Sean Nelson (1, 3, 5, 7)
Michael Willson (2, 4, 6, 7)

A3: Arc Consistency Algorithms

1. Introduction

In this project we compared two arc consistency algorithms, AC-1 and AC-3, on the N-queens problem. The N-queens problem is when N queens are placed on an NxN chessboard such that no two queens threaten each other (no two queens share the same diagonal, row, or column). Arc consistency is a filtering process which allows us to eliminate nodes from a graph when they don't have support from other nodes. More specifically, a binary constraint is arc-consistent if every value of one variable has a value of the second variable such that they satisfy the constraint (every value has 'support'). AC-1 loops through the entire queue of arcs repeatedly, indicating when there is no support, until no change is found. AC-3, on the other hand, removes arcs from the queue with each iteration. When a change occurs, if any of the removed arcs could be possible supporters of other arcs in the queue, they are added back. The process repeats until the queue is empty. The two algorithms are described in more detail in the next section.

For N=4 to N=10, for each N we will generate at least 200 random D matrices, where the percentage, p, of one's varies from 0 to 1 in steps of 0.2, and answer the following questions:

- How does the expected reduction in labels compare to the starting number of labels for each N?
- What is the difference in execution time for AC-1 and AC-3?
- If there isn't a significant gain in performance using AC-3 over AC-1, what is a possible reason?

Our hypothesis is that, on average, AC-3 will perform significantly better than AC-1 for our trials.

2. Method

For this assignment, 4 functions were written in order to test our hypothesis. The 2 most important methods are the AC-1 and AC-3 functions, and we will start with the AC-1 algorithm.

AC-1

Our AC-1 takes in 3 inputs: a neighborhood logical graph G that represents which nodes have a relation to another, a domain matrix D which tells us the initial state of the possible values for the 4 queens and a string that names the predicate function. With these inputs, the AC-1 algorithm will determine if, and how, the domain can be reduced based on the consistency analysis done by the predicate function.

To begin, the AC-1 function runs through the entire G matrix and creates a list of variable index pairs to indicate the arcs that must be checked for consistency. Once this list is created, we then run a our consistency predicate on every possible input of each pair of variables in the list, taking care to only check values that are possible based on the domain provided. If an inconsistency is found, we change the appropriate cell in the domain from 1 to 0. Furthermore, once we have iterated through the entire list, we check if any inconsistencies have been found; and if so, we then initiate another run through the entire list and will continue to do so until no inconsistencies have been found, at which point the algorithm is done.

AC-3

Like the AC-1 algorithm, our AC-3 algorithm takes in 3 inputs: a neighborhood logical graph G, a domain matrix D and the name of our predicate function. Much like the AC-1 algorithm, our AC-3 algorithm uses these inputs to determine if, and how, the domain D can be reduced based on some consistency checking.

To begin, the AC-3 function runs through the entire G matrix to create a queue of pairs to indicate every possible arc. Once we have our queue, we will continually check the arcs on the queue, while the queue is not empty. To check each arc, we take off the top pair of variables on the queue and compare every possible value for each variable to check

for inconsistency. If an inconsistency is found, we then add back into the queue every arc that relies on the original first variable, except the one we just checked, and every other arc that the first variable relies on, as long as the queue is not empty and they are **not** currently in the queue currently. As said above, this process will continue until the queue is empty, at which time the domain will be reduced.

Predicate Function (CS4300_P_No_Attack)

Our predicate function turned out to be a relatively easy one to implement. It took in 4 arguments: the index of both variables and the values of both variable. With this, we determined consistency of the first variable by checking that both values were not equal (i.e. the queens could not be in the same row) and that the distance between the values were not equal to the distance between the indices of the variables (i.e. the queens could not be in the same diagonal). It would return 1 if the values and indies were consistent with the restrictions or 0 if not.

Run Trials Function (CS4300_A3_Run_Trials)

To run our trials for the number of queens N = 4:10 where the percentage, p, of one's varies from 0 to 1 in steps of 0.2, we created a nested for loop where in the innermost loop we would run our AC-1 and AC-3 on a randomly generated domain matrix D with the determined percentage, p, of one's. With each individual trial, we recorded the number of 1's in D both before and after it had been revised as well as the time it took for AC-1 and AC-3 to complete using the tic/toc methods provided in Matlab. Also, to determine the means for each N, at the end of each loop for the N, we would take an average for all the times take for that N and put it in it's own array for both AC-1 and AC-3. Once done, our function would then return both the list of raw data and the list of average times.

3. Verification

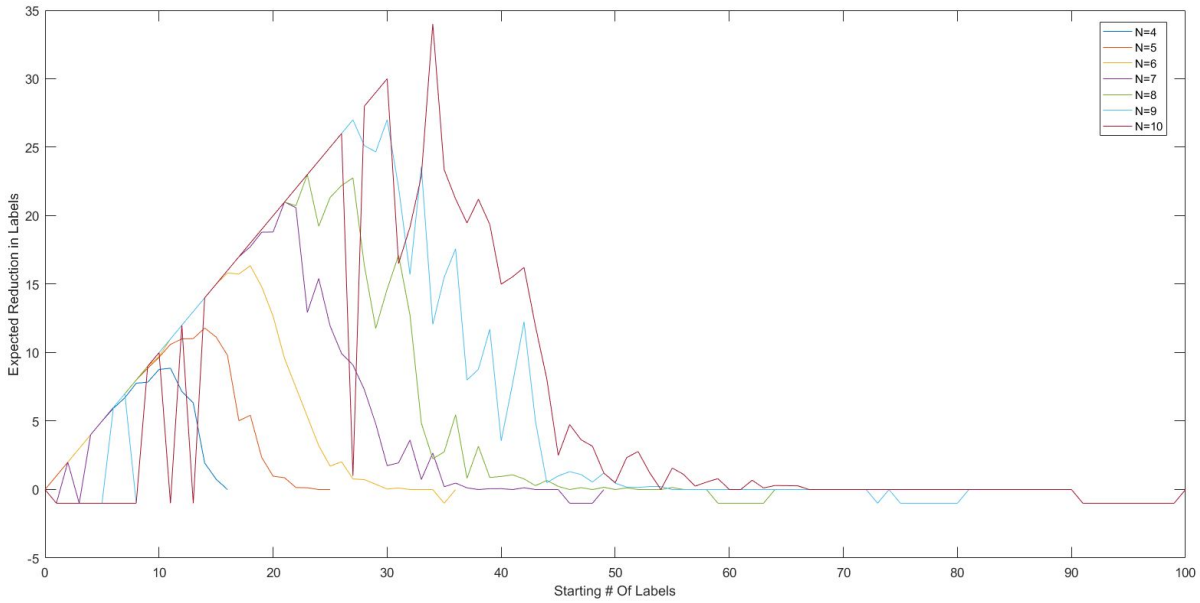
To verify that our arc consistency algorithms were working correctly, for each N = 4:10 we applied the arc consistency algorithm by hand for a random matrix of 0's and 1's (seven matrices total). Then we compared these results to the results produced by our functions. For the sake of brevity, we'll only look at one of those matrices and explain how the algorithm was applied by hand. Let D = [0 1 0 1; 1 1 1 1; 1 0 0 0; 1 1 1 1]. A location will be represented as '(row,column)' where indices start at 1.

We'll start with the first row. We can skip any 0's so we move on to (1,2). It's supported by (2,4), (3,1), and three different locations on row 4 so it remains unchanged. The next location to check is (1,4). It's unchanged because it's supported by (2,1), (3,1), and (4,1). Moving on to the next row, we check (2,1). It's unsupported because (3,1) is the only usable space on row 3 and it would threaten the queen. So, we change the value to a 0. When we use AC-1 we would just move on here and check against other locations the next time around. Intuitively, we can do something more like AC-3 and check against possible other locations that could be affected by this change. Locations (1,2) and (1,4) are still supported so we move on. (2,2) isn't supported either for the same reason as (2,1). We change it to a 0 and find that none of our previous 1's were affected. (2,3) is threatened by (1,2) and (1,4) so it's not supported. None of the previous 1's were affected by the change. Location (2,4) is supported by (1,2), (3,1), and (4,1) so we can move on. Now that we've looked at two rows, we repeat this general process for the next two rows and find the resulting D matrix to be [0 1 0 0; 0 0 0 1; 1 0 0 0; 0 0 1 0]. Our functions for AC-1 and AC-3 produced the same result.

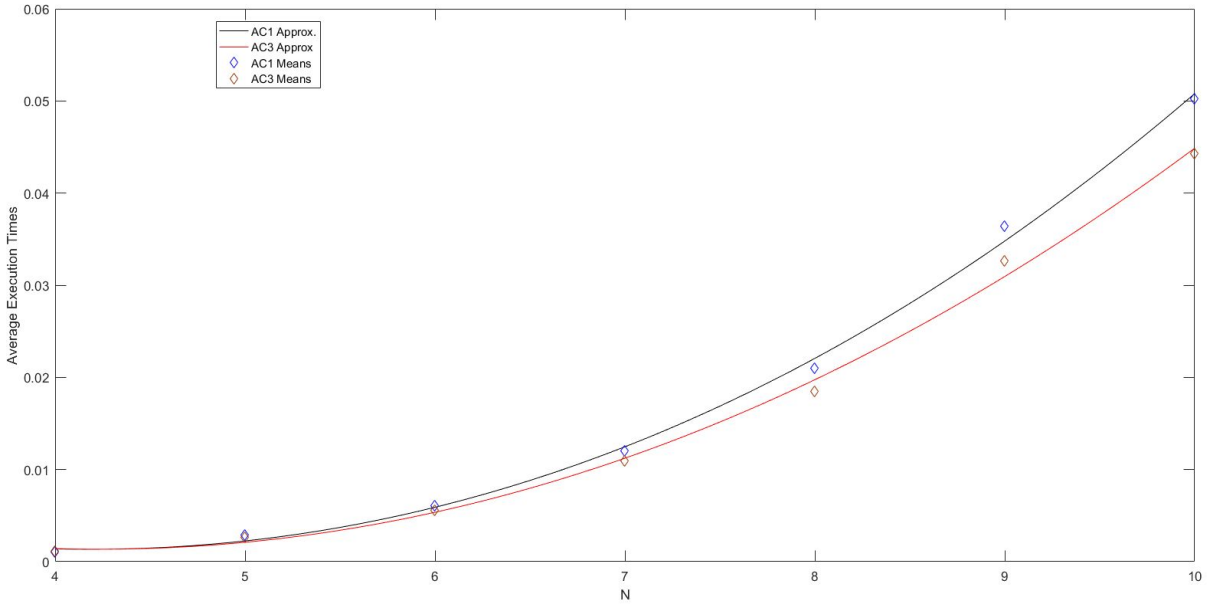
It's also important to note that the method of how our functions work is consistent with how we get the resulting D matrix by hand. For AC-1 we simply move on and don't bother looking at previously checked locations until the next iteration starting at (1,1). For AC-3 we take possibly affected locations into account. Again, the details of the algorithms are described in section 2. We found our functions to produce exactly the same results for all N = 4:10 we tested by hand.

4. Data and Analysis

Below is a graph of the expected value reduction for any domain of 1's ranging from 0 to N*N, or in our case 100.



Next is the graph of the average times of AC-1 and AC-3, as well as their approximate functions, for N ranging from 4 to 10.



Their approximate functions (based on the polyfit function in matlab) are as follows:
AC-1 - $f(n) = 0.00002n^3 + 0.0011n^2 - 0.0103n + 0.0235$
AC-3 - $f(n) = 0.000006n^3 + 0.0012n^2 - 0.0104n + 0.0236$

5. Interpretation

To answer our first question, comparing the expected reduction in labels to the starting number of labels for each N, we examine the first graph above. To generate this graph, the starting number of 1's was calculated during our 'run trials' function. The expected reduction in the number of of labels was calculated by taking the average change in the number of 1's for each N. The graph gives us interesting insight on how arc consistency algorithms work. We'll look specifically at two N's from the graph: 4 and 5. Both of the N's (and all of the N's) start and end with 0 expected reduction in labels. This makes sense because for a starting matrix D of entirely 0's or entirely 1's the board doesn't change. The most change occurs at about 12 starting labels for N=4 and about 15 starting labels for N=5. We expected the reduction in labels to change at about the halfway mark (8 starting labels and 12-13 starting labels respectively), but this isn't the

case. For all of N's above you'll notice the lines peak past their halfway points. The general trend is about 60-70% of the of the number of labels instead of our expected 50%. In summation, to see the maximum amount of filtering (by arc consistency) before a backtrack search, a starting matrix D would need to have **more** than half of the values as 1's.

To answer our second question above, comparing execution time for AC-1 and AC-3, we examine the second graph above. The specific times for each N and arc consistency algorithm are as follows:

- N = 4; AC-1 = 0.001066070703; AC-3 = 0.001119491518
- N = 5; AC-1 = 0.002863946021; AC-3 = 0.002649295383
- N = 6; AC-1 = 0.006091812405; AC-3 = 0.00557161148
- N = 7; AC-1 = 0.01202419909; AC-3 = 0.0109036352
- N = 8; AC-1 = 0.02097175102; AC-3 = 0.01848745263
- N = 9; AC-1 = 0.0364338184; AC-3 = 0.03267024974
- N = 10; AC-1 = 0.05021534697; AC-3 = 0.04428269987

For each N, AC-3 performs better than AC-1 with the exception of N=4. This could be related to the setup for AC-3 which requires removing/adding from the queue, an additional check for possible supporters, etc. We can also look at the 95% confidence interval for N=4 to verify that it may have just been an unusual case:

- AC-1 95% CI = (0.00099269, 0.0011)
- AC-3 95% CI = (0.0011, 0.0012)

It could just be a matter of precision, but based on the confidence intervals and what we can see visually on the graph, for N=4 the execution times are almost identical. AC-1 might be very slightly better in this case but, as mentioned above, that's probably just related to AC-3's algorithm setup. We can get a better comparison of the two algorithms by comparing N = 5:10.

Looking at the graph starting at around N=7 we can definitely see that AC-3 is performing better than AC-1. But is that difference significant? The gap between the two lines grows and N increases. In order to see the true benefits of AC-3 over AC-3 we would want to look at cases where N=20 or even N=50. Then, we would definitely see a significant improvement. Thus, our initial hypothesis is correct for a larger value of N.

6. Critique

A3 seemed more straightforward. Both algorithms seemed to be relatively easy to understand and implement, bar a little confusion about how to add back in the arcs into the queue, but a quick check against the AC-3 wikipedia page cleared that up. For me personally, as Sean seemed to figure this out much faster than I did, my biggest stumbling block was understanding that when checking for consistency, we were only supposed to check 2 queen variables at a time and not check the entire board at once; however, after the example given on thursday (9/15/2016), my misunderstanding was quickly corrected.

The only other issue we had was with A3 was the method header declarations that we had to use to program for. The problem which we faced (which we also had in A2) was that the inputs for the AC-1 and AC-3 functions were a little too vague. Sean and I discussed what G was supposed to be at length and still were a little fuzzy on how to use it. If we are to use certain inputs for our functions then we would suggest a little more detail be given about the inputs and how they relate to the algorithm, i.e. G (nxn array): neighborhood graph for n nodes to be used to determine which arcs to test.

7. Log

Sean - 16 hours

- 12 hours programming
    - 5 hours analyzing/discussing the project with Mike
    - 4 hours coding
    - 4 hours debugging/testing
- 3 hours lab write up

Michael - 18 hours

- 15 hours programming
    - 5 hours analyzing/discussing with Sean how the AC-1 & AC-3 algorithms should be written in relation to the N queens problem
    - 6 hours coding
    - 4 hours debugging/testing
- 3 hours working on the lab write up