Regular Languages, Property Satisfiability, and Shortcuts

Patrick Melanson

June 2021, Halifax, Nova Scotia

A Thesis Submitted to Saint Mary's University, Halifax, Nova Scotia in Partial Fulfillment of the Requirements for the Degree of Bachelor of Science, Double Honours Mathematics and Computer Science

Dr. Stavros Konstantinidis

Professor, Supervisor

Dr. Mitja Mastnak

Professor, Reader

Date: June 6th 2021

Regular Languages, Property Satisfiability, and Shortcuts

By: Patrick Melanson

Abstract

The Language Server (*LaSer*) is a website created to ask and answer various questions pertaining to regular languages. One of its main features is testing property satisfiability, that is, does a given regular language satisfy a particular property. If a regular language does satisfy the property, we can then ask if the language is *maximal* with respect to the property. That is, L is maximal if it is not properly contained in any language satisfying the property. Deciding if a language is maximal reduces to deciding if a language is universal, which is known to be $PSPACE$-*complete*. However, for some practical purposes, we need only know if a language is *approximately* maximal. That is $p\%-$maximal. Using a randomized algorithm, we can check if a language is as maximal as we want, by repeatedly adding words and testing whether the language still satisfies the property. This new property is called pseudo-maximality, and is much easier to test.

Date: June 6$^{\text{th}}$ 2021

Contents

1 Introduction

In many math classes, someone has raised their hand at the end of a long lesson, and asked the dreaded question, "When will I ever actually use this?" Theoretical computer science, a related field, is similar in that it seeks to answer questions which may or may not have uses down the line. Part of that is the study of automata and formal languages, which in part can be used to model certain types of computation. This paper will serve as an overview of regular languages and automata, as well a deeper dive into *property satisfiability* and *property maximality*. For those who require practical uses, the *Language Server* (*LaSer*) is a website designed to answer questions about property satisfiability and property maximality, and we will be discussing why trying to compute the latter is resource intensive, and how we can create an algorithm which "guesses", which we can use instead and is much more efficient.

This thesis will be broken down into eight sections, working up towards the concept of pseudo-maximality. We will first go over some basic definitions that will aid in simplifying many concepts, then have a discussion about regular languages and automata, explain what they are and how they can be used, as well as various operations under which they are closed. Following that we will look into various constructions one can make with automata, as well as introduce NFAs. A final look into automata theory will have us defining transducers, and looking at how we can use those, giving us the final piece of information we need before discussing what properties, and property satisfiability are. Finally, we will need some basic information about computational complexity to understand the motivation behind pseudo-maximality, before finally explaining what that concept is, and discuss the possible uses for it.

2 Preliminary Definitions

1. Set: A set is any well defined collection of objects where order doesn't matter, and the elements are unique.

   Eg. $\{a, b, c\}$ is a finite set. Note that sets can be (and often are) infinite, for example the integers, $\mathbb{Z} = \{0, 1, -1, 2, -2, ...\}$.

2. Union: Let A and B be sets. Then $A \cup B$ (read "A union B") is a new set, C, which consists of all the elements of A as well as all the elements of B.

3. Alphabet: an alphabet is a non-empty set of characters or symbols which can be used to create words. Two common examples are the English alphabet $\{a, b, c, ..., y, z\}$ and the binary alphabet $\{0, 1\}$. We use an alphabet to make words, that is, finite sequences of symbols from the alphabet.

4. Concatenation: Let x and y be words, then xy, (read "x concatenated with y"), is the new word z which is x, followed directly by the word y.

   Eg. $x =$ "mouse", $y =$ "trap", then xy would be "mousetrap". If we are concatenating the same word multiple times, it is convenient to express that as an exponent, that is $aaaaa = a^5$.

   More generally, we can concatenate sets of words (that is, languages), that is. if A and B are sets, then $AB = \{ab : a \in A, b \in B\}$.

5. Empty word: ϵ will denote the empty word, which has length 0 and the property that for any w which is a word, $w\epsilon = \epsilon w = w$. Sometimes also denoted as λ.

6. Kleene star: Let A be a set, then $A^* = \bigcup_{i=0}^{\infty} A^i$. That is, the set of all words that are comprised of zero or more copies of a word found in the set A.

   Eg., if $A = \{a\}$, then $A^* = \{\epsilon, a, aa, aaa, aaaa, ...\}$ (note that $A^0 = \{\epsilon\}$).

3 Regular Languages and Automata

3.1 Inductive Definition and Examples

Regular Languages: We can inductively define the regular languages over some alphabet Σ as follows [1]:

1. The empty language, denoted $\varnothing$, and the language consisting of the empty word, denoted $\{\epsilon\}$, are regular.
2. $\forall a \in \Sigma$, $\{a\}$ is a regular language
3. If A and B are regular languages, then $A \cup B$, AB, and A^* are all regular (note that $\varnothing^* = \{\epsilon\}$)
4. Finally, a language is regular *if and only if* it can be constructed from a finite number of steps of 1, 2, or 3

A syntactic representation of regular languages is via regular expressions. That is, a language is regular *if and only if* it can be represented by a regular expression. For example, a is a regular expression, which represents the regular language $\{a\}$. Another one is ab^* which represents the regular language $\mathcal{L}(ab^*) = \{a, ab, abb, abbb, abbbb, ...\}$. Every finite language (that is a language which contains only a finite number of words) is a regular language. For simplicity, it is generally easier to talk about regular expressions, as oppose to regular languages, as they are more succinct to describe. It is also important to note, however, that a regular expression is *not* a regular language. It is merely a representation. Thus, if r is a regular expression, then $\mathcal{L}(r)$ is the regular language represented by r.

Some more examples may be helpful:

1. $\mathcal{L}(b) = \{b\}$
2. $\mathcal{L}(a^3b^2) = \{aaabb\}$
3. $\mathcal{L}(b + a^*) = \mathcal{L}(b) \cup \mathcal{L}(a^*) = \{b, \epsilon, a, aa, aaa, ...\}$
4. $\mathcal{L}(ab^*) = \{a, ab, abb, abbb, ...\}$
5. $\mathcal{L}((a+b)^2) = \mathcal{L}((a+b)(a+b)) = (\mathcal{L}(a) \cup \mathcal{L}(b))(L(a) \cup \mathcal{L}(b)) = \{aa, ab, ba, bb\}$

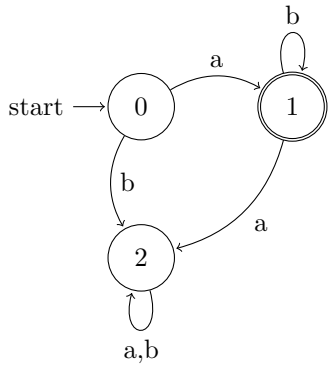
3.2 Equivalence with Deterministic Finite Automata

Another helpful equivalent description of regular languages is via finite state automata. Informally, they can be thought of as the simplest model for a computer, one without memory, which can either accept a word as being in a given regular language, or reject it, indicating that it is not in the language. More formally, we think of a Deterministic Finite Automaton (DFA) M as being a 5-tuple: $M = (Q, \Sigma, \delta, q_0, F)$, where Q is the finite set of states which make up M, Σ is the allowable alphabet which our language is over, δ is a function $\delta : Q \times \Sigma \to Q$ (which one can think of as the transitions between states), $q_0 \in Q$ which is the unique start state, and then finally $F \subseteq Q$, which is the set of final or accepting states. As we can see, every automaton has exactly one start state, but may have multiple final states. As well, let $w = a_0a_1...a_n$, where $a_i \in \Sigma$. Then we say M accepts w if we can find $r_0, r_1, ..., r_m \in Q$ such that:

1. $r_0 = q_0$
2. $r_{i+1} = \delta(r_i, a_{i+1}), 0 \leq i \leq n - 1$
3. $r_n \in F$

That is, if we can find a sequence of states which ends at an accepting state, the word is accepted.

DFAs are much more useful as a visual guide though, so I think it is a good idea to go over an example. Let's take the regular expression ab^*. A (complete) DFA which accepts it may look like this:



Note that the above is a *complete* DFA, compliant with our rigorous definition. Therefore each state has two transitions, one for each letter in our alphabet. You may notice though that if a word enters state 2 it would just get "stuck" and loop over the rest of the word, never reaching the accepting state. In practice, these "dump" states are usually left out of the diagrams and are just assumed to exist. Now, what does this picture represent? First we take our candidate word, say *abb* (note that this is the *string abb* and NOT the regular expression). Then we start at the start state (usually denoted as state 0), checking if our word meets the criteria to transition to the next state. In this case, we see that the first character of our word is a, therefore we've "used up" that character, transitioning us to the appropriate state, in this case state 1. Now we look at the rest of the word bb, and see if we meet the criteria for this new state. The next character is b, and we see that the diagram is telling us, when we see a b, go from state 1 back to state 1. So we do that, and now we're left with one last character, also b. Similar as before, that means we go from state 1 back to state 1. At this point, we've "used up" the entire word, and we're at the

final state, denoted by the double circle in the diagram. That means that the word is accepted by the DFA, which means the word is in the regular expression represented by the DFA. That is, we've verified that $abb \in \mathcal{L}(M)$, where M is the DFA which represents ab^*. Formally, this acceptance can be represented by the following sequence of transitions:

1. $r_0 = 0$
2. $r_1 = \delta(0, a) = 1$
3. $r_2 = \delta(1, b) = 1$
4. $r_3 = \delta(1, b) = 1 \in F$

Now, what would it look like for a word to *not* be accepted? Let's take bb to be our word, and use the same DFA. Then at the very start, we would have to transition to state 2, thus ending on a non-final state. Thus the word is not accepted. Similarly, if our word was $abba$, then for the first three characters, we would play the same game as before, but then after we accepted the final b, we would have one final a, and have to, again, transition to state 2, so we don't accept that word. In both of these cases, we've verified that $bb, abba \notin \mathcal{L}(M)$. And again, we can represent the rejection of bb as the sequence:

1. $r_0 = 0$
2. $r_1 = \delta(0, b) = 2$
3. $r_2 = \delta(2, b) = 2 \notin F$

3.3 Closure Properties

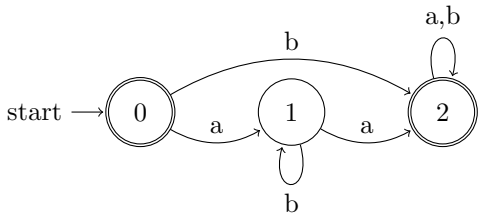
Since regular expressions (which represent regular languages) are equivalent to DFAs, this gives us a visual way to see that regular languages are closed under taking complement. That is, if L represents a regular language, then $L^c = \Sigma^* \setminus L$ is also regular (remember that Σ^* is all possible words over an alphabet). Why is that the case? Below will be a rigorous proof, followed by an easier to follow example.

Lemma 3.1. *If L is regular then L^c is regular.*

Proof. Let L be a regular language, and M be a DFA that accepts L. We know that $M = (Q, \Sigma, \delta, q_0, F)$. Thus, for all $w \in L$, there is a path from q_0 to some $q_f \in F$. As well for all $u \notin L$, there is a path from q_0 to some $p_f \notin F$. Let $\bar{M} = (Q, \Sigma, \delta, q_0, Q \setminus F)$. Since δ is a function, the path we get from w is unique. Thus, $\bar{M}(w)$ will have the same path as $M(w)$, except now $q_f \notin Q \setminus F$ (our new accepting states). Similarly for u, we know that the final state for $M(u)$ is p_f, which in $\bar{M}$ is an accepting state, since $p_f \in Q \setminus F$. Thus, $\bar{M}$ accepts precisely the words NOT accepted by M, and nothing more. Thus $\bar{M}$ accepts L^c, therefore L^c is regular. □

Example: Suppose $L = \mathcal{L}(ab^*)$ as in Section 3.2. We know that all words which are accepted by L must end at the accepting state. Therefore, if we turn every accepting state into a non-accepting state, and every non-accepting state into an accepting state, then we would have a DFA which precisely accepts words which were not in the original language. Note that here it is important that our DFA explicitly be complete.

Thus we can see that the automaton which accepts L^c is going to be:



Therefore, because we can express the complement of a regular language as a DFA, and because DFAs are equivalent to regular language, the complement of a regular language must also be regular, as we'd expect. An important consequence of this then is that regular languages are also closed under intersection:

Theorem 3.2. *If L and K are regular languages, then $L \cap K$ is regular*

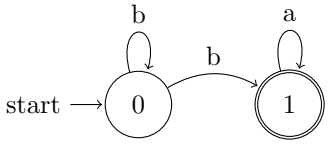
Proof. Let L and K be regular languages. Then by DeMorgan's Law, $L \cap K = (L^c \cup K^c)^c$. However, we know that L^c and K^c are regular (Lemma 3.1). We also know that the union of two regular languages is also a regular (see Section 3.1). Finally, we must take the complement again, but since we're taking it from a regular language, the resulting language must still also be regular. Thus, $(L^c \cup K^c)^c$ is regular, therefore $L \cap K$ is regular. □

This fact, that regular languages are closed under intersections, will lead us to a very useful conclusion. However, please note that the method used to *prove* that $L \cap K$ is regular is *not* the same method that will be practically used to compute the intersection (which we will see in Section 4.2). If K and L are NFAs(which will be explained in Section 4.1), the above method requires converting them to DFAs (which can be exponential in cost) whereas the product construction method explained further down does not (only having quadratic cost).

4 Constructions on Regular Languages

4.1 NFA and Subset Construction

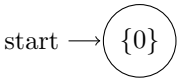
A related type of automaton is called a Non-deterministic Finite Automaton (NFA). These are very similar to DFAs, except when leaving a state, there may be more than one transition with the same symbol (thus non-deterministic). Formally, this can be thought of as having δ represent a *relation*, as opposed to a function. NFAs can also be specified by listing their transitions, that is, their labelled edges. Take the following NFA as an example:



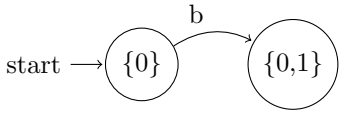
Above, the set of transitions would be $\{(0, b, 0), (0, b, 1), (1, a, 1)\}$. We can see that if I take the word ba, I could do one of two things. One accepting path might be to transition to state 1, then loop onto state 1, thus accepting the word. Instead, I could loop onto state 0, but then I'd be stuck at state 0 needing to consume a b, but unable to do so. So it would seem that ba is both accepted and not accepted by this automaton? That's okay, since as long as we can find *at least* one path, the word is accepted by the NFA. Now, it may seem like NFAs are unimportant to our discussion of regular languages, but in fact they are incredibly valuable. At first glance, it may seem that NFAs must accepts strictly more languages than DFAs, and thus are outside the scope of the regular languages. However, we will shortly show that in fact, DFAs and NFAs are equivalent. This will be important when we want to use regular expressions in any sort of computation. While it is easiest to understand this equivalence via an example, it is worth understanding abstractly what is happening.

Suppose A is the NFA we want to convert to a DFA A' over the alphabet Σ. Let $\sigma \in \Sigma$, $\{p\}$, and $\{q_1, q_2, ..., q_n\}$ be states of A, with $\{p\}$ being the start state. We start by saying that (p) is the start state of A'. Suppose now that we have the transitions (p, σ, q_1), (p, σ, q_2),..., (p, σ, q_n) in A. Then our DFA A' will have the transition $(\{p\}, \sigma, \{q_1, q_2, ..., q_n\})$. We now repeat this process, looking at all transitions from the states $q_1, q_2, ..., q_n$ that share a common transition label. Finally, if q_f is a final state of A, then all states which contain q_f in A' are also final states. This construction is called the *subset construction*. As stated previously, the subset construction is easier to understand with an example. We shall turn the previous NFA into an equivalent DFA.

Example: First we'll start with the start state, writing as so:

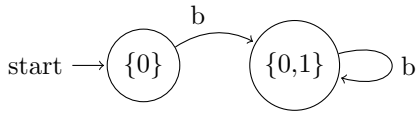


Next step, we see where all transitions of the NFA from state 0 go to. We can see that there's a transition b that loops back, and as well a transition b that goes to state 1. We express these options by creating a state that is a combination of both of those:

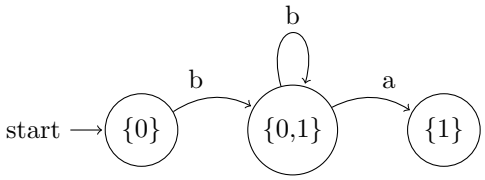


We now essentially repeat the above step, asking what transitions go out from the states 0 OR state 1. We've already said that from state 0, b goes to either state 0 or state 1. As well, we can see that state 1 has no other b transitions.

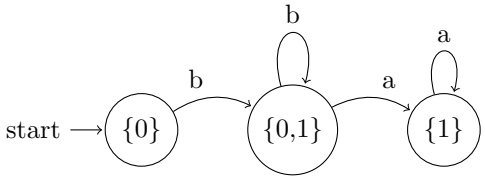
We can denote that via a loop:



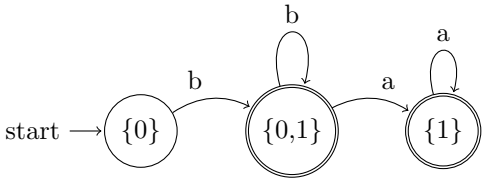
Also notice that from state 1, if we see an a we transition ONLY to state 1. Thus we need add another state:



Almost done! We then check to see what transitions go out of state 1, and we see that it's only a that loops back onto itself. That is, it goes from state 1 back to state 1. Thus, we need not add another state, but instead just a transition like so:



The very last thing we need to do is to add the final states, which is easy. Any state that contains a final state (in this case, state 1) becomes a final state in our new DFA. So the final DFA (which is equivalent to the original NFA) is:



Which means that, after all that work, we can say that our original NFA represents the following regular expression $bb^* + bb^*aa^* = bb^*(\epsilon + aa^*)$. As we have shown, this process can be used to convert any NFA into an equivalent DFA. Therefore, we can see that any language which is accepted by an NFA is also accepted by a DFA. As well, any language accepted by a DFA is also accepted by an NFA, since DFAs are just a special case of NFA. Thus, DFAs are equivalent to NFAs, since they accept the types of languages.

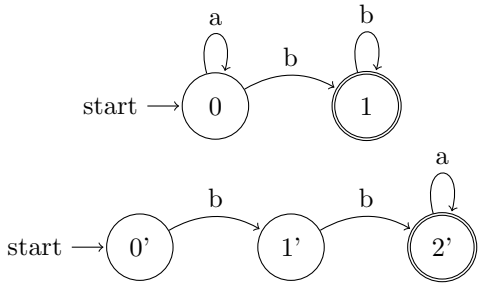
You may now be wondering what the importance of this is, and in fact there are a few. First it means we need not worry about the transitions going out from the stats, since if we create an NFA, we can just convert it. As well, when creating an automaton to accepts $\mathcal{L}(r)$, for some regular expression r, the automaton we end with will almost certainly be an NFA. Finally, and arguably most importantly, note that the subset construction is just that, a construction made of subsets of the NFA's states. That means that if we have an NFA of size n (here, we denote the size as being the number of states and transitions), then to convert it to a DFA we may end up with an automaton of size up to 2^n, that is, exponential. Therefore, it is clear that one would want to avoid this technique as much as possible. However, this may be difficult, as when we convert a regular expression to an automaton, it must be first converted to an NFA, then to a DFA.

4.2 Product Construction

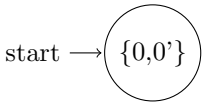
Another important construction is the *product construction* of two NFAs. Remember that regular languages are closed under intersection (see Section 3.3), so we should be able to find an automaton that accepts the intersection. And

we can! Suppose we have two automata, A and A', and we want to find $A \cap A'$. Let p, q be states in A and p', q' be states in A'. Now suppose $x \in \Sigma$ (here the alphabet we are concerned about is the intersection of the alphabet for A and for A'). Then if (p, x, q) is a transition in A, and (p', x, q') is a transition in A', then $((p, p'), x, (q, q'))$ is a transition in $A \cap A'$. And if q_f and q'_f are final states of A and A', respectively, then state (q_f, q'_f) is a final state (note that both states *must* be final states for it to be a final state in the new automaton).

Once again, the above description is much easier to understand with an example: Suppose we have the following two regular expressions: a^*bb^* and bba^*, which are represented by the following two NFAs, respectively:

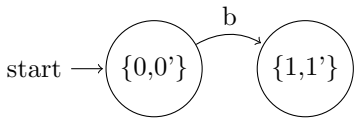


By inspection, we might be able to tell that the only word that they both share is bb, but we will confirm that this is true via the product construction. We start with a state that is the combination of the starting states of both automata, like so:

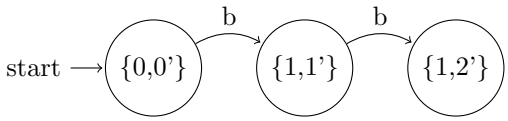


We then look at all transitions out of both starting states in both automata, and make note of all similar transitions between the two of them. That is, all transitions from state 0 to state 1 which take in the same symbol. In this case, we have the b transition from state 0 to state 1 in both automata, so we add

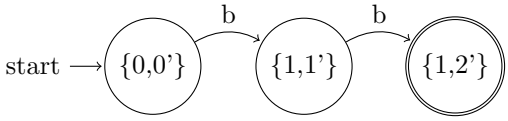
that going out from our new state {0,0'}:



Repeating the process, we see the only similar transition between states 1 and 1 is a b transition, though in this case they go to different states (one changes states the other just loops). But that's okay! We just add a new state to get the following automaton:



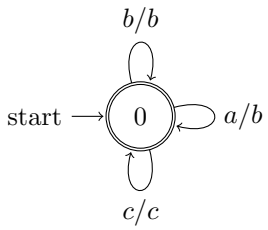
The final step is to find all of the new states that contain one final state from BOTH automata, and that becomes the new final state. Therefore, $a^*bb^* \cap bba^*$ is accepted by:



Which confirms our intuition that it is just bb. However, this gives us a concrete way, that is to say, an algorithm, for taking the intersection of two automata. And importantly, we can do this much quicker than the subset construction. Let n be the number of states and transition of our DFA (that is, the size). Then, this new DFA will have around n^2 new states, as oppose to the subset construction which was on the order of 2^n. This is because instead of taking all possible *subsets* of the states, our worse case scenario only uses all possible *pairs* of states.

5 Transducers

A special type of automata (different from both NFAs and DFAs) are transducers. These have the same "form" as regular automata, but instead of checking if a given word is in a certain regular language, a transducer takes a given word, and outputs a set of words (though it is possible that this set is empty). For example, take a transducer t over $\Sigma = \{a, b, c\}$ that takes a word, and changes every instance of a to b. For example, $t(abca) = \{bbcb\}$. How would one describe such a thing? Probably how you would imagine:

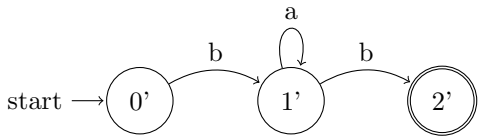


So if we take the word $abca$, the first a is used to loop over state 0. However, in doing so, we must substitute the a for a b (that's what a/b means). That is, a is the *input* label and b is the *output* label. Next up are b and c which we just loop over, since every instance of b or c gets replaced with an instance of b or c, respectively. Finally we hit the last a, where we loop again, consuming the a and outputting a b. Thus, our new word is $bbcb$. Technically our transducer outputs the set $\{bbcb\}$, however when it is just a singular word it is convenient to talk about the word directly. An important distinction between transducers and NFAs is that the transitions for a transducer have an input label and an output label, as oppose to an NFA which has only an input label.

A formal definition for a transducer T is similar to that of a DFA, but is a 6-tuple instead of a 5-tuple (see Section 3.2) and looks like so: $T = (Q, \Sigma, \Gamma, I, F, \delta)$. Like before, Q is a finite set of states, and Σ is the *input* alphabet. This is opposed to Γ, which is the *output* alphabet, and can be different to Σ (smaller, larger, or completely separate). We then also have the set of start states $I \subset Q$ (that is, we can have multiple start states, however for our purposes one start state will be sufficient), as well as $F \subseteq Q$, which is the set of final or accepting states. Finally, we have δ which is the transition relation, defined as $\delta \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times (\Gamma \cup \{\epsilon\}) \times Q$, where once again ϵ is the empty string (this allows us to add or delete symbols from our word). Note that δ is a *relation* and not a *function*, since it may be non-deterministic.

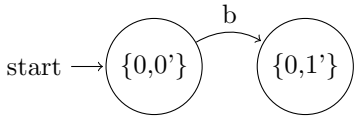
As noted, a transducer need not, and often will not, be deterministic. In that case, we really do get as output a set of words, that being all the possible paths the word could have taken through the transducer. This new set is regular, assuming that the input language is regular. It would stand to reason then that you should be able to take an entire regular language, and run that through a transducer, to get a new (modified) regular language. In fact, we can, by using an altered version of the product construction.

Example: Let's use the above transducer (which changes all a's into b's) and run it on the following language $\mathcal{L}(ba^*b)$, defined by this automaton:



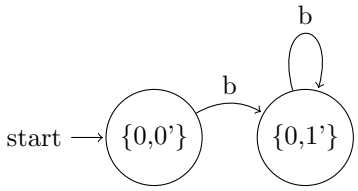
For convenience, we're going to simplify, our alphabet to just $\{a, b\}$, removing all the c transitions. As before, our first step is to create a start state from the

start state of both automata, and see which transitions they have in common. In this case, they both have transitions for the character b going into both state 0 from the transducer and state $1'$ from the DFA. Graphically, we can represent it like so:



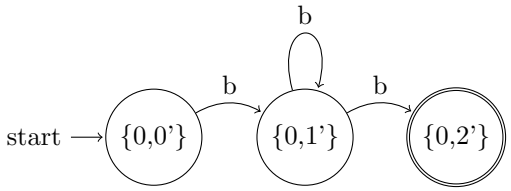
Here is now a very important point. While in the standard product construction we need only worry about transitions on similar symbols, because the transducer also produces an output, that must also be reflected in the final automaton. That is, we use the input label of the transition for matching, but when creating the automaton, we must create a new transition which has as label the output label of the transducer, since we're creating a new regular language, not a transducer. This will be much easier to see via the next step.

In this step, we consider both transitions from state 0 and state $1'$. We have both a loop over state 0 with input label a, altered to a b, as well as a loop over state $1'$ via a. Both of these combined give us this:



Note how we changed the a transition to a b, as that is what the transducer does, so that must be reflected in our new automaton. Finally, we look at our new states, and see if we need to add any more. From state $\{0, 1'\}$ we use the same a/b transition to work with over state 0, so that will be added. Also, we notice that there are no transitions out of state 2, so that is done. Last step

is checking which of our new states contain the final states of both automata, and designating that as a new final state, similarly to the product and subset constructions from before. Thus, our final automaton looks like this:



Now, this may seem like a pretty trivial example. And it is. All we've done is take the language $\mathcal{L}(ba^*b)$ and converted it to $\mathcal{L}(bb^*b)$, that is $t(ba^*b) = \mathcal{L}(bb^*b)$. However, the idea is incredibly powerful. This means that if we have a certain "property" defined by a transducer t, we can take an automaton A and ask what happens if we run A through t, which will be explained in Section 6.1. That is we can ask questions about the regular language $t(A)$.

6 Theoretical Aspects of *LaSer*

6.1 Properties and Property Satisfiability

The primary purpose of *LaSer* is deciding property satisfiability for a given regular language, thus before we can discuss that it is important to build up to what that term means. Please note that definitions 1 and 2 are from [4].

Definition 1. *Let t be a transducer and L be a regular language. Then we say that L is t-independent if for all $u, v \in L$ and $v \in t(u)$ that implies that $u = v$.*

This is sometimes also stated as L satisfying the property t. Thus, we can think of a transducer t as representing a language property. The only problem with this is that it isn't practical to check every word, especially if the language is infinite. However, we can simplify our search in certain special situations. For example, if a transducer is *input-altering*.

Definition 2. *Let t be a transducer over an alphabet Σ. t is input-altering if $\forall w \in \Sigma^*$, then $w \notin t(w)$.*

Lemma 6.1. *Let t be an input-altering transducer and L a regular language. Then L is t-independent, if and only if $t(L) \cap L = \varnothing$*

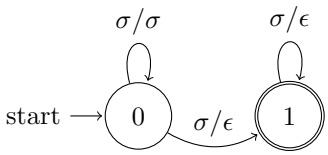
Proof. 1. Let $w \in L$ and $K = t(w)$. Since L is t-independent, in general we would have that $K \cap L = \{w\}$, since if we had another $u \in L$ also being in K, $u = w$. However, since t is input-altering, we also know that $w \notin K$, thus in actuality, $K \cap L = \varnothing$. We now note that $t(L) = \cup_{w \in L} t(w)$, thus our expression $t(L) \cap L$ is made up of a bunch of unions of the empty set. Thus $t(L) \cap L = \varnothing$.

2. Suppose $t(L) \cap L = \varnothing$. It is sufficient to show that it is not possible to have $u, w \in L$ and $v \in t(w)$. Let $w \in L$. As well, since $t(L) \cap L = \varnothing$, there is no $v \in L$ such that $v \in t(L)$, since then the intersection would contain v and be non-empty.

□

Lucky for us, we know that $t(L) \cap L = \varnothing$ can be tested quickly, as we need only use the product construction. An example may now be helpful.

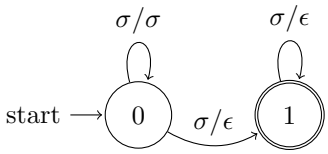
Example: Let $L = \mathcal{L}(a^*b) = \{b, ab, aab, aaab, ...\}$ and $K = \mathcal{L}(ab^*) = \{a, ab, abb, abbb, ...\}$. Within *LaSer* there are pre-set transducers, which represents certain common properties. One of these properties is called the prefix code, which checks if there are any words in the language which are prefixes of other words, also in the language. For this property, the transducer, let's call it px, is as follows (note that px is *non-deterministic*):



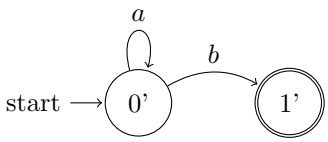
At a high level, we can see that this transducer takes in a word x and then outputs a set of words, which are the non-trivial prefixes of the original word. For example $px(aaba) = \{aab, aa, a, \epsilon\}$. Note that for this example, we are working over $\Sigma = \{a, b\}$, and $\sigma \in \Sigma$, that is, it is any letter in that alphabet. This allows us to generalise the prefix code transducer over other alphabets. This is input-altering, since when we run a word of length n through it, out will pop a word of length at most $n - 1$, or else it wouldn't be a prefix (note that technically the word w is a prefix of the word w, however we ignore it) and thus, $\forall x \in M, x \notin px(x)$ (note that M is any non-empty regular language), so px is

input-altering, and our search is simplified. Without even looking at the actual implementation of the process, we can see that L satisfies the property. Why? Note that besides the word b, no other word starts with a b. Thus, b satisfies the property. Note also how ab is the only word in L which contains a b in the second position. This repeats on with aab being the word word to start with a b in the third position, etc. Thus, it is clear that L satisfies the prefix code property. For K, we can see that it does not satisfy the property, since a is a prefix of ab (note that a is in fact a prefix of all the words in the language, but it is sufficient to just find one such example).

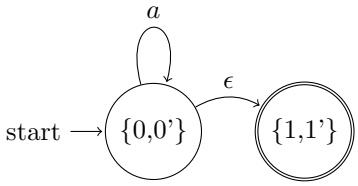
We can also see concretely that L does satisfy the property by checking that $px(L) \cap L = \varnothing$. Reminder that the transducer looks like this:



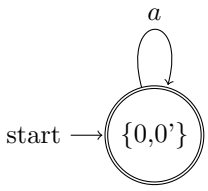
Now let's take the following DFA for the language L:



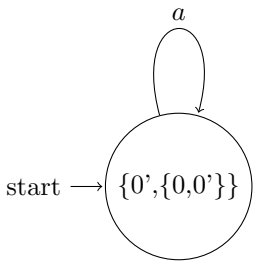
To see if our language L satisfies the property, we must first perform the product construction to get $px(L)$, which becomes the following NFA:



Note that there are algorithms to remove the empty transitions [2] (also called ϵ-transitions), so the above NFA becomes the following DFA:

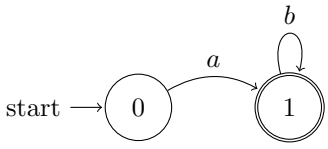


Now that we have the DFA $px(L)$, we must now take the product construction of $px(L) \cap L$, which will return the following DFA

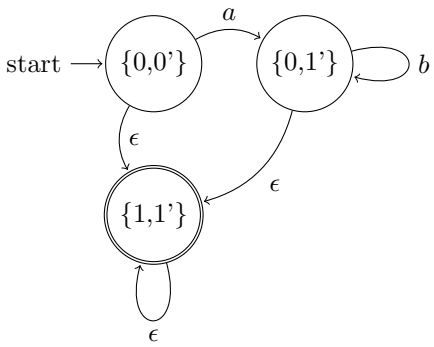


When we take the product construction, we find that while L has transition out of the state via b to an accepting state, $px(L)$ has no b transitions. Therefore, they never "share" an accepting state, which means $px(L) \cap L$ does not have an accepting state, as we can see from the automaton. Thus we've confirmed that $px(L) \cap L = \varnothing$, and therefore as we predicted, L does satisfy the prefix code. That is, there is no sequence of transitions which leads to an accepting state, in this case since no accepting state exists.

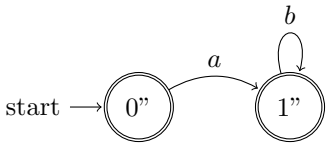
Conversely, we'll check that $px(K) \cap K \neq \varnothing$. Recall that $K = \mathcal{L}(ab^*)$ can be represented like so:



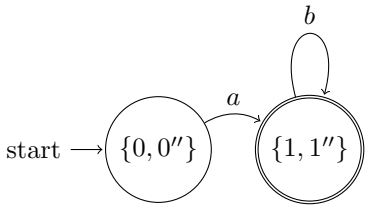
First note that $px(K)$ can be represented by the following automaton:



Which we can simplify to the following by removing ϵ-transitions:



If we then take $px(K) \cap K$ we get this final automaton:



This is precisely the automaton K that we started with, and evidently it is non-empty. Thus K does not satisfy the property. However, we don't have a way of knowing what word "breaks" this property satisfaction. Thus, we will have to develop some new machinery to create a *witness*, that is, a word in the language that does not allow the property to be satisfied, which would be helpful information.

Remember that the above "tests" are only valid if t is input altering. However, this does not give us any information as to what the problem is, so we need to introduce some more notation. Let t be a transducer and L a regular language. Then we have the two following definitions:

Definition 3. *The transducer* $t \downarrow L : y \in (t \downarrow L)(x)$ if and only if $y \in t(x)$ *and* $x \in L$

Definition 4. *The transducer* $t \uparrow L : y \in (t \uparrow L)(x)$ if and only if $y \in t(x)$ *and* $y \in L$

Both $t \downarrow L$ and $t \uparrow L$ are themselves transducers, the former being the transducer t with *inputs* limited to words in L and the latter being the transducer t with *outputs* limited to words in L. Thus, in a sense, we can take our transducer t and limit it (in both input and output) to words found only in L. As well, we need to define what a *relation* is.

Definition 5. *If* t *is a transducer, then* $R(t) = \{(x, y) : y \in t(x)\}$.

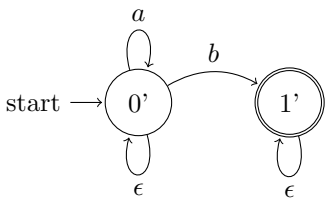
Our original condition to test satisfiability can be expressed by the following equation:

$$R(t \downarrow L \uparrow L) = \varnothing.$$

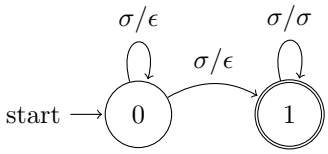
This is great, as it means that if $R(t \downarrow L \uparrow L) \neq \varnothing$, then we can find a witness pair that is causing us our troubles, as well as not requiring t to be input-altering. An important question one may ask is how exactly one computes either $t \downarrow L$ or $t \uparrow L$. The answer, as it always seems to be, is via product construction. However, while before we took a transducer and an automaton and got as output a new *automaton*, we will now be outputting a *transducer*. Below will be an explanation for $t \downarrow L$. A very similar computation can be made to calculate $t \uparrow L$. Abstractly, suppose we have a transition from t of the form $(p, x/y, q)$, which means if you're at state p with input symbol x, change it to symbol y and move to state q. Now suppose in our automaton L we have the transition (p', x, q'), where x in both cases are the same. Then our new transducer $t \downarrow L$

will have transition $((p, p'), x/y, (q, q'))$ [9].

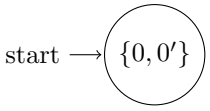
Example: Let L be the language represented by a^*b, which as an automaton can be viewed as follows:



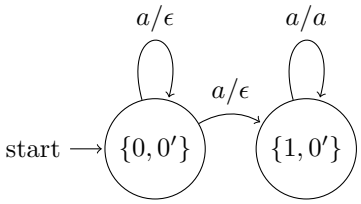
Note that the ϵ-transitions do not change the behaviour of the automaton, but they will be necessary later on. This time, we want to check if any of the words in this language are the *suffix* of any of the other words. We will use the following transducer, sx:



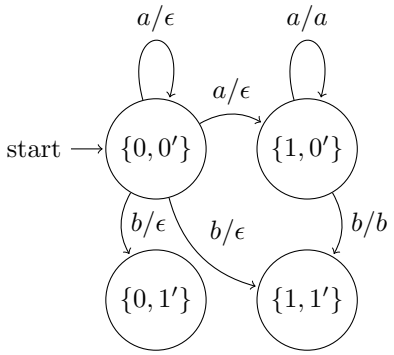
Note that $sx(w)$ = all proper suffixes of w. To get $sx \downarrow L$, we need to start from both start states of our automataj as before:



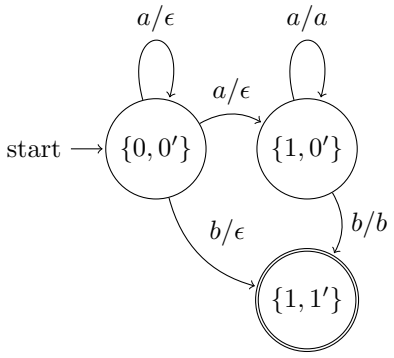
Now, from state $0'$, we just loop over if we see an a, and in state 0, we loop over, replacing with an ϵ. We can also see that from state 0, we can replace a with ϵ, and move to state 1. From state 1, we can see that we don't replace any letters, and again on state $0'$, if we see an a, we loop over it. Thus we get the following new states:



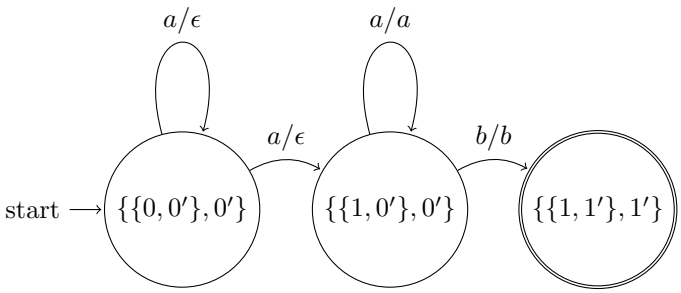
As we will see, we have dealt with all the a transitions. As for b transitions, from state 0 we know that we will be deleting it, and from state $0'$, we can only transition to state $1'$, however this transition will ultimately not be necessary. We can also see that from state 0, we can instead delete b and transition to state 1. Finally, from state 1 we recall that we don't change the letter, and loop back to state 1, and from state $0'$ we simply transition to state $1'$. All this means that we can add two new states and three new transitions, as follows:



Finally we notice that the state $\{0, 1'\}$ is unnecessary since there are no transitions out of it, and it is not an accepting state. We can denote state $\{1, 1'\}$ as an accepting state for our final automaton $sx \downarrow L$:



Now that we have restricted the input of our transducer to words in $\mathcal{L}(a^*b)$, we now want to restrict its output to those same words, denoted as $(sx \downarrow L) \uparrow L$. Doing so will yield the following transducer:



From here, we get the following pair of values, (ab, b) with the following path:

$$(\{\{0, 0'\}, 0'\}, a/\epsilon, \{\{1, 0'\}, 0'\}) \rightarrow (\{\{1, 0'\}, 0'\}, b/b, \{\{1, 1'\}, 1'\})$$

This means, in our case, that b is a suffix of ab, which are both in $\mathcal{L}(a^*b)$, and thus it does not satisfy our property. Importantly though, this technique gives us a way to test for property satisfiability, while also explicitly giving us a witness. It is easy to find a witness, as all we need to do is find a path from a start state and an accepting state. Our input word will just be a concatenation of all the input labels on our path, and our output word will be the concatenation of all the output labels (or simply passing our input word through the transducer).

6.2 Universality and Maximality

Universality is a decision problem that asks whether a given regular expression or NFA represents all possible words over the alphabet. For example, $\mathcal{L}((a+b)^*)$ represents all words over the alphabet a, b. However, the language $\mathcal{L}(a^*)$ is not universal, as it does not contain any words with a b. Thus, given a regular language L and an alphabet Σ we can ask: $L = \Sigma^*$? Since regular languages are closed under complements, this is equivalent to the statement: $L^c = \varnothing$. Note also that since regular languages are closed under intersection, we can restrict our language to any regular subset of Σ^*. That is, if L, M are regular languages, and $L = \Sigma^*$, then $L \cap M = \Sigma^* \cap M = M$. Now suppose that we want to see if a given language is *maximal* with respect to a given property.

Definition 6. *Let L be a regular language and t a transducer. If $\forall w \in \Sigma^*$, $w \notin L$, $L \cup \{w\}$ is not t-independent, then L is maximal with respect to t.*

First, we need to make sure the language actually satisfies the property t. After we've checked that it does satisfy the property, our high level strategy will be to check whether all the words that we can get out from the transducer, $t(L)$, as well words can form words in our language, $t^{-1}(L)$, as well as the L itself form Σ^*. That is, does $t(L) \cup t^{-1}(L) \cup L = \Sigma^*$? We can then take the complement on both sides to simplify to $t(L)^c \cap t^{-1}(L)^c \cap L^c = \varnothing$. And since on both sides we have regular languages, we can again restrict the alphabet to whatever other regular expression we want. At this point, an example may be helpful. Let's take our language $L = \mathcal{L}(a^*b)$ from Section 6.1, and a new language $K = \mathcal{L}(ab^*a) = \{aa, aba, abba, ...\}$. We know from before that L satisfies the prefix code, and similarly it can be shown that K also satisfies that property. Thus, we can also ask, are either L or K maximal with respect to px? Indeed, L is maximal, however K is not. That is because we can add the word b to K. Since b is not the prefix of any of the words in K (as they all start with

an a), and none of the words are prefixes of it (for a similar reason). Therefore K is not maximal with respect to px.

7 Computational Complexity

7.1 Big O Notation

Big O notation is a corner stone of computational complexity theory. Formally, we have that a function $g(n)$ is $\mathcal{O}(f(n))$ if there exists a constant c such that $g(n) \leq cf(n)$ for all non-negative values of n [1]. At a high level, it attempts to a describe the worst case performance for a certain algorithm. Some well known big O complexities are:

1. Bubble Sort: $\mathcal{O}(n^2)$
2. Merge Sort: $\mathcal{O}(nlog(n))$
3. Solving the traveling salesman problem: $\mathcal{O}(n^2 2^n)$ (Held-Karp algorithm)[3]

For our purposes, the specifics aren't exactly important. What is important is understanding that this is a measure of how well an algorithm scales in time or space with different amounts of inputs. In our case, we want to make a distinction between those algorithms which have a polynomial big O, something of the form $\mathcal{O}(n^k)$, versus something which is exponential, $\mathcal{O}(2^n)$. This all to say, problems which have a $\mathcal{O}(2^n)$ are much more time and/or space consuming then $\mathcal{O}(n^k)$, since for large enough n, $2^n \gg n^k$.

7.2 Complexity Classes and the Problem with Maximality

Keeping in mind this polynomial versus exponential division, we can create classes of problems which have a similar type of complexity. However, it should be noted that these complexity classes deal with decision problems. That distinction isn't very pertinent to our discussion, though I'd be remiss not to mention it. One of the "easiest" types of problem to solve are in a class of problems

with a polynomial time algorithm, which is usually called P, or more explicitly *P-time*. In fact, there are many hierarchies of classes, the most famous of which are $EXPTIME$, $PSPACE$, NP, and P. Without going into too many details, suffice to say, $EXPTIME$ is all languages which can be decided in at most an exponential amount of time, $PSPACE$ are those languages/problems which can be solved using at most a polynomial amount of space (but possibly an exponential amount of time). NP is those which have a *non-deterministic* polynomial time algorithm to solve (essentially, it is those problems which are difficult to solve directly but it is easy to check if a given answer is correct), and then finally P, which as previously discussed, take a polynomial amount of time to solve.

For most of these classes we have an associated concept of *completeness*. That is, a problem is *complete* with respect to its class if all other problems in that class can be reduced to it. For example, we have a set of problems called *NP-complete*, which are the hardest problems in NP. A classic example is 3-SAT (for more information see the *Cook-Levin Theorem*). In practice, that means that all problems in NP can be reduced to any of the *NP-complete* problems. In $PSPACE$ we have a similar concept called *PSPACE-complete*, which are the hardest problems in $PSPACE$. Why is this important? Well, being as $PSPACE$ are those problems solvable in a polynomial amount of space, we can imagine them as the most reasonable problems to solve. It doesn't mean that they are easy (they aren't), but they can (in theory) take up less resources than $EXPTIME$. Thus, problems which are *PSPACE-complete* can be viewed as the most difficult of the "reasonable" problems. The importance here is that deciding maximality (or in general, universality) is known to be *PSPACE-complete*. Asking if the intersection of m regular languages is empty

is also *PSPACE-complete* [7]. An interesting point to make here is that even if $\Sigma = \{a\}$ (a single letter), deciding if $L = \Sigma^* = a^*$ is *NP-complete* [5]. Which means that, in general, and for a complex enough language or property, it may take a very long time to compute the answer. This may become an issue as *LaSer* is an active server, which requires resources to run and operate. Thus it would be nice if we didn't have to hog all the resources to ask a single question. Thus, pseudo-maximality was developed!

8 Pseudo-Universality and Pseudo-Maximality

8.1 Pseudo-Universality: Why Even Bother?

As stated before, checking whether a given regular language is maximal is expensive (see Section 7.2). However, in some practical scenarios, we need only know if the language is *approximately* maximal. If we thus relax our notation of maximality, we can try and answer a different problem called *pseudo-maximality*, which decides if a regular language is p%-maximal. Now, what does it mean for a given language to be p%-maximal? Suppose I only care if my language is 95%-maximal. Remember that a given language is maximal (with respect to a given property) if we cannot add any new words to it that also satisfy the property. If I'm not bothered by the idea that my language may have a small percentage of *not* being maximal, we can be clever and make the question much more efficient to solve. However, since the two are intimately related, we shall take a quick detour to discuss pseudo-*universality*. That is, we want to check with 90%, 95%, 99%, etc. certainty that our language L is Σ^n. We shall first focus on finite languages. Let A_s be a generic alphabet of the form $\{0, 1, 2, ..., s-1\}$, for some s, and M an NFA. Then if M accepts only words from A_s^n, we say M is a *block NFA*. First, we shall focus our attention on words from A_s^n, that is, words of a fixed length $n \in \mathbb{N}$. That is $L \subseteq A_s^n$ is p%-universal if $\frac{|L|}{|A_s|^n} \geq \frac{p}{100}$.

The following is the pseudo-code for one possible way of testing/deciding p%-universality for a block NFA:

UnivBlockNFA(a, ϵ):

```
/* a is an automaton, ϵ is our desired precision          */
```

$n :=$ the block length of $\mathcal{L}(a)$;

$l := \lceil \frac{1}{\epsilon^2} \rceil$;

for $i = 0;\ i < n;\ i++$ **do**

  $w =$ **selectUnif**(A_s, l);

  if $w \notin \mathcal{L}(a)$ **then**

    return *False*;

  else

end

return *True*;

Essentially what we are doing is checking if we've found a word not in our original language, or until we've iterated enough times that we are happy with the result. As well, we have the function **selectUnif** which selects a word of length l, over the alphabet $\{0, 1, 2, ..., s-1\}$ using uniform distribution. If L is universal, then our algorithm will always return that it is universal. That is, we won't get any false positives. However, it is still entirely possible to have a non-universal language, and iterate a million times, and decide that it is pseudo-universal. That being said, as we iterate more, the probability that that is the case decreases. It can be shown that this probability decreases with the number of tries quadratically to zero [5].

8.2 Proofs

You may recall that all of our technical solutions are based on finding a word in a language that ought to be empty. While we showed above that pure universality and pure emptiness are logically equivalent, we should check that the same holds

true in the pseudo case. That is, we want to show that a regular language is $p\% - universal$ if and only if it is $(1-p)\% - empty$. Recall that for regular languages m_i, $m_1 \cap m_2 \cap ... \cap m_n = \varnothing \iff m_1^c \cup m_2^c \cup ... \cup m_n^c = \Sigma^*$. In general, we want to check that $\frac{number\ of\ words\ in\ our\ language}{total\ number\ of\ words} \leq p$. Also note that that we will be proving this for an arbitrary language. Since when we pass a regular language through a transducer we get out a regular language, we need only check that our condition holds when taking intersections. First we shall show that for a finite language, this holds.

Lemma 8.1. *If $L_1, L_2, ..., L_m$ are subsets of $A = \{a_1, a_2, ..., a_m\}^n$, then $\frac{|L_1 \cap L_2 \cap ... \cap L_m|}{|A|^n} \leq p \iff \frac{|L_1^c \cup L_2^c \cup ... \cup L_m^c|}{|A|^n} \geq 1 - p$*

Proof. Done in two parts, first going to the right.

1. Let $\frac{|L_1 \cap L_2 \cap ... \cap L_m|}{|A|^n} \leq p$, $0 < p < 1$. Then we know that $\frac{|(L_1^c \cup L_2^c \cup ... \cup L_m^c)^c|}{|s^n|} = \frac{|s^n \backslash L_1^c \cup L_2^c \cup ... \cup L_m^c|}{|A|^n} = \frac{|s+A|^n - |L_1^c \cup L_2^c \cup ... \cup L_m^c|}{|A|^n} = 1 - \frac{|L_1^c \cup L_2^c \cup ... \cup L_m^c|}{|A|^n} \leq p$ therefore $1 - p \leq \frac{|L_1^c \cup L_2^c \cup ... \cup L_m^c|}{|A|^n}$

2. This direction is incredibly similar. Suppose $\frac{|L_1^c \cup L_2^c \cup ... \cup L_m^c|}{|A|^n} \geq 1 - p$ then $\frac{|L_1^c \cup L_2^c \cup ... \cup L_m^c|}{|A|^n} = \frac{|(L_1 \cap L_2 \cap ... \cap L_m)^c|}{|A|^n} = \frac{|s^n \backslash (L_1 \cap L_2 \cap ... \cap L_m)|}{|s^n|} = \frac{|A|^n| - |L_1 \cap L_2 \cap ... \cap L_m|}{|A|^n} = 1 - \frac{|L_1 \cap L_2 \cap ... \cap L_m|}{|A|^n}$. So, $1 - \frac{|L_1 \cap L_2 \cap ... \cap L_m|}{|A|^n} \geq 1 - p$ thus, $\frac{|L_1 \cap L_2 \cap ... \cap L_m|}{|A|^n|} \leq p$

□

Therefore, if a given regular language is $p\% - empty$, it's complement is $(1-p)\% - universal$, which is what we would expect, given that language is *finite*. That is, this seems to be a concept that we can in fact extend to the pseudo case. Now that we have an understanding of the finite case, we can now generalize this for infinite languages. Let $W : \Sigma^* \to [0, 1]$, $\Sigma_{x \in \Sigma^*} W(x) = 1$ (that is, W is a probability distribution). We want the following lemma:

Theorem 8.2. $W(L_1 \cap ... \cap L_m) \leq p \iff W(L_1^c \cup ... \cup L_m^c) \geq 1 - p$

Proof. For this, we must note that $W(L^c) = 1 - W(L)$, as $W(L) + W(L^c) = 1$ (by definition)

1. Suppose $W(L_1 \cap ... \cap L_m) \leq p$. Then $W(L_1 \cap ... \cap L_m) = W((L_1^c \cup ... \cup L_m^c)^c) = 1 - W(L_1^c \cup ... \cup L_m^c) \leq p$. Thus $W(L_1^c \cup ... \cup L_m^c) \geq 1 - p$

2. Suppose $W(L_1^c \cup ... \cup L_m^c) \geq 1 - p$. Then $W(L_1^c \cup ... \cup L_m^c) = W((L_1 \cap ... \cap L_m)^c) = 1 - W(L_1 \cap ... \cap L_m) \geq 1 - p$. Therefore $W(L_1 \cap ... \cap L_m) \leq p$

□

And thus, we can see that $p\% - empty$ is equivalent to $(1-p)\% - universal$, which is what we should expect, in both the finite and the infinite case. Which is great for us, as it allows to focus just on $p\% - empty$, as that allows us to create a witness, or counter-example.

8.3 Pseudo-Maximality: There's More?

Now the question is, is there a similar process for discussing *maximality*? Perhaps a *pseudo*-maximality? We shall, as before, restrict ourselves to the finite case, that is to a block code. Suppose we have some transducer t which describes a property, and an automaton M which satisfies this property, and a generic alphabet A_s, with $\mathcal{L}(M) \subseteq A_s^l$, for some l. Thus, to show that M is *not* maximal, we need to find $w \in A_s^l \setminus \mathcal{L}(M)$ such that $\mathcal{L}(M) \cup \{w\}$ satisfies the property t. It would be convenient as well to know what that this word w is, so that we can adjust accordingly. In this case, our language is $p\%$-t-maximal if $\frac{|A_s^l \cap (t(M) \cup t^{-1}(M) \cup M)|}{|A_s|^l} \geq p$ [6].

One possible randomized algorithm that achieves this, is the following:

pseudoMax(M, t, n):

`/*` M `is an automaton,` t `is our property (transducer), and` n `is the number of iterations */`

$l :=$ length of the words in $\mathcal{L}(M)$;

for $i = 0;\ i < n;\ i++$ **do**

  $w :=$ **selectUnif**(A_s, l);

  if *(*$w \notin \mathcal{L}(M)$ *and* ***SAT****(*$\mathcal{L}(M) \cup \{w\}, t$*)* **then**

    return w;

  else

end

return *None*;

Here, A_s is, as before, our generic alphabet $\{0, 1, 2, ..., s-1\}$. Similarly, the function **selectUnif** simply takes a randomly generated word of length l to be tested, as in the **UnivBlockNFA**. As well, the **SAT** simply checks if the given language satisfies the given property, as discussed in section 6.1. The final part of the algorithm takes this new randomly generated word, and checks if after we add it to our original language, the language still satisfies the property. If we can find a $w \notin \mathcal{L}(M)$, then we know that M is not maximal. Testing whether a given w is in $\mathcal{L}(M)$, and adding a given w to $\mathcal{L}(M)$ are standard methods in automata software, for example *FAdo* [8]. Recall that $|M|$ is the total number of states and transitions in the automaton. If M is a DFA, then these methods take time $\mathcal{O}(|M| + |w|)$. Moreover, since M is a block automaton of length l and $|w| = l$, we have that $|M| + |w| = \mathcal{O}(|M|)$. Essentially what we are doing is checking if our language satisfies the property over and over again, picking new words each time. Similarly to pseudo-universality, if $\mathcal{L}(M)$ is maximal, our algorithm will return that result. Recall that checking for maximality is *PSPACE-complete*, as well. The time complexity of the above algorithm is

$\mathcal{O}(n|M|^2|t|)$ [6], where, $|t|$ is the total number of states and transitions. Importantly, this is polynomial, much better than our, possibly exponential result from before.

Now we discuss that we can actually do better. Notice how in **pseudoMax**, we have to test for property satisfiability every time, which in practice can be a bit expensive [6]. This is in part because we need to run the product construction on $\mathcal{L}(M) \cup \{w\}$ and t on every iteration. We could, instead, do some of the hard work at the beginning, by only creating one new automaton once. That is to say, instead of checking for satisfiability every time, we can create the new automaton $N = (t \cup t^{-1})(M) \cup M$, which we know should be universal (see Section 6.2). Note that taking the union of two automata can be done by creating a new start state, then creating an ϵ-transition to the start states of the two other automata. Since we know how to remove ϵ-transitions, this is fine [2]. Thus, our new algorithm need only check if some $w \notin \mathcal{L}(N)$. Thus, the above algorithm can be modified as follows:

betterPseudoMax(M, t, n):

```
/* M is an automaton, t is our property (transducer), and n
   is the number of iterations                                */
N := (t ∪ t^{-1})(M) ∪ M;
l := length of the words in L(M);
for i = 0; i < n; i++ do
    w := selectUnif(A_s, l);
    if (w ∉ L(N)) then
        return w;
    else
end
return None;
```

As we can see, in the iterative step here, we only check if our word w belongs to the language, which can be done in $\mathcal{O}(|w||N|) = \mathcal{O}(l|N|)$. As well, computing N, which need only be done once, can be done in $\mathcal{O}(|M||t|)$. Thus, the time complexity for the whole algorithm is $\mathcal{O}(nl|M||t|)$. Note that in general, this will be more efficient than our previous attempt, because in general $|M| \gg l$.

9 Conclusion

As one can see, pseudo-maximality is a practical answer to a theoretical question. While it would be preferred to feasibly check for maximality efficiently, being a *PSPACE-complete* problem, this is not likely to be the case. However, using techniques related to pseudo-maximality, and other randomized algorithms, we know that we can get as high a confidence as we want for a given property maximality. Like most things in life, this is a gentle game of balance between absolute certainty and use of resources. I hope that by the end of this paper you would agree that pseudo-maximality is an acceptable way to test for such an attribute, while also understanding how it works, and having confidence that it will in fact do its job.

In terms of future research, we should address the problem of pseudo-maximality for automata accepting infinite languages. In this case, it seems that random words would be picked according to a chosen probability distribution. What methods could be used, or which probability distributions give the best results are still open for discussion.

References

[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Publishing Company, 1974.

[2] David Mix Barrington. Lecture #22: From λ-nfa's to nfa's to dfa's. https://people.cs.umass.edu/∼barring/cs250s13/lecture/22.pdf, 04 2013. University of Massachusetts Amherst.

[3] Gregory Gutin, Abraham Punnen, Alexander Barvinok, E. Gimadi, and Anatoliy Serdyukov. The traveling salesman problem and its variations. *Kluwer Academic Publishers*, page 33, 06 2001.

[4] Stavros Konstantinidis. Applications of transducers in independent languages, word distances, codes. In *Descriptional Complexity of Formal Systems*, volume 10316, pages 4,5, 2017. Lecture Notes in Computer Science.

[5] Stavros Konstantinidis, Mitja Mastnak, Nelma Moreira, and Rogério Reis. Deciding approximate nfa universality and related problems. Unpublished manuscript.

[6] Stavros Konstantinidis, Nelma Moreira, and Rogério Reis. Randomized generation of error control codes with automata and transducers. *RAIRO - Theoretical Informatics and Applications*, 11 2018.

[7] Klaus-Jörn Lange and Peter Rossmanith. The emptiness problem for intersections of regular languages. volume 629, pages 1–9, 2005. Lecture Notes in Computer Science.

[8] Rogério Reis and Nelma Moreira. Fado - v.1.3.5.1. http://fado.dcc.fc.up.pt/software/, 2011–2018.

[9] Meng Yang. Application and implementation of transducer tools in answering certain questions about regular languages. Master's thesis, Saint Mary's University, 12 2012.