	2.78	1.07	2.7	3.4

As you can see rather plainly, the reset value of 10,000 gives the fastest time. A lower reset value will give you a longer time, as will a higher reset value. The valuable part to observe here is *how* we arrived at 10,000. Yes, I had this piece of data all along, I just didn't show it until now. But, for good reason. Although a reset value of 100,000 offered less frequent resets on average, and although a reset value of 1,000 offered faster resets on average, we needed to balance the costs and benefits of scaling the reset value up or down. And therein lies the art.

With a board of size 100, how is 10,000 related to it? It is the square of the board size. I generalize this in my min-conflicts implementation, setting the reset value equal to the board size squared.

Conclusion

We made it. We first got our feet wet by diving into backtracking and exploring how 1) it worked and 2) how efficient it was on boards of varying sizes. We saw that boards of an even size appear to be far slower than boards of an odd size, and I encouraged that you

look more into this yourself. And then we jumped into min-conflicts, where we went through a list of six solutions, each time opting to find a better, faster, more efficient method, but always getting hit by a new problem or question, until we finally reached something that looked promising, at which point we had another question to answer: how do we determine the reset value?

When I first gave a description of min-conflicts it was vague. But these algorithms have vague descriptions for a reason; there are countless ways to implement them. There's always a better implementation right around the corner. You only need to ask the right questions. So, I urge you; don't let the solution presented in this paper be the end; it is merely a local maximum. I can promise you, a more efficient, elegant method lies at the summit of another peak; we are just to blind to see it from here.

Appendix A

Backtracking Algorithm Data:

Board Size	Number of Steps	Time
5	5	0
6	56	0
7	11	0
8	218	0
9	73	1
10	194	0
11	93	0
12	510	1
13	209	1
14	3784	11
15	2703	9
16	20088	81
17	10731	56
18	82580	424
19	5071	29
20	399250	2701
21	17103	118
22	3474354	31687
23	50833	447
24	823192	8837
25	97341	1010
26	795372	8954
27	908399	10362
28	6012568	75373
29	306449	40965
30	112859208	1650918

Appendix B

All relevant min-conflict data is supplied below. Any chart/table referenced in the paper can be reconstructed through the use of the data below.

Reset Value: 1,000

# of Steps	Time (seconds)	# of Resets
1.1k	2.5	9
3.5k	7.7	30
2.4k	5.3	21
4.7k	9.9	40
350	0.72	2
1.4k	2.8	11
1.2k	2.5	10
200	0.44	1
400	0.88	3
1.9k	4.0	16
300	0.68	2
1.9k	3.9	16
1.5k	3.0	12
300	0.65	2
86	0.20	0
460	0.93	3
89	0.23	0
720	1.5	5
3.2k	6.7	28
590	1.2	4
1.2k	2.7	10

Reset Value: 10,000

# of Steps	Time (seconds)	# of Resets
450	0.92	1
2.2k	3.8	6
430	0.88	1
310	0.57	0

1.3k	2.6	4
670	1.5	2
170	0.40	0
480	0.96	1
690	1.5	2
110	0.34	0
370	0.88	1
1.4k	2.7	4
250	0.45	0
640	1.1	1
120	0.34	0
210	0.43	0
120	0.33	0
690	1.3	2
160	0.38	0
680	1.3	2
100	0.32	0
140	0.36	0
220	0.43	0
1.2k	2.0	3

Reset Value: 50,000

# of Steps	**Time (seconds)**	**# of Resets**
90	0.74	0
1.2k	2.8	1
410	1.1	0
190	0.83	0
260	0.92	0
5.5k	9.6	5
1.2k	2.6	1
2.4k	4.4	2
1.2k	2.5	1
100	0.74	0
1.2k	2.5	1
2.5k	4.5	2
4.5k	7.9	4

1.1k	2.5	1
2.7k	4.8	2
130	0.81	0
1.3k	2.8	1
120	0.78	0
1.4k	2.7	1
120	0.77	0
140	0.78	0

Reset Value: 100,000

# of Steps	**Time (seconds)**	**# of Resets**
150	1.4	0
6.5k	11.6	3
230	1.5	0
2.5k	5.1	1
148	1.3	0
2.3k	4.8	1
2.2k	4.8	1
2.3k	4.9	1
90	1.4	0
87	1.3	0
2.4k	5.0	1
130	1.3	0
630	1.9	0
120	1.3	0
110	1.3	0
140	1.2	0
97	1.2	0
200	1.4	0
6.3k	11.5	3
2.2k	4.8	1
2.2k	4.8	1
110	1.3	0

Image Citations

"Queen Chess Piece Icon #047426." *Icons Etc*. N.p., n.d. Web. 02 Dec. 2015.

"Stock Photography: Search Royalty Free Images & Photos - IStock." *Stock Photography: Search Royalty Free Images & Photos - IStock*. N.p., n.d. Web. 02 Dec. 2015.