A3 - AC-1/AC-3 Algorithms

CS 4300
Fall 2016
Leland Stenquist - 1,3,5
Ryan Keepers - 2,4,6

1. Introduction

The intent of this experiment is to explore the differences between two similar constraint satisfaction problems. These are namely AC-1 and AC-3 where AC-3 is hopefully an optimized version of the AC algorithm. To compare these two algorithms the N-Queens problem was used. This is given an NxN board, can you fit N queens onto that board in such a way that none of the queens could attack each other. Comparisons were made for a N from 4 to 10. Random boards were generated where each cell had a probability, P, of having a 1 in it. P was incremented from 0 to 1 by steps of 0.2. For each P and N combination the experiment was run 200 times.

This experiment was meant to answer the following questions:

1. How does the AC-1 algorithm compare to the AC-3 algorithm in efficiency?
2. For a given starting number of ones, what would be the expected reduction in ones for each N?
3. Can a linear regression model be used to fit this data? What is that model?
4. Is one of the algorithms, AC-1 or AC-3, better than the other?

2. Method

The purpose of the AC1 and AC3 algorithms is to take in a starting state on a given graph of connected components (connections are called Arcs, in this case) and remove from that state any situation which is totally nonviable. For the scope of this assignment the algorithms are run on an N-Queens problem. The graph of connected components represents the rows on the board, with the graph being fully dense except in the case of components referencing themselves. The starting state is a vector containing domains of values dictating the population of queens within each row. At the end of the algorithms, the resultant set will contain only a population of queens which are capable of producing a viable result.

The mechanisms which determine whether an element is viable or not are the CS4300_REVISE function and a predicate evaluation function, CS4300_n_queens_predicate in this case. The Revise function takes in two domains *A* and *B*, and iterates through pairs of populated queens (i,j) where $i \in A | i = 1$ and $j \in B | j = 1$, sending each pair into the predicate function. The predicate function returns a boolean value stating whether the pair conflict (ie, the queens would attack each other), or provide support (ie, would not attack). For any given queen $i \in A$, if no $j \in B$ provides support, then the queen is deemed non-viable and is removed from the domain. At the end of the Revise function, the potentially modified domain is returned.

AC1 and AC3 are largely similar algorithms. Each one follows the following general behavior: populate a queue with all arcs in G, iterate over the queue until the state of D no longer changes, then return the potentially revised set of domains. Both algorithms, during this time, call the same Revise and predicate functions to resolve conflicts and revise D.

The critical difference of each algorithm involves the reduction of operations as D becomes modified by the Revise function. If any change occurs to D during AC1, at the end of its iteration through the queue of arcs it will start again at the top of the queue and resume Revising. This continues until D sees no change in a full iteration on the queue.

AC3 will pop each arc off the queue through its iteration. In the event that Revise returns a modified D, it will push a small subset of arcs onto the top of the queue and revisit them. The arcs which are reintroduced must not already be present in the queue, must not match the rows which caused the revision, and could be affected by the change in the domain. When the AC3 has fully exhausted the queue the algorithm is finished.

3. Verification of Program

Verification of this program was done in two ways.

1. A simple simulation was run by hand and compared to the results of the program. This simulation runs by (row, col) and will only take into account situations where a change happened.

AC-1			
Manual		**Program**	
G = 0 1 1 1 1 0 1 1 1 1 0 1 1 1 1 0 (Row, Col)	D = 0 1 0 0 0 0 1 1 1 0 0 0 0 0 1 0 CHANGE = false	G = 0 1 1 1 1 0 1 1 1 1 0 1 1 1 1 0 (Row, Col)	D = 0 1 0 0 0 0 1 1 1 0 0 0 0 0 1 0 CHANGE = false
(1,2),(1,3),(1,4)	No Changes CHANGE = false	(1,1) - (1, 4)	No Changes CHANGE = false
(2,1)	2 => (0,0,1,1) 1 => (0,1,0,0) For 2 at index 3 there is no support so we replace that 1 with a 0. D = 0 1 0 0 0 0 0 1 1 0 0 0 0 0 1 0 CHANGE = true	(2,1)	2 => (0,0,1,1) 1 => (0,1,0,0) For 2 at index 3 there is no support so we replace that 1 with a 0. D = 0 1 0 0 0 0 0 1 1 0 0 0 0 0 1 0 CHANGE = true
(2,3),(2,4),(3,1),(3,2),(3,4),(4,1),(4,2),(4,3)	N0 Changes CHANGE = true	(2,2)-(4,4)	N0 Changes CHANGE = true
Repeat	CHANGE = false	*Repeat*	CHANGE = false
(1,2),(1,3),(1,4),(2,1),(2,3),(2,4),(3,1),(3,2),(3,4),(4,1),(4,2),(4,3)	No Changes CHANGE = false	(1,1)-(4,4)	No Changes CHANGE = false
Complete	D = 0 1 0 0 0 0 0 1 1 0 0 0 0 0 1 0	*Complete*	D = 0 1 0 0 0 0 0 1 1 0 0 0 0 0 1 0
AC-3			
Manual		**Program**	

G = 0 1 1 1 1 0 1 1 1 1 0 1 1 1 1 0 (Row, Col)	D = 0 1 0 0 0 0 1 1 1 0 0 0 0 0 1 0 CHANGE = false	G = 0 1 1 1 1 0 1 1 1 1 0 1 1 1 1 0 (Row, Col)	D = 0 1 0 0 0 0 1 1 1 0 0 0 0 0 1 0 CHANGE = false
Queue = [[1,2] [1,3] [1,4] [2,1] [2,3] [2,4] [3,1] [3,2] [3,4] [4,1] [4,2] [4,3]]	Create a queue	Queue = [[1,2] [1,3] [1,4] [2,1] [2,3] [2,4] [3,1] [3,2] [3,4] [4,1] [4,2] [4,3]]	Create a queue
De-Queue = [1,2],[1,3],[1,4],[2,1]	There were no revisions		
Queue = [[2,3] [2,4] [3,1] [3,2] [3,4] [4,1] [4,2] [4,3]]	2 => (0,0,1,1) 1 => (0,1,0,0) Revision at (2,3) because of the queen at (1,2). D = 0 1 0 0 0 0 0 1 1 0 0 0 0 0 1 0	Que at revision Queue = [[2,3] [2,4] [3,1] [3,2] [3,4] [4,1] [4,2] [4,3]]	2 => (0,0,1,1) 1 => (0,1,0,0) Revision at (2,3) because of the queen at (1,2). D = 0 1 0 0 0 0 0 1 1 0 0 0 0 0 1 0
En-Queue Queue = [[2,3] [2,4] [3,1] [3,2] [3,4] [4,1] [4,2] [4,3]]	See if we need to add arcs back to the queue. No new arcs are added.	Que after we en-queue En-Queue Queue = [[2,3] [2,4] [3,1] [3,2] [3,4] [4,1] [4,2]	See if we need to add arcs back to the queue. No new arcs are added.

		[4,3]]	
De-Queue = [1,2],[1,3],[1,4],[2,1] [2,3],[2,4],[3,1],[3,2] [3,4],[4,1],[4,2],[4,3]	D = 0 1 0 0 0 0 0 1 1 0 0 0 0 0 1 0	Queue = []	D = 0 1 0 0 0 0 0 1 1 0 0 0 0 0 1 0

2. Different scenarios with clear solutions were tested and compared to the program's solutions.

Input Domain	**Expected Output Domain**	**AC1 Output Domain**	**AC3 Output Domain**

[1 1 1 1;
1 1 1 1;
1 1 1 1;
1 1 1 1]
	[1 1 1 1;
	1 1 1 1;
	1 1 1 1;
	1 1 1 1]
		ans =
		
		1 1 1 1
		1 1 1 1
		1 1 1 1
		1 1 1 1
			ans =
			
			1 1 1 1
			1 1 1 1
			1 1 1 1
			1 1 1 1

[1 1 1 1;
1 0 0 0;
1 1 1 1;
1 1 1 1]
	[0 0 1 0;
	1 0 0 0;
	0 0 0 1;
	0 1 0 0]
		ans =
		
		0 0 1 0
		1 0 0 0
		0 0 0 1
		0 1 0 0
			ans =
			
			0 0 1 0
			1 0 0 0
			0 0 0 1
			0 1 0 0

[1 0 0 0;
1 1 1 1;
1 1 1 1;
1 1 1 1]
	[0 0 0 0;
	0 0 0 0;
	0 0 0 0;
	0 0 0 0]
		ans =
		
		0 0 0 0
		0 0 0 0
		0 0 0 0
		0 0 0 0
			ans =
			
			0 0 0 0
			0 0 0 0
			0 0 0 0
			0 0 0 0

[0 1 0 0;
1 1 1 1;
1 1 1 1;
1 1 1 1]
	[0 1 0 0;
	0 0 0 1;
	1 0 0 0;
	0 0 1 0]
		ans =
		
		0 1 0 0
		0 0 0 1
		1 0 0 0
		0 0 1 0
			ans =
			
			0 1 0 0
			0 0 0 1
			1 0 0 0
			0 0 1 0

[0 1 0 0 0;
1 1 1 1 1;
1 1 1 1 1;
1 1 1 1 1;
	[0 1 0 0 0;
	0 0 0 1 1;
	1 0 1 0 0;
	1 0 1 0 0;
		ans =
		
		0 1 0 0 0
		0 0 0 1 1
			ans =
			
			0 1 0 0 0
			0 0 0 1 1

1 1 1 1 1]	0 0 0 1 1]	1 0 1 0 0 1 0 1 0 0 0 0 0 1 1	1 0 1 0 0 1 0 1 0 0 0 0 0 1 1

[1 0 0 0 0 0 0 0 0 0;
1 1 1 1 1 1 1 1 1 1;
1 1 1 1 1 1 1 1 1 1;
1 1 1 1 1 1 1 1 1 1;
1 1 1 1 1 1 1 1 1 1;
1 1 1 1 1 1 1 1 1 1;
1 1 1 1 1 1 1 1 1 1;
1 1 1 1 1 1 1 1 1 1;
1 1 1 1 1 1 1 1 1 1;
1 1 1 1 1 1 1 1 1 1;]
	[1 0 0 0 0 0 0 0 0 0;
	0 0 1 1 1 1 1 1 1 1;
	0 1 0 1 1 1 1 1 1 1;
	0 1 1 0 1 1 1 1 1 1;
	0 1 1 1 0 1 1 1 1 1;
	0 1 1 1 1 0 1 1 1 1;
	0 1 1 1 1 1 0 1 1 1;
	0 1 1 1 1 1 1 0 1 1;
	0 1 1 1 1 1 1 1 0 1;
	0 1 1 1 1 1 1 1 1 0;]
		ans =
		
		1 0 0 0 0 0 0 0 0 0
		0 0 1 1 1 1 1 1 1 1
		0 1 0 1 1 1 1 1 1 1
		0 1 1 0 1 1 1 1 1 1
		0 1 1 1 0 1 1 1 1 1
		0 1 1 1 1 0 1 1 1 1
		0 1 1 1 1 1 0 1 1 1
		0 1 1 1 1 1 1 0 1 1
		0 1 1 1 1 1 1 1 0 1
		0 1 1 1 1 1 1 1 1 0
			ans =
			
			1 0 0 0 0 0 0 0 0 0
			0 0 1 1 1 1 1 1 1 1
			0 1 0 1 1 1 1 1 1 1
			0 1 1 0 1 1 1 1 1 1
			0 1 1 1 0 1 1 1 1 1
			0 1 1 1 1 0 1 1 1 1
			0 1 1 1 1 1 0 1 1 1
			0 1 1 1 1 1 1 0 1 1
			0 1 1 1 1 1 1 1 0 1
			0 1 1 1 1 1 1 1 1 0

4. Data and Analysis

1 reduction of 1's
2 linear regression for

Table 1
Average AC1 run time in milliseconds

0.0477	0.1067	0.2142	0.5342	1.0947	0.9609
0.0791	0.1977	0.464	1.3546	1.9941	1.7532
0.122	0.3086	0.7874	2.6116	2.7194	2.9036
0.1741	0.3773	1.6962	4.4146	4.0371	4.5539
0.2818	0.8094	5.1865	6.0522	5.9582	6.2898
0.2896	0.7489	5.2122	6.8453	7.2767	9.281
0.4535	1.5833	10.1664	8.3263	9.7729	13.4998

Table 2
Average AC3 run time in milliseconds

0.0475	0.1056	0.2299	0.4622	0.7632	0.951
0.0735	0.179	0.4396	0.995	1.4232	1.7319

0.1117	0.2866	0.7912	1.7658	2.3148	2.8624
0.1552	0.3155	1.4194	2.8347	3.6337	4.5129
0.243	0.7296	3.8169	4.4223	5.7408	6.1995
0.2506	0.6395	3.3117	5.4958	7.0738	9.1342
0.3917	1.3468	6.1962	7.2658	9.6206	13.3511

Table 3
Difference in average run time (AC1-AC3)

0.0002	0.0011	-0.0157	0.072	0.3315	0.0098
0.0055	0.0187	0.0243	0.3596	0.5708	0.0213
0.0104	0.0221	-0.0038	0.8458	0.4046	0.0412
0.0189	0.0618	0.2768	1.5799	0.4035	0.041
0.0388	0.0798	1.3696	1.6299	0.2173	0.0904
0.0391	0.1094	1.9005	1.3495	0.203	0.1468
0.0618	0.2365	3.9702	1.0605	0.1523	0.1487

Table 4
Mean number of 1’s generated

0	3.18	6.38	9.7	12.985	16
0	5.295	9.91	15.08	19.955	25
0	7.455	14.325	21.785	28.89	36
0	9.35	19.475	29.12	38.83	49
0	13.285	25.525	38.85	50.88	64
0	15.84	32.035	48.44	64.915	81
0	20.08	40.09	59.64	80.115	100

Table 5
Mean number of 1’s after AC algorithms

0	0	0.22	1.92	8.075	16
0	0.095	0.75	5.64	17.14	25

0	0	0.54	13.115	28.025	36
0	0	2.725	23.21	38.525	49
0	0	7.535	36.05	50.815	64
0	0	13.44	47.095	64.895	81
0	0.12	25.195	58.915	80.105	100

Table 6
Least Squares Regression:
Board size: 4

Board size: 4			
	P1	P2	P3
AC1	1.116	0.767	-0.017
AC3	1.276	0.365	.044

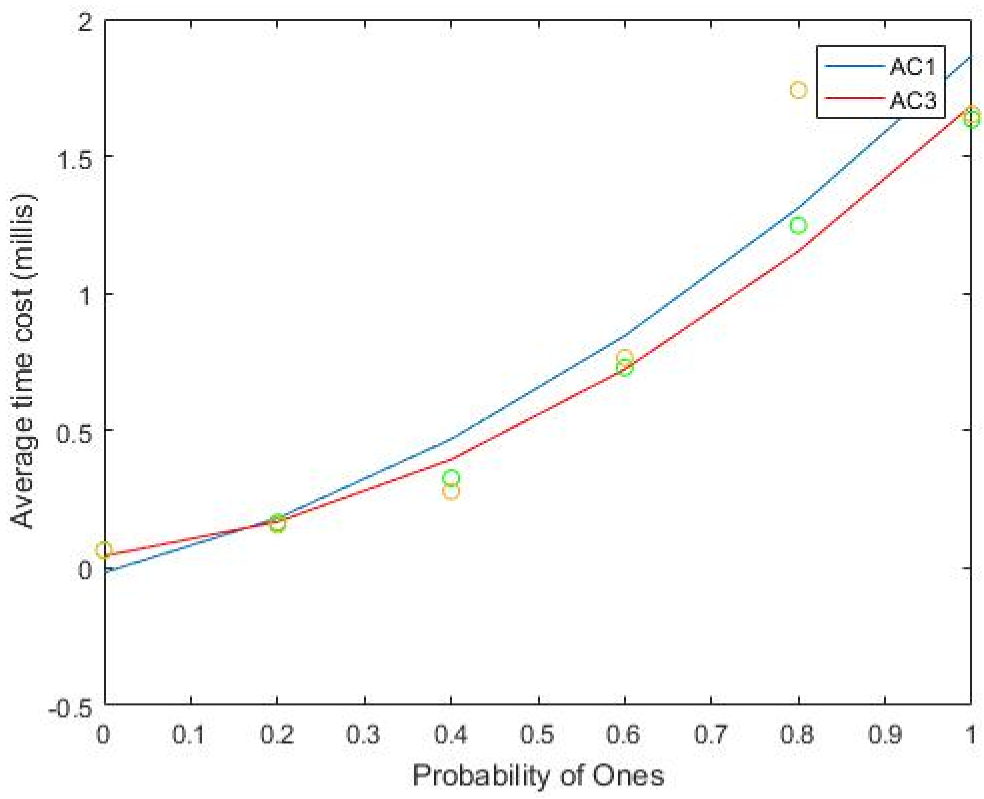


Table 7

Board size: 5			
	P1	P2	P3
AC1	0.274	3.455	-0.190
AC3	1.782	1.376	0.003

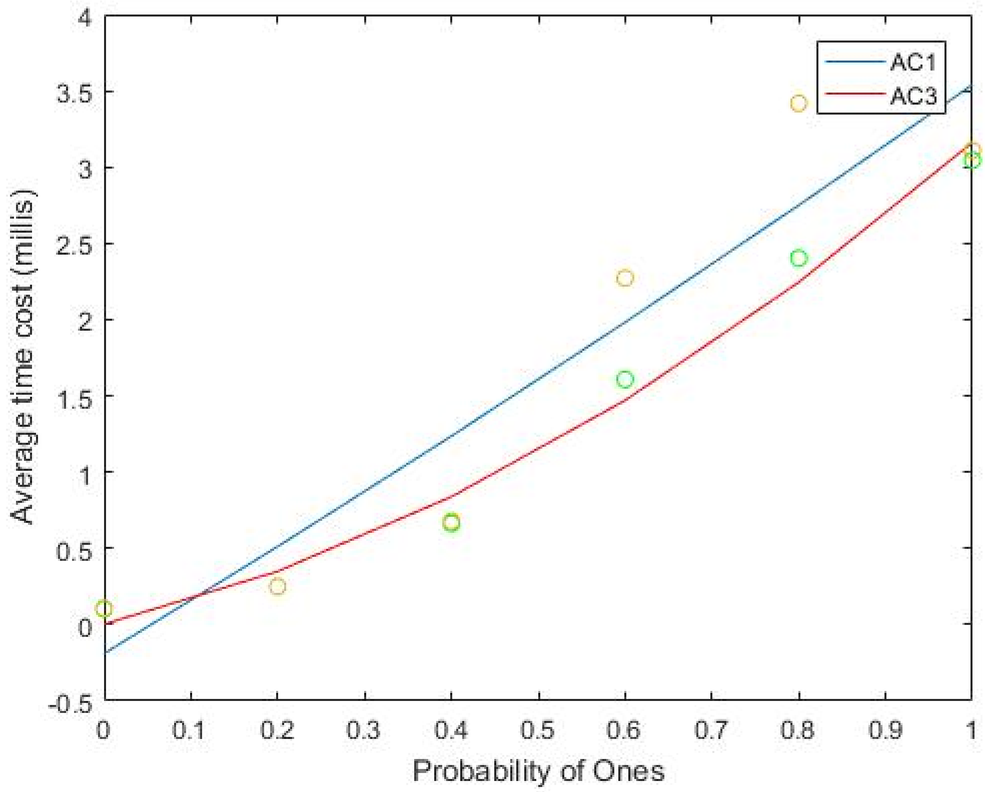


Table 8

Board size: 6			
	P1	P2	P3
AC1	-3.207	9.159	-0.478
AC3	0.2355	5.158	-0.184

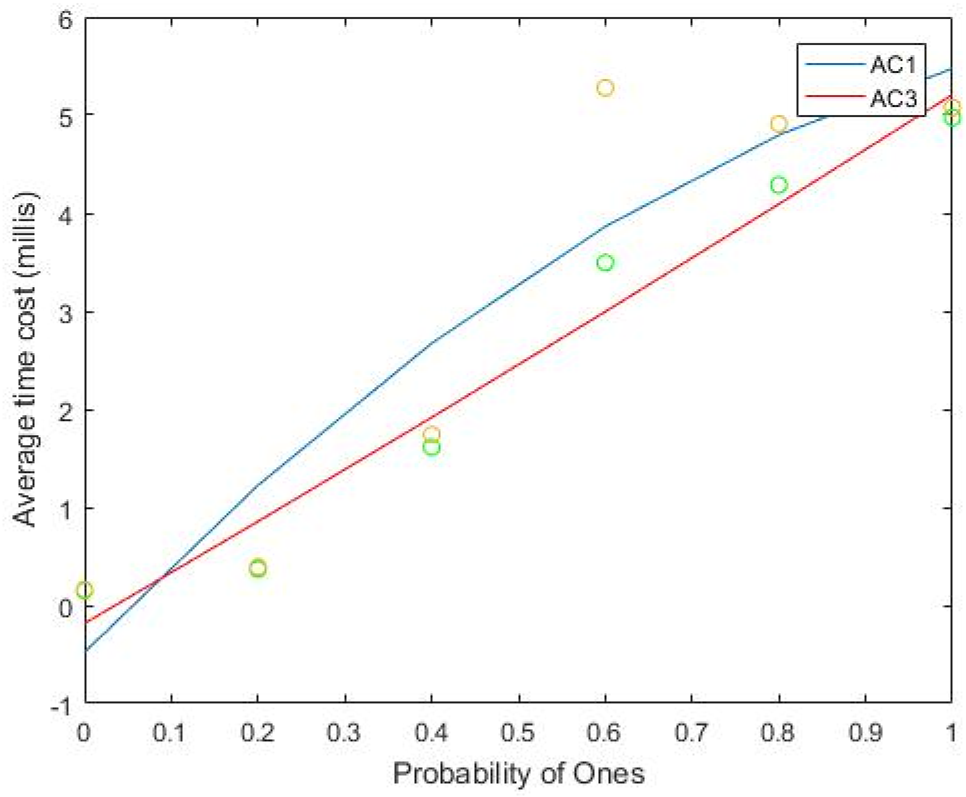


Table 9

Board size: 7			
	P1	P2	P3
AC1	-3.923	12.213	-0.577
AC3	1.539	6.215	-0.191

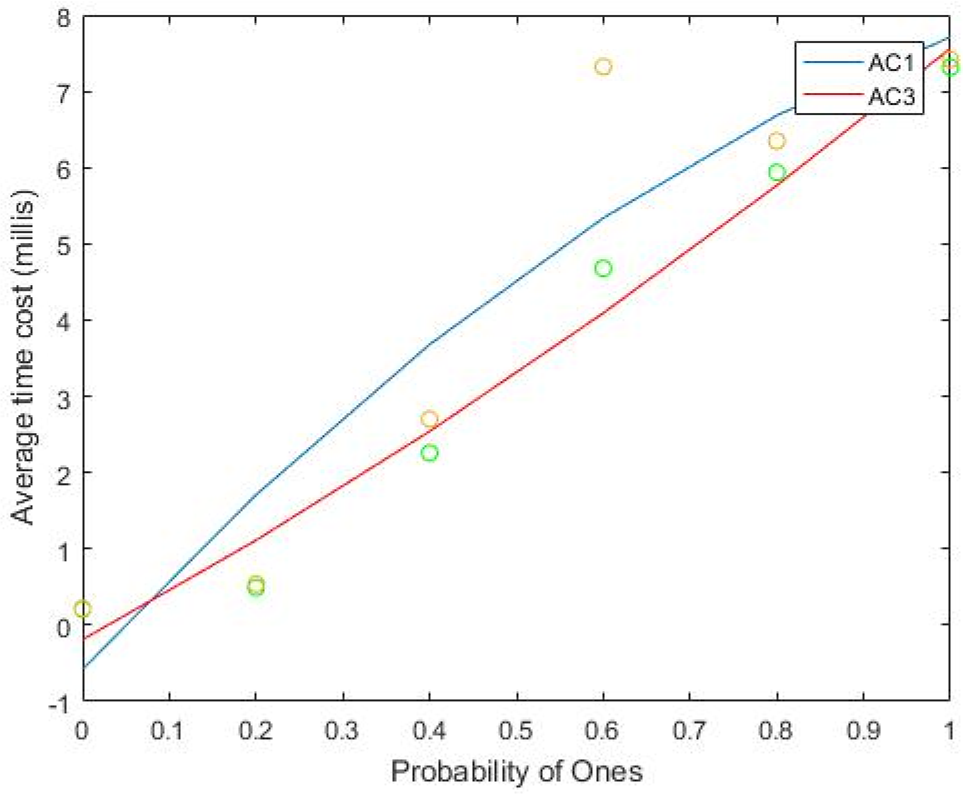


Table 10

Board size: 8			
	P1	P2	P3
AC1	-6.082	17.530	-0.670
AC3	1.116	9.939	-0.306

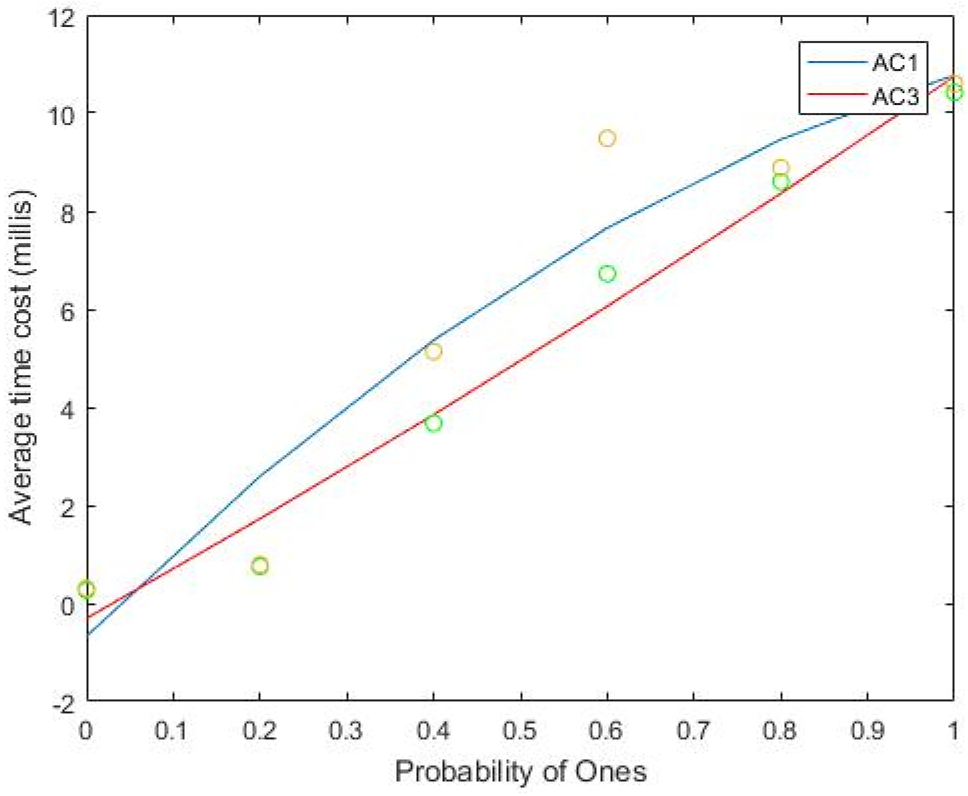


Table 11

Board size: 9			
	P1	P2	P3
AC1	-8.847	24.472	-0.732
AC3	1.483	14.071	-0.433

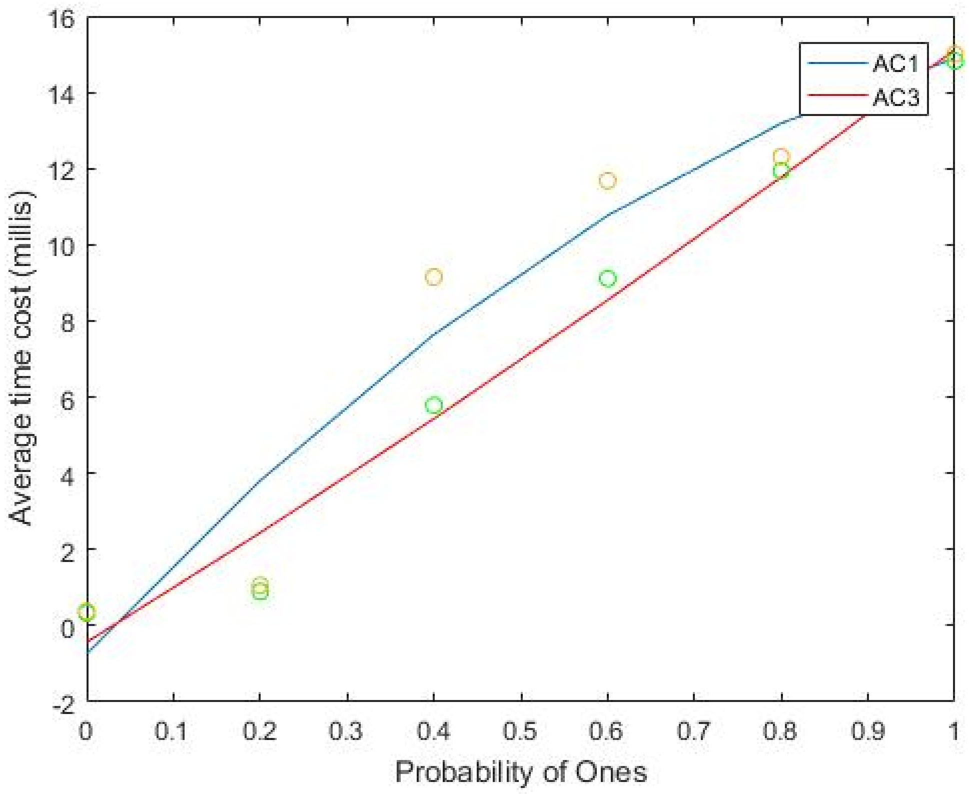


Table 12

Board size: 10			
	P1	P2	P3
AC1	-9.411	30.688	-0.975
AC3	0.915	20.674	-0.8043

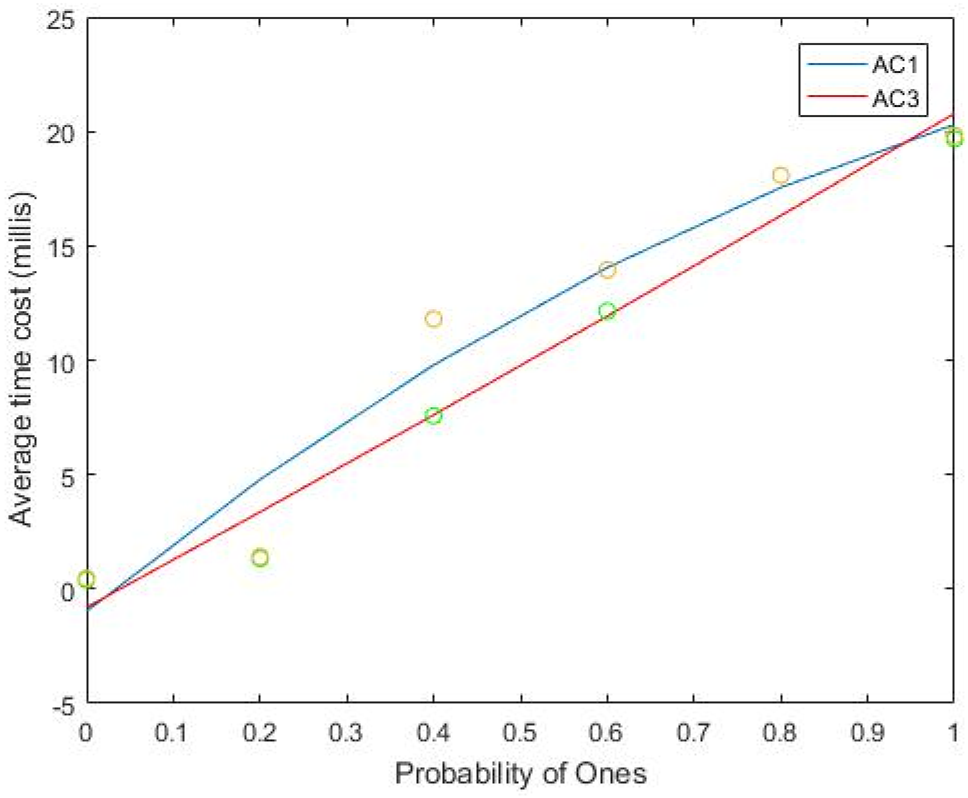
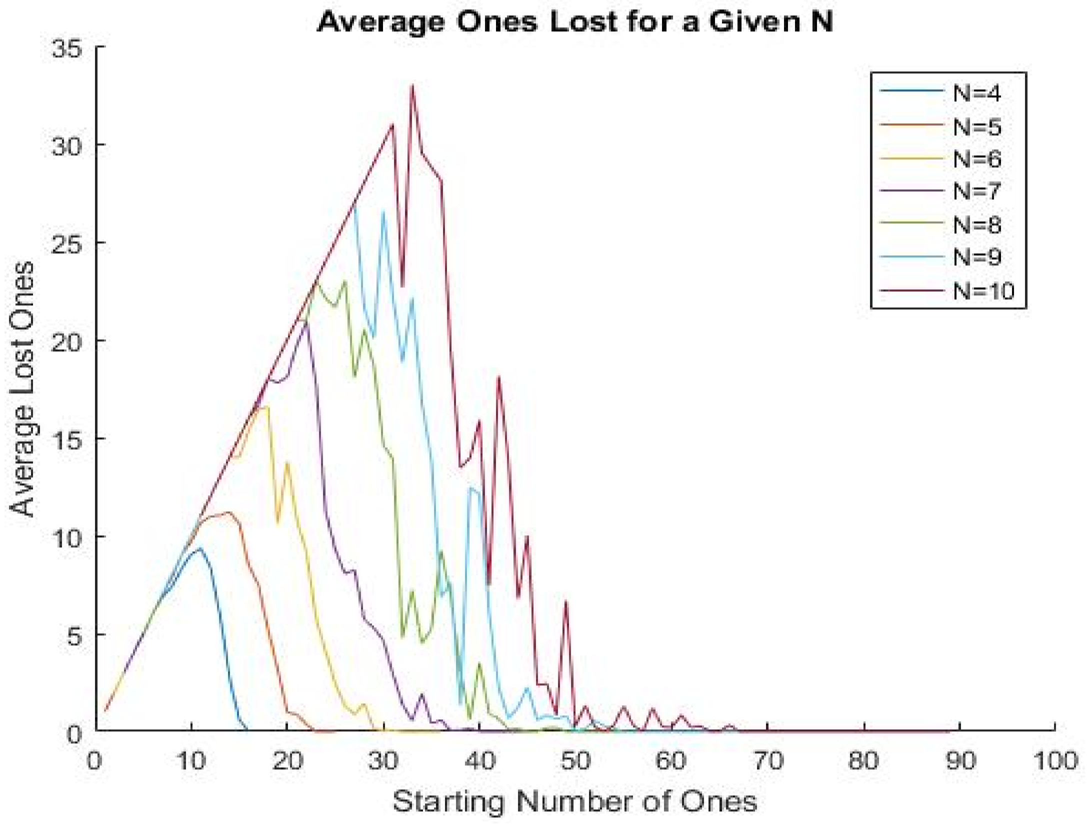


Table 13



5. Interpretation

Reduction in Ones

There were very interesting trends in the statistics for the reductions in ones. Of course when P=0.0 and P=1.0 the reduction in ones is zero all around. As P increases between 0 and 1 the reduction in ones drastically decreases. This is to be expected. If there are many ones it there are more options for support and therefore less reductions. It is interesting that for a percentage P=0.2 as N increases the reduction in ones increases drastically from ~3 to ~20. For P=0.8 though the reduction in ones actually decreases as N increases, but not nearly so drastically from ~4 to ~0.

The interpretation of this seems that Domains with fewer ones will result in more one reductions. It could be hypothesised that this is because for domains with fewer ones each one is less likely to have support. Whereas if there are many ones each one is very likely to have support.

Algorithm Run Times

According to the experiments run, the AC-3 algorithm is, in general, faster. This becomes extremely apparent in at the case where P=0.4 and N=10. In this case AC-3 run time averaged 3.97 second faster. This is by far the greatest difference in average run times. None of the other average run times breaks 2 seconds. This seems to indicate that AC-3 is better in most circumstances, but it shows the most improvement over AC-1 as N increases where P is around 0.4. The conclusion is that AC-3 is better than AC-1.

6. Critique

The implementation of AC1 and AC3, within the scope of the assignment, most importantly involved critical investigations into clever optimizations on otherwise mundane behaviors. The purpose of the comparison was to track the efficiency of AC1 relative to AC3. However, in our initial naive designs, AC3 actually ended up more costly for most board sizes. It wasn't until we picked over the algorithm for smaller optimizations that we found AC3 to be faster.

On one hand, this is the result of operating the algorithms on boards which are (I assume to be) relatively small. A 4x4 matrix will be highly expedient for a computer. Within such small board sizes we found that AC3 was spending extra computation on issues like misuse of preallocated memory which led to some thrashing. However, AC1 operated solely within a singular cache space. The additional overhead on AC3 is ultimately what ended up putting it behind AC1.

Of course, we also came across many optimizations which benefit the total run time of the analysis as a whole. For example, our naive Revise method would check the entire domain for the sum of support. A simple, but highly effective change was to continue onto the next iteration as soon as any support was discovered for a given element.

This attention to optimization, we feel, represents the purpose of learning both algorithms. The high-level intent of AC3 sounds immediately reasonable (eg, reduce computation by minimizing unnecessary behavior), yet at the same time the implementation is not immediately obvious. It requires a thorough understanding of the components in use to appreciate.

7. Log

Ryan : 14.5 hrs.

Sections 2, 4, 6

Leland : 16 hrs

Sections 1, 3, 5