1-2017

Automated Grading and Feedback of Regular Expressions

Himesh Kakkar
hxk1496@rit.edu

Automated Grading and Feedback of Regular Expressions

by

Himesh Kakkar, B.Sc.

THESIS

Presented to the Faculty of the Golisano College of Computer and

Information Sciences

Rochester Institute of Technology

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science in Computer Science, Department of Computer Science

Rochester Institute of Technology

January 2017

Automated Grading and Feedback of Regular Expressions

APPROVED BY

SUPERVISING COMMITTEE:

Dr. Edith Hemaspaandra, Chair

Dr. Ivona Bezáková, Reader

Dr. Zack Butler, Observer

ACKNOWLEDGEMENT

I would first like to thank my advisor, Dr. Edith Hemaspaandra, for her time and patience. She helped me understand how research works and guided me in the right direction when she felt this was needed. In addition, our weekly meetings were extremely motivating and helped me improve the quality of my work for which I will be forever grateful.

Besides my advisor, I would like to thank the rest of my thesis committee: Dr. Ivona Bezáková, and Dr. Zack Butler, for their encouragement, feedbacks, and questions.

I would also like to thank the graders, Nishtha Ahuja and Priyank Singh, for their time and efforts in grading my dataset.

Lastly, I would like to thank all my family and friends for their support, feedback and time.

ABSTRACT

To keep up with the current spread of education, there has arisen the need to have automated tools to evaluate assignments. As a part of this thesis, we have developed a technique to evaluate assignments on regular expressions (regexes). Every student is different and so is their solution, thus making it hard to have a single approach to grade it all. Hence, in addition to the existing techniques, we offer a new way of evaluating regexes. We call this the regex edit distance. The idea behind this is to find the minimal changes that we could make in a wrong answer to make its language equivalent to that of a correct answer. This approach is along the lines of the one used by Automata Tutor to grade DFAs. We also spoke to different graders and observed that they were in some sense computing the regex edit distance to assign partial credit.

Computing the regex edit distance is a PSPACE-hard problem and seems computationally intractable even for college level submissions. To deal with this intractability, we look at a simpler version of regex edit distance that can be computed for many college level submissions. We hypothesize that our version of regex edit distance is a good metric for evaluating and awarding partial credit for regexes. We ran an initial study and we observed a strong relation between the partial credit awarded and our version of regex edit distance.

CONTENT

1 INTRODUCTION

Grading and feedback (G&F) continue to be an essential part of education. However, the growth in education has made it hard to cope with G&F needs. This growth in education has encouraged the development of tools which can provide grades and feedback without human intervention. Having such tools not only saves much time spent on grading by teaching assistants but also equip the students to practice and improve their skillset independently. In this study, we have developed a tool for one of the introductory concepts of computer science called "Regular Expressions" (aka regexes).

We studied two tools, Automata Tutor (AT) [1] and Regex Equivalence [2], which attempt to evaluate regexes. Regex Equivalence only provides feedback while AT provides both feedback and grades. The above two tools have two inputs, the correct solution, and the submitted solution, of which the submitted solution needs to be evaluated. The evaluation techniques of both the tools provide feedback only for an incorrect submission. The feedback mechanism of both the tools is focused on providing counter examples by identifying strings which are accepted by the correct solution but not by the incorrectly submitted solution and vice versa. Such strings are referred to as witness strings, and a subset of them is provided as a feedback. Also, AT tries to estimate the language of the witness strings, which is then used as feedback. We make a further attempt to try and improve the feedback given by identifying the minimal changes we could make to a submitted solution for it to be equivalent to the correct one. We call this criterion 'Regex Edit Distance' (RegED) which is inspired by the 'edit distance' used by AT for DFA evaluation [1] [3].

To compute RegED, we perform a set of operations, which are quite laborious to compute even for the problem we are trying to solve. Due to this, we perform only a subset of the operations conducted in RegED. This new modified approach is called simplified RegED (aka SRegED). Additionally, we are trying to identify if SRegED could be utilized to compute the partial credit that can be given to a partially incorrect submission. Currently, AT computes partial credit by approximating the count of witness strings and a total number of strings in the language. Then, based on the ratio of the two counts, a grade is allocated. We say that AT uses an approximation approach because a language may have an infinite number of strings due to which they limit themselves to only counting strings up to a certain length. This technique fails to accurately compute the grade in certain cases. An example of an inaccurate computation is: if a correct solution is 1(0+1)* and the submitted solution is 1(0+0)*. In such a case AT would give out a much lesser grade than the student deserves. Thus, we believe that SRegED can help improve the computation of partial credit in a subset of such cases.

2 BACKGROUND

1.1 DETERMINISTIC FINITE AUTOMATA (DFA)

DFA is a model of computation. As the name suggests, it is an automaton with a finite set of states. We call it deterministic because for each state and each input there can only exist a single predetermined way for the machine to behave. The machine consists of states and transitions. States represent the progress the machine has made on an input. The rules that help in defining the behavior of each state on each input symbol are called transition functions. A DFA has a single starting state, which is referred to as a start state. The input to a DFA is a string made of symbols from a predefined set of symbols (the alphabet). Every string accepted by the DFA is said to be a member of the DFA's language. The DFA is defined by a 5-tuple notation $(Q, \Sigma, \delta, q_0, F)$ as follows:

- Set of states (Q)
- Set of symbols (Σ)
- Transition Function ($\delta: Q \times \Sigma \rightarrow Q$)
- Start state (q_0)
- Final states (F)

The set of states holds all the states of the system including the start and the final state. The set of symbols contains the symbols, which are accepted by the machine. A start state (q_0) is the state of the machine at the beginning of every input string. For each symbol a DFA transitions from its current state to any member of Q (including itself). The decision for transitions made by each state for each symbol is defined by using the transition function (δ). The transition function is defined as $\delta(q,c) \rightarrow q_1$ where q_1, $q \in Q$, $c \in \Sigma$ and it defines the behavior on the input symbol c. For example, in this case, q on input symbol c transitions to q_1. To determine if a string is a member of a DFA's language, the DFA must first process it completely. We process one symbol at a time and then move to the next symbol until the end of the string. The processing step involves transitioning from one state to another depending on the current state and the symbol. On processing the whole string if the state of the DFA is a member of final states (F) we say the string has been accepted.

1.2 NON-DETERMINISTIC FINITE AUTOMATA (NFA)

An NFA is the same as a DFA with the same rules and definition. Transition rules are the only component that changes here. As opposed to before, the definition of transition rules has changed to $\delta(q,c) \rightarrow s_1$ where $q \in Q$, $c \in \Sigma \cup \varepsilon$ and $s_1 \subseteq Q$. Which means, now on a single input symbol a state can transition to multiple states. Another difference is that it can even transition on Epsilon. This implies that the machine can change its state even without any input symbol. Due to such transition rules, while checking for

acceptance of a string, the NFA may end in multiple states, and we say that the NFA accepted the string if at least one of those states is a member of F. DFA and NFA are equally powerful i.e., any NFA can be converted to a DFA [4]. Any language for which an NFA or a DFA can be constructed is called a regular language.

1.3 REGEX

Regex is another form of representation for regular languages. Every regular language can also be presented as a regular expression (regex). Also, any language that can be represented as a regex is a regular language. A regex is represented as a string of characters which comprise of operators, symbols and grouping characters. The atomic symbols of a regex are as follows [5]. Each of these individual characters is also considered to be a regex:

- Any symbol from the set of symbols (Σ)
- Ɛ for the empty string
- ∅ for the empty language

The operators of a regex are as follows. For in-depth details of this concept, please see [5]:

- Union (+)
- Concatenation (.)
- Kleene star (*)

Where union and concatenation are binary operators and Kleene star is a unary operator. The union operator represents the union of two languages. For example, the expression (R_1+R_2) would accept any string which is either accepted by R_1 or R_2. The concatenation operator represents the concatenation of two languages. For example, the expression R_1R_2 would accept any string where any one of its prefixes must be accepted by R_1 and the remainder of the string by R_2. The Kleene star of a regex R, i.e., R* means it accepts Epsilon or strings which can be formed by the concatenation of 0 or more strings in R. For example, if R accepts {110, 10} then an example of a string accepted by R* would be 11010110110 (made by concatenation of 110, 10, 110 and 110). However, the string 110010 would be rejected since no matter how we divide it can never be represented as a concatenation of members of R. To check if a string is accepted we could convert a regex to an NFA or a DFA [6]. We say a string is accepted if the DFA reaches a final state on processing the string. This process has been explained in depth under the heading REGULAR EXPRESSION EQUIVALENCE, see Section 8.

Just like any mathematical operation, regexes too have operator precedence and to override it we use the grouping characters (and). A regex is closed under concatenation, union, and Kleene star.

Regexes provide an excellent compact representation of the language and hence are widely used in text processing engines for identifying patterns of strings.

3 WHAT IS RegED?

In the world of regexes, two regexes are equivalent if and only if their languages are identical. However, it is not necessary for two regexes which are describing the same language to look identical. For example, a regex (1+0)* describes the same language as (1*0*)* or (1*(01*)*) or (0*1*)*. Furthermore, there could be infinitely many such representations, for example (1+0)* is also equivalent to (1*0*+11*)*, (1*0*+11*+111*)* … The tool consists of only one correct solution and uses that as a reference to evaluate any submitted solution. With this constraint, we considered two ways to evaluate to what extent the answer is inaccurate:

First, we could try by making the representation of the submitted and correct solution identical. For example, if the correct solution is (1+0)* and the submitted solution is (10+1), we modify the representation of the submitted solution to make it look identical to the correct solution. To do so, we apply the following transformations:

- Delete the one before the 0. Editing it to (0+1)
- Change the 0 to 1. Transforming it to (1+1)
- Change the 1 to 0. Making it (1+0)
- Finally, add a * at the end. Thus making it (1+0)*

We have limited ourselves to using just the add, delete and change transformations. These transformations can be applied to any position and any character in the expression. Each transformation is usually referred to as an edit, and finding the minimum number of edits to make the two regexes look identical is called string edit distance. Using the string edit distance proves to be an ineffective approach because the difference in the representation of regexes is not an accurate measure of the amount of error in the submitted solution. For example, the string edit distance between (1+0)* and (0*1*0*1*) would be 7. However, adding a Kleene star to (0*1*0*1*) would be sufficient to make them equivalent. In such cases using string edit distance would not be a good metric for partial credit. This brings us to our second approach which aims at making the language of the two regexes equal rather than their representation. In the above case, if we performed only 2 of the four edits namely:

- Delete the one before the 0. Editing it to (0+1)
- Finally, add a * at the end. Thus making it (0+1)*

It would have been sufficient to make the language of the submitted and correct solution identical. Trying and finding the minimum number of edits to make the language of the submitted solution identical to that of the correct one is what we call RegED. In this study, we refer to edit distance between two regexes as

the minimal number of edits computed by RegED. To the best of our knowledge, this has never been tried before.

4 WHY RegED?

Before we proceed with our motivation to use edit distance, two of the relevant abbreviations that are going to be utilized in this section are RC and RS, where RC is the correct regex and RS is the submitted regex. We have three primary reasons for choosing RegED as an evaluation metric. Firstly, interacting with the graders gave us an insight into the manual grading process, secondly, we felt this was a good approach to take on, and thirdly RegED has never been used before.

While speaking to the graders, we realized that in some manner they are performing RegED too. While grading an assignment, if they are not confident about the correctness of RS, they usually start by looking for a witness string. If a witness string is found, the next task is usually to identify to what extent the RS is incorrect. To do this, their first attempt is to try and see if they can modify RS in a way such that it becomes equivalent to RC. This is exactly what we try to do with RegED. If finding RegED fails, they try to estimate the degree of incorrectness by checking RS with strings which students usually get wrong. A grade is awarded depending on the fraction of strings RS behaves correctly on.

In addition to the grader's input, we observed that even a small error in the RS could cause its language to be significantly different from that of RC. For example, if the RC is (1+0)(11+00)*.

(a) (10)(11+00) (b) (1+0)(11+00) (c) (1)(11+00)* (d) (1+0)*(00+1)* (e) (1+0)*(11 + 10)*

Figure 1 Error Introduced in RC to produce different RS

In Figure 1, we have introduced different kinds of errors in RC. One can observe that with a single error in (a), (b) and (c) the resultant expression describes a language which vastly differs from the correct one. Consider the case of (a) where there is no string in common which both RC and (a) accept. Introducing a combination of 2 errors as in (e) and (d) further reduces the number of strings on which RS behaves correctly on. With RegED we intend on identifying those solutions that deviate from the correct answer by a misplaced, missing or misspelled character or two and ended up producing a completely different language. We believe that these errors could be either due to a minor flaw in understanding or could just be by accident. Moreover, the grades for such submitted solutions should be computed robustly by using edit distance. Similarly, RegED also gives us an opportunity to detect the exact area of error, which, could then be used to produce feedbacks. It is due to this reason that we limit ourselves to look for a maximum combination of 5 edits in which we attempt to make RS equivalent to RC. If an RS needs more than five edits, it is a clear indicator of the incorrectness of the solution, and we do not seek to improve the grading of these solutions. The number of edits was chosen as five by looking at the assignments in Introduction

to the Theory of Computation course [5] whose solutions usually have lengths of 10-80 characters. Moreover, looking for solutions beyond the depth of 5 would become computationally unviable.

5 HARDNESS OF THE RegED

While doing our research, we did not find anyone who has previously ventured to compute RegED. Intuitively, we know it is a hard problem, but we need to confirm this by proof. We do so by picking a different problem X. The properties of X are that it is a decidable problem and it has been proved to be a hard problem. We now need to show that a solution to X can be found using RegED. Once this is done, it implies that if there exists a polynomial time algorithm to solve RegED, then X too can be solved in polynomial time. However, that would be a contradiction since X has been proven to be a hard problem. Hence, it makes it impossible for RegED to be solved in polynomial time as well. The key trick here is choosing an X and identifying how we could use our problem to solve X. This technique is called reduction. In our case, the X chosen is regex equivalence, whose hardness has been proven to be PSPACE-complete [7]. We now try to prove RegED is at least PSPACE-hard by showing we can check for regex equivalence using RegED.

Consider a function FuncRegEDit(R_1,R_2) that accepts two regexes as parameters (R_1, R_2) and returns the list of edits needed to make R_1 equivalent to R_2. Then, to check for equivalence of any two regexes we call FuncRegEDit with the two regexes as parameters. If the length of edits returned by FuncRegEDit is greater than 0, it implies there exist edits which, indicates that R_1 and R_2 are not equivalent and if the length returned was 0 it would mean that they are equivalent. This confirms that finding the edit distance is at least as hard as checking regex equivalence.

6 TEST SET

This section will provide insight on the test set used to evaluate our algorithm. On studying assignments handed out by different schools, we realized that the majority of the questions were identical to those mentioned in the book Introduction to the Theory of Computation [5]. Due to this we chose the questions for our test set from this book. Our test set is comprised of 10 questions asking the student to provide the regex. The length of the solution and its complexity was considered while choosing each question. The complex nature of a question was determined by the number of constraints present in it. For example, a question like 'describe a language which does not have substring 11 and 00' is more complex than a question like 'describe a language which does not have the substring 11' because the previous question asks for a language without two substrings, i.e., two constraints. We broadly classified each question into two categories namely, mathematical and non-mathematical. The mathematical category, consists of questions which deal with counting, for example, questions like 'describe a regex which contains an even number of 1s' or 'describe a regex which contains at most three 1s'. The non-mathematical category comprises of questions such as 'describe a regex which has the substring 001' or 'all strings that do not begin with 11 or 01'. In our test set, we had six mathematical and four non-mathematical questions.

We tried to include harder problems with longer answers as part of the test set to be able to accurately measure the robustness of our algorithm. Mathematical problems were usually of the above-mentioned nature because of which we included more of those. These questions were posted online as a survey where graduate students answered them anonymously. Only 15 students chose to answer the survey. Additionally, not all students answered all questions hence we had only 117 questions answered. Following are the questions and solutions of the test set:

1. {w| w begins with a 1 and ends with a 0} **Sol**: 1(1+0)*0
2. {w| w contains at least three 1s} **Sol**: (1+0)*1(1+0)*1(1+0)*1(1+0)*
3. {w| w contains the substring 0101 (i.e., w = x0101y for some x and y)} **Sol**: (1+0)*0101(1+0)*
4. {w| w has length at least 3 and its third symbol is a 0} **Sol**: (1+0)(1+0)0(1+0)*
5. {w| w starts with 0 and has odd length, or starts with 1 and has even length} **Sol**: (0+1(1+0))((1+0)(1+0))*
6. {w| w doesn't contain the substring 110} **Sol**: (0+(10)*)*1*
7. {w| the length of w is at most 5} **Sol**: (ε+1+0)(ε+1+0)(ε+1+0)(ε+1+0)(ε+1+0)
8. {w| w is any string except 11 and 111} **Sol**: (1+0)*0(1+0)* +(1111(1+0)*)+(1)+(ε)
9. {w| w contains at least two 0s and at most one 1} **Sol**: 0*((100) + (010) + (001))0*
10. {w| w contains an even number of 0s, or contains exactly two 1s} **Sol**: (1*01*01*)* + 0*10*10*

7 ALGORITHM FOR CALCULATING RegED

In this section, we talk about how we chose to compute RegED. In our approach, we look for different combinations of edits using depth-first search (DFS). We apply one edit at each depth until we reach the cut off depth. Every time, we apply an edit we convert the resultant expression to a regex. This way at depth one we get all regexes after applying a single edit, at depth two all regexes made of a combination of 2 edits and so on. We call this the Brute Force Search. We further analyzed the running time of brute force search and realized that it would be a very expensive operation. To avoid this, we introduced an adjustment in which we simplified the definition of RegED called SRegED. Despite this simplified definition, the run time of the search was still very high. Therefore, we introduced four more enhancements to make it run faster. We call each of these improvements as optimizations. These optimizations enabled us to look for combinations of up to 5 edits. It also helped us to bring the runtime of our code from over 10 hours to 110 seconds for the entire test set. Additionally, we only compute SRegED for syntactically correct solutions.

7.1 BRUTE FORCE SEARCH

In Section 3 we spoke of 3 ways in which we could edit, namely add, delete and change. Since Regex is a string of characters, we can perform edits on any of these characters. At the end of each edit, we must check that the edited regex is syntactically correct. Which means due to this limitation we are not able to edit opening and closing parenthesis. To simplify the add operation, let us assume that we make additions behind the character we are editing. Out of the three ways of editing, delete edit can only remove the character. On the other hand, add edit and change edit can add or change to a plus (+), concatenation (.), Kleene star (*), Epsilon (ε) or any symbol s (s ∈ ∑). This means that for any character we are trying to edit, we have (4+|∑|-1) ways to change[1] it, or we can simply delete or we could add any of the (4+|∑|) characters behind it. Thus, making the total number of possible modifications for each character in the regex as:

$$((4 + |\sum|) + (4 + |\sum| - 1) + 1) = (2 * (4 + |\sum|))$$

Equation 1 Calculation of possible option for each character

Every time a regex is edited, it is expected to produce a different language. This may not always be the case, for example, in (1*1)* if we delete a 1 to make it (1*)*, doing so doesn't change its language.

[1] The value 4 in (4+|∑|-1) has been derived considering the 3 operators (concatenation (.), plus (+), Kleene star (*)) and an Epsilon and we subtract one since we do not change the character to itself

Ignoring this and a few other cases, we provide an upper bound of the number of regexes that could be produced applying edits up to a depth D:

$$\text{Number of Expressions} = \sum_{k=1}^{D} \binom{M}{k} (2 * (4 + |\sum|))^k$$

Equation 2 Number of Possible Expressions

Where,

D = Maximum depth of our search, i.e., a maximum combination of edits.

M = Size of regex i.e. all characters excluding grouping characters '(' and ')'.

$\binom{M}{k}$ = M choose k. Count of combination of k characters from M.

$|\sum|$ = Number of symbols

In Equation 2, the summation is adding all possible regexes produced with 1 edit, then 2 and so on, all the way to depth D. k represents the number of edits being performed. If k = 2, it implies we are editing two characters in the regex. The number of ways in which we can choose these k characters is counted by $\binom{M}{k}$. Each of the k characters has (2*(4+|∑|)) possible ways to be edited. Under the assumption that each character is independent of the others in the group of k characters, we would have $(2*(4+|\sum|))^k$ ways to edit it. It can be clearly seen that the growth of this function with respect to the increasing depth is high. For example, for M = 10, D = 3 and |∑|= 3 we may have 263,640 expressions and if we increased the depth by 1 i.e. D = 4 it would be 6,269,185. This is only an upper bound since some expressions would be eliminated due to syntax error after the introduction of the edits. To make matters worse each of these expressions must be checked for equivalence with the correct regex (RC). This is necessary to find out if any of the edits produces RC or not. In the worst case when no solution is found and all combination of edits must be exhausted, which leads to a runtime of over an hour for each evaluation of a submitted solution. This means that if there was a class of 30 students and each student made 3 regex submissions, it would take ~6 days to grade them all. We realized that this does not scale very well and hence we tried to make it faster by using our first enhancement which is OPTIMIZATION 1.

7.1.1 OPTIMIZATION 1

This and all the following optimizations are completely focused on reducing the run time of our search. This optimization tries to do so by reducing our search space. On studying Equation 2, we observed that the exponential growth in the number of regexes produced was due to $(2*(4+|\sum|))^k$ term and decreasing this would help reduce the number of regexes produced. To lower the number of regexes produced we simplify the problem by performing only two types of edits i.e. change and delete. We call this the simplified regex edit distance or SRegED. On performing SRegED, the term now looks like $(4+|\sum|)^k$.

Furthermore, we intended to use this technique to grade, and in our opinion, an edit such as changing a symbol to an operator or vice versa is a severe mistake. To handle this, we added one more restriction by only changing a symbol with another symbol. It also reduced the number of regexes produced. This questions all the scenarios in which RS needs an added edit to be equivalent to RC. To tackle this, we not only compute SRegED for RS but RC as well.

Intuitively, if RS and RC were close, then there should be a common ground between the two where they could be equivalent after undergoing the edits. This could also be viewed as a bi-directional search. For RS, we search up to depth 2 and in RC we search up to depth 3. The reason that the depth for RS is limited to 2 is because student submissions may not always be quantitatively small. Doing this, in the worst case, produced ~4000 new regexes for RC and ~1250 new regexes for RS. We now check for every modified RC produced against every modified RS. This makes the total number of equivalence checks to ~5,000,000 (4000 X 1250). Which raises the question as to why is this an optimization? To begin with, we are now able to test up to 5 combinations of edits which in the previous case would have been impossible. Apart from 5 edits, it offers two major advantages which can be clearly seen when compared with the approach without optimization 1.

Firstly, without optimization, all modifications were made to RS, which meant that these could never be pre-computed since every RS is different. Now, we can pre-compute all the edits performed to RC and save it in a single file. The size of the file usually wouldn't be very large because the maximum number of regex we ever store for each RC would be ~4000 in the worst case. Hence, with the help of space, we can save computation time. The second, biggest advantage it offers is reusability. Assume with the optimization; we have produced X regexes by editing RC and similarly Y regexes by editing RS. Since each member of X is compared to each member of Y, it implies that each member of X is reused Y times and vice versa. As opposed to the unoptimized case, where if an expression was checked once for equivalence it was never used again. This observation helped us develop our OPTIMIZATION 2.

7.1.2 OPTIMIZATION 2

OPTIMIZATION 2 was inspired by two important pieces of information. The first was that every regex could be converted to a DFA. The second being that the equivalence between two DFAs can be tested in nearly linear time [8]. We decided to use this information to make our equivalence checks faster. We did so by converting each regex to a DFA and then use DFA equivalence to test them. However, converting a regex to a DFA is an exponential operation[2] and could produce a DFA with an exponential number of states. In such cases, we would not benefit from the near linear algorithm because the algorithm's complexity is near linear with respect to the number of states. And, if the number of states is exponential then the equivalence would take exponential time as well. We attempt to mitigate this by minimizing the DFAs produced from regexes.

Getting back to our optimization, assume we have set X of ~4000 regexes and a set Y of ~1250. Running an equivalence check for each member of X against each member of Y would be ~5,000,000 exponential time operations. However, what if we converted each member of X and Y to a DFA and then minimized it? This would cost us only ~10,500 (2*(4000 + 1250)) exponential operations (converting and minimizing). Once converted, we could now do our ~5 million exponential time operations in ~5 million near linear time operations (by using DFA equivalence). This way at the expense of a bit more space we saved ~11,986,000 exponential time operations. We further reduced the time by storing pre-computed DFAs for RC. For every evaluation task, we would simply load these pre-computations. Doing so significantly brought down the runtime from days to hours. However, it still did not scale well as a couple of hours for grading 117 assignments is still quite a lot of time. Moreover, making a student wait for 10 minutes for each feedback would be too inconvenient for the student. On further analysis, we identified that one of the ways to improve the running time would be by reducing the number of equivalence checks that were performed. The more obvious way to achieve this was by further reducing the number of edited regexes being produced. However, this was not a tradeoff we were willing to make. This led us to develop our third optimization, which brought our runtime down from hours to minutes.

7.1.3 OPTIMIZATION 3

DFA equivalence can be done in near linear time, but DFA string acceptance[3] can be done even faster in O(N) time where N is the length of the string being tested. Running tests, like checking acceptance of ~120 million strings on different DFAs, was extremely fast. These results inspired us to develop this optimization. Let us begin with an example, assume we have 2 DFAs D_1 and D_2 and we want to test their

[2] If we could convert a regex to a DFA in polynomial time, then regex equivalence wouldn't be a PSPACE-complete problem [7].

[3] Acceptance: checking if the string is a member of DFA's language.

equivalence. To do this, we choose a set of strings that D_1 accepts which we call P_1 and a set of strings which D_1 rejects which we call N_1. Similarly, we generate two sets of strings for D_2 called P_2 and N_2. We now check if D_2 does not accept at least one string from P_1 or accepts at least one string from N_1. It means there exists a string on which the two DFAs behave differently which proves D_1 is not equivalent to D_2 thus avoiding the need to perform an equivalence test. Suppose if D_2 accepted all strings in P_1 and rejected all strings in N_1 it implies that D_2's behavior is the same as D_1 for the strings in P_1 and N_1. We then check if D_1's behavior is the same as D_2 i.e. if D_1 accepts all strings in P_2 and rejects all strings in N_2. If D_1 finds a contradiction, it again implies D_2 and D_1 are not equivalent. If D_1 too doesn't find any contradiction, then it could be that D_1 is equivalent to D_2 and to verify that we now run an equivalence algorithm (near linear DFA equivalence algorithm).

It must be noted that using the additional check with strings increases the running time of equivalence checking algorithm. However, this happens only in the worst case; in most cases, it allows us to prove inequivalence without having to run the equivalence algorithm. Adding this to our algorithm helped it eliminate the need to execute the equivalence algorithm by ~99.11%. To obtain this result, we tried a different combination of two parameter values on our test set. With parameter 1 being the maximum number of string in each set and parameter 2 being the maximum length of each string in the set. We had best results with a maximum of 6 strings in each set with the maximum length of each string being 6. This configuration was chosen based on the ratio of running time of the algorithm and the amount of disk space consumed if we pre-computed these string sets for each produced RC.

The benefits of computing such strings far outweigh the space and time cost involved in the computation. To further improve the running time, we pre-computed these strings for each edited RC and saved it on the disk. This lowered our running time for grading from a couple of hours to ~609 seconds. Though the grading time was reduced, the time for pre-computations was ~1000 seconds. To improve this time for pre-computation we introduced our next optimization.

7.1.4 OPTIMIZATION 4

On conducting our analysis, we realized a lot of the edited regexes being produced were nothing but repetitions of previously produced regexes. For example, on editing (1+0)*, we generated (1+1)* and (1)* which represent the same language. To avoid such repetitions, one would have to look at all the previously generated regexes and determine if an equivalent regex had been previously generated. Which means every time we generate a regex we would have to run an equivalence test against all the previously generated regexes. To avoid this exponential check, we developed the concept of unitization and used that to generate regexes.

7.1.4.1 UNITIZATION

To perform unitization on a regex R, we first identify the smallest possible regexes from whose union R can be made. For example, 11+(110)* is made from the union of 11 and (110)*. Similarly, 11(1+0)* is made from a single sub-regex 11(1+0)*. Once this is computed, now for each of those regexes we divide it into smallest possible regexes from whose concatenation they can be made. In the case of the two examples above, once we have 11 and (110)* we now divide 11 to 1 and 1 and (110)* stays as is. Similarly, 11(1+0)* is made from the concatenation of 1,1 and (1+0)*. We call each of these smaller regexes a unit.

Unitization has been inspired by the concept of derivative [9]. The derivative is used for checking acceptance of a string without converting a regex to an NFA or a DFA. At a high level, to check for acceptance of a string X, we compute the derivative of a regex with respect to X. To calculate the derivative, it does not treat a regex as a string of characters but rather as a string of units. This process has been explained in depth in the regex equivalence section (Section 8).

7.1.4.2 USING UNITIZATION

To see how we use unitization, let us assume we have a regex R of the form $R_1R_2R_3$. We first unitize R and then get the units R_1, R_2, and R_3. Now unlike the previous case, we do not edit the whole expression but just the individual unit instead. Now, as we perform edits on each unit we maintain three different lists L_1, L_2, and L_3. L_1 contains all the regexes produced by editing R_1. Similarly, we have L_2 and L_3, which are made from regexes produced by R_2 and R_3 respectively. Once we have these lists, the concatenation of a single member from L_1, L_2 and L_3 produces an edited form of R. The total number of possible regexes generated is as follows:

$$\text{Total number of equations} = \prod_{k=1}^{N} L_k$$

Equation 3 Total number of possible regexes

Where,

L_k: is the size of the list L_k

N: is the number of units formed from the regex R

But since we are not looking beyond 3 edits for each RC and 2 edits for each RS each regex is chosen with the following constraint:

$$\sum_{k=1}^{N} E_k <= 3$$

Equation 4 Constraint for choosing an edit for creating an edited RC

Where,

E_k: is the number edits performed in the edited regex chosen from list L_k

N: is the number of units formed from the regex R

The equation ensures that we only produce regexes that aren't made from more than 3 edits. The fact that these units are very small allows us to check if a language of an edited unit has been previously produced. With this we ended up reducing the total number of generated regexes by ~24,000 and finally bringing the grading process to ~289 seconds, out of which 70 seconds was the time taken to load everything from the disk and 219 seconds was the time taken to evaluate. We felt this was good running time for evaluating 117 questions. However, the concept of using strings to filter regexes for equivalence (OPTIMIZATION 3) gave us an opportunity to further reduce the running time which in turn led to the implementation of our final optimization.

7.1.5 OPTIMIZATION 5

In this optimization, we started by creating a list of 80 strings that are accepted by RC. This list of strings is stored on the disk. Whenever an edited RC is generated we check how many strings out of these 80 strings are accepted by the regex. Similarly, whenever, we generate an edited RS we check how many of these 80 are accepted by it. Going further we sort all the generated RSs and RCs using two keys, first the number of edits that generated it and second the number of strings (out of the 80) which are accepted by it. Now when we compare each member of the generated RC against each member of the generated RS, we first check if the number of strings (out of the 80) accepted by the generated RC is the same as that of the RS. For example, if the generated RC R_1 had accepted 20 of the 80 strings, then any generated RS that accepts more than 20 strings implies that the generated RS is not equivalent to R_1. Since, if a generated RS, R_2 accepts 21 of the 80 strings, then it means that no matter which 21 of the strings it accepts there would always exist at least one string that R_2 accepts and R_1 doesn't. So, any generated RS that accepts more than 20 strings of the 80 can never be equivalent to R_1. With the same logic, any generated RS that accepts less than 20 strings can also not be equivalent to R_1. Hence, the only regexes with which R_1 has a possibility of being equivalent to are the ones, which accepted exactly 20 strings. Thus, by just storing the count we can determine if an equivalence check is required or not. Moreover, since the lists are sorted we used binary search to look for the regexes from the list of generated RSs. This decreased the runtime to 110 seconds for the entire test set. Finally, with all the optimization our pre-computation time for 15 assignments with 10 questions each, was 620 seconds.

In our current system, we have allowed a maximum combination of 3 edits for RC and 2 edits for RS. With this setting, we managed to identify edits for 18 incorrect solutions out of 32. From the ones we

missed, 8 of the solutions scored a low grade, which, imply that their answers were disparate from the correct one. On the remaining 6, the students roughly scored around 60% – 70%.

We intend on developing a formula to compute the grade using the number of edits and length of the submitted solution. The formula for grade computation is yet undecided. The cases that fail this technique are those where without addition it would be impossible to make the submitted solution identical to the correct solution by using just five edits. For example, if the student submission is (1*0*1*0*1*0*) and the right answer is (1+0)* then with an addition of a single Kleene star we could make the two equivalent. However, our system fails to do so.

8 REGULAR EXPRESSION EQUIVALENCE

8.1 OVERVIEW:

One of the necessary steps for regex evaluation is regex equivalence. We say that two regexes are equivalent if the language represented by the two regexes is equal. That is, two given regexes R_1 and R_2 are equivalent if and only if every string accepted by R_1 is accepted by R_2 as well and every string rejected by R_1 is rejected by R_2. This problem has been proven to be a PSPACE-complete problem [7]. With the lower bound already being proved the various approaches discussed are an optimization. Some of the optimizations being proposed such as [8] provide an opportunity for the algorithm to terminate faster for certain cases or help skip steps [10] in the standard procedure such as eliminating the need to convert a regex to NFA, which is explained later in the section under Approach 4. Some of those approaches experimented with as a part of the thesis are as follows:

8.2 APPROACH 1

The naïve way of equivalence checking is by converting both the regexes to NFAs (using the Thompson construction [6]) and then further converting the NFAs into DFAs (using the Rabin–Scott power set construction [4]). Once we have both the DFAs, we minimize them. If the languages of the two DFAs are equal, then their minimized versions are identical and checking that can be done using breadth first search. We can improve the algorithm by avoiding the minimization step and by directly comparing the two DFAs by using Hopcroft and Karp's algorithm [4], which, runs in near linear time. With this, we improve the worst case running time of the algorithm. However, it continues to be exponential in nature.

The first step for Approach 1 is converting the regex to an NFA. The technique used for this is Thompson construction [6]. The algorithm provides rules for handling concatenation (.), union (+) and star (*) operations on a regex, which are applied to the whole expression and then recursively, applied to its sub-expressions. The constructed NFA has a start state and a single final state.

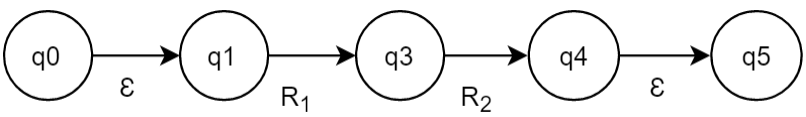

Figure 2 Concatenation Operation of Regex (.)

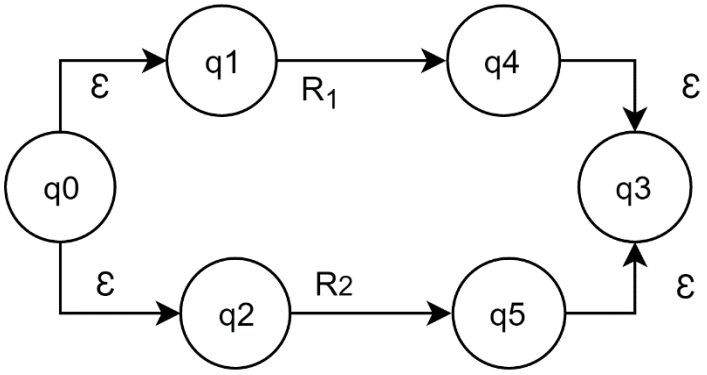

Figure 3 Union Operation on Regex (+)

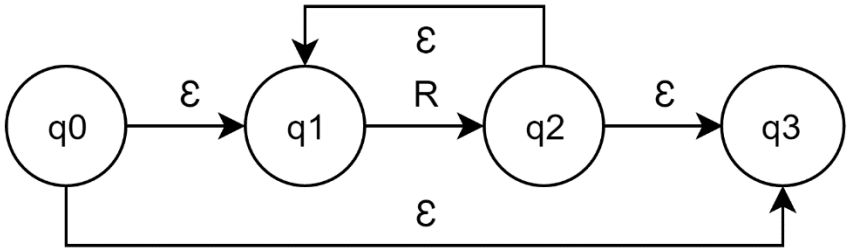

Figure 4 Star Operation on Regex (*)

The figures above are the NFAs that are used to recursively substitute for each operation for regexes R, R_1 and R_2 as follows:

1. R_1R_2 (concatenation): Figure 2 Concatenation Operation of Regex (.) represents concatenation.
2. R_1+R_2 (union). Figure 3 construction allows strings from either of the two regexes to be transition ahead.
3. R* (Kleene star) is defined by Figure 4.

Using the above transformation, any regex can be converted to an NFA. We recursively keep applying the above rules until a single symbol represents every transition between two states. For example, suppose we have a regex R defined by (1+0)*+(10). We begin with a single start and final state (Figure 5). We see that R accepts any string from the regex (1+0)* or regex 10 hence we apply the union rule as in Figure 6 .

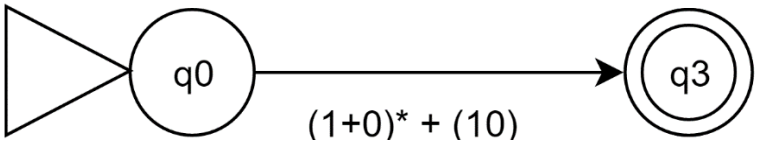

Figure 5 Regex to NFA Step 0

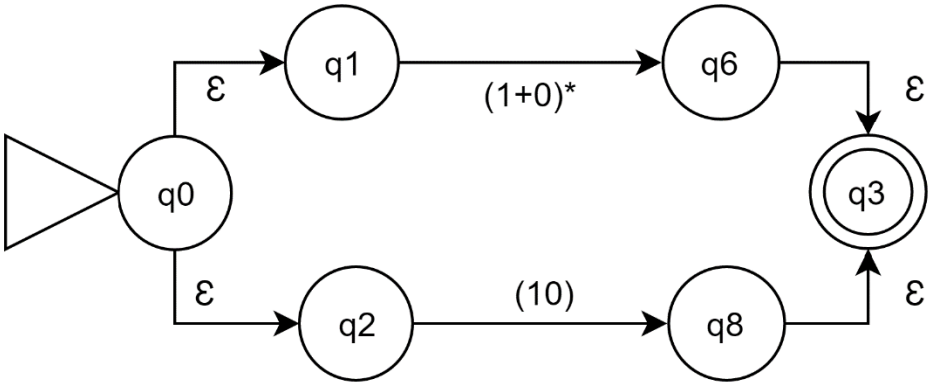

Figure 6 Regex To NFA Step1: Applying the Union rule

The presence of (1+0)* and 10 provides us with an opportunity to substitute further using the star and concatenation rule as in Figure 7.

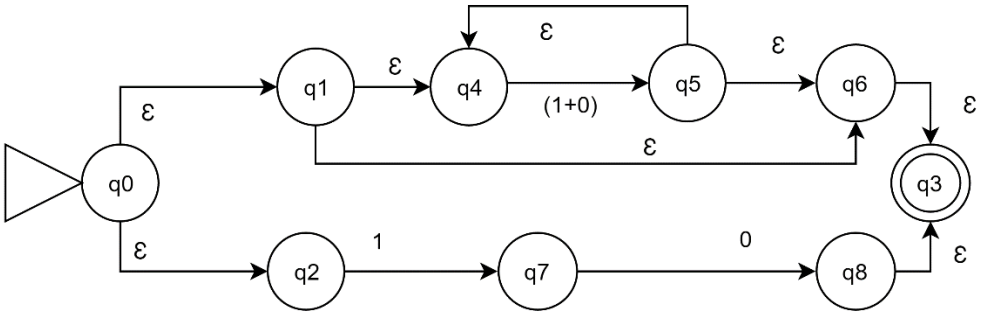

Figure 7 Regex To NFA Step2: Applying the Star(*) and Concatenation(.) rule

The transitions from q0 to q2 to q7 to q8 to q3 are all represented by a single symbol and cannot be substituted further. However, the transition between q4 to q5 is still a union of 1 and 0 and thus, we finally apply the union rule as in Figure 8.

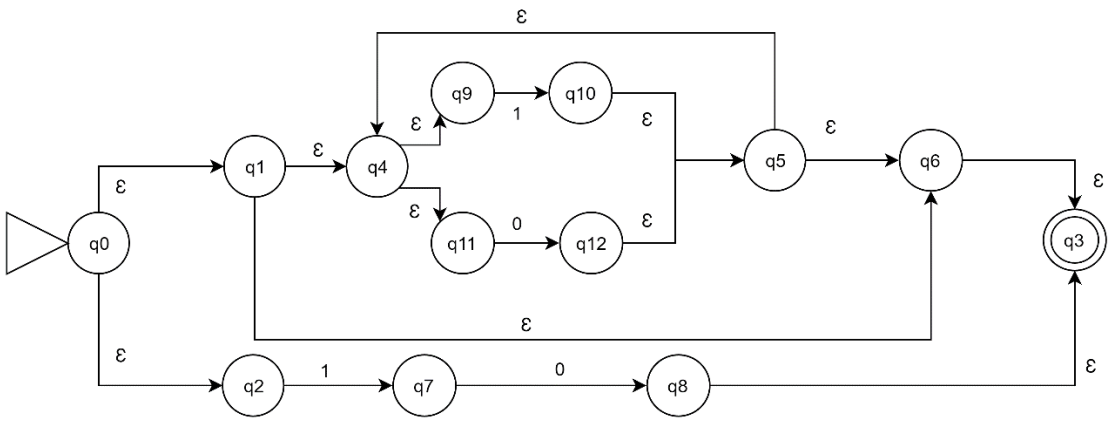

Figure 8 Regex To NFA Step 3: Applying the Union rule

Once we have an NFA N, we then use Rabin–Scott power set construction [4] to convert it to a DFA D. The major difference between a DFA and a NFA is that there are no Epsilon(ε) transitions in a DFA and that any state in the DFA can have a maximum of one outgoing transition per symbol. Since D is being constructed from N, we will be using all the transition information from N to build it. To avoid any confusion with the states in N, we name the states of D as Q0, Q1, Q2, …. The states of D will be sets of states of N. The rules for construction remain the same across all states of D.

The NFA being used for conversion from NFA to DFA is the one in Figure 9. We are going to be calling this NFA as N and the DFA being constructed as D. We begin with the start state of N, i.e., q0. This will also be a member of the start state for D. We also include states reached by the Epsilon transition from the start state in N. This is because those states would also be reachable without any input. In our case, there are no Epsilon transitions therefore D's start state (Q0) is comprised of only one state i.e. q0. It is important to maintain the set of states that Q0 is made of since it is this set that helps in determining the transitions made by Q0 (Figure 10).

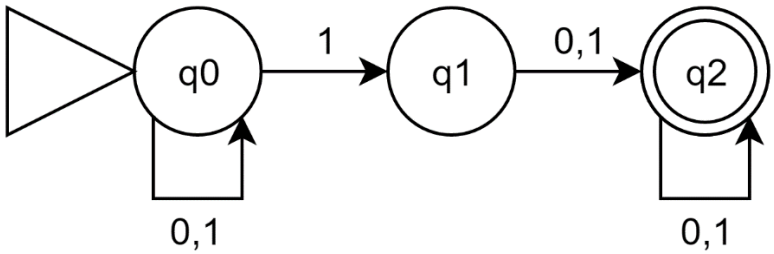

Figure 9 NFA N for conversion

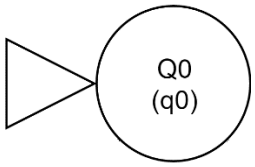

Figure 10 NFA to DFA Step 1: Start State

From now on for D, the name of the state means Q0,Q1,Q2 … and by 'set' we refer to the states of the NFA, which make the state of D. For example, the set of Q0 would be {q0}. Now, we check if any of the states from the set of Q0 is a final state in N. The presence of a final state implies that a final state is reachable, hence, if present we mark the state as a final state in D. In this case, no state is final.

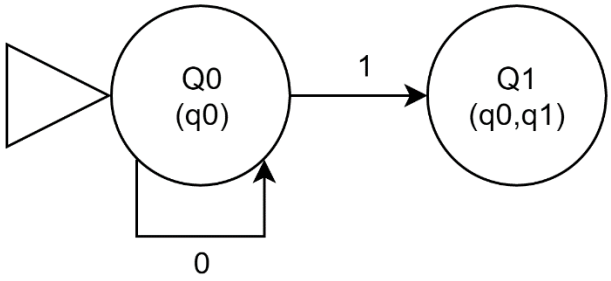

Figure 11 NFA To DFA Step 2: Q0 on input {0,1}

Now for each symbol, we look for all the states that can be reached from the set of Q0. For example, in the current case for the transition symbol 1 we check for transition from the set {q0}. For 1 the set transitions to q0 and q1. While doing this, we must keep the Epsilon closure in mind. Since there aren't any Epsilon transitions, we get our next state in D Q1 made of the set {q0, q1}. Yet again, since no state in the set is a final state we don't mark it final. Similarly, for the symbol 0 Q0 transitions to the set {q0} (Figure 11). Please note the set of states reached by Q0 on 0 is the same as itself hence we do not create a new state.

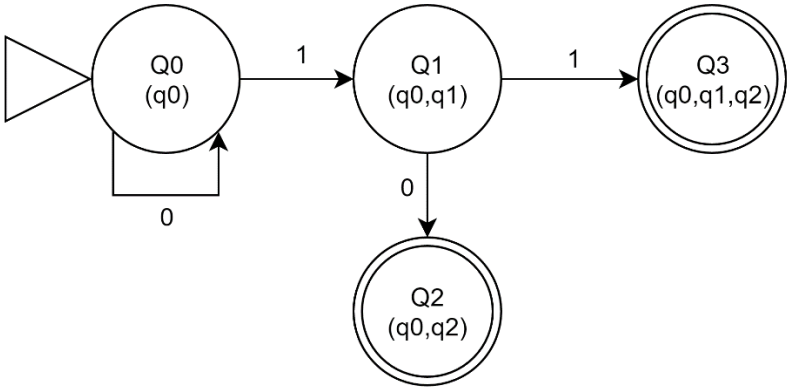

Figure 12 NFA To DFA Step 3: Q1 on input {0,1}

Continuing with the process we then process Q1 and we have 2 new states Q2 and Q3. They are marked as final as the set of both states have the final state q2 as a member (Figure 12). After this no new states are added and the process is stopped once the computation for Q3 (Figure 13) and Q2 (Figure 14) have been finished.

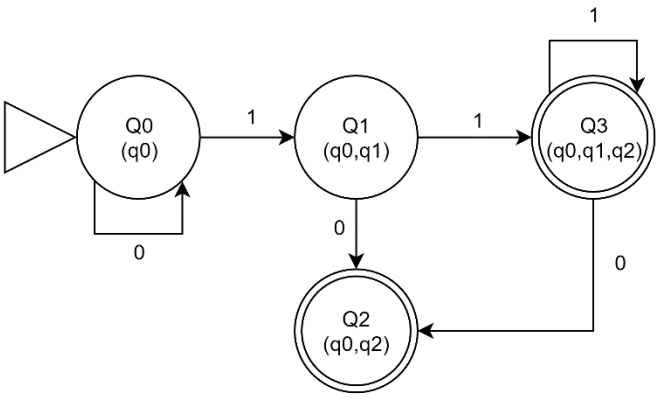

Figure 13 NFA To DFA Step 4: Q3 on input {0,1}

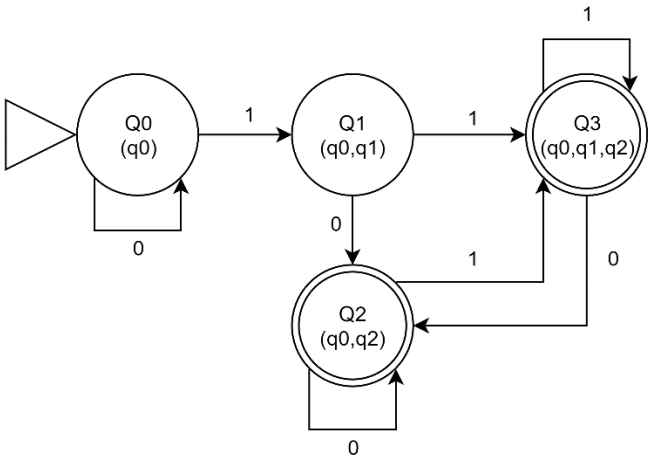

Figure 14 *NFA To DFA Step 5: Q2 on input {0,1}*

Observe that the NFA had 3 states and the produced DFA 4. It can get much worse; if the NFA has n states then the DFA could have 2^n states. For example, if one tries the convert NFA in Figure 15 would end up with a DFA with 7 states.

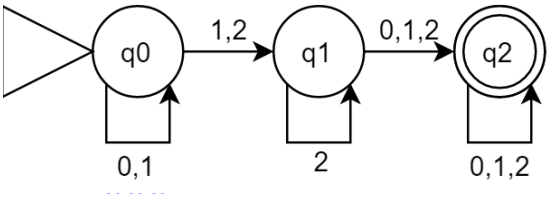

Figure 15 NFA example with exponential states in DFA

Using the above procedure any regex can be converted to a DFA. Once we have the DFA of the two regexes we need to check for equivalence. We will be focusing on Hopcroft and Karp's algorithm [8], which does not use minimization.

One of the ways for checking equivalence would be by beginning with the start states of the two DFAs, and checking if both are final or not. If only one of them is final it would imply that one DFA accepts Epsilon and the other doesn't. After ensuring both start states behave the same, we check if the states they transition to on each symbol also behave in the same way. For example, in Figure 16 we check if q2 and q6, the states reached by q1 and q5 on symbol 1, behave alike. Similarly, we also check if states reached on symbol 0 behave the same. If the behavior is identical, we then check if the transitions made by q2 and q6 behave the same. Notice, we don't compare q1 and q5 again because they have been previously compared. We continue our state comparison process until no new pairs of state are reached or we find a pair where one member of the pair is final while the other is not (i.e. a contradiction) which would mean that the two DFAs are not equivalent. This is the naïve approach. The worst case running time for this algorithm will be O(MN) where M is the number of states in DFA 1 and N the number of states in DFA 2. This would be a case where every state would end up being paired with the other.

To implement this algorithm, we need a couple of data structures.

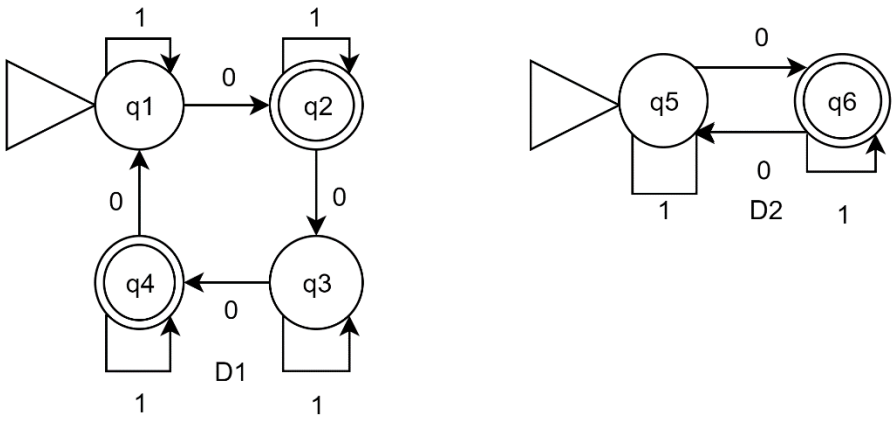

Figure 16 DFA D1 and DFA D2

To begin with, we maintain a queue which we call Q. This is used to add the pair of states for comparison. We initialize Q with the start states of the two DFA. Then for each symbol the start state transitions to is added to the queue for example we would add (q2, q6) to Q in Figure 16. Which would further add (q3, q5) on being processed. However, before we add a pair of states to the Q we need to check if the pair has been previously added. To keep track of the previously added pairs we maintain a set called VisitedPair. Using these data structures the algorithm is as follows:

Algorithm 1.1 for naïve comparison:

```
// Algorithm for checking if two DFAs are equivalent.
Function CheckEquivalenceDFA(DFA D1, DFA D2):
 Initialize Set VisitedPair -> {}
 Initialize queue Q -> []
 Q.push((D1.Initial, D2.Initial)) // adding the initial states of the two DFA
 VisitedPair.add((D1.Initial, D2.Initial)) // Adding it to the set
 While not Q.empty() // run loop while Q not empty
        currentStateDFA1, currentStateDFA2 = Q.pop() //getting the current state pair in Q
        if currentStateDFA1 ≠ currentStateDFA2 // check for if both final or non-final
                return False // if that is not the case
        for eachSymbol in Sigma: // for each symbol we check transitions
                DFA1Transition = δ(currentStateDFA1, eachSymbol) // the transition made by
the current state on the symbol
                DFA2Transition = δ(currentStateDFA2, eachSymbol)
                If ((DFA1Transition, DFA2Transition)) ∉ VisitedPair // check if present in
```

```
                    Q.push((DFA1Transition, DFA2Transition))
                    VisitedPair.add((DFA1Transition, DFA2Transition))
return True
```

In the previous algorithm, we keep adding unexplored pairs of states to Q and check if all pairs from those states also behave the same. What if both members of the pair of states have been visited before as members of different pairs? For example, in the naïve algorithm for some DFA D1 and D2 we find a new pair (q8, q9) to explore. Assume q8 was previously explored as part of the pair (q8, q11) and q9 as a part of (q10, q9) and we also had explored the pair (q10,q11). Now, if all paths from q8 behave the same as q11 and all paths from q9 behave the same as q10 then definitely all paths from q8 will behave the same as all paths from q9 (using transitivity). Moreover, if there is a contradiction for the pair (q8, q9) then it would be caught while exploring (q8, q11) or (q10, q9). Thus, eliminating the need to add (q8, q9) to Q for comparison. Using this concept Hopcroft and Karp developed their near linear algorithm to avoid unnecessary checks.

To keep track of all pairs previously explored, they merge the states in a pair into a group. Groups are sets of states. Initially all states belong to individual groups comprising of themselves. Whenever a state pair is added to Q (same as that of naïve algorithm) we merge the groups of the two states into a single group using an auxiliary data structure. The groups in no way affect the two DFA structurally, they are merely a tracking mechanism for the algorithm. Now, a state pair is added to Q if and only if the members of the pair belong to different groups. Finally, after Q is empty we check if any groups are comprised of both final and non-final states at the end of the algorithm. If there exists such a group, we say that the DFAs are not equivalent. The running time of this algorithm is $O(|\sum|(M+N))$ where M and N are the number of states in the two DFA being compared. Lastly, complexity of this algorithm heavily depends on the time taken to identify if two states belong to the same group. To do so effectively we use a special data structure. It is called union-find.

To implement union and find we first number all the states of both the DFAs. Let us assume the total number of states we have is N. We then create a 1-D array of size N called Arr1. Initially, each index is given its own index value for example, the 3rd index would be initialized with the number 3. This means every state forms its own group. Now each group is represented by single member of the group we call this the representative of the group. If we were to merge any two members we would set the value of one member in Arr1 equal to the other. For example, if we have 4 members 1,2,3 and 4. Now, if we merge 1 and 2 we would simply make Arr1[1] = Arr1[2]. Similarly, if we were to merge 3 and 4 we would make Arr1[3] = Arr1[4]. Now, if were to merge 4 and 2. We want to make all member of 4's group merged

with that of 2's group. Hence, in such a case we make the representative of 4's group point to the representative of 2's group. Therefore, for merging 4's and 2's group we would make Arr1[1] = Arr1[3]. But now, the question is how do we get the representative of the group? It must be noted in this structure; the representative of the group is the only member that points to itself. Hence, this allows us to look for it recursively till we find a member that points to itself. In our example, the representative of 4 would be the value of Arr1[Arr1[4]].

Let us run through an example. Assume, we have two DFAs D1 and D2 (Figure 17) with $\sum = \{0,1\}$. Initially the queue is empty and each state forms its own group.

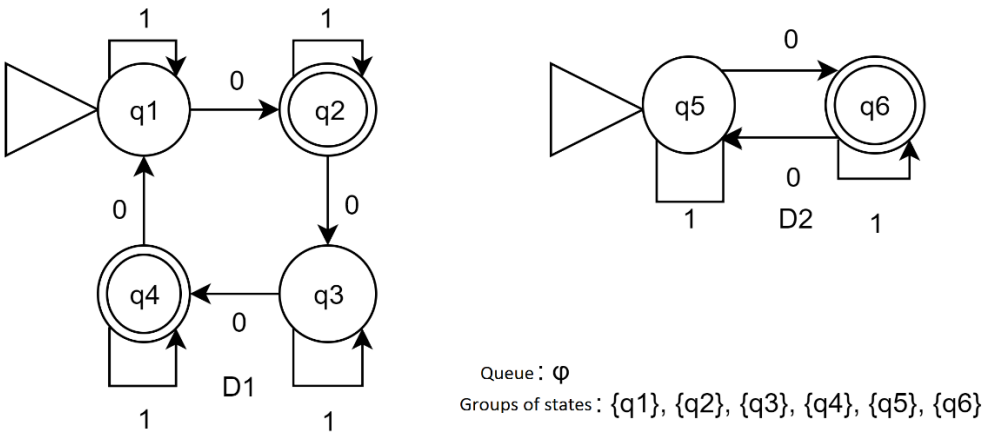

Figure 17 DFA D1 and DFA D2 Comparison Iteration 0

Just like the naïve algorithm we begin with the initial states of both D1 and D2 and add it to the queue, Q. For example, in this case the two initial states (q1,q5) are added. The first state in the pair of states will always be a member of D1 and the second of D2.

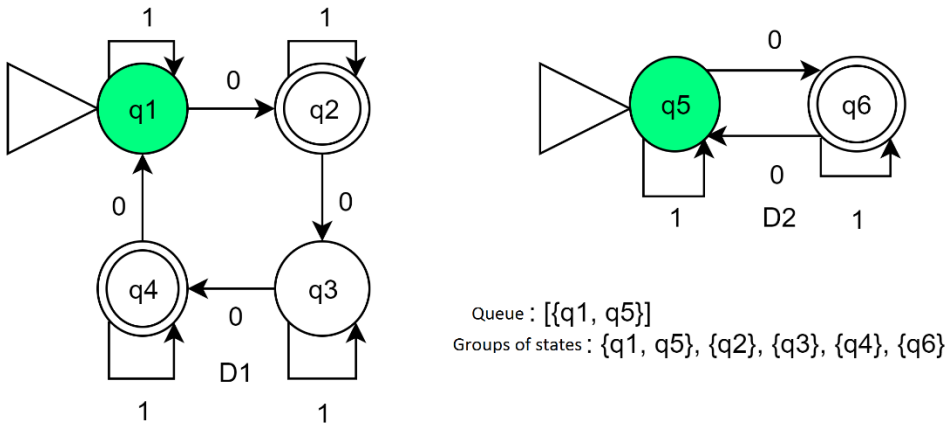

Figure 18 DFA D1 and DFA D2 Comparison Iteration 1

We begin with popping the first pair from Q. For the popped pair, we look for the transitions made by each member of the pair, for each symbol and also merge the pair into a single group. In our example, the first popped pair is (q1, q5) (Figure 18) which is merged in a single group. For the input symbol 1 q1 transitions back to q1 and q5 also back to q5 (Figure 18) thus getting the pair (q1, q5). We do not add this to Q since both the member of the pair belong to the same group. Similarly, for symbol 0 we get new pair (q2, q6) (Figure 19). Before adding we first check if both q2 and q6 belong to the same group. If not, only then they are added to the queue.

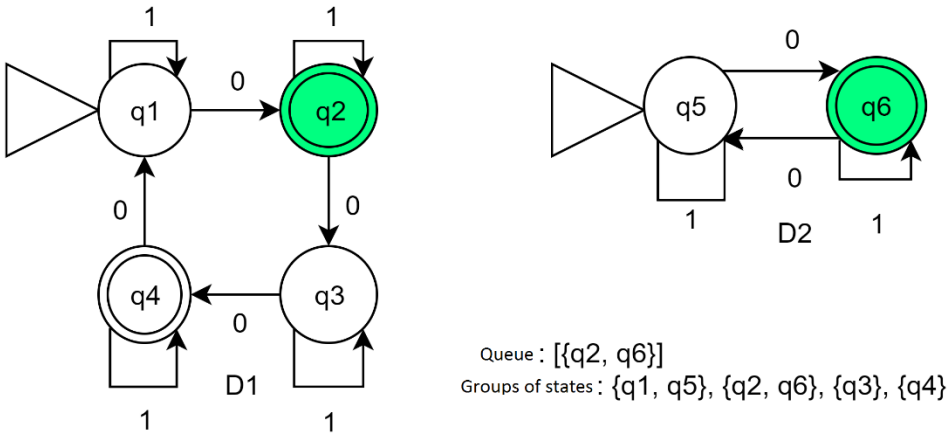

Figure 19 DFA D1 and DFA D2 Comparison Iteration 2

We now pop the next element in Q i.e. (q2, q6). We merge them into a single group and then check for transitions they make as before and add all new state pairs to the Q. For example, for the input symbol 1, q6 and q2 loop back. Since, the pair (q2,q6) belong to the same group we do not add it to Q. This process of popping, merging and adding continues until either a contradiction is found or all pairs have been tested. Finally, at the end we have, different groups of different sizes (Figure 20). Now, we perform an additional step to check if any group is comprised of both final and non-final states at the end of the algorithm. If there exists such a group, the DFAs are not equivalent.

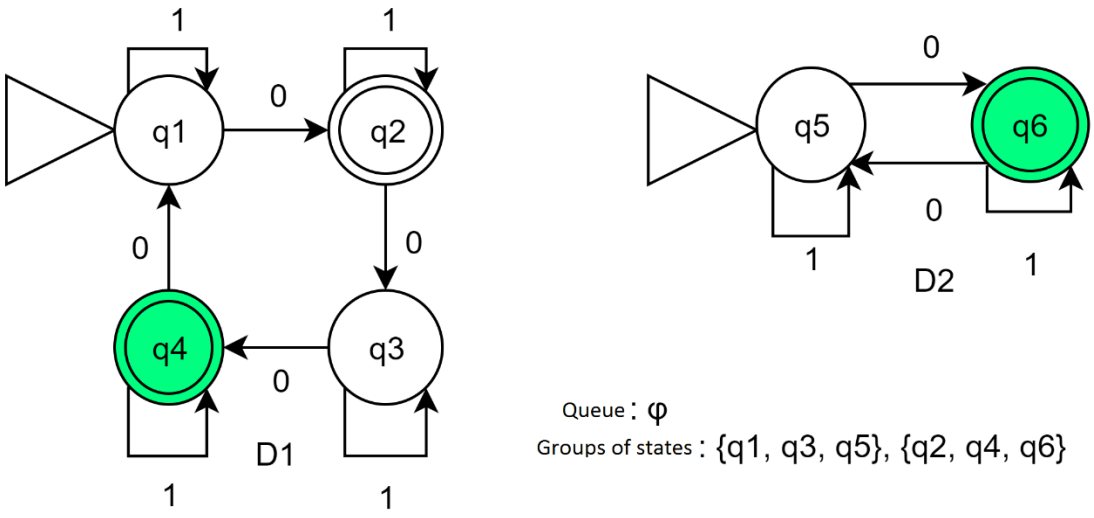

Figure 20 DFA D1 and DFA D2 Comparison Iteration 3

The near linear time algorithm from Hopcroft and Karp [8] pseudo code is as follows:

Algorithm 1.2 by Hopcroft And Karp:

```
// function for checking equivalence of DFA using Hopcroft-Karp algorithm
Function CheckEquivalenceDFA(DFA D1, DFA D2):
Initialize Set H -> {}
Initialize queue Q -> []
Q.push((D1.Initial, D2.Initial)) // adding the initial states of the two DFA
GroupSet -> For each State in D1 and D2 // initializing groups for each state in D1 and D2
Merge(GroupSet[D1.initial],G[D2.initial]) // merging group of D1.initial and D2.initial
While not Q.empty() // run loop while Q not empty
        currentStateDFA1, currentStateDFA2 = Q.pop() //getting the current state pair in Q
        for eachSymbol in Sigma: // for each symbol we check transitions
                DFA1Transition = ʒ(currentStateDFA1,eachSymbol) // the transition made by
the current state on the symbol
                DFA2Transition = ʒ(currentStateDFA2,eachSymbol)
                If  GroupSet [DFA2Transition] == GroupSet [DFA1Transition] // check if group of both
not the same
                        Q.push((DFA1Transition, DFA2Transition))
                        Merge(GroupSet [DFA1Transition],G[DFA2Transition])
```

```
For g in GroupSet // for each group in G
        InFinal = False
        InNonFinal = False // initializing if member in final or not
        For m in g // for each member of g
                If m in Final:
                        InFinal = True
                If m in Non-Final:
                        InNonFinal = True
        If InFinal and InNonFinal: // this would only be if there exists a member in each group
        return False
return True
```

An important question here is what does it mean to be in the same group and why are two members from the same group never added to our queue (Q)? Assume we have a single group of states q0, q1, q2 and q3. Assuming, this was formed due to the pairs (q0, q2), (q0, q3) and (q1, q2). We notice that all transitions from the pair (q0, q2) also get merged in a different group. For example, let us assume for symbol 1 (q0, q2) transitioned to (q4, q5) this means that q4 and q5 get merged into a single group as well. Similarly, all transitions from (q4, q5) are also merged together into a single group. Similarly, for the pair (q0, q3) merging happens. Suppose for symbol 1 q3 transitions to q6, hence for the pair (q0, q3) on input symbol 1, q6 would be merged with {q4, q5}. Due to this, we do not put the states belonging to the same groups as their transitions have already been merged into a single group. That explains why the final part of the algorithm checks for any contradiction within each group.

8.3 APPROACH 2

In the above approaches, the improvement (Hopcroft and Karp algorithm) introduced was after converting the NFA to a DFA. This brings us to an important observation. The part of the algorithm that happens to have the highest time complexity is always being computed to completion. This is the NFA to DFA conversion whose worst-case complexity is $\Omega(2^n)$. Hence, the obvious thing to do is to apply Hopcroft and Karp directly to NFAs [11]. To do this we merge the DFA to NFA algorithm and algorithm 1.2. We continue computing the states of the DFA but instead of adding the state to the DFA we add it to Q. Once its added to the queue we continue to proceed with the exact same set of steps we used in Hopcroft and Karp with an additional step of checking if both states are final or non-final. If at any point there is a mismatch we stop the process and return False.

Algorithm 2.0 Hopcroft and Karp for NFA:

```
// Applying HK to NFA for equivalence
Function CheckEquivalenceDFA(NFA N1, NFA N2):
Initialize queue Q -> []
// Getting the initial state of DFA
D1Initial,D2Initial = CollapseEpsilonTransitions(N1.Initial), CollapseEpsilonTransitions(N2.Initial)
Q.push((D1Initial, D2Initial)) // adding the initial states of the two NFA
GroupSet -> {} // this is empty since we don't have a DFA and don't know how many states we will have
Merge(GroupSet[D1Initial], GroupSet[D2Initial]) // merging group of D1.initial and D2.initial
While not Q.empty() // run loop while Q not empty
        currentState1, currentState2 = Q.pop() //getting the current state pair in Q
        for eachSymbol in Sigma: // for each symbol we check transitions
                D1Transition = GetAllReachedState(N1,eachSymbol,currentState1)
// the transition made by the current state on the symbol
                D2Transition = GetAllReachedState(N2,eachSymbol,currentState2)
                If GroupSetp[D1Transition] == GroupSet D2Transition] // check if group of both not the
same
                        If hasFinalState(N1,D1Transition) ≠ hasFinalState(N2,D2Transition):
                                Return False //means one belongs to final while other doesn't
                        Q.push((D1Transition, D2Transition))
                        Merge(GroupSet [D1Transition], GroupSet[D2Transition])
return True
```

Filippo Bonchi and Damien Pous [11] further improve the algorithm. Their improvement is loosely based on the idea that we could further reduce the search space of an NFA by not having to search through all the new pairs of states reached in each iteration. To explain this further, let us consider an example. On some input NFA1 reaches the set {x} and NFA2 on the same input reaches the set {1,2,3}. Similarly, on another input NFA1 reaches {y} and NFA2 reaches {4,5}. So far, the algorithm continues to behave exactly as above. The brilliance of the algorithm arises when we reach a point where on some input NFA1 reaches {x,y} and NFA2 reaches {1,2,3,4,5}. Unlike our implementation, they do not explore this as a new state, since {x} was already checked with {1,2,3} and {y} with {4,5} if there exists an inequivalence it would be caught in that branch. However, this is a shallow review of the idea. This is because we did not explore it in greater depth owing to the discovery of a better approach for regex equivalence (

APPROACH 4).

8.4 APPROACH 3

A. Ginzburg's technique [12] like all the previous techniques requires the regex to be converted to an NFA. The notion of his algorithm is based on the concept of 'derivatives' introduced by Brzozowski [9].

8.4.1 DERIVATIVE OF REGULAR EXPRESSION

When a DFA accepts a string, it reaches a final state after processing it. Similarly, for an NFA if any state reached at the end of the string is a final state, we say that the string has been accepted. However, with regex, the notion of states does not exist. To perform such transitions in the regex world, we use the concept of derivatives. Derivatives may be seen as transition rules for a regex. As defined in [9] "Given a set R of sequences and a finite sequence s, the derivative of R with respect to s is denoted by $s^{-1}R$ and is $s^{-1}R = \{t|st \in R\}$.".

To explain it further, let us assume that we have a symbol s and a regex R. Now on computing the derivative of R with respect to s ($s^{-1}R$) we end up with another regex R_S. The notation $s^{-1}R$ indicates computing the derivative with respect to s. sR_S represents all the string obtainable from R with the prefix s. But how do we conclude if R accepts s? To do this we check if $\varepsilon \in L(R_S)$. To explain it further, the set of strings that can be generated with the prefix s is s.x where $x \in L(R_S)$. Now if ε is a member of R_S then one of the strings with prefix s would be $s.\varepsilon$ which is nothing but s hence if $\varepsilon \in L(R_S)$ we say s is accepted by R. The subscript s in R_S is used to identify the prefix for which the derivative was computed. Some of the rules for computing derivatives are as follows:

1. $s^{-1}s = \varepsilon$
2. $s^{-1}s_1 = \emptyset$ where $s \neq s_1$
3. $s^{-1}\varepsilon = \emptyset$
4. $s^{-1}(R^*) = s^{-1}RR^*$ (Kleene star rule)
5. $s^{-1}(R_2+R_1) = s^{-1}R_2 + s^{-1}R_1$ (union rule)
6. $s^{-1}(R_2R_1) = s^{-1}R_2R_1 + v(R_2)s^{-1}R_1$ where $v(R_2)$ is ε if $\varepsilon \in L(R)$, $\emptyset$ otherwise (concatenation rule)

The function v(R) is defined as follows:

1. $v(\emptyset) = \emptyset$
2. $v(\varepsilon) = \emptyset$
3. $v(R^*) = \varepsilon$
4. $v(R_2R_1) = v(R_2)v(R_1)$
5. $v(R_2+R_1) = v(R_2) + v(R_1)$

These rules are quite intuitive in nature. For example, when a regex is of the form (R_1+R_2) it means it accepts any string which belongs to the language of R_1 or R_2. Hence, in the union rule, whenever a derivative is computed with respect to a symbol s the resultant expression is the union of the derivatives of each individual expression. Similarly, if a regex is of the form R_1R_2, then the strings with prefix s can either be formed by concatenation of each string with prefix s in $L(R_1)$ with every string in $L(R_2)$ and if $\varepsilon \in L(R_1)$ then all strings with prefix s in $L(R_2)$ would also be included. Another, case from the above rules is when language of the derivative is $\emptyset$. This is the case for a symbol s where s and any string beginning with s is not a part of the regex's language. For example, if a regex R represents a language that accepts all string beginning with 1. Then on computing $0^{-1}R$ we get $\emptyset$, which implies that no string with the prefix 0 is accepted by R. Furthermore, the derivative of $\emptyset$ with respect to any symbol is always $\emptyset$ ensuring that all strings with prefix 0 are always rejected. This is very like reaching a dead state in a DFA. The function v() too works on a similar principle for identifying if a regex accepts ε or not. For example, in the above rule for v(R*) it always returns ε but for a regex R of the form R_1R_2 it ensures both R_1 and R_2 accept ε. Since, if either R_1 or R_2 doesn't accept ε then it means R too cannot accept ε. Let us try an example, by computing a derivative of (0)*11 with respect to 0 and 1.

We will first compute the derivative with respect to 0:
$0^{-1}(0)*11 = (0^{-1}(0)*)11 + v(0*)0^{-1}11$ (applying concatenation rule)
Now computing derivative for the first part of the union i.e. $0^{-1}(0)*11$
$(0^{-1}(0)*)11 = (0^{-1}0)(0)*11$ (applying the Kleene star rule)
$(0^{-1}0)(0)*11 = \varepsilon(0)*11 = (0)*11$
Now computing derivative for the second part of the union i.e. $v(0*)0^{-1}11$
$v(0*) = \varepsilon$ as $v(R*) = \varepsilon$ which means we can compute $0^{-1}11$
$(0^{-1}1)1 = \emptyset 1 = \emptyset$
Finally putting it all together language of the strings with prefix 0 would be:
$0^{-1}(0)*11 = 0^{-1}(0)*11 + v(0*)0^{-1}11 = (0)*11 + \emptyset = (0)*11$

Then compute the derivative with respect to 1:
$1^{-1}(0)*11 = 1^{-1}(0)*11 + v(0*)0^{-1}11$ (applying concatenation rule)
Now computing the derivative for the first part of the union i.e. $1^{-1}(0)*11$
$(1^{-1}(0)*)11 = (1^{-1}0)(0)*11$ (applying the Kleene star rule)
$(1^{-1}0)(0)*11 = \emptyset(0)*11 = \emptyset$
Now computing derivative for the second part of the union i.e. $v(0*)1^{-1}11$
$v(0*) = \varepsilon$ as $v(R*) = \varepsilon$ which means we can compute $1^{-1}11$

$(1^{-1}1)1 = \varepsilon 1 = 1$

Finally putting it all together language of the strings with prefix 1 would be:

$1^{-1}(0)^*11 = 1^{-1}(0)^*11 + v(0^*)1^{-1}11 = \emptyset + 1 = 1$

Assume if we have a string $T = t_1t_2t_3$ then to check if it is accepted by R we would check if $\varepsilon \in L(t_3^{-1}(t_2^{-1}(t_1^{-1}R)))$ which could also be written as either $\varepsilon \in L(T^{-1}R)$ or $\varepsilon \in L(R_T)$. From this point on the regex computed after a derivative will be referred to as remainder regex. Finally, each regex can have only a finite number of unique derivatives. Which means there is only a finite number languages which a remainder regex can represent. Which then means that computing $s^{-1}R$ for any $s \in \sum^*$ will always produce a remainder regex which represents one of these finite languages. Each unique language produced by the derivatives could be thought of as an equivalence class and these equivalence classes could be used to convert a regex directly to a DFA as in Berry and Sethi [13]. One can read Brzozowski [9] to know more about the rules for computing the derivative.

8.4.2 GINZBURG ALGORITHM

At a high level, Ginzburg generalizes the idea that any regex R with symbol set $\sum$ could be written as:

$$R = v(R) + \sum_{s \in \sum} s(s^{-1}R)$$

Equation 5 Regex as sum of its derivatives

Where the function v() is the same as before i.e. it returns ε if $\varepsilon \in L(R)$ and $\emptyset$ if $\varepsilon \notin L(R)$.

The L(R) could consist of ε and/or strings with prefix p where $p \in \sum$. This is exactly what Equation 5 handles with v(R) and the summation. Since every other string except ε must begin with some symbol, we compute the derivative ($s^{-1}R$) with respect to every symbol. Now when we concatenate the symbol with its derivative ($s(s^{-1}R)$) it basically represents all the strings accepted by R with the prefix s. And each of these symbols would be accepted if their remainder regex accepts ε. We must note that the result on the RHS is also a perfectly valid regex because each derivative produces a valid regex and regexes are closed under union.

Inductively, we could further write each remainder regex as:

$$R_{s1} = v(R_{s1}) + \sum_{s \in \sum} s(s^{-1}R_s) \text{ where } s1 \in \sum$$

Equation 6 Representation of each remainder regex

Ginzburg begins with the two regexes R_1 and R_2 and computes the derivative with respect to each symbol as in Equation 5. If $v(R_1)$ is equal to $v(R_2)$ then it means that both accept or reject Epsilon. He then compares each of the remainder regexes to one another by further writing it as Equation 6 and compares their v() value. This process continues till either a mismatch is found or all comparisons have been exhausted. In the example below one will be able to see how the exhaustion happens. The algorithm consists of two parts, the first part involves a preprocessing step which pre-computes a table of strings. The second part involves comparing the precomputed table of strings from the previous step. Ginzburg's algorithm has been created for NFAs. As we move with our example, we show in what manner what we explained above has been performed by him in the NFA world.

In part one of the algorithm, we compute the table of strings. The property of the strings is, when the NFA transitions on it, the states reached by each string could not have been reached by any shorter string. However, there may be longer strings which may reach the same set. To find such strings we use a brute force approach i.e. we begin with strings of length one then two and so on (using breadth first search BFS). The process stops after finding all such strings. Ginzburg has proved there can only be a finite number of such strings. Intuitively, there can exist only finite such strings because the number of symbols and states is finite.

For example, if we had the 2 NFA (NFA1 and NFA2) for the regex R and Q as in Figure 21 with $\sum = \{1, 0\}$. Before we move ahead with our example the states marked green are the states that are reached by the string which is mentioned in the figure. An NFA is highlighted orange when the states reached by the current string have been previously visited by a shorter string.

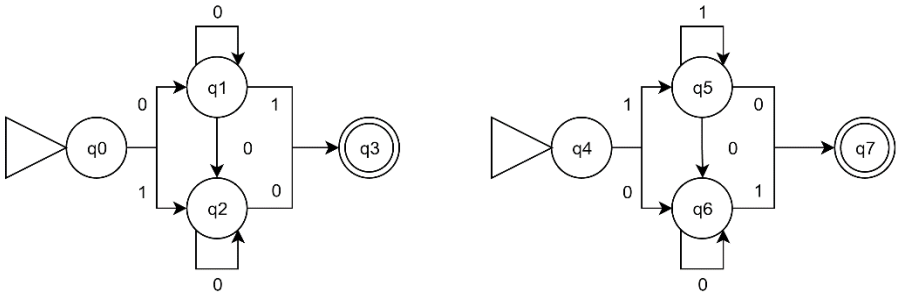

Figure 21 2 NFAs

We now start looking for a string using breadth-first search (BFS). Before beginning with any symbol, we check for all reachable states on ε. All the states reachable on ε are now used as the start states (Figure 22). This is like finding the start state of the DFA. We now begin with the shortest possible string of size 1 i.e. in this case 1 or 0. On 1, NFA1 reaches q2 and NFA2 q5 (Figure 23). Similarly, the transitions happen for 0 as in Figure 24. Now, imagine that the states that have been reached on these symbols were

made the starting states of the NFAs. For example on input of 1 NFA1 reaches q2, let us call NFA1 with the start state as q2 $NFA1_{NEW1}$. Similarly, NFA1 with the start state as the states reached on input 0 as $NFA1_{NEW0}$. Then we could say that:

$$L(NFA1) = 1.L(NFA1_{NEW1}) + 0.L(NFA1_{NEWB}) + v(NFA1)$$

Then NFA_{NEW1} and NFA_{NEW0} both represent is the same language as regex 'R_1' and 'R_0' respectively. v(NFA1) is ε if NFA1 accepts Epsilon. In some sense at every prefix the marked states in the NFA represent the remainder regex. And since there is only a finite number of remainder regexes, what Ginzburg is doing is he is identifying the shortest possible strings to reach each unique remainder regex.

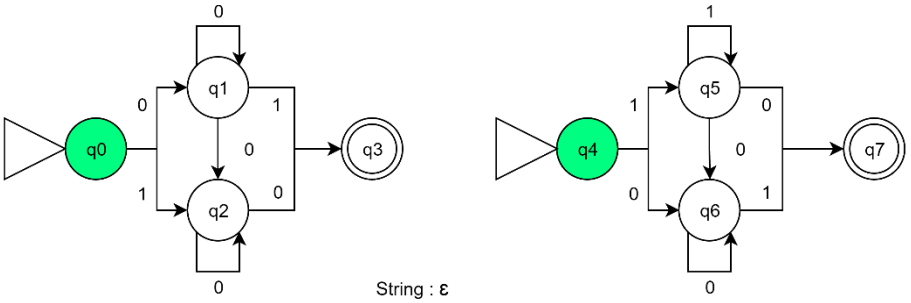

Figure 22 States reached on ε

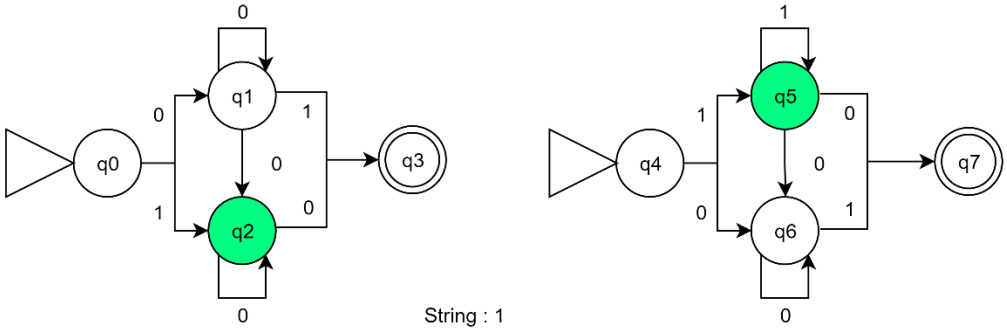

Figure 23 Input string 1

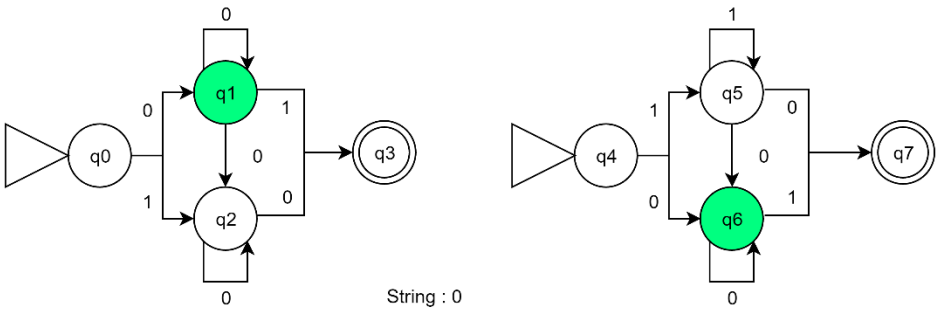

Figure 24 Input string 0

Now, we progress to the next shortest possible strings i.e. of size 2 in this case would be 11, 10, 01, 00. We observe in Figure 25 that no state is marked green for NFA1, which indicates that the strings with prefix 11 are rejected by the language. This implies that the strings with prefix 111, 110 … will also be rejected. The continuation of search on this branch (beginning with 11 so far in BFS) will always exhibit the same behavior as 11. Due to this we stop our search on the branch 11. This also means that the smallest possible string with no state marked in green is 11. NFA2 reaches the same set of states as on input 1. This means that searching further on 11 for NFA2 will also not result in the gain of any new information as 110 will act the same as 10, owing to the behavior of 11 and 1 in which will both end up on reaching the same set of states.

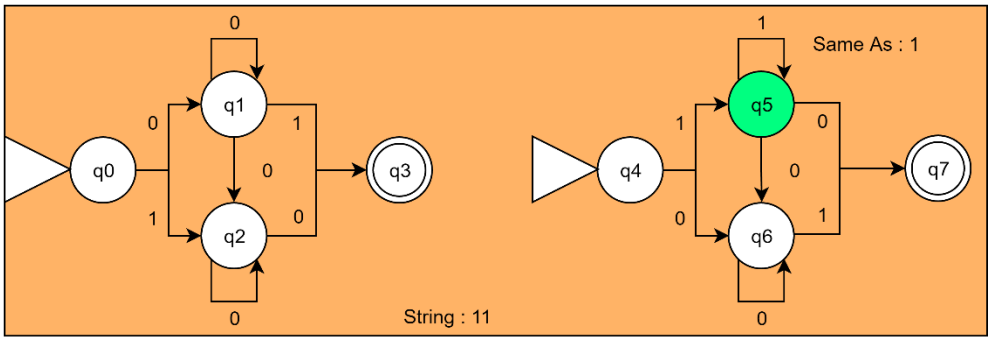

Figure 25 Input string 11

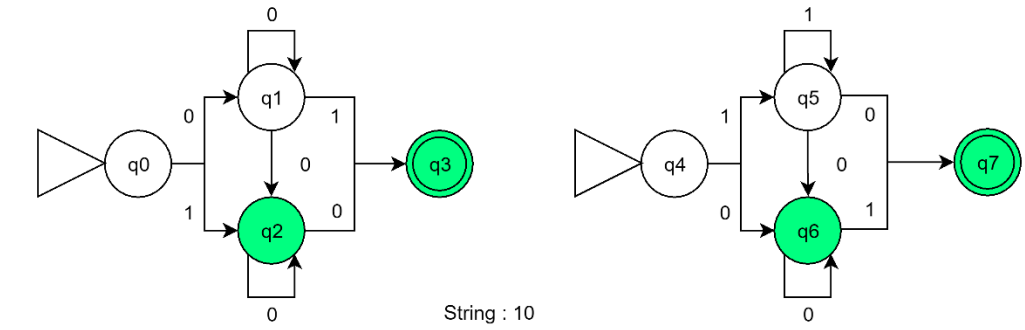

Figure 26 Input string 10

Unlike for 11, NFA1 on 10 (Figure 26) reaches a set it has not reached before and we will continue to search on the branch 10 for new sets. Additionally, we notice that NFA1 on 10 reaches the final state, which means that 10 is accepted by NFA1. If we were to interpret this in the regex world it would mean that the remainder regex on input 10 accepts Epsilon. However, the search on 10 branch is stopped on NFA2 as it reaches the same set as 0. Following the process for 01 and 00 we get Figure 27 and Figure 28 respectively.

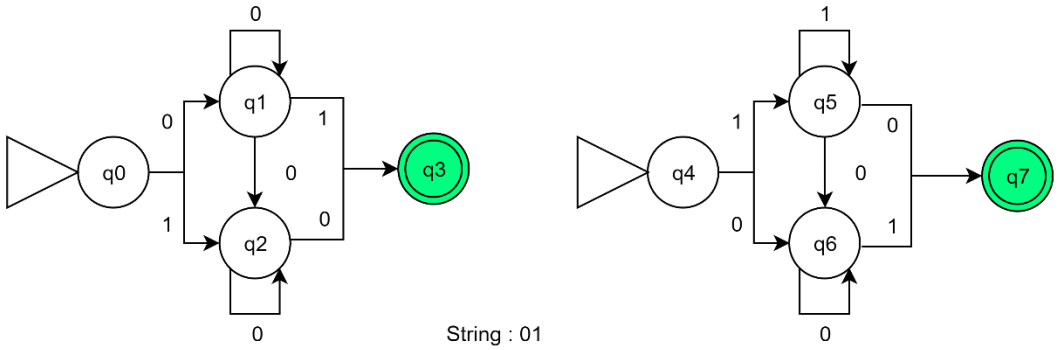

Figure 27 Input string 01

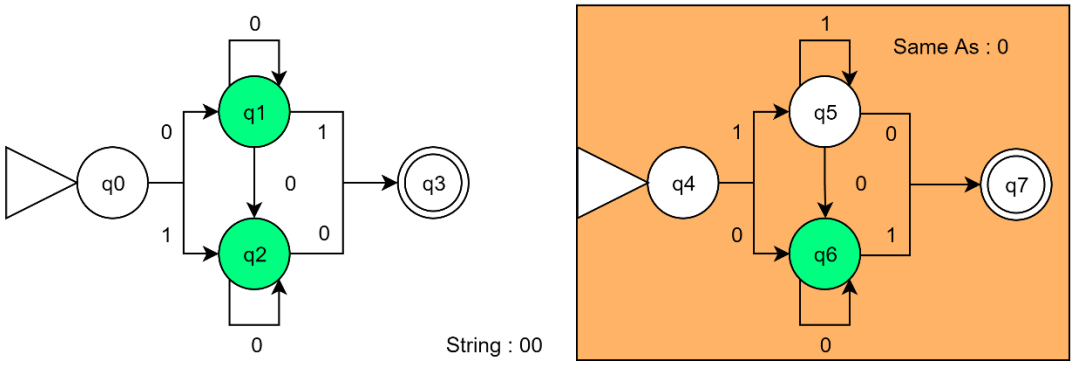

Figure 28 Input string 00

On looking at strings of size 3 (Figure 29, Figure 30, Figure 31, Figure 32, Figure 33 and Figure 34) we observe that all the set of states reached for each input have been previously reached by shorter strings. With no new set found at length 3 strings, it marks the end of our search.

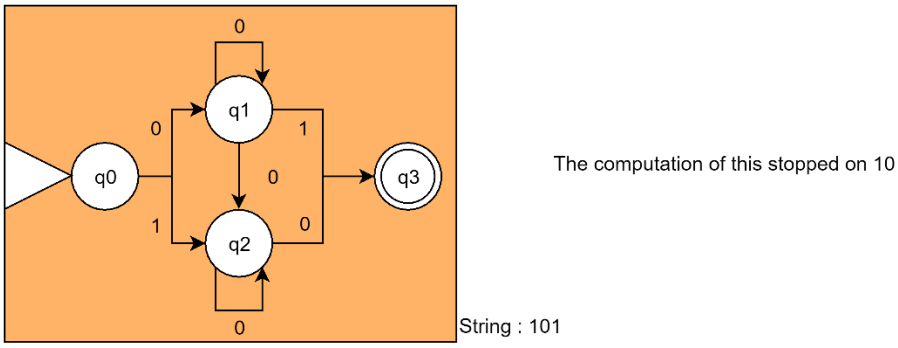

Figure 29 Input string 101

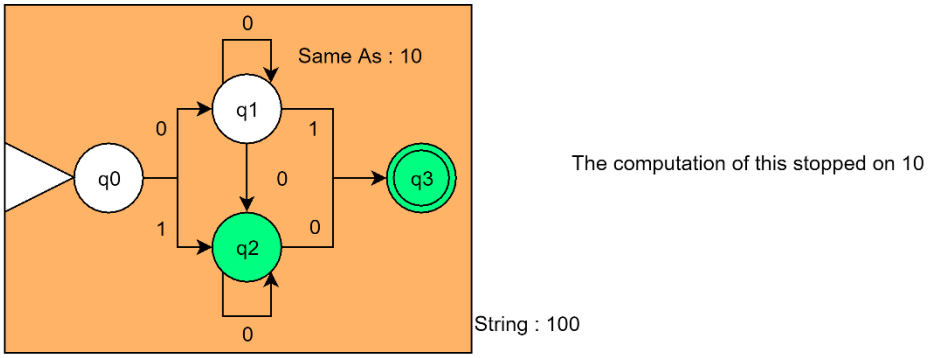

Figure 30 Input string 100

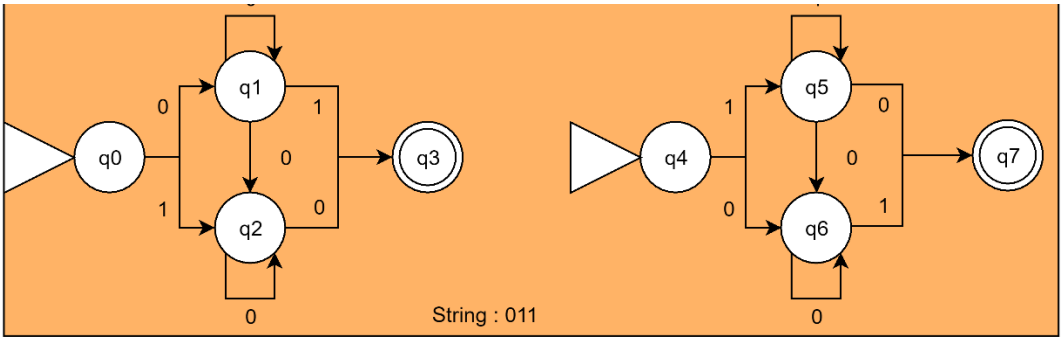

Figure 31 Input string 011

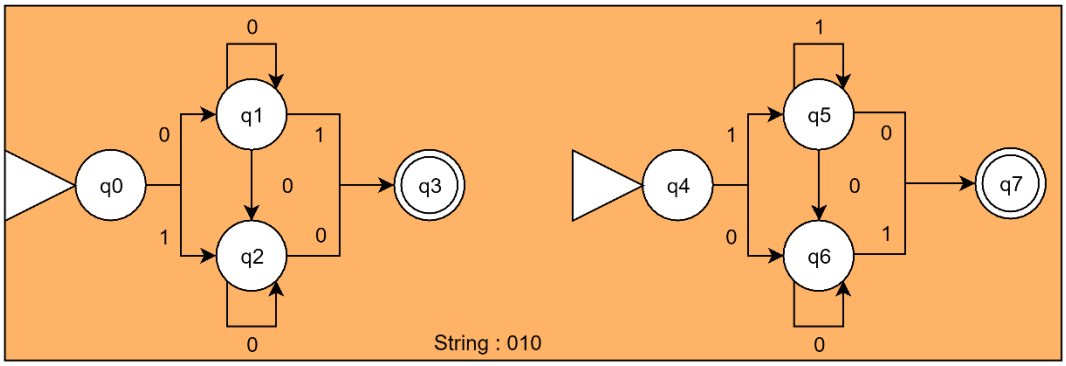

Figure 32 Input string 010

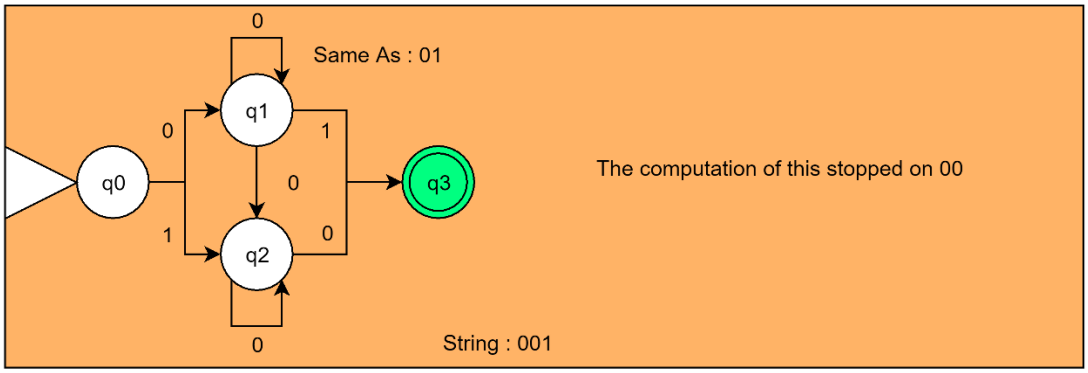

Figure 33 Input string 001

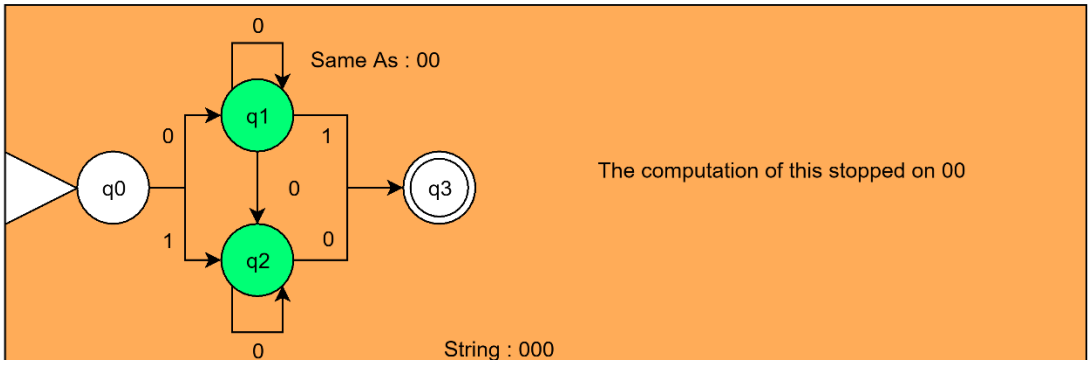

Figure 34 Input string 000

The process above has been presented using figures, however in his paper Ginzburg [12] uses tables. The columns of the tables are inputs, vertices, equal to and includes. Input is the string, vertices are the states reached on the input and equal to has the string for which the set of states have been previously reached. The 'includes' column is checked if the input string is accepted. The "includes" column helps us in determining if two NFA behave identically on each string. The table for both NFA1 and NFA2 has been shown in Table 1 and Table 2. States reached on an input have been ticked just like we highlighted the

green ones above if these are the same set of states that have been reached before we fill the equal to column.

Table 1 NFA 1

Input	q0	q1	q2	q3	Equal to	includes
ε	√					
1			√			
0		√				
11						
10			√	√		√
01				√		√
00		√	√			
101					11	
100			√	√	10	√
011					11	
010					11	
001				√	01	√
000		√	√		00	

Table 2 NFA2

Input	q4	q5	q6	q7	Equal to	includes
ε	√					
1		√				
0			√			
11		√			1	
10			√		0	
01				√		√
00			√		0	
011						

010					011	

With the completion of the tables above we are done with part one of the algorithm.

Algorithm 3.0 for Table Construction:

```
// table for precomputation step of Ginzburgs algorithm
Function ProduceGinzburgTable(NFA N1):
startState = CollapseEpsilonTransitions(N1.Initial)
tblPreviousReached <- [] // this table holds previously reached states and their strings
tblPreviousReached[startState] = ε
tblOfString <- []
tblOfString[ε] = (ε, hasFinalState(N1,startState)) //we store information of both string and if it reaches final
dq <- deque (deque of states reached)
dq.append((ε,startState)) //holds the string so far and the associated string with it
while !(dq.empty):
        stringSoFar , setOfStates  = dq.pop()
        for eachSymbol in Sigma:
                reachedState = GetAllReachedState(N1,eachSymbol, setOfStates)
                if reachedState not in tblPreviousReached:
                        tblPreviousReached[reachedState] = stringSoFar+ eachSymbol
        // now adding symbol to the string so far and adding it dq since a new set found
                        dq.append((stringSoFar+ eachSymbol, reachedState))
        // also now this string is associated with states reached
                        tblOfString[stringSoFar+ eachSymbol] =  ((stringSoFar+ eachSymbol, hasFinalState(N1,reachedState))
                Else:
                        tblOfString[stringSoFar+ eachSymbol] = tblPreviousReached[stringSoFar+ eachSymbol] // since this is already present means there is a smaller string to get there assigning the smaller string to this one
return tblOfString
```

The second part of the algorithm comprises of using the tables above to check for equivalence. This part works exactly like the naïve algorithm for DFA comparison. The important part here is to understand the interpretation of the strings we just got. Each string that is a part of the table effectively transitions to a unique set of states. Which if treated as a starting state would be the remainder regex for the string as prefix. Since, once a set of states is reached it doesn't matter which string helped it to get there, we keep track of only the shortest string that helps it get there. To compare the two NFA we start by checking if they exhibit the same behavior on prefix of length 0. What we mean by same behavior remains the same as before basically checking if one is in accept state for a prefix while the other is not. Proceeding further, we check for strings of size 1 then 2 so on and so forth. Since the size of the two tables is limited, which means so is the comparison. Having said that let's run an equivalence test for the above two NFA. Before we begin, we are going to be talking in terms of the regex R and Q instead of NFA1 and NFA2. The '/' symbol is used as a separator to denote the two expressions that need to be compared. Also, the value of v() is true if it accepts Epsilon and false otherwise.

We begin with Epsilon:

We first check if both regex accept the Epsilon

1. $(R/Q) = (1R_1/1Q_1) + (0R_0/0Q_0) + (v(R)/\ v(Q)) = (1R_1/1Q_1) + (0R_0/0Q_0) + (\text{False} / \text{False})$

The comparison of R and Q is done by (v(R)/ v(Q)) the other two terms provide the next step for comparison if the two regex behave the same after input 1 and 0.

Now we check the behavior for remainder regexes above

2. $(R_1/Q_1) = (1R_{11}/1Q_{11}) + (0R_{10}/0Q_{10}) + (v(R_1)/\ v(Q_1)) = (1R_{11}/1Q_1) + (0R_{10}/0Q_0) + (\text{False} / \text{False})$
3. $(R_0/Q_0) = (1R_{01}/1Q_{01}) + (0R_{00}/0Q_{00}) + (v(R_0)/\ v(Q_0)) = (1R_{01}/1Q_{01}) + (0R_{00}/0Q_0) + (\text{False} / \text{False})$

However, above we must notice while we replaced $0Q_{10}$ with $0Q_0$ since in our table we have the equal to column set on 10 for NFA2. It means as same set of states are reached by 10 and 1 we could continue to check for equivalence using 1 instead. This is done because if we didn't then when we further split $0Q_{10}$ we would need to compute Q_{100} and Q_{101} which is also ignored since it doesn't behave any different than Q_{10} and Q_{11}. Similarly, we also replace Q_{11} with Q_1.

Neither of the two reach the final hence we now continue to strings of length 2. Please note we continue with the replaced versions.

4. $(R_{11}/Q_1) = (1R_{111}/1Q_{11}) + (0R_{110}/0Q_{10}) + (v(R_{11})/\ v(Q_1)) = (1R_{11}/1Q_1) + (0R_{11}/0Q_0) + (\text{False} / \text{False})$

Now we don't add (R_{11}/Q_1) back to the queue again since it has already been compared once.

$(R_{10}/Q_0) = (1R_{101}/1Q_{01}) + (0R_{100}/0Q_{00}) + (v(R_1)/v(Q_1)) = (1R_{11}/1Q_{01}) + (0R_{10}/0Q_0) + (\text{True}/\text{False})$

We find a mismatch before all comparisons have finished which implies the two NFA or regexes behave differently on the string 10. Regex R accepts it while regex Q doesn't. Hence the two regexes are not equal.

Algorithm 3.1 for Ginzburg Comparison:

```
// Ginzburg algo for computing equivalence of NFA
Function GinzburgEquivalence(NFA N1 , NFA N2):
tblForN1 = ProduceGinzburgTable(N1)
tblForN2 = ProduceGinzburgTable(N2)
strForN1 = ε
strForN2 = ε
dq <- queue // queue of comparisons
setOfCompared <- {}
dq.append((tblForN1[strForN1], tblForN2[strForN2]))
setOfCompared.add((tblForN1[strForN1], tblForN2[strForN2]))
while !(dq.empty):
        currEl1, currEl2 = dq.pop()
// remember each entry in the table is a tuple of string and final state acceptance
        If currEl1[1] ≠ currEl2[1] // check if one is final while the other is not
                return False
        for eachSymbol in Sigma:
                currStr1 = currEl1[0]+eachSymbol
                currStr2 = currEl2[0]+eachSymbol
// if this combination hasn't been compared before
                if (tblForN1[currStr1], tblForN2[currStr2]) not in setOfCompared:
                        dq.append((tblForN1[currStr1], tblForN2[currStr2]))
                        setOfCompared.add(((tblForN1[currStr1], tblForN2[currStr2]))
 Return True
```

APPROACH 4

Finally, we come to our last approach. This is the only approach out of all the approaches which does not convert the regex to a NFA. It does exactly what Ginzburg does except without converting it to an NFA. The running time of this approach is faster than the time taken to convert a regex to an NFA.

Like the previous approach, we are trying to compute the remainder regex. To do so Almeida et al [10] [14] use the concept of linear regex which was developed by Antimirov and Mosses [15]. In their definition, a linear regex(R_{lin}) is a regex which can be represented by the following CFG:

$$A \rightarrow C \mid C \cdot B \mid A + A$$
$$B \rightarrow C \mid B + B \mid B \cdot B \mid B^*$$
$$C \rightarrow a \in \Sigma$$

Equation 7 CFG Linear

The CFG in Equation 7 provides a definition for a regex which is a union or concatenation of 1 or more regexes. Each of the regex can either be a symbol or a symbol followed by a regex (as all transitions are of the form C.B). Which brings us to the conclusion that a regex R is linear if it can be written as a union a_iR_i where $a_i \in \Sigma$ and each R_i is a regex. In the paper, the a_i is referred to as the head of the expression and R_i the tail. The function tail(R) represents the set of all R_i and head(R) set of all heads. More than one expression may have the same head. If every head is occurring exactly once we call it a deterministic linear regex(R_{det}).

The concept of linear regex is similar to the concept of remainder derivatives described in APPROACH 3. The prime difference being, in remainder regexes we always had the form union of a_iR_i where $i <= |\sum|$ which means they were always in deterministic linear form. This is not always possible with a linear regex due to their CFG. The second major difference is, it was possible to have an empty language (∅) in remainder regex which is not in this case. To deal with these two differences, the authors define a pre-linear regex which is defined by the CFG:

$$A' \rightarrow \emptyset \mid D$$
$$D \rightarrow A \mid D \cdot B \mid D + D$$
$$A \rightarrow C \mid C \cdot B \mid A + A$$
$$B \rightarrow C \mid B + B \mid B \cdot B \mid B^*$$
$$C \rightarrow a \in \Sigma$$

Equation 8 CFG for Pre-Linear

Where, we begin with the non-terminal A'. From Equation 8, we can clearly see that now the language of the regex can also be ∅. Furthermore, the non-terminal D provides an opportunity to bring together different regexes without losing the general idea i.e. they would still begin with a symbol followed by a regex. Like linear, we have the concept of head and tail and a pre-linear expression with unique heads is called a deterministic pre-linear expression. For example, if in the linear case we could write regex R_1's linear form as $a_2R_2 + a_3R_3 + a_4R_4$. Now, if a_2 was equal to a_3, using rules of pre-linear grammar we could also write it as $a_2(R_2+R_3) + a_4R_4$ which wasn't possible to generate using the CFG rules of the linear form.

It has been proven that any regex can be converted to an equivalent deterministic pre-linear form. To do so we follow 3 main steps:

1. Convert regex to its pre-linear form
2. Convert the pre-linear form as a union of the linear form with {∅}
3. Convert the output from step 2 to its deterministic form

We begin, with the rules of step one i.e. converting the regex to its pre-linear form. To do so we call the function that converts it to pre-linear as lin1:

1. $lin1(\emptyset) = \emptyset$
2. $lin1(\varepsilon) = \emptyset$
3. $lin1(a) = a$
4. $lin1(R_1 + R_2) = lin1(R_1) + lin1(R_2)$
5. $lin1((R_1)^*) = lin1(R_1)(R_1)^*$
6. $lin1(aR_1) = aR_1$
7. $lin1((R_1 + R_2)R_3) = lin1(R_1R_3) + lin1(R_2R_3)$
8. $lin1((R_1)^*R_2) = lin1(R_1)(R_1)^*R_2 + lin1(R_2)$

Where, R are regexes and $a \in \sum$.

The rules above are very intuitive in nature. For example, the regex in rule 8 implies all strings that belong to its language would either contain strings from $L(R_1)$ concatenated with strings from $L(R_2)$ or only strings from $L(R_2)$. Then all possible heads for this expression would be all unique first characters of strings that belong to $L((R_1)^*)$ and $L(R_2)$. Which is exactly what the rule attempts to do by making it a union of $lin1(R_1)(R_1)^*R_2$ and $lin1(R_2)$. Similarly in rule 4, where all strings that belong to $L(R_1 + R_2)$ can be in $L(R_1)$ or $L(R_2)$ hence we apply lin1 to them individually. Additionally, one might observe there is no rule to handle concatenation of two regexes of the form R_1R_2. This is because, in such a case R_1 can be either a symbol (which is handled by rule 6), ε which then would mean $R_1R_2 = R_2$, ∅ which then would mean $R_1R_2 = \emptyset$, $(R_1)^*$ (handled by rule 5) and lastly, if neither of the previous cases hold then it means it can simply be expanded which is handled by rule 7.

For example, if we have a regex R = (0*11) + (1*0*1) + ε then using lin1 we would get:

lin1(R) = lin1((0*11) + (1*0*1) + ε)

lin1((0*11) + (1*0*1) + ε) = lin1(0*11) + lin1(1*0*1) + lin1(ε) using rule 4

Now solving for lin1(0*11),

lin1(0*11) = lin1(0)0*11 + lin1(11) using rule 8

lin1(0) = 0 using rule 3

lin1(11) = 11 using rule 6

Summing it up we get:
lin1(0*11) = 00*11 + 11

Now solving for lin1(1*0*1),
lin1(1*0*1) = lin1(1)1*0*1 + lin1(0)0*1 + lin1(1) recursively applying rule 8
lin1(1) = 1 using rule 3
lin1(0) = 0 again using rule 3
lin1(1) = 1

Similarly, putting it together for lin1(1*0*1) we get
lin1(1*0*1) = 11*0*1 + 00*1 + 1

Finally solving for last part of our regex lin1(ε) with direct application of rule 2 we get its ∅. Putting together the pre-linear we computed for expression of (0*11) + (1*0*1) + ε we get:
lin1((0*11) + (1*0*1) + ε) = 00*11 + 11 + 11*0*1 + 00*1 + 1 + ∅

Once, we have the pre-linear form we convert it as a union of linear form and {∅}. The rules for converting it to linear form are as follows, we call this function lin2. lin2 accepts the output from lin1:

1. $lin2(R_1 + R_2) = lin2(R_1) + lin2(R_2)$
2. $lin2((R_1 + R_2)R_3) = lin2(R_1R_3) + lin2(R_2R_3)$
3. $lin2(R_1) = R_1$ if none of the above rule applies

Applying the above rules doesn't produce any different expression for example, the only rule we apply here is rule1 and 3 as follows:
lin2(00*11 + 11 + 11*0*1 + 00*1 + 1 + ∅) = lin2(00*11) + lin2(11) + lin2(11*0*1) + lin2(00*1) + lin2(1) + ∅ using rule 1
Except rule 3 none of the rules apply to the above produced regexes (on RHS) hence the output of lin2 for regexes continue to look the same as that of lin1 i.e. 00*11 + 11 + 11*0*1 + 00*1 + 1 + ∅.
Finally, we now pass this to our final function which basically brings together the regexes which have the same head. We call this function det. The rules for det are as follows:

1. $det(aR_1 + aR_2 + R_3) = det(a(R_1 + R_2) + R_3)$
2. $det(aR_1 + aR_2) = a(R_1 + R_2)$
3. $det(aR_1 + a) = a(R_1 + \varepsilon)$
4. $det(R_1) = R_1$

We now apply the above rules to our example.

$\det(00*11 + 11 + 11*0*1 + 00*1 + 1 + \emptyset) = \det(0(0* + 0*1) + 1(1 + 10*1 + \varepsilon) + \emptyset)$ using rule 1

$\det(0(0*11 + 0*1) + 1(1 + 10*1 + \varepsilon) + \emptyset)) = 0(0* + 0*1) + 1(1 + 10*1 + \varepsilon) + \emptyset$

Hence, using the above function we can convert any regex R to its deterministic pre-linear form by simply applying the functions in the order det(lin2(lin1(R))). The more mathematical proof can be read in [14].

This technique we can now be used for regex equivalence. For example, we have two regexes R_A and R_B and $\sum = \{0,1\}$. We compute the deterministic pre-linear form of each expression, let us call this form R_{PLINA} and R_{PLINB}. Then, R_{PLINA} would look like $1R_{A1} + 0R_{A0}$ where R_{A1} or R_{A0} could be $\emptyset$. Similarly, R_{PLINB} would look like $1R_{B1} + 0R_{B0}$. Just like the previous approach $1R_{B1}$ represents the regex of all strings with prefix 1. Now to check for equivalence, we check if both R_A and R_B behave the same on Epsilon. If they do, we now want to compare R_{A1} with R_{B1} (basically the behavior of the regexes on input 1) and R_{A0} with R_{B0}. To do so, we compute the derivative of R_{PLINA} and R_{PLINB} for symbol in our case 1 and 0. The output of the derivative for each symbol would then be compared in a similar manner i.e. check for behavior on Epsilon and then compute the deterministic pre-linear expression. Since there can only be a finite number of unique derivatives we continue this process till there are no more comparisons left or we find the behavior of the two expressions different. For our purposes, we used an existing implementation of linearization in the FAdo library [16]. Let us run a short example to check for equivalence of regex R = (11+111)* and regex Q = (1)*

Step 1: Check if both behave same on Epsilon. Yes, both accept. Now, we compute R_{PLIN} and Q_{PLIN}.

$R_{PLIN} = 1(1(11+111)* + 11(11 + 1)*)$, $Q_{PLIN} = 1(1)*$

We now compute the derivative with respect to 1:

$1^{-1}R_{PLIN} = 1(11+111)* + 11(11 + 1)*$, $1^{-1}Q_{PLIN} = (1)*$

Since $1^{-1}Q_{PLIN}$ accepts Epsilon and $1^{-1}R_{PLIN}$ doesn't, we conclude that the two regexes are not equal.

8.5 RESULT OF EACH APPROACH

To identify the best approach for our test set, we ran an experiment on 12 different sets of regexes. Each set contained 150 regexes which were randomly chosen from the regexes generated for the correct solutions. To measure the performance, we compared the time taken by each approach to check for equivalence of each regex against all the other regexes in its set. The data set was chosen this way because we were going to use the equivalence technique for grading and our objective was to find a function which performs best for testing purposes. Each algorithm was run five times for each set, and the average has been plotted with 95% confidence interval.

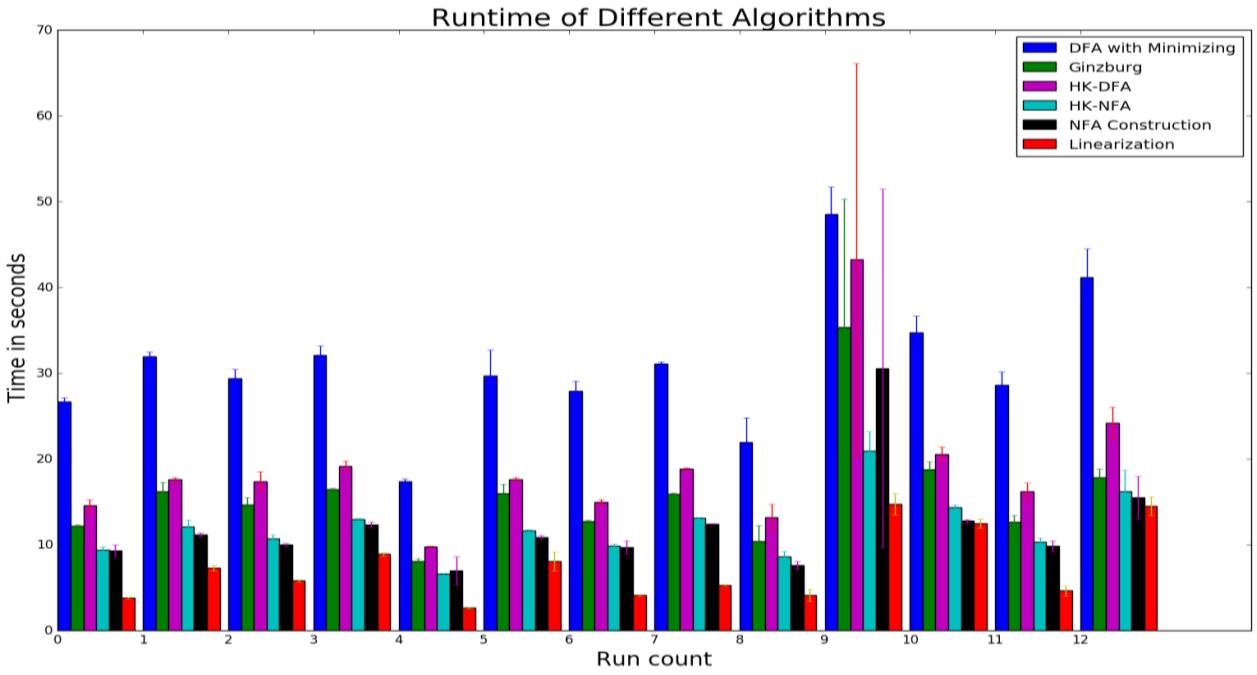

Figure 35 Running time comparison for different algorithms

In the graph above, all the algorithms on the left of the black line need the regex to be converted to an NFA and all on the right don't. The black bar is the time taken to convert a regex to an NFA. We observe that in all the tests linearization was faster than the time taken to convert a regex to an NFA (red bar vs black bar)! Thus, making it faster than all approaches whose first step is to convert a regex to an NFA. Hence, for our purposes, we chose linearization for regex equivalence.

9 FUTURE WORK

There are 3 main areas we identify for future work:

9.1 IMPROVEMENT IN EDIT RULES

We believe our edit rules could be improved in a lot of ways such as allowing an operator to be changed too. Or allowing add edit in a limited way. Our implementation of the algorithm has shown that it can very quickly compute for up to ~4000 generated regexes for RC and ~1250 generated regexes for RS. Which means any edit rules producing roughly these many regexes should be able to be run quickly.

9.2 FURTHER REDUCING THE SEARCH SPACE

In our existing algorithm, we apply edits to the whole expression, however, that is not always necessary. We could avoid applying edits to those parts of the regex which we know are incorrect. For example, in our test set one of the submitted solution was 1*(01*01*)*+0*10*10* and the correct solution was (1*01*01*)* + 0*10*10* in such a case we could completely avoid editing 0*10*10* and only try to make 1*(01*01*)* and (1*01*01*)* equivalent. We have this coded in a function called DPComparison(). We further could reduce the search space using our concept of unitization. For example, if the submitted regex was 101*0*1* and the correct solution was 1*01*01* then we simply convert it to units and then start matching single or combination of units from RC to RS. And then only run our algorithm for parts of the units which need to be matched. In case of our example above we would only run our algorithm to make 1 and 1* and 0* and 0 equivalent. We have this coded in the function VerifySolution() in the class FeedbackPureBruteForce.

9.3 TRY A MORE GOAL ORIENTED APPROACH

Our search was more focused on finding all edits up to a certain depth and checking if any of the produced regex was our solution. However, a more goal oriented approach would not only be a lot faster but also more successful since, it wouldn't need to try any edits which don't really contribute towards getting to the right answer using Salomaa's axioms [17]. Assume, if we have a regex R and Q and we want to make Q equivalent to R. Then we begin with string of size 0 i.e. Epsilon and check if both R and Q behave the same on Epsilon. If yes, we then try strings of length 1. If no, we modify Q such that it behaves the same as R on Epsilon. We continue this process till either all branches of our search have exhausted i.e. have reached a combination of 5 edits or we have found an answer. For example, if we have a regex R as (1+0)* and Q as (1+0), then a couple of ways in which we could make Q accept Epsilon would be (1+ ε)+(0+ ε), (1+0+ ε), (1+0)*, 1*+0*, 0*, 1* or ε and we check if any of these new regex make Q equivalent to P. If we observe in the first round of our search itself, we have introduced 3-4 edits in most of these expressions and it would be cut off in the next depth or two.

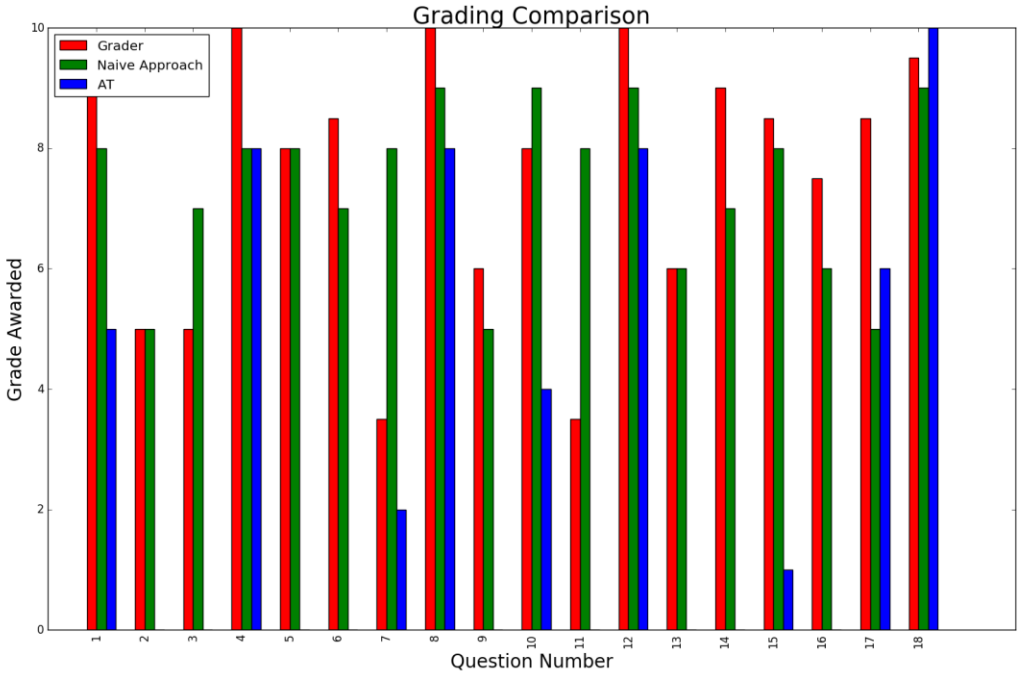

Figure 36 Grades awarded

10 CONCLUSION

Building an automated grading tool is an extremely difficult task. The fact that we found very little literature for grading theoretical computer science is a proof that this area needs more study. Some of the literature we found was Automata Tutor [3], D. Norton [18] and A. Shaikh [19]. As for our algorithm, performing bi-directional search by editing both the correct and the submitted regexes introduces some edge cases which need to be handled carefully. Despite this we feel it is still better than the unidirectional search since bi-directional search helped us significantly speed up the process of finding SRegED. Moreover, we believe by choosing better edit rules such as allowing changing of an operator with another operator we could significantly improve the results. To handle this flexibility to try different rules we have coded our algorithm in a way where it is very easy to add new rules and test them.

Finally, we believe this could be an effective technique to explore and help provide more robust grades. For example, in Figure 36 we have graded the assignment in a naïve way i.e. the number of points a student has lost is equal to the number of edits found for the student solution. Even with this approach we have performed better than Automata Tutor in most cases. In some cases, the blue line is missing because

the grade awarded by AT was 0. However, due to the small size of the test set we believe this is only a preliminary result and would need more systematic testing.

11 REFERENCE

[1] S. Gulwani, M. Vishwanathan and D. Kini, *Automata Tutor,* United States: Microsoft Research, University Of Illinois, University Of California Berkeley, University of Pennsylvania, 2013.

[2] York College of Pennsylvania, *Regular Expression Equivalence Checker,* United States: York College of Pennsylvania, -.

[3] S. Gulwani, M. Vishwanathan and D. Kini, "Automated grading of dfa constructions," *IJCAI,* 2013.

[4] D. Scott and M. Rabin, " Deterministic Finite automata and their decision problems," *IBM J. Research and Development,* vol. 3, pp. 114-125, 1959.

[5] M. Sipser, Introduction to the Theory of Computation, Third Edition, Boston: Cengage Learning, 2013.

[6] K. Thompson, "Regular expression search algorithm," *Communications of the ACM,* vol. 11, no. 6, p. 419–422, 1968.

[7] A. R. Meyer and L. J. Stockmeyer, "The Equivalence Problem for Regular Expressions with Squaring Requires Exponential Space," in *Proc. 13th Symp. on Switching and Automata Theory*, 1972.

[8] J. Hopcroft and R. Karp, "A linear algorithm for testing equivalence," Technical Report 114, Cornell University, 1971.

[9] J. Brzozowski, "Derivatives of Regular Expressions," *ACM,* vol. 11, no. 4, pp. 481-494, 1964.

[10] M. Almeida, M. Moreira and R. Reis, "Testing the Equivalence of Regular Languages," in *DCFS*, Madenburg, 2009.

[11] F. Bonchi and D. Pous, "Hopcroft and Karp's algorithm for Non-deterministic Finite," HAL, 2011.

[12] A. Ginzburg, "A Procedure for Checking Equality of Regular Expression," *Journal of the Association for Computing Machinery,* vol. 14, no. 2, p. 8, 1967.

[13] G. Berry and R. Sethi, "From Regular Expression to Deterministic Automata," *Theoretical Computer Science,* vol. 48, pp. 117-126, 1986.

[14] M. Almeida, N. Moreira and R. Reis, "Antimirov and Mosses's rewrite system revisited," *International Journal Of Foundations Of Computer Science,* vol. 20, no. 04, pp. 669-684, 2009.

[15] V. M. Antimirov and P. D. Mosses, "Rewriting extended regular expressions," *Theoretical Comput. Sci.,* vol. 143, pp. 51-72, 1995.

[16] A. Almeida, M. Almeida, J. Alves, N. Moreira and R. Reis, *FAdo: Tools for formal languages*

manipulation, http://fado.dcc.fc.up.pt/.

[17] A. Salomaa, "Two complete axiom systems for the algebra of regular events," *Journal of the ACM (JACM),* vol. 12, no. 1, pp. 158-162, 1966.

[18] D. Norton, *Algorithms for testing equivalence of finite automata, with a grading tool for JFLAP. Thesis,* Rochester: Rochester Institute of Technology, 2009.

[19] A. Shaikh, "Automatic grading and feedback in theory of computing courses," Rochester Institute of Technolgy, Rochester, 2015.