Yale University
Department of Computer Science

Analysis of Optimizations of Angluin's L^* Machine Learning Algorithm

David Beam

YALEU/DCS/TR-1498
September 2014

Analysis of Optimizations of Angluin's L^* Machine Learning Algorithm

David Beam[*†]

Abstract

This paper provides the results of a study conducted on various versions and optimizations of Angluin's L^* learning algorithm. Tests were done on randomly generated DFAs and prefix-closed DFAs using both the original algorithm, and an optimized version under several different conditions. Results were analyzed in terms of queries and counter examples used, as well as overall run time. The optimized algorithm consistently outperforms the original in terms of queries and run time, and even outdoes an optimization made specifically for prefix-closed machines presented in a previous study.

1 Introduction

Recently, effective modeling techniques have become essential to the field of computer science as we attempt to use software to describe complex, oftentimes evolving systems. Many such systems can be described using deterministic finite automata (DFA), a type of state accepter that uniquely accepts or rejects any finite string of a given alphabet. One method for automated generation of DFAs, Angluin's L^* machine learning algorithm, has been widely used since its introduction in 1987, in areas including communication, big data, and the NASA Mars rover project [Cobleigh et al., 2003]. Within the last decade, optimizations of L^* have been developed that dramatically increase efficiency in terms of run time and queries. This study sought to analyze the effectiveness of Angluin's original algorithm (henceforth be referred to as L_0^*) and one such optimization, presented by Rivest and Schapire [1993] (referred to in this article as L_1^*), in a number of

[*]Stanford University, Stanford, CA 94305

[†]The author is grateful for the support and guidance provided by Professor Dana Angluin throughout the project.

different test environments. This paper does not seek to offer an in depth explanation of L_0^* as it has been done multiple times. For more information on the original version refer to "Learning Regular Sets from Queries and Counterexamples" [Angluin, 1987]. L_1^* forgoes the consistency check in L_0^*, instead optimizing each counter example and adding the optimized string to the experiments section. The optimization step ensures the table is always consistent, and, as shown later, drastically increases overall performance despite using more counter examples. For a more in depth discussion of L_1^*, read the presentation in section 4.5 of "Inference of Finite Automata using Homing Sequences" [Rivest and Schapire, 1993].

2 Preliminaries

L_0^* and L_1^* were compared in a number of different scenarios. While these trials are discussed in depth in Section 3 some factors remained constant throughout most trials. All machines were generated using Python's pseudo-random generator. First, a machine size between 4-100 non-minimized states was randomly selected. Then each state-symbol pair was assigned to lead to a state. Finally, states were randomly chosen to be accepting or rejecting. This creation method did not guarantee all of the states in the originally generated machines were reachable or unique. The states reported in the results sections data tables refer to the minimized states in the final guessed machine. While deterministic and random equivalence tests were used at different times, it can be assumed the deterministic test was used unless it is stated otherwise. The default search method for optimizing counter examples was binary, and in fact binary search was always used except in section 3.4. Machines with only one minimal state were excluded from the results, since all such machines always took both algorithms 3 queries, 1 equivalence test (0 counter examples), and a relatively small run time. All machines were run with an alphabet size of 2. The total average presented at the bottom of each table weights each range of states, not each machine, equally. Run time was measured from immediately after the random machine was generated until the time the guessed machine passed an equivalence check. Time is always reported in seconds in this paper. Random counter examples were generated by using Python's pseudo-random generator to first pick a string length between 1 and a given max length, then randomly pick 0's and 1's to make a string of the given length. In order to add some optimization to this process, the max length of these counter examples was first limited to ten symbols. If a counter example was not found after 500 strings, the max

length was increased to the theoretical maximum (two times the number of states in the true machine plus one, note the number of non-minimal states was used as the number of minimized states was not known at runtime). If a counter example was still not found after another 500 strings, the machine was assumed to be equivalent and the timer was stopped. The machine's equivalence was however verified deterministically before recording the trial as data. With the exception of Figure 2, all graphs show third order polynomial best fit lines generated by Microsoft Excel using all data pertinent to the given figure (not just the averages shown in the tables).

3 Results

3.1 Results Overview

In order to compare L_0^* to L_1^*, both algorithms were tested on the same randomly generated DFAs containing up to 100 non-minimized states with an alphabet size of 2. The main metrics used to evaluate the algorithms' relative performance were the average number of queries per state (q/s) and counter examples per state (c/s). These tests were also complemented by a run time test, which, though subject to code inefficiencies and computer performance, still proved a useful measure of effectiveness. Machines were tested using deterministic equivalence tests, and a random counter example brute force approach (where the randomly checked machines were verified deterministically at the conclusion of each run). In general L_1^* used less queries than L_0^* but more counter examples. This seemed to be a favorable trade-off however, as L_1^* consistently out performed L_0^* in terms of run time, especially on larger DFA's.

3.2 L_0^* versus L_1^*, with Deterministically Generated Counter Examples

L_1^* frequently used more c/s than L_0^* but less q/s. In all cases, both c/s and q/s were dependent on the number of states so averages were taken accordingly. The results are shown both graphically in Figure 1 and numerically in Figure 3.

The first graph in Figure 1 accurately depicts a trend of rapid growth in q/s while the second seems to suggest that c/s also grow rapidly after a brief period of decline. This second graph was the only case in this paper where the trend line was misleading. Though the trend line was skewed by large deviation and a relatively sparse sample of large machines, c/s actually

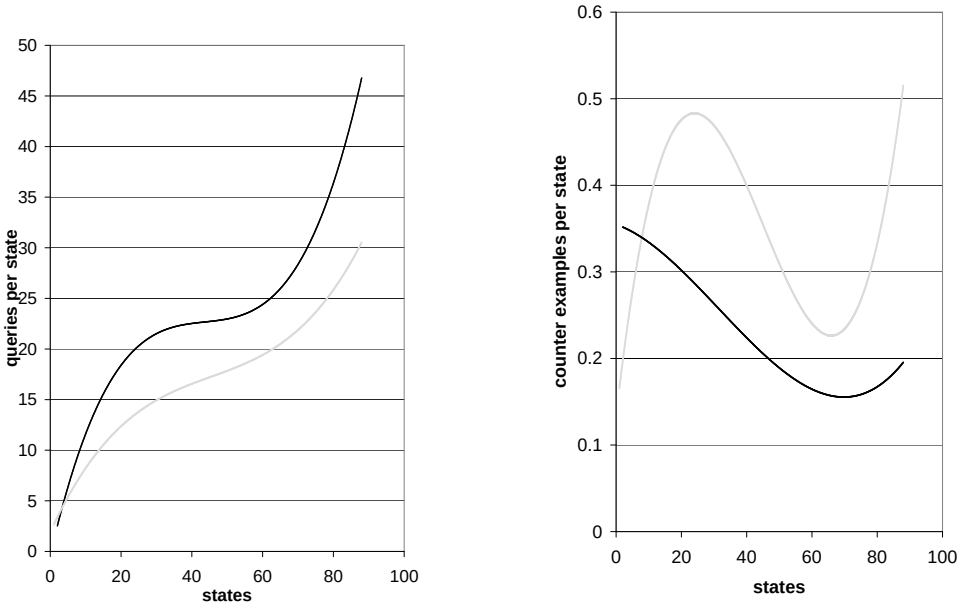

Figure 1: Trend lines comparing L_0^* (black line) and L_1^* (gray line) q/s (left) and c/s (right)

seemed to decline as the number of states grew. To better convey this trend Figure 2 offers the full scatter plot data of c/s used by every machine in this section, along with the trend lines from the second graph in Figure 1.

Overall on average L_1^* used 27.16% fewer q/s than L_0^*, but 60.49% more c/s. Both numbers are a direct result of the lack of consistency checking in L_1^*. Though L_0^* avoids some equivalence queries and subsequent counter examples by checking for consistency and adding experiments, it also ends up querying some unnecessary strings. L_1^* avoids checking these extra strings, and the consistency check altogether, by optimizing each counter example (finding the earliest point of divergence) and adding the optimized string to the experiments instead. The latter strategy proves much more efficient from a time standpoint, as evidenced in Figure 3.

On average L_1^* ran 4.76 times faster than L_0^*. The difference in run time is largely a result of two major differences between L_0^* and L_1^*: counter

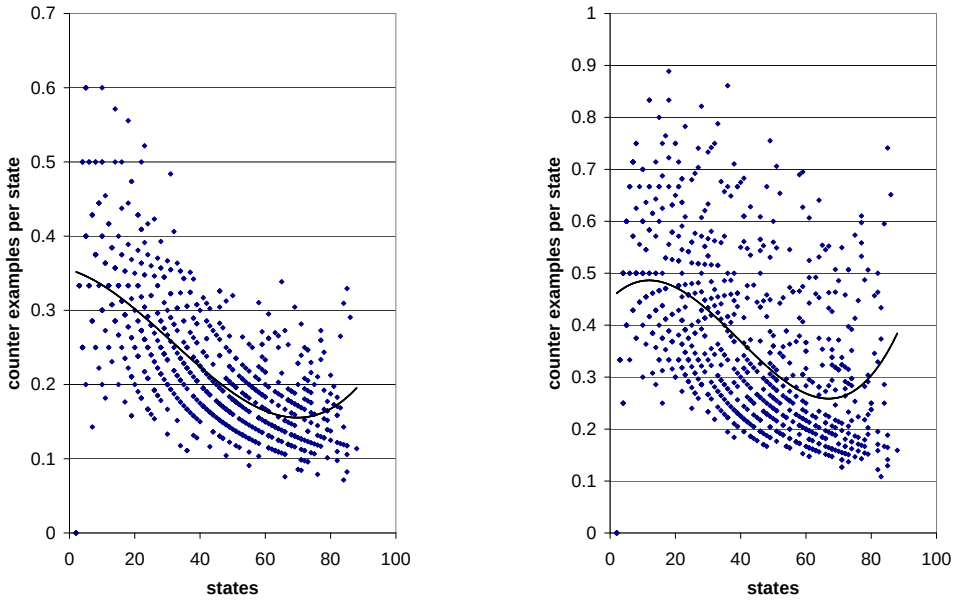

Figure 2: Full Scatter plot data of L_0^* c/s (left) and L_1^* c/s (right) with trend lines from the second graph of Figure 1

States	L_0^* q/s	L_1^* q/s	L_0^* c/s	L_1^* c/s	L_0^* time	L_1^* time	Trials
2-10	8.57	6.08	0.44	0.48	0.04	0.02	75
11-21	15.28	10.49	0.32	0.50	0.27	0.09	97
21-30	19.73	13.64	0.27	0.44	0.85	0.24	133
31-40	22.44	16.28	0.24	0.39	1.59	0.50	107
41-50	23.51	17.35	0.20	0.33	2.69	0.72	124
51-60	24.01	18.84	0.18	0.29	3.99	1.09	104
61-70	26.09	20.46	0.16	0.27	5.61	1.48	97
71-80	30.05	23.10	0.16	0.27	10.43	2.43	73
81+	42.86	28.59	0.16	0.31	27.72	4.59	23
Total	23.62	17.20	0.23	0.37	5.91	1.24	833

Figure 3: L_0^* vs L_1^*, deterministically generated counter examples

States	L_0^* q/s	L_1^* q/s	L_0^* c/s	L_1^* c/s	L_0^* time	L_1^* time	Trials
2-10	12.99	7.46	0.28	0.50	0.19	0.14	26
11-21	21.43	12.68	0.22	0.45	1.15	0.32	36
21-30	29.43	15.99	0.19	0.39	5.87	0.57	42
31-40	41.46	19.11	0.16	0.35	20.38	0.86	30
41-50	48.39	20.49	0.14	0.29	47.07	1.20	46
51-60	61.46	24.39	0.14	0.29	106.10	1.94	48
61-70	63.06	24.05	0.12	0.25	176.25	2.09	28
71-80	73.84	25.40	0.11	0.23	325.36	2.82	29
81+	150.03	31.96	0.11	0.26	1,766.46	4.37	8
Total	55.79	20.17	0.16	0.33	272.09	1.59	293

Figure 4: L_0^* vs L_1^*, randomly generated counter examples

examples vs queries, and optimizing counter examples versus checking for consistency. While L_1^* uses more counter examples than L_0^* than it saves on queries percentage wise, in terms of magnitude both algorithms use many more queries than counter examples. So from a run time standpoint it is better to use more counter examples if it saves on queries. Additionally, checking for consistency is an extremely costly function, that involves looping through the observation table multiple times. However, optimizing a counter example has a relatively low cost and the point of divergence can be found in $O(\log n)$ run time using a binary search. These two factors account for the significantly better run times using L_1^*.

3.3 L_0^* versus L_1^* with Randomly Generated Counter Examples

The performance discrepancy between the two algorithms is even larger when using randomly generated counter examples. As seen in Figure 4, L_0^* used 2.5 times more q/s than L_1^* in these trials. L_0^* also used more than twice as many q/s as it did with deterministic counter examples (55.78 vs. 23.62). L_1^* meanwhile only saw a 17% increase in q/s (17.20 vs. 20.17). The average run time for L_0^* on these machines was over two orders of magnitude larger than that of L_1^* under the same conditions (272.09s vs 1.59s). Two of the 81+ state machines took L_0^* over 100 minutes to run each, while L_1^* finished them in 9 and 8 seconds respectively. The main reason why random counter examples cause L_0^* to run so inefficiently is the consistency check. As explained before, the consistency test has to loop through the

Counter Example Type	Avg q/s Linear	Avg q/s Binary	Trials
Deterministic	16.57	16.54	461
Random	20.32	19.73	458

Figure 5: Queries used with binary and linear optimization searches

table multiple times checking each pair of rows for equivalence and resulting in a run time of $O(mn^2)$, where m is the width of the observation table and n is the length. An upper bound to the worst-case size of the observation table for L_0^* was calculated by Angluin to be $(l|\sum| \cdot |Q|^2)$ [Ang87] where l is the length of the longest counter example provided, $|\sum|$ is the alphabet size (always 2 in these tests) and Q is the theoretical maximum number of queries in a given machine. If the teacher always gives the shortest possible counter example, as it does when using the deterministic equivalence test, the maximum table size simplifies to $(|\sum| \cdot |Q|^3)$. This slight efficiency is lost when using random counter examples. More significantly however, the tendency of random counter examples to be longer than necessary means the table size has a much larger chance of approaching this maximum bound than when deterministic counter examples are used. The larger table size leads to a more costly consistency test.

3.4 A Comparison Between Binary and Linear Search when Optimizing Counter Examples in L_1^*

As part of L_1^* the counter example optimizer searches through the provided string for the point of divergence between the guessed machine and the true one. The piece of the string up to and including this point is used as the optimized counter example. In order to find divergence the optimizer must query portions of the string, which counts against the overall membership query total. This search process can be carried out using either binary or linear search. While binary search would automatically seem advantageous, there is a caveat. The linear search can instantly stop when it finds a divergence point and know that it is the first one. The binary search must also test the point before any different point it finds to ensure it is actually returning a divergence point. These two methods were compared using both a deterministic equivalence test and with randomly generated counter examples. The results are shown in Figure 5. In the deterministic case, the difference between queries was almost negligible. Oftentimes the two methods used the exact same number of queries. The similarity can be explained given the fact that all membership queries are cached, and can

be reused without counting against the overall total. So an extra query in a given run of the optimizer will often have no effect overall. Despite the small difference it is noteworthy that, in the deterministic trials, the binary search never used more queries than the linear search, and used less queries on roughly half of the trials, meaning it is slightly superior. Runtime was also measured for all trials in this test, and no significant difference was observed. Given that the optimizer is only one small piece of the overall L_1^* algorithm the run time similarity was expected. As expected in this case both methods always used the same number of counter examples. With the random counter examples there was a more notable difference in terms of number of queries. The binary search, used 3.38% fewer queries than its linear counterpart. This number becomes significant when considering the queries used in the optimization phase are only a fraction of the algorithm's total number, which is what gets compared in the end. In these tests however there were some rare cases where the linear test used fewer queries. Regardless, binary search was clearly the superior method.

3.5 L_0^* versus L_1^* on Prefix-closed Automata

Berg et al. [2005] note that L_0^* struggles with prefix-closed automata. A prefix-closed machine is one such that every prefix of an accepted string is accepted and every suffix of a rejected string is rejected. L_0^* and L_1^* were both run on prefix-closed DFAs to see how much these machines affected performance, especially on L_1^* which was not tested by Berg et al. The machines were generated by first using Python's pseudo-random generator to make a non-minimized DFA, as above, then setting all but one of the reachable states to accepting, then putting self loops on the rejecting state. Berg et al. offer an optimization that uses 20% fewer membership queries than L_0^*. L_1^* however used an average of 36.21% fewer membership queries (which translates to the 36.93% fewer q/s shown in Figure 7), in exchange for 140.68% more counter examples. Meaning that in terms of queries, L_1^* was a better optimization than the prefix-closed specific one presented by Berg et al. [2005] for the machines tested. L_1^* also ran 3.7 times faster than L_0^* on these machines. Figure 6 shows how prefix-closed machines change the growth curve, for both algorithms. The growth of the prefix-closed machine's q/s appears to be much more linear than that of standard automata.

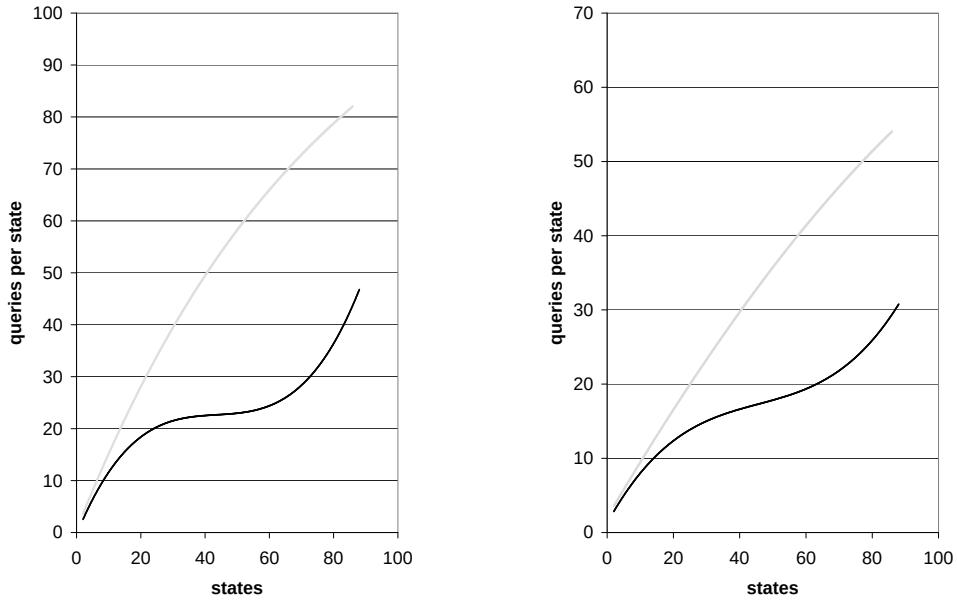

Figure 6: Trend lines comparing standard (black line) and prefix-closed (gray line) automata q/s using L_0^* (left) and L_1^* (right)

States	L_0^* q/s	L_1^* q/s	L_0^* c/s	L_1^* c/s	L_0^* time	L_1^* time	Trials
2-10	8.39	5.86	0.31	0.47	0.03	0.02	50
11-21	23.44	13.92	0.33	0.68	0.47	0.18	65
21-30	33.81	20.38	0.29	0.68	1.86	0.59	62
31-40	44.77	25.82	0.27	0.63	5.20	1.29	55
41-50	53.24	32.67	0.25	0.63	10.24	2.70	64
51-60	63.89	38.94	0.22	0.63	25.73	6.12	57
61-70	67.36	43.66	0.21	0.59	53.20	13.81	65
71-80	77.81	49.81	0.21	0.60	54.92	13.93	52
81+	74.40	50.93	0.18	0.55	45.53	14.43	6
Total	49.68	31.33	0.25	0.61	21.91	5.90	476

Figure 7: L_0^* versus L_1^* on Prefix-closed Automata

4 Conclusion

The data clearly shows the superiority of L_1^*, which outperforms L_0^* in almost every metric. It also shows the viability in terms of run time of L_1^* in multiple scenarios. Even on prefix-closed machines, a supposed weakness of the algorithm, L_1^* finished with an average run time of 14.4 seconds on machines with 81 or more minimized states. While run times certainly may grow for machines with larger alphabets, the overall results are encouraging, especially in the light of the continuously growing range of applications for the algorithm. New optimizations and ways of using L^* continue to be discovered. For example, Cobleigh et al. [2003] have used compositional verification to eliminate large table sizes on big machines. The compositional approach not only saved memory space, but also may have presented an alternate solution to the run time issues shown when using L_0^* with random counter examples (as the problems occurred with large observation tables). Further work may seek to build on these results with larger alphabet sizes, more non-minimized states, or different optimizations. This study also did not examine memory usage of the tables which becomes an important factor with larger machines.

References

Angluin, D. (1987). Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106.

Berg, T., Jonsson, B., Leucker, M., and Saksena, M. (2005). Insights to Angluin's learning. *Electronic Notes in Theoretical Computer Science*, 118(0):3–18. Proceedings of the International Workshop on Software Verification and Validation (SVV 2003) Software Verification and Validation 2003.

Cobleigh, J., Giannakopoulou, D., and Păsăreanu, C. (2003). Learning assumptions for compositional verification. In Garavel, H. and Hatcliff, J., editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2619 of *Lecture Notes in Computer Science*, pages 331–346. Springer Berlin Heidelberg.

Rivest, R. and Schapire, R. (1993). Inference of finite automata using homing sequences. *Information and Computation*, 103(2):299–347.