

Computer Science 345/711: 2021 - Project for Term 4

Short project summary

In this project students will explore the *Angluin learning algorithm* for symbolic finite state automata (SFA). Angluin learning consists of two conceptual parts, the *learner* and the *teacher*. The learner wants to learn a specific SFA, and in order to achieve this, may ask the teacher two kinds of questions.

Membership queries: Is the string “w” accepted by the automaton I am trying to learn?

Equivalence queries: Is my current hypothesis of the automaton I am learning correct? If I am not correct, provide a counterexample on which my current hypothesis is wrong.

SFA enhances classical automata by allowing what is referred to as predicates on transitions. For example, we might label a transition with the predicate “[a-z] union [0-9]”, which implies that the given transition can only be taken if the next symbol is a lower case letter or a digit.

Due Date

Monday 1 November at 23:59

Project teams

The project may be completed on your own or in a team of two. This is taken into account in the marking rubric given below.

Purpose of the project

The aim of this project is to encourage students to familiarize themselves with some of the recent research on automata theory.

Outcomes

After completing this project, students will have a better understanding of some of the shortcomings of automata models that were studied in the first part of this course. Students will also understand the main aspects of the Angluin learning algorithm and SFA.

Why use SFA?

Finite automata are used in applications in software engineering, including software verification, text processing, and computational linguistics. Despite their many useful applications, classical automata models suffer from the major drawback that in most common forms they can only handle finite and small alphabets. To overcome this limitation, SFA were introduced. They allow transitions to carry predicates over Boolean algebras. For this project students may assume that they are working over the unicode Boolean algebra where predicates are unions of unicode intervals. Students should start by watching the first 20 minutes of [3] to get a basic understanding of SFA.

Angluin learning

The purpose of this project is to experiment with SFA active learning algorithms, in particular Angluin learning. Angluin learning for classical DFA was introduced by Dana Angluin in 1987 [see 7]. A summary of Angluin learning for classical DFA is described in [8]. This automata learning algorithm by Dana Angluin was also discussed in class. Angluin learning can make use of one of two base data structures, namely an observation table or a discrimination tree. In this project the focus will be on the simpler of the two, the observation table (which is also less efficient in various ways compared to the discrimination tree approach). The discrimination tree approach for Angluin learning of classical DFA is described in [11], and for SFA in [1], and also in Chapter 8 in [4].

One of the aims of this project is to study the efficiency (in terms of number of required membership and equivalence queries) of SFA Angluin learning from an experimental point of view. The SFA that should be used to evaluate Angluin learning are those obtained from a list of regular expressions provided below.

Read the first 9 pages of [2], which is also summarized in [5] (look at the first 47 slides) in order to understand the basics of SFA Angluin learning.

For this project, students are given a skeleton framework for the algorithm described in [2]. The skeleton consists of a maven project with the framework needed to implement the algorithms from [2]. In the *src* directory of the code skeleton there are 7 packages (inside *src/main/java*). The packages are:

- **Learning**: Contains the skeleton for the implementation of the algorithm described in [2]. This is where you will be working for the majority of the project.
- **algebralearning**: Contains the implementation for the algorithm described in [1]. You may need to use the code in this directory for the benchmarking of the algorithm in [1] as a challenging aspect of the project.
- **automata,logic,strings,theory,utilities**: Contains the framework necessary for the implementation of this project. You may ignore the code contained in these directories.

The main differences between the classical and SFA Angluin learning algorithms are as follows: In the SFA case, a partition function is used that makes guesses in terms of transitions of the automaton being learned, consistent with concrete evidence previously received from the teacher. Also, in the symbolic case the observation table is populated in a sparse way.

Students should start the project by completing the following four methods in the code skeleton (as described in [2]):

- In `src/main/java/learning/sfa/Learner.java`: [fill](#), [close](#), [make_consistent](#)
- In `src/main/java/learning/sfa/PartitionFunc.java`: [intervalPartition](#)

Benchmarking

When referring to benchmarking below, it is implied that the number of membership and equivalence queries required for learning each SFA, derived from the given set of regular expressions, should be determined. To run the framework for benchmarking, use the provided Makefile. To get information on how to use the Makefile, type `make help`. This will give a detailed explanation on how to use the Makefile. The framework takes a regular expression (`r="..."`), logging flag (`l="..."`), a partition function choice, and a teacher length and an ordering strategy. For the purpose of this assignment, you may ignore the teacher length strategy. You may leave out all of the arguments except the regular expression, and you may provide the arguments in any order. The benchmarking should be done for a lexicographic least (i.e. shortest and lexicographic least counterexamples are returned by the teacher) and a lexicographic random teacher.

A given regular expression is converted to a SFA by the `RegexOracle.java` class found in the `learning/sfa` directory, and uses the `regexToDFA` framework found in the `learning` directory. This SFA is then given as a teacher/oracle to the learner to learn.

Benchmarking regexes:

- | | |
|---------------------|----------------------------------|
| 1. a | 11. a[b-e]cd[x-z]* |
| 2. a* | 12. [\u0-\u10]*[a-z] |
| 3. [a-z]* | 13. \u0*[\u0-\u10](\u1 \u65534)* |
| 4. ab* | 14. (0 [\u0-\u10])*[\u0-\u10] |
| 5. abca | 15. [\u50-\u65]*\u0[\u247-\u257] |
| 6. (a e)* | 16. ((abc)*e)*z |
| 7. a(b h)*d | 17. (0 a A)* |
| 8. a[g-kxz]d | 18. (\u0 \u65534)* |
| 9. 0[a-x]z* | 19. (\u1 \u754 \u9872 \u65534)* |
| 10. [a-g][a-m][m-z] | 20. ([a-z][0-9])* |

All of the above regexes are available in the code skeleton in the file *benchmarks.txt*. The results (both equivalence and membership queries) for the first 5 regexes are also given, for testing purposes, in *results.txt*. These results are obtained when using the interval partition function (described below), and when BSQ (also described below) is not used.

Binary search Querying (BSQ)

The algorithm described in [2] performs well if the teacher is well-behaved, i.e. returns lexicographic least counterexamples, but performs badly when a teacher returns random counterexamples. To simplify the project, you may assume that the teacher always returns shortest counterexamples.

Consider the situation where we want to learn the SFA that accepts the language described by the regular expression: $([a-z])^+$. This SFA accepts all strings having only lowercase letters. To learn this, we now consider a teacher which returns random counterexamples in terms of lexicographic ordering (but still shortest counterexamples). Assume our learning algorithm hypothesizes a single state SFA that rejects all strings. The teacher will then return a counterexample string of length 1, consisting of a single character in the range $[a-z]$, and since it is a random teacher, it may return any one of these. Assume our teacher returns the letter “g”. If we use the interval partition function (described below), our learner will now assume that on any character up to and including “g” and also after “g”, the SFA should transition to an accepting state. The teacher may then return the string “H” (i.e. capital ‘H’), as a counterexample, since this string is not accepted, but the learner hypothesized that it should be. The learner then assumes that on strings starting with a character smaller than or equal to “H”, and also on strings starting with a character between “H” and “g”, that the SFA should transition to a non-accepting state, but on strings starting with a “g” or a larger, it should transition to an accepting state. This process of guessing by the learner (using the partition function) and counterexamples being returned by the teacher, will continue, and sometimes for a large number of steps, as the teacher may provide suboptimal counterexamples in terms of learning.

To counter this, Binary Search Querying (BSQ) can be used to find boundary tuples $[b, b']$, i.e. b and b' are neighboring characters in the unicode alphabet (so the interval $[b, b']$ contains only two characters) on which the SFA should transition to distinct states, from a given state. For the example above, we have the boundary tuples $[_, a]$ and $[z, \{]$. Also, the interval $[a, z]$ contains “g” and the rows for “a”, “g” and “z” are the same in the observation table, thus the learner hypothesizes that the SFA should transition to the same state on the interval $[a, z]$. To find boundary tuples, we use binary search to find neighbouring unicode characters on which the SFA should transition to different states (in our example, “a” is accepted, and “” is rejected, so

the SFA should transition to different states on these two inputs from the initial state). This method is referred to as Binary Search Querying (BSQ) and works as follows: Assume we start with an observation table and thus a SFA that is consistent with the evidence which the learner has received thus far, but that the learner next receives a counterexample, and thus adds one or more rows (and perhaps also columns) to the observation table. Let “py” be one of these rows, having “y” as the right most character. We now consider the smallest interval $[x,z]$ in the unicode alphabet, such that the observation table has rows labelled with “px” and “pz”, with $[x,z]$ containing “y”. If there are no such strings “px” and “pz”, we add rows labelled by “pm” and “pn” to the observation table, where “m” and “n” are the minimum and maximum unicode values respectively. If “px”, “py” and “pz” are labeled by identical rows, the learner will conjecture that the SFA should transition to the same state for all characters in $[x,z]$, from the state reachable when reading “p”. If the rows for “px” and “py” differ, do BSQ on the interval $[x,y]$. If the rows for “px” and “py” are identical, the rows for “py” and “pz” must differ, so then do BSQ on $[y,z]$. BSQ is done by iteratively subdividing the current interval $[e, g]$ into two intervals $[e,f]$ and $[f,g]$ (overlapping on “f”), with “f” (approximately) the middle value of $[e,g]$, and repeating BSQ on one of these intervals until an interval of length 1 is obtained, which must be a boundary tuple. Note that we also add the row for “pf” to the observation table, when dividing $[e,g]$.

Parts of the BSQ algorithm are already implemented in the *Learner.java* class. There are two parts to the algorithm that you need to implement, namely:

1. **Boundary value initialization:** For each access string corresponding to a given state, append the lex min and lex max (of the unicode Boolean algebra) to the right of an access string (to a given state of the currently hypothesized SFA) and add these two strings as rows to the observation table.
2. **Binary search:** For each string “py” labelling a row in the observation table, where “p” is a string and “y” a unicode character, find “x” and “z” closest to “y” in the unicode alphabet such that “px” and “pz” are labelling rows in the observation table and the interval $[x,z]$ contains “y”. If the rows for “px”, “py” and “pz” are not all identical, do BSQ (as described above) on $[x,z]$. See TODO in the *Learner.java* class.

You will need to consult *src/main/java/theory/BooleanAlgebraFiniteLoSet.java* (which represents a Boolean algebra that has a total ordering) for methods that will be helpful in implementing BSQ.

Challenging aspects of the project

A small part of this project is much more difficult than the rest. For this, select some of the following aspects to investigate.

- Implement your own partition function and motivate the design of your partition function in the project report, by benchmarking your custom partition

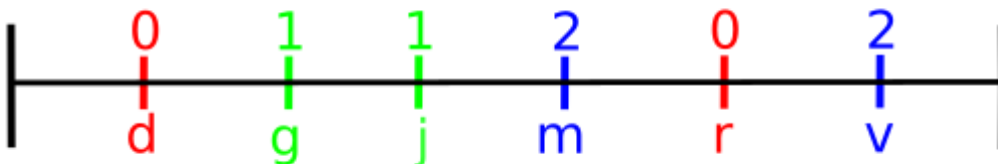
function and comparing the results obtained with those obtained when using the interval partition function and BSQ.

- Adapt the BSQ algorithm in the following way: From a training set of regular expressions (defined above), identify commonly used boundary tuples $[b,b']$ from amongst the regular expressions and sort them by the number of occurrences of that boundary tuple from low to high (break ties with lexicographic ordering). For BSQ, if the interval contains the boundary tuple $[b,b']$ (where $[b,b']$ is the earliest possible value in the sorted list of tuples), then add pb and pb' to the observation table. If they differ, then $[b,b']$ is a boundary value in this regex, so BSQ can return with this, otherwise continue in the intervals $[e,b]$ and $[b',g]$. If there is no boundary tuple $[b,b']$ contained in $[e,g]$, do BSQ as usual.
- Benchmark the implementation described in [1] (and [4], Chapter 8), which makes use of a discrimination tree rather than an observation table. The code for this is contained in the *algebralearning* directory as indicated above.
- Investigate empirically the benefits of applying the “distributes” procedure described above Theorem 1 in [2], i.e. benchmark the efficiency of learning when applying, and also when not applying the “distributes” procedure.

Description of the interval partition function

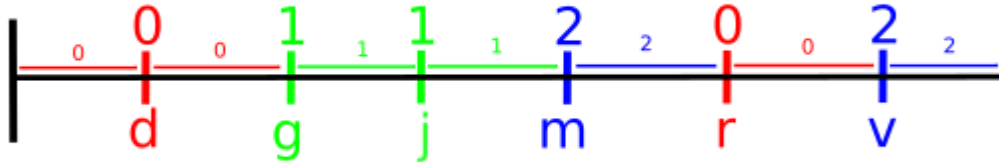
The goal of a partition function is to take individual evidence elements and divide the full domain of the Boolean algebra into smaller partitions based on a particular strategy which is consistent with the evidence elements.

The interval partition function does this by assuming every evidence element represents an interval from itself to the next evidence element. There is an implicit ordering on unicode characters, and so we represent them on a “number line”. The figure below gives an example of a character line for some evidence elements that we might have.



Fix a state p in the SFA from which we are busy learning transitions. Then in this example we have that on both ‘d’ and ‘r’ we go to state 0 (from state p), on ‘g’ and ‘j’ we go to state 1 (from state p), and on ‘m’ and ‘v’ we go to state 2 (from state p). The

interval partition function will then use this information to assign the remainder of the characters to a particular state, according to the following strategy. Every character in between an evidence element is sent to the state the last evidence element went to. This is shown in the following figure:



Note that the one exception is characters below the first evidence element. These characters are sent to the state the first evidence element is sent to as well.

Description of helper methods for partition functions (available in the code skeleton)

In the framework provided, there are a few helper functions provided to make the partition functions implementation easier. These helper functions are contained within the *PartitionFunction.java* class. They are outlined below:

1. **Predicate add(Predicate original, int lower_bound, int upper_bound):**

This method takes an original predicate from the Boolean algebra, and a lower and upper bound, and returns a predicate including what was in the original predicate, and the characters between the lower and upper bound (both inclusive).

For example:

$\text{add}([a-d], g, j) = [a-g] \cup [g-j],$
 $\text{add}([A-T], U, x) = [A-x],$
 $\text{add}([A-T], Z, a) = [A-T] \cup [Z,a],$
 $\text{add}([a-z], g, m) = [a-z]$

2. **Element shift(Element original, int d):**

This method takes an element from the domain of the boolean algebra and an integer (-1 or 1) representing the direction, and returns either the previous or next element in the domain depending on the direction.

For example:

$\text{shift}('d', 1) = 'e',$ (assuming unicode is the domain of the BA)
 $\text{shift}('s', -1) = 'r',$

Implementing your own partition function

In order to implement and run your own partition function you must do two things:

1. Add the custom partition function for which the skeleton is given inside `src/main/java/learning/sfa/PartitionFunc.java`
2. Uncomment the case statement for the custom partition function within the switch statement used in the `partition()` method of `PartitionFunc.java`

You may also rename your partition function (instead of `customPartition`) for better description and readability.

Marking Rubric - Including a short description of the required components

5 x 4 = 20	Implementation of the four required methods listed above: fill , close , make_consistent , intervalPartition
20	BSQ implementation and benchmarking when using BSQ.
10	Benchmarking when using the interval partition function.
10 x number of challenging aspects attempted.	Challenging aspects - see above.
20	Project report
10	Project video

For a single 3rd year student, 90 will be full marks, for a group of two 3rd year students or a single honours student, 100 will be full marks, and for two honours students, 110 will be full marks. Note that it is possible to obtain more than 100% for this project.

In both the project video and report (which should be approximately 10 pages in length) outline your implementation, the benchmarking results obtained, and also the aspects listed as challenging that were attempted. Your project report should also be on the gitlab server of Computer Science, in conjunction with your implementation. Also add a Readme file on gitlab with a link to your project video. Describe the benchmarking results obtained and list the aspects implemented in the Readme. The video should not be hosted on gitlab.

References

1. [The Learnability of Symbolic Automata](#) - George Argyros and Loris D'Antoni (this is a bit on the hardcoe side, but an extended version of this paper is available as Ch8 in the PhD thesis listed below as reference [4])
2. [Learning Symbolic Automata](#) - Samuel Drews and Loris D'Antoni
3. [Youtube - The Power of Symbolic Automata and Transducers - Loris D'Antoni - CAV 2017](#)
4. [Symbolic Model Learning - New Algorithms and Applications](#) - George Argyros
5. [Learning Symbolic Automata - Slides](#) - Samuel Drews and Loris D'Antoni
6. [Automata Modulo Theories](#) - Loris D'Antoni, Margus Veanes
7. [Wikipedia entry for Dana Angluin](#)
8. [Computer Science 345/711 lecture notes on Angluin learning](#)
9. [Loris D'Antoni's Symbolic automata page](#)
10. [A quick survey of active automata learning](#)
11. [The TTT Algorithm: A Redundancy-Free Approach to Active Automata Learning](#) - Malte Isberner, Falk Howar, and Bernhard Steffen
12. [An Abstract Framework for Counterexample Analysis in Active Automata Learning](#) - Malte Isberner and Bernhard Steffen
13. [LearnLib](#) - An open framework for automata learning