6.858/18.428 Machine Learning	Lecture 24: December 7, 1994
Lecturer: Ron Rivest	*Scribe: Ping Huang*

Outline

- Learning DFAs: Angluin's L* algorithm
- Learning DFAs without resets using homing sequences: Rivest & Schapire

24.1 Learning DFAs: Angluin's L* algorithm

In review, Angluin's L* algorithm progresses by iteratively building a table such as that in Figure 24.1. Upon every iteration, one of these actions is taken:

If table is not closed: I.e., the bottom part of the table contains a row which is different from every row in the top part of the table. The algorithm promotes that row to the top part, and appropriate rows are added to the bottom part to preserve the $S \cdot A - S$ property.

If table is not consistent: I.e., the top part of the table has two states (S_j, S_k) whose rows are identical but those two states' successors have rows which are *not* identical. The algorithm expands the set of experiments to differentiate between S_j and S_k by adding the experiment $a \cdot e$, where e is an experiment which $S_{j \cdot a}$ and $S_{k \cdot a}$ differed on.

If table is closed and consistent: The algorithm constructs a conjecture DFA and issues an equivalence query. If it receives the answer yes, then the algorithm successfully terminates. If it receives the answer no, it adds the received counterexample and all its prefixes to S, and extends $S \cdot A - S$ accordingly.

Proof that the L* algorithm is correct is informally sketched:

1. For any regular language, there exists an unique minimal DFA which recognizes that language.

2. We can generate a DFA consistent with a closed consistent table.
3. Any DFA which is consistent with the table has at least as many states as unique rows in the top part of the table.
4. Once the algorithm builds the table to have n different states in S, then it will have the target DFA.

Proof that the L* algorithm terminates is informally sketched:

1. Each closure failure increases the number of distinct rows in S by 1.
2. Each consistency failure increases the number of experiments in E by 1. This must concomitantly increase the number of distinct row patterns in S by at least 1, because the added experiment is precisely one which distinguishes between two formerly-identical rows.
3. Each counterexample increases the number of distinct rows in S by at least 1, since the counterexample was not handled by the table before.
4. The number of distinct rows in S is therefore strictly increasing, and reaches n, at which point the table must represent the target DFA.

Theorem 1 *L^* is a polynomial-time algorithm for inferring the exact target DFA using membership queries and equivalence queries.*

Proof: More specifically, the running time of the algorithm can be shown to be polynomial in n (number of states in the target machine) and m (length of the longest counterexample). Initially $|S| = |E| = 1$. At all times, $|E| \leq n$ since at most n consistency failures could have occurred; similarly, the maximum length of the longest string in E is $n-1$. At all times, $|S| \leq n + (n-1)m$ (the first term is a bound on the number of states added to S by closure or consistency failures, the second term a bound on the number of states added to S by counterexamples and their prefixes); the maximum string length in S is $n + m - 1$. The number of table entries is $|(S \cup S \cdot A)| \cdot |E| \leq (k+1)(n + (n-1)m)n = O(mn^2)$, if the size k of the alphabet representation is assumed to be constant; this is also the number of membership queries issued. The maximum string length in $(S \cup S \cdot A) \cdot E$ is $\leq m + 2n - 1$, so the size of the table is polynomial. Finally, at most $n-1$ equivalence queries are issued.
■

S \ E	λ	0
λ	0	0
0	0	0
1	1	1
00	0	1
01	0	1
10	1	1
11	1	1
000	1	1
001	0	0

Figure 24.1: Example of the table constructed by Angluin's L* algorithm for learning DFAs when resets, membership queries, and equivalence queries are available.

Note that while membership queries are quite reasonable, availability of equivalence queries is a more difficult requirement to meet. We can, however, simulate equivalence queries by using randomized tests, in a PAC-framework where ϵ and δ provide leeway for statistical errors in the simulation process, allocating $\frac{1}{2^i}$ of the δ budget on each successive equivalence query if we do not know *a priori* how many will be issued.

24.2 Learning DFAs without resets using homing sequences: Rivest/Schapire

24.2.1 Scenario

L* makes the assumption that it is possible to "reset" the DFA being learned. This may not be realistic; or, it may be too restrictive. For example, suppose we have a robot trying to explore its environment, 15 squares which the robot may travel to, plus 1 which it cannot travel to, arranged in a 4x4 grid (Figure 24.2). In addition to which of the 15 squares it is in, the robot's state includes which of 4 directions it is

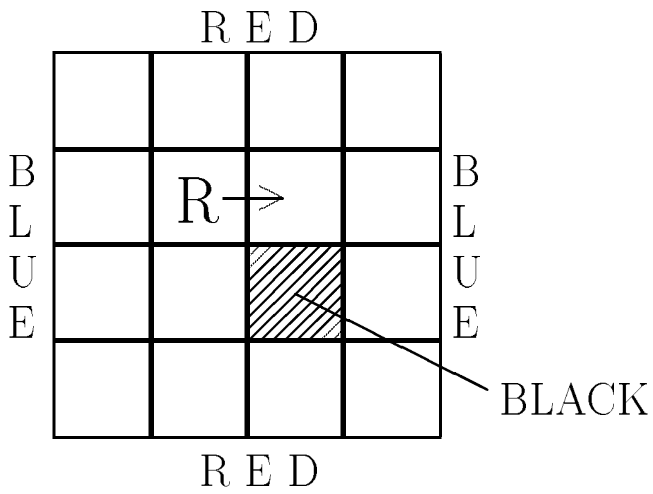
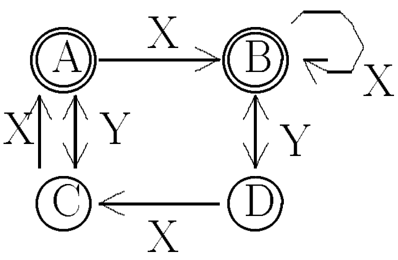


Figure 24.2: A robot can learn its environment by modelling it as a DFA, but one which cannot be reset for learning with L*. The robot's location is marked by "R", its orientation by the arrow, and the current DFA "output" is blue. The black square represents an inaccessible location.

facing. The north and south walls are red, the east and west walls are blue, and the barrier square is black. The robot may step forward (`S`), turn left (`L`), or turn right (`R`); a step forward when the robot is up against a wall or barrier is a no-op.

From the robot's point of view, the *environment* can be modeled by a DFA: the robot feeds this DFA a string of symbols from the alphabet {`S`, `L`, `R`} (via its motor mechanisms), and gets back the DFA output {`blue`, `red`, `black`}, which represents the color the robot sees in front of itself.

To do membership queries, strictly speaking, as required by the L* algorithm, the robot would need a "RESET" button to get it back to the state it was to start with; such teleportation is an unrealistic requirement. (Note that with prior knowledge about the semantics of {`S`, `L`, `R`} and under particular circumstances, e.g., wrap-around world without barriers, after the robot issues some membership query q, it could construct and issue q' to undo q. However, in general there may not exist inverses, or even if one existed it might not be constructable given just q. Even in the kind of "strongly connected" environment pictured in Figure 24.2, constructing q' from q would require having *already* learned the environment.)



Started in:	Observed		End state is:
A	◎	◎	B
B	◎	◎	B
C	○	◎	A
D	○	○	C

Action: X

Figure 24.3: Homing sequences. The observation sequence for the action sequence `X` determines exactly what state we end up in.

24.2.2 What is a homing sequence?

The key insight is that while there is no procedure for general finite state automata to restore the initial state, a *homing sequence* always exists, and can help us. A homing sequence is a sequence of actions whose output sequence fully determines what final state the DFA is in, even if we did not know what the start state was. For example, in Figure 24.3, the action sequence `X` is a homing sequence; even if we do not know what was the start state, the observation sequence can still tell us where we must have ended up.

Theorem 2 *Every minimal DFA has a homing sequence.*

Proof: Given a minimal DFA, we construct a homing sequence as follows. Check if the null sequence suffices as a homing sequence (i.e., if every state has an unique output). If the null sequence is insufficient, for every pair of states which currently look the same, by the definition of a minimal DFA, there must exist some sequence to distinguish them, of at most length n—we append that sequence to our candidate homing sequence. We iterate at most $n-1$ times to distinguish every state; the resulting homing sequence has length $O(n^2)$. ■

(Note that the Rivest/Schapire procedure works much faster when used with a homing sequence generator that adapts to the current state rather than a statically generated and usually unnecessarily long one as generated by the above construction.)

24.2.3 Using homing sequences

Assuming a robot is given some homing sequence `HS`, what should it do with it? Let us plan to run n copies of the L* algorithm, each with one of the n different states as its initial state. (For now, we assume n is known; this assumption is actually unnecessary.) We then can get the semantics of membership queries with resets by time-slicing. Suppose we are currently running copy i of the L* algorithm and it just issued a membership query; we then issue the homing sequence, which takes us to state S_k. We then put copy i to sleep and wake up copy k — for which the DFA is in the correct initial state. Each time, *some* copy of the algorithm must make progress, hence at worst we've introduced a factor of n into the running time.

24.2.4 Generating homing sequences

We've solved the problem that there is not a reset, but now we must be able to generate a homing sequence for a DFA whose structure we do not already know. To do this, we add another layer of iterations to the procedure. We start with the trial homing sequence $\mathtt{HS}_1 = \lambda$. In the i^{th} iteration, we try running n copies of the L* algorithm afresh, as above, using $\mathtt{HS}_i$ as the trial homing sequence. (Note: the "afresh" implies a tremendous loss of efficiency, because all the work done by previous sets of n copies are thrown away, but there does not seem to be an easy way to incorporate the learning already successfully done.) If any copy of the L* algorithm ever successfully completes in some iteration (i.e., its equivalence query returns the answer "Yes"), then we're definitely done.

How do we figure out when $\mathtt{HS}_i$ is not a homing sequence for the DFA being learned (so that we know to stop running iteration i and go on)? During a given iteration, we know we're running $\leq n$ copies of L*, since any trial homing sequence can at generate at most n different observation sequences. For any given copy of L*, the generated table may be messed up, but we know that after a certain number of wake-ups, it will have promoted more than n unique row patterns to the top part of its table (this assumes we know n). When any of the copies of L* in an iteration has more than n unique row patterns in the top part of its table, we can stop that iteration and go on to the next one, because clearly by then something has gone wrong, and we will need to try a new homing sequence.

How do we construct $\mathtt{HS}_{i+1}$ from $\mathtt{HS}_i$? We will have need to extend our prospective homing sequence if and only if $\mathtt{HS}_i$ could not distinguish between some two states S_j and S_k. Therefore, for the table of some copy of the L* algorithm, some table entries were filled in using membership queries when the DFA was in S_j, and some when the DFA was in S_k. Choose an experiment column (e) such that the table entries for the experiment in any two randomly chosen rows (S_1, S_2) differ in value. (These two rows appear to be different, but may actually be the same.) Pick one of the table entries and test it (i.e., with $S_1 \cdot e$ or $S_2 \cdot e$). If we happened to have chosen the right two rows (probability $\mathrm{O}(\frac{1}{n^2})$) and the right experiment, then we will have found how to extend the homing sequence: $\mathtt{HS}_{i+1} = \mathtt{HS}_i \cdot e$.

As for the assumption that n is known, we can always add another layer of iteration which keeps doubling our estimate for n.

24.2.5 In practice...

Schapire implemented a program to try to learn the random graph walk problem modeled as a DFA. There were 400 nodes, each of degree 3, which were randomly colored black or white. Equivalence queries were simulated with random walks. One run to learn the graph used 258 copies of L^* in parallel (in a particular iteration), issued 33,000 membership queries, took 1.7 million actions, and consumed 1 hour 20 minutes of CPU time.

References

[1] Dana Angluin: "Learning regular sets from queries and counterexamples." *Information and Computation*, 75:87-106, 1987.

[2] Ronald L. Rivest & Robert E. Schapire: "Inference of Finite Automata Using Homing Sequences." *Information and Computation*, 103:299-347, 1993.