A Library for Extracting Regular Expression for Log File Analysis

by Simon Yan Lung Yip
Student ID: B424047

Loughborough University

17COP326: Internet Computing Network Security (ICNS)

Supervisor: Dominik D. Freydenberger
SUBMITTED 7^{th} SEPTEMBER 2018

Acknowledgements

I want to express my thanks to my supervisor, Dr Dominik D. Freydenberger for the tremendous support during the project. I want to thank you for all your advice and stories you told throughout my time with you in order to encourage me when I was dishearten and upset. Allow me to apologise for all the troubles i have given you. Without your support, I do not think the project would have gone well. It was truly fun and enjoyable working with you on this project. I will miss all the meeting we had and I hope to work with you again during my lifetime if possible.

"No amount of thanks can describe my gratitude to you. I am very fortunate to have you as my supervisor. Thank you very much" - Simon Yan Lung Yip

Abstract

In rule-based information extraction, a formal framework called document spanners for text querying was introduced by Fagin, Kimelfeld, Reiss and Vansummeren [2]. One of the major models in this frameworks are core spanners which use regular expressions with capture variables, knowns as ”regex formulas”, extended with relational operations and string-equality selection to extract relations of spans from the text. Spans are intervals specified by bounding indices. Extracting these relations of spans implies that we can locate the specific data from the text and store the relations between these data. Evaluating the relations of span using document spanners are expensive which require efficient evaluation algorithms to generate the extracted data.

In this project, the main objective was to create the first implementation of the first polynomial delay evaluation algorithm by Freydenberger, Kimelfeld and Peterfreund [6] that evaluate document spanners and extract the relations of spans with polynomial delay along with creation of a library for text querying using document spanners and to demonstrate that the evaluation algorithm works in an implementation.

This report introduces the topic of document spanners and terminologies require to understand the concept of the evaluation algorithm along with detail descriptions of algorithms used to support text query for the library. A thorough explanation of functions in the library were given in the report. Various tests were carried out demonstrate the use of library for querying text and a detailed analysis on the performance of the evaluation algorithm and the string-equality selection function were presented in the report.

Contents

1 Introduction

Information Extraction (IE) is the task of automatically extracting structured information including events, entities or relationships from unstructured sources. Information Extraction had its start with the DARPA Message Understanding Conference in 1987 [7] where its early work focused on military applications and current work focused on research of rule-based IE [9] with topics in logic [5], automata [10] and relational languages [1].

Recently in the field of (IE), Fagin, Kimelfeld, Reiss and Vansummeren [2] introduced document spanners, a formal framework for IE that formalize the annotation query language (AQL) used in IBM's SystemT. The process of querying a text with document spanners can be described as given a text t with search term e, document spanners P is used to extracts all interval of positions of p that contains e from t. These intervals are called spans and it is represented in the form $[i, j\rangle$ where intervals are specified by bounding indices. A spanner P is a function that maps every input string into relations over the spans of string.

One of the main models in document spanners are core spanners which uses regular expressions with capture variables, extended with relational operations (union, projection and join) and string-equality selection, to extract relations of spans from the text. Regular spanners are core spanners without string-equality selection.

Document spanners is a method to bring the expressive power of relational queries to searching all kinds of texts. However, it generate a large amount of output for evaluation. Based on this framework, Freydenberger, Kimelfeld and Peterfreund [6] constructed the first polynomial delay evaluation algorithm that allows the evaluation of document spanners with polynomial delay. Polynomial delay refer to the time complexity of generating partial solutions of a whole solution.

The algorithm was officially published on June 2018, where it was presented to evaluate unions of conjunctive queries (CQs) with fixed-parameter tractable delay given that every CQs has a bounded number of atoms. Before this project, previous implementation of this algorithm did not exist.

In this project, our main objective is to create first implementation of first polynomial delay evaluation algorithm along with the creation of a library featuring the use of document spanners including string equality selection. We also want to demonstrate that the evaluation algorithm does work for the evaluation of document spanners and to investigate feasibility of its practical application.
following objectives for this project:

- To read and understand the topic of document spanners which includes definitions and terminologies used, spanner representation and spanner algebra.
- To understand the concept and mechanism of the polynomial evaluation algorithm and other algorithms used in the library.
- Creation of a library of functions featuring the functionality of documents spanners for query searching.
- Implementation of the polynomial delay evaluation algorithm in the library and uses the algorithm to evaluate the spanners to obtain our relations of spans (Details in section 2).
- Creating the conversion of regex formula into vset-automaton in the library (the algorithm uses automaton as inputs)
- Testing of the evaluation algorithm and analyse the performance of using the library and evaluation algorithm. in order to use the algorithm to evaluate the spanners.
- Produce a report to show our understanding of the project, providing detail descriptions of the theory, thorough explanations of our implementation, results of our investigations and conclude on the overall success of the project.

The report will first introduce the necessary definitions and terminologies in order for the reader to understand the document spanners and along with spanner algebra. Section 3 provide a detailed description of polynomial delay evaluation algorithm along with descriptions and pseudo-codes of functionality test and enumeration algorithms used in the library. Next, a thorough description of the implementation and library along with explanations the process of implementation are given in Section 4. Then, Section 5 present examples of testing the library to prove the evaluation algorithm works for evaluating spanners and present a investigation of performance of the library and algorithm. The report finishes with the final conclusion of this project. We advise readers to read Freydenberger et al paper [6] where the algorithm was first presented for further insights.

2 Definitions

In this section, we will introduce the terminologies needed to understand the topic of document spanners. The essentials terms and notations introduced will be used throughout the paper.

2.1 Basic Definitions and Terminologies

Throughout this paper, let Σ be a fixed finite alphabets of terminal symbols and assume that $|\Sigma| \geq 2$. We denote Σ^* to be the set of all finite strings over Σ. A string s is a sequence of symbols $\sigma_1\sigma_2\sigma_3 \cdots \sigma_N$ from Σ where N is the length of s which is denoted by $|s|$. We denotes $|s|_\sigma$ to be the number of occurrence of some symbol σ in s and we use ϵ denote the empty word. We say a string u is a sub-string of s if there exist strings $x, y \in \Sigma^*$ such that $s = xuy$. We denote this by $u \subseteq s$.

Definition 1. Let $s = \sigma_1\sigma_2\sigma_3 \cdots \sigma_N \in \Sigma^*$. A span of s expresses an interval in s by the use of bounding indices with the form $[i, j\rangle$ where $1 \leq i \leq j \leq N + 1$. We say $[i, j\rangle$ is a span of s and we write $s_{[i,j\rangle}$ to denote the sub-string $s_{[i,j\rangle} = \sigma_i\sigma_{i+1} \cdots \sigma_{j-1}$ of s.

Example 1. Let $s = abcbca$, and $[1, 4\rangle$ to be a *span* of s. Then, the sub-string is $s_{[1,4\rangle} = abc$.

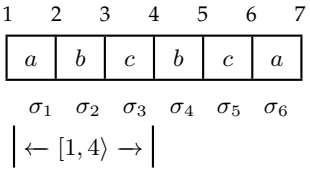


Figure 1: Showing the break down information of a string.

In Figure 1, the numeric value above the table represent interval of positions between each letter σ of the string. The σ's shows how each letter correspond to each σ in the string. We can see that $s_{[1,4\rangle} = \sigma_1\sigma_2\sigma_3 = abc$.

The empty spans are represented as $[i, i\rangle$ which has an empty interval. Two spans are equal $[i_1, j_1\rangle = [i_2, j_2\rangle$ if and only if $i_1 = i_2$ and $j_1 = j_2$. It is possible to obtain the same substring with different spans, $s_{[i_1,j_1\rangle} = s_{[i_2,j_2\rangle}$ where $i_1 \neq i_2$ and $j_1 \neq j_2$.

Example 2. Let s be the string $s = WinterWind$, hence $|s| = 10$. There are multiple empty spans in s, one example of an empty span in s is $[1, 1\rangle = \epsilon$. We can see that $s_{[1,4\rangle} = s_{[7,10\rangle} = Win$, has the same substring with different spans $[1, 4\rangle \neq [7, 10\rangle$.

Definition 2. Let $\mathbb{V}$ is a infinite set of variables, disjoint from Σ, then $V \subset \mathbb{V}$ is finite and $s \in \Sigma^*$. A (V, s)-tuple is the function μ that maps each variable V to span of s.

Definition 3. A set of (V, s)-tuple is called a (V, s)-relation or a span relation over s.

Definition 4. Given $s \in \Sigma^*$, a spanner P with the set of finite variables V, denoted as $\mathbb{V}(P)$. Then $P(s)$ maps every $s \in \Sigma^*$ to a (V, s)-relation.

2.2 Spanner Representations

In the previous section, we described that a spanner is a function that maps every string to a (V, s)-relation. In this section, we introduce two primitive spanner representations *regex formula* and *vset-automata* introduced by Fagin et al [2] and the concept of *ref-words* [6], short for reference words that are used define the semantics of these representations.

2.2.1 Ref-Words

Ref-words are strings that includes variable operations for all $x \in V$.

Definition 5. Let $V \subset \mathbb{V}$ be finite. A ref-word for V is a word over the extended alphabet $\Sigma^* \cup \Gamma_V$, where $\Gamma_V := \{x\vdash, \dashv x \mid x \in V\}$ and $\Sigma \cap \Gamma_V = \emptyset$.

The symbols $x\vdash$ and $\dashv x$ represent the opening and closing (beginning and ending positions of an interval) of a span for the variable $x \in V$, respectively. We denote these opening and closing operations as *variable operations*.

Definition 6. Given a ref-word $r \in (\Sigma \cup \Gamma)^*$. We say r is valid if for all $x \in V$ such that $|r|_{x\vdash} = |r|_{\dashv x} = 1$ and $x\vdash$ must take place before $\dashv x$.

A valid ref-word requires that all variables $x \in V$ to be opened and closed exactly once, such that all variable must be opened first before closing ($x\vdash$ must occur before $\dashv x$).

Example 3. Let $V := \{x\}$ and

$$r_1 = x\vdash Hello \dashv x, \qquad r_2 = x\vdash\dashv x,$$
$$r_3 = \dashv x World x\vdash, \qquad r_4 = x\vdash choco \dashv x\ x\vdash late \dashv x$$

We see that r_1 and r_2 are valid ref-words since all variables $x \in V$ are opened before closing.

The function $\mu^s(x)$ give the span of variable $x \in V$ for the string s. The bounded indices from the produced span will represent the occurrence of opening and closing variable operation of x, so for $\mu^s(x) = [i, j\rangle$, $x\vdash$ appear between $\sigma_{i-1}\sigma_i$ and $\dashv x$ appear between $\sigma_{j-1}\sigma_j$.

Example 4. Given $s = hello$ and $r = x\vdash hello \dashv x\ y\vdash world \dashv y$. Then $\mu^s(x) = [1, 6\rangle$ and $\mu^s(y) = [6, 11\rangle$.

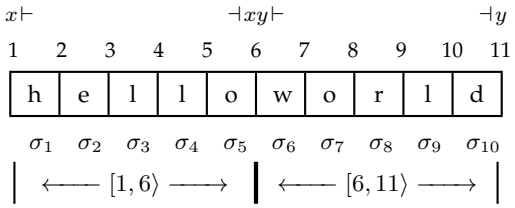


Figure 2: Showing where the variable operations of $V := \{x, y\}$ occurs in a string.

We can see in Figure 2, we have $x\vdash$ occurring between σ_0, σ_1, $\dashv x$ occurring between σ_5, σ_6, $y\vdash$ occurring between σ_5, σ_6, and $\dashv y$ occurring between σ_{10}, σ_{11}. We denote $Ref(s)$ to be all valid ref-words $r \in (\Sigma \cup \Gamma)^*$.

2.2.2 Regex Formula

Definition 7. A regex formula is a type of primitive spanner representation. Intuitively, regex formula is an extension of regular expression by including variables. The syntax for regex formula is as follows:

$$\varphi \ = \ \emptyset \mid \epsilon \mid \sigma \mid (\varphi \ \vee \ \varphi) \mid (\varphi \ \cdot \ \varphi) \mid \varphi^* \mid x\{\varphi\}$$

where $\sigma \in \Sigma$ and $x \in \mathbb{V}$.

The syntax of a regex formula is not different to a regular expression except with the added variable $x\varphi$, where $x \in \mathbb{V}$ which allows the formula to define spans of s for variables in V, $V \subset \mathbb{V}$. We denote $\mathbb{V}(\varphi)$ to be the set of all variables $x \in \mathbb{V}$ that occurs in φ. As short-hands, we also denote φ^+ as $(\varphi \cdot \varphi^*)$, and Σ to be the set of all letter σ.

For each regex formula φ, we define its ref-language $\mathcal{R}(\varphi)$ over $\Sigma \cup \Gamma_{\mathbb{V}(\varphi)}$. The ref-language $\mathcal{R}(\varphi)$ is all the possible ref-words that can be generated from the regex formula. $\mathcal{R}(\varphi)$ is the same as language $\mathcal{L}(\varphi)$ except for the form $x\{\varphi\}$ where $\mathcal{R}(x\{\varphi\}) = x\vdash \varphi \dashv x$.

Definition 8. We define the ref-word language $\mathcal{R}(\varphi)$, which gives

$$\mathcal{R}(\emptyset) := \emptyset, \qquad \mathcal{R}(\sigma) := \{\sigma\}, \qquad \mathcal{R}(\varphi_1 \vee \varphi_2) := \mathcal{R}(\varphi_1) \cup \mathcal{R}(\varphi_2)$$
$$\mathcal{R}(\varphi_1 \cdot \varphi_2) := \mathcal{R}(\varphi_1) \cdot \mathcal{R}(\varphi_2), \qquad \mathcal{R}(\sigma^*) := \mathcal{R}(\sigma)^*, \qquad \mathcal{R}(x\{\varphi\}) = x\vdash \mathcal{R}(\varphi) \dashv x$$

A ref-word $r \in \mathcal{R}(\varphi)$ is valid, if and only if for all $x \in \mathbb{V}(\varphi)$, $x\{\ \}$ appears in φ and $|r|_{x\vdash} = |r|_{\dashv x} = 1$ and $x\vdash$ occur before $\dashv x$.

Let $RV(\varphi) := \{\ r \in \mathcal{R}(\varphi) \mid r \text{ is valid for } \mathbb{V}(\varphi)\ \}$, the set of all valid ref-words obtainable by φ. We say that φ is functional if the set of all valid ref-words equals to the set of all obtainable ref-words from φ so $RV(\varphi) = \mathcal{R}(\varphi)$.

Definition 9. We define $RV(\varphi, s) = RV(\varphi) \cap Ref(s)$, where $Ref(s)$ is the set of all ref-words that can be obtained by inserting variable operations into s. We say $RV(\varphi, s)$ is the set of all valid ref-words from $\mathcal{R}(\varphi)$ and also maps can each letter to s.

Let $[\![\varphi]\!]$ to be spanner in regex formula, μ^r is a (V, r)-tuple, which show all $x \in V$ as spans in the string r. We define the $(V(\varphi), s)$-relation for every string $s \in \Sigma^*$ as:

$$[\![\varphi]\!](s) := \{\ \mu^r \mid r \in RV(\varphi, s)\ \}.$$

This equation imply that spanner in regex formula with a string s will produce all spans of $x \in V$ for all valid ref-words from the spanner of s.

2.2.3 Variable Set Automata

A variable set automaton or (vset-automaton) is another type of primitive spanner representation. This is the input used for the evaluation algorithm and we focus on converting all spanner inputs to variable set automata before processing.

Intuitively, a vset-automaton with variables from $V \subset \mathbb{V}$ is an ϵ-NFA over Σ extended with additional transitions which has its values as variable operations $x\vdash$ or $\dashv x$ for $x \in V$. Simply put, a vset-automaton is an ϵ-NFA that allow edges with opening and closing of variables as proper values.

Definition 10. Let $V \subset \mathbb{V}$ be finite set of variables and $\Gamma_V := \{\ x\vdash, \dashv x \| \text{ for all } x \in V\ \}$. A variable set automata over Σ is a tuple $A := (V, Q, q_0, q_f, \delta)$, where V is a set of finite variables, Q is the set of states, q_0 is the initial state, q_f is the terminal state (and $q_0, q_f \in Q$) and δ is transition function where $\delta : Q \times (\Sigma \cup \Gamma_V \cup \{\epsilon\}) \to 2^Q$.

Similarly with φ, the set of all variables that occurs in A is denoted as $x \in \mathbb{V}(A)$. A vset-automaton A is a directed graph where nodes are states $q, p \in Q$, and every transitions is represented $p \in \delta(q, a)$ which means an edge from p to q with the value of a where $a \in (\Sigma \cup \Gamma_V \cup \{\epsilon\})$.

For a vset-automaton A, we define $\mathcal{R}(A)$ to be its ref-language where $\mathcal{R}(A)$ is all the possible ref-words that can be generated from A (on some path from q_0 to q_f).

Definition 11. Given a vset-automaton A over the alphabet $(\Sigma \cup \Gamma_V)$. Let $RV(A)$ to be the set of all valid ref-words obtained from A. If $RV(A) = \mathcal{R}(A)$, then A is a functional vset-automaton. $[\![A]\!](s)$ is the (V, s)-relation for every $s \in \Sigma^*$

Example 5. Let A to be a vset-automaton in Figure 3.

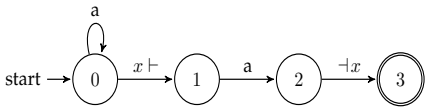


Figure 3: Showing automaton A

We see that $\mathbb{R}(A) = \{\ a^*, x\vdash, a, \dashv x\ \}$ and $RV(A) = \{\ a^i x\vdash\ a\ \dashv x \mid i \geq 0\ \}$. Then $\mathbb{R}(A) = RV(A)$, so A is functional. $[\![A]\!](s)$ contains all possible $(\{x\}, s)$-tuples for all $s \in \Sigma^*$.

2.3 Spanner Algebra

The concept of spanner algebras come from Fagin et al [2], which is to extend the primitive spanner representations with algebraic operator symbols. Spanner algebras are the results of using algebraic operations with spanners. There are five algebraic operations for spanners. These are Union $\cup$, Projection π_x, Natural Join $\bowtie$ and Selection $\zeta^{=}_{x_1,\ldots,x_k}$.

Let P, P_1, P_2 be spanners and s be a string. We present the formal definition for these spanner algebras below. Examples on how we use these spanner algebras will be given in section 4.

Definition 12. Union: The union of P_1 and P_2, $(P_1 \cup P_2)$ is valid if and only if $\mathbb{V}(P_1) := \mathbb{V}(P_2)$. Then $\mathbb{V}(P_1 \cup P_2) := \mathbb{V}(P_1)$ and $(P_1 \cup P_2)(s) := P_1(s) \cup P_2(s)$.

Definition 13. Projection: Let $Y \subseteq \mathbb{V}(P)$. If $\pi_Y(P)$, then $\mathbb{V}(\pi_Y(P)) := Y$ and $\pi_Y(P)(s) := (P)|_Y(s)$, for all $s \in \Sigma^*$, where $(P)|_Y(s)$ will only display (V, s)-relation to (Y, s)-relations of $P(s)$.

Definition 14. Natural Join: Let $V_i := \mathbb{V}(P_i)$, $i \in \{1, 2\}$. The natural join $(P_1 \bowtie P_2)$ of P_1 and P_2 is defined by $\mathbb{V}(P_1 \bowtie P_2) := \mathbb{V}(P_1) \cup \mathbb{V}(P_2)$ and for all $s \in \Sigma^*$, where $(P_1 \bowtie P_2)(s)$ is the set of all $(V_1 \cup V_2, s)$-tuple μ for which that there exists $\mu_1 \in P_1(s)$ and $\mu_2 \in P_2(s)$ with $\mu|_{V_1}(s) = \mu_1(s)$ and $\mu|_{V_2}(s) = \mu_2(s)$.

In the case where $\mathbb{V}(P_1) \cap \mathbb{V}(P_2) := \emptyset$, then the joined spanner is equivalent to the Cartesian product $(P_1 \times P_2)$.

In the case where $\mathbb{V}(P_1) = \mathbb{V}(P_2)$, then the joined spanner is equivalent to the intersection of $(P_1 \cap P_2)$.

Definition 15. Selection: Let $x_1...x_k \in \mathbb{V}(P)$. We define the string equality selection $\zeta^{=}_{x_1...x_k}P$ by $\mathbb{V}(\zeta^{=}_{x_1...x_k}P) := \mathbb{V}(P)$ and for all $s \in \Sigma^*$, $\zeta^{=}_{x_1...x_k}P(s)$ is the set of all $\mu \in P(s)$ for which $s_{\mu(x_1)} = ... = s_{\mu(x_k)}$.
In other words, the string equality selection is parametrised by k variables. The selection will match the same substring that is extracted from the spans and does not distinguish between the values of the span.

Definition 16. Let us denote the primitive spanner representation as SR and spanner algebra as SA. Then SR^{SA} is the set of all possible spanner representation that can be constructed using algebra operators from SA with the spanner from SR.

Definition 17. Let us denote the primitive spanner representation as SR. We call a spanner as a regular spanner is when SR is extended with the algebra operations of $[\ \pi, \cup, \bowtie\ ]$ of spanner algebra. A core spanner is when SR is extended with all of the algebra operations of spanner algebra, $[\pi, \cup, \bowtie, \zeta^{=}]$.

Fagin et al. [2] proved that using regex formula as spanner representations for regular spanners have the same expressive power as using vset-automata as spanner representations for regular spanners. Hence, vset-automata core spanners have the same expressive power as regex formula core spanners. Regex formulas can be transformed into vset-automata. In our library, we can convert regex formula to vset-automaton and the evaluation takes a vset-automaton as input (Details on the method to convert regex formula is in section 4).

3 The Algorithms

In this section, we will give a detailed description of the polynomial delay evaluation algorithm by Freydenberger [6]. Polynomial delay refer to the time complexity of generating partial solutions of a whole solution. Each partial solutions can be completed within $n, nlogn, n^i$ for some $i \in \mathbb{N}$. Given a functional vset-automaton A with n states and m transitions, and a string s, one can enumerate $[\![A]\!](s)$ with polynomial delay $O(n^2|s|)$ following a polynomial preprocessing of $O(n^2|s| + mn)$ [6].

3.1 Introducing Variable Configurations

Let us recall the definition of a functional vset-automaton. A vset-automaton $A = (V, Q, q_0, q_f, \delta)$ is functional if the set of all generated ref-word is equal to the set of valid ref-words, $RV(A) = \mathcal{R}(A)$. All ref-words in A, must have a path from q_0 to q_f. Recall that a valid ref-word $r \in (\Sigma^* \cup \Gamma_V)$, if for all $x \in \mathbb{V}$, where $|r|_{x\vdash} = |r|_{\dashv x} = 1$ and $x\vdash$ occur before $\dashv x$. We can interpret this condition into for every states $q \in Q$ and forall $x \in \mathbb{V}$, each state must satisfies exactly one of these conditions:

- $|r|_{x\vdash} = |r|_{\dashv x} = 0$
- $|r|_{x\vdash} = 1$, $|r|_{\dashv x} = 0$
- $|r|_{x\vdash} = |r|_{\dashv x} = 1$ and $x \vdash$ occur before $\dashv x$.

The first condition describes the case where the span of x has not started, no variable operation is read so we label this as waiting case. The second condition describes that only the opening variable operation for some x is read, label as opened case. The last condition describe that the span for x has been done, opening and closing variable operation is read for x, so label as closed case. If none of these conditions are met, this implies that either there exists $x\vdash$, $\dashv x$ twice in r or $\dashv x$ occur before $x\vdash$. Then there exists a state that does not meet the requirement for a valid ref-word which implies that the vset-automaton is not functional as these exists $r \in \mathcal{R}(A)$ but $r \notin RV(A)$, so $\mathcal{R}(A) \neq RV(A)$. Therefore, every state in the A have these implicit information about each variable $x \in V$ and Freydenberger [6] has formalized this notion.

Definition 18. We define the set of *variable states* $\mathcal{V} := \{w, o, c\}$, which stands for waiting, open and close respectively, for variables $x \in V$.

Definition 19. Let A be a functional vset-automaton. For each state q in A, the variable states for all $x \in V$, is recorded as *variable configuration* of state q, where $\vec{c}_q : V \to \mathcal{V}$.

Let q_0, q_n, q_m be states in a vset-automaton and q_0 is the starting state. If $\vec{c}_{q_n}(x) := w$, this implies that any incoming edges from q_m to q_n does not contain any variable operations for x and the ref-word r generated from state q_0 to q_m does not contain the opening or closing operation of x, so $r \notin \Gamma_V$. If $\vec{c}_{q_n}(x) := o$ and $\vec{c}_{q_m}(x) := w$, this implies that state q_n has edges from q_m to q_n containing opening operations of variable x, then the ref-word r from q_0 to q_n only contain one opening operation of x, so $x\dashv$ exists in r. If $\vec{c}_{q_n}(x) := c$ and $\vec{c}_{q_m}(x) := o$, this implies that state q_n has edges from q_m to q_n containing closing operations of variable x, then the ref-word r from q_0 to q_n contain the opening and closing operation of x, so $r \in (\Sigma \cup \Gamma_V)$.

Example 6. Let A to be a vset-automaton with $V := \{x, y\}$, start state $q_0 = 0$, terminal state $q_f = 4$ and with edges displayed Figure 4.

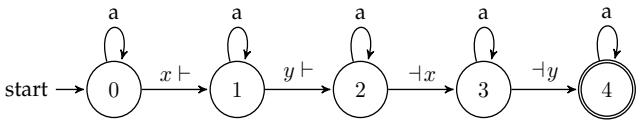


Figure 4: Showing automaton A

From Figure 4, we obtain the following variable configuration for $\{x, y\}$ for all states in A:

$$\vec{c}_0(x) = w, \quad \vec{c}_0(y) = w, \quad \vec{c}_1(x) = o, \quad \vec{c}_1(y) = w, \quad \vec{c}_2(x) = o, \quad \vec{c}_2(y) = o,$$
$$\vec{c}_3(x) = c, \quad \vec{c}_3(y) = o, \quad \vec{c}_4(x) = c, \quad \vec{c}_4(y) = c,$$

Enumerating though the automaton in Figure 4, we can see that at state 0 does not have any open nor close operations for any variables from q_0 to 0. Therefore, we have $\vec{c}_0(x) = w,\ \vec{c}_0(y) = w$. From state 0 to state 1, we see that there exists a edge from state 0 to 1 with opening operation of variable x. Therefore, $\vec{c}_1(x) = o$. From state 1 to state 2, we see the opening operation of variable x, y. Therefore, $\vec{c}_2(x) = o,\ \vec{c}_2(y) = o$. For each states,

we check whether we have a valid ref-word r from q_0 to q_n. To do this we compare the variable configuration of state q_m and q_n, where q_m has an edge to q_n which contain a variable operation. If the variable state from q_m to q_n have the pattern w to o or o to c, then q_n is contains a valid ref-word because these pattern correspond to the conditions for where $|r|_{x\vdash} = 1$ and $|r|_{\dashv x} = 0$ or $|r|_{x\vdash} = |r|_{\dashv x} = 1$ and $x \vdash$ occur before $\dashv x$.

3.2 The Main Algorithm

In this section, we will give a description of process of the polynomial delay algorithm and a detail explanation of the pseudocode for each process and possible alternative ways of achieving process.

Let $s[i]$ denote a character σ_i in s and s_i denote the position w_i where $s = w_0\sigma_1 w_1\sigma_2 w_2 ... w_{N-1}\sigma_N w_N$ where $N = |s|$. The idea of the polynomial delay evaluation algorithm by Freydenberger et al [6] is to enumerates (V, s)-tuples of $[\![A]\!](s)$ by enumerating the variable configurations for V from s_0 to s_N. To enumerate the variable configurations, we denote the set $\mathcal{V}config := \{\ \vec{c}_q \mid q \in Q\ \}$ to be the alphabet for the NFA called A_G. The NFA A_G is an automaton that contains all combinations of paths available in A from q_0 to q_f by traversing A with every symbol from 0 to N where σ_0 is the empty word. Edges from state $(i-1, q_m)$ to (i, q_n) in A_G where q_m, q_n are states in A and i represent the position $s[i]$ denoting the character σ_i in s, store the variable configuration of q_n in s_{i-1}, where s_{i-1} denote the position w_{i-1} which the interval between $\sigma_{i-1}w_{i-1}\sigma_i$. Enumerating A_G, we will get sequences of variable configurations for V from s_0 to s_N, allowing us to work out the boundary indices for spans of variables $x \in V$ and obtain set of (V, s)-tuples of $[\![A]\!](s)$.

The polynomial delay algorithm has the following process:

1. Takes a vset-automaton A and a string s as inputs. Assuming that the vset-automaton has already run through the functionality test and it is functional. The functionality test is a separate algorithm created by Freydenberger [4] which is shown in section 3.3.
2. We proceed to construct an NFA A_G.
   (a) To construct A_G, first, we construct a graph G which contain all possible path of A when reading the each letter in s.
   (b) In graph G, each state is represented as tuple (i, q) where i indicate the position of the string and q is the state in A. If there exists an edge between state $(i-1, q_n)$ to (i, q_m), this implies there exists a path from q_n to q_m that accept the symbol σ_i in s.
   (c) For each symbol $\sigma_0 ... \sigma_{|s|}$, we search for edges in A from state q_n to q_m which accept only a symbol of σ_i for some $i \in |s|$ and add edges of $(i-1, q_n)$ to (i, q_m) to graph G for all accepting edges.
   (d) Edges with empty word value where $\vec{c}_{q_n} = \vec{c}_{q_u}$ can be traversed freely when searching for an edge that matches with σ_i from q_n. Edges with either empty word value or variable operation where $\vec{c}_{q_n} \neq \vec{c}_{q_r}$ can be only be traversed in A after finding an edge that matches with σ_i. States with variable configuration changes from q_n to q_r where it passing through q_m which has an edge from q_n to q_m that accept σ_{i-1} are also added to graph G, so edges from $(i-1, q_n)$ to (i, q_r) are also added to graph G
   (e) Once a valid path is found for σ_i from some state q_n to q_m in A, an edge from $(i-1, q_n)$ to (i, q_m) with value $\vec{c}_{q_m}$ is added to G. Note that state $(i-1, q_n)$ must exists in G, so we only need to look for path from q_n in A if node $(i-1, q_n)$ exists in G.
   (f) G starts from q_0, all other states have the format of (i, q_n). We iterate from 0 to N. Starting from 0, we link the accepted states from q_0 to (i, q_n). To create states linking to q_0, we first search for paths within A from q_0 that does not contain any edges that can process any terminal symbols. Since σ_0 is a empty word in s, the condition of finding exactly one edge that accept the terminal symbol does not exist either. These paths allows us to create edges from q_0 to $(0, q_n)$ for some q_n as destination nodes. Next, we look for valid paths from q_n in A, to match the symbol σ_1. The process continue up to $i = N$.
   (g) To obtain A_G from G, we only keep nodes in G that are reachable from q_0 (the starting node) and $(|s|, q_f)$ (the terminal node). This particular process is called pruning. This process can easily be done by iterate through G from q_f to q_0. Since we created the graph G from q_0 all states are already reachable from q_0
3. Finally, we enumerate $\mathcal{L}(A_G)$ with out repetitions to find all sequences of variable configuration which extract all spans of V or the (V, s)-relations for $[\![A]\!](s)$.

Example 7. Let A be a functional vset-automata as shown below. We illustrate the graph G produced by A and the NFA A_G produced by G.

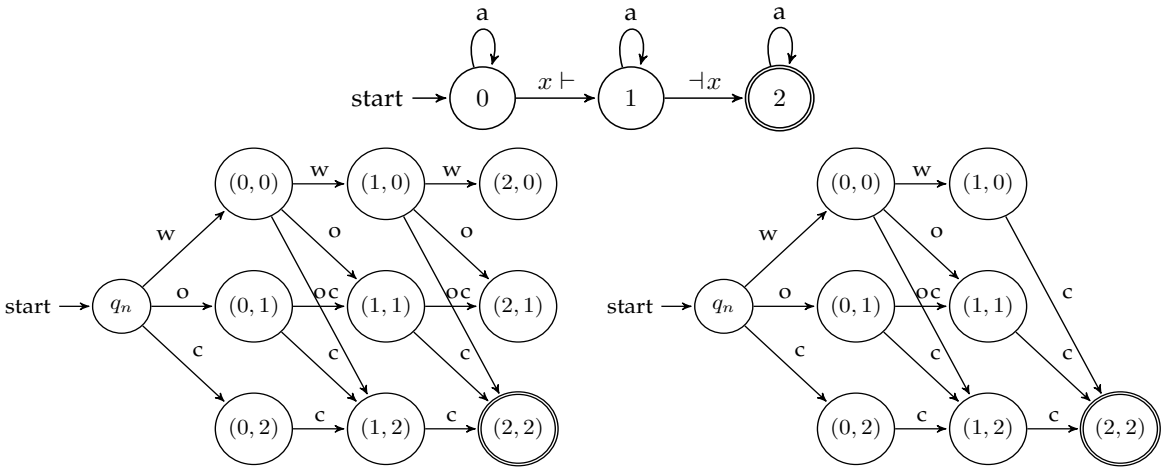


Figure 5: Showing automaton A, which graph G on bottom left and A_G on botton right

The automaton in Figure 5 on the left show the result of constructing A_G without pruning and the automaton on the right show us the NFA A_G we get from A shown on top.

3.3 Functionality Test Algorithm

This algorithm is use to test if the vset-automaton is function by checking every state in the automaton whether it satisfies the three conditions shown in section 5.1. Let $q \in \delta(p, a)$ denote the edge from state q to state p with value $a \in (\Sigma^* \cup \Gamma_V)$. Below show the pseudo-code for the functionality test for vset-automaton

Algorithm 1: Functionality Test

Input : A vset-automaton $A = (V, Q, q_0, q_f, \delta)$ with variable V
Output: if A is functional, return the set O and C, then return False

Initialization of $O, C, Seen, ToDo$
for $q \in Q$ **do**
  $O_q \leftarrow$ undefined;
  $C_q \leftarrow$ undefined;
$O_{q_0} \leftarrow \emptyset$; $C_{q_0} \leftarrow \emptyset$;
$Seen \leftarrow \{q_0\}$; $ToDo \leftarrow \{q_0\}$;
while $ToDo \neq \emptyset$ **do**
  $node \leftarrow$ ToDo.pop() `// Get any value from ToDo`
  foreach q *such that* $q \in \delta(p, a)$ *for some* $a \in \Sigma$ **do**
    if $p \in Seen$ **then**
      if $O_p \neq O_q$ *or* $C_p \neq C_q$ **then return** *False;*
    else
      $Seen \leftarrow Seen \cup \{p\}$; $ToDo \leftarrow ToDo \cup \{p\}$;
      $O_p \leftarrow O_q$; $C_p \leftarrow C_q$;
  foreach q *such that* $q \in \delta(p, x\vdash)$ *for some* $x \in V$ **do**
    if $x \in O_q$ **then return** *False;*
    if $p \in Seen$ **then**
      if $O_p \neq (O_q \cup \{x\})$ *or* $C_p \neq C_q$ **then return** *False;*
    else
      $Seen \leftarrow Seen \cup \{p\}$; $ToDo \leftarrow ToDo \cup \{p\}$;
      $O_p \leftarrow (O_q \cup \{x\})$; $C_p \leftarrow C_q$;
  foreach q *such that* $q \in \delta(p, \dashv x)$ *for some* $x \in V$ **do**
    if $x \notin O_q$ *or* $x \in C_q$ **then return** *False;*
    if $p \in Seen$ **then**
      if $O_p \neq O_q$ *or* $C_p \neq (C_q \cup \{x\})$ **then return** *False;*
    else
      $Seen \leftarrow Seen \cup \{p\}$; $ToDo \leftarrow ToDo \cup \{p\}$;
      $O_p \leftarrow O_q$; $C_p \leftarrow (C_q \cup \{x\})$;
if $C_{q_f} \neq V$ **then**
  return *False*
else
  return *All set of O and C*

Going through the algorithm, lines 1-6 are to initialise the sets *O*, *C*, *Seen* and *ToDo*. For each state $q \in Q$, the set *O* and the set *C* are used to stores variables $x \in V$ that has been opened and closed respectively. The set *Seen* contains all states which has its set *O* and *C* processed. The set *ToDo* contains states to be processed.

Once all sets had been initialised, we use a while loop with condition set *ToDo* not empty to process all states in the automaton. Within the while loop (from line 8), we pop a state q from *ToDo* and iterate through all edges δ from q to some state p with value a. For each edge, we check whether value a is terminal letter Σ or an open variable operation or a close variable operation. After the algorithm determined the type for a, it will proceed to check for any error occur in the set *O* and *C*. Take the current variable configuration and determine which variable configuration $a \in (\Sigma \cup \{\epsilon\})/x\vdash\ /x\dashv$. If next state is new, we set its set *O* and *C* to be the ones we compute. If the next state is in set *Seen*, check if set *O* and *C* are correct. If not, we return false. If true, we continue.

Different type of a will have a different input for the set O_p and C_p (see line 15, 22, 29). The state p will be added to set Seen to indicate that the state has been processed. p will be added to set ToDo to iterate its edges from p in the while loop. Lastly, once it has processed all states in the automaton, set C for the terminal state q_f should contain all variable $x \in V$ because all $x \in V$ should be closed at the terminal node. The algorithm will return set O and C, which can be used to produce variable configuration for each state.

3.4 Enumeration Algorithm

This enumeration algorithm is split into three functions, two sub-functions and one main function that use sub-functions. Below shows all three functions in pseudocode.

Algorithm 2: minString(l)

Input : An integer l with $0 \leq l \leq N$
1 **Assumptions:** state stack $S = S_0, ..., S_l$
Output: a smallest string $t = k_0, ..., k_l$ such that $k_i \in \mathcal{V}Config$ and t is accepted by A_G starting in state S_l
2 **Data Changes:** Updates the states stack $S = S_0, ..., S_l$ according to t
3 **for** $i := l$ **to** $N - 1$ **do**
4 *Find the minimal letter a from $q \in \delta(p, a)$ for all states $q \in S_i$, function: minletter*
5 k_i =minletter(S_i);
6 **if** $i \leq N - 1$ **then**
7 S_{i+1} :=*All destination nodes p where it has edge value of k_i from all nodes in S_i;*
8 *Add S_{i+1} to state stack S;*
9 $k_N := \vec{c}_{q_f}$;
10 **return** $k_l ... k_{N-1} \cdot k_N$;

Algorithm 3: nextString(k)

Input : A string $k = k_0 ... K_N, k_i \in \mathcal{V}Config$
1 **Assumptions:** state stack $S = S_0, ..., S_{N-1}$ contain nodes
Output: a new smallest string t where $t \in \mathcal{L}(A_G)$ with k has smaller letters then t or Empty String if t does not exist.
2 **Data Changes:** Updates the states stack $S = S_0, ..., S_l$ according to t
3 **for** $i := N - 1$ **to** 0 **do**
4 *Find the minimal letter a that is not equal to k_i from $q \in \delta(p, a)$ for all states $q \in S_i$, function: nextLetter*
5 k_i =nextLetter(S_i, k_i);
6 **if** k_i = *EmptyLetter* **then** *Remove $S_i fromS$;*
7 **else**
8 **if** $i \leq N - 1$ **then**
9 S_{i+1} :=*All destination nodes p where it has edge value of k_i from all nodes in S_i;*
10 *Add S_{i+1} to state stack S;*
11 **return** $k_l ... k_i \cdot$*minString*$(i + 1)$;
12 **return** *Empty String*;

Algorithm 4: Main Enumeration

1 $S_0 := \{q_0\}$;
2 **k** :=minString(0);
3 **while** ***k*** $\neq$ *EmptyString* **do**
4 **output k**;
5 **k** :=nextString(k);

The enumeration algorithm is used to obtain a list of outputs from the A_G graph by enumerating through the $\mathcal{L}(A_G)$ from 0 to N to get all set of (V, s)-tuples of $[\![A]\!](s)$. First, it define the state stack $S = S_0, ..., S_i$ where S_0 contains the start state q_0. The state stack is used to store all states available for position i. Next, the algorithm call the function ***minString(0)*** with value 0 to construct the smallest string of $\mathcal{L}(A)$. In ***minString*** function, we perform a iteration from input value to position $s[N - 1]$ to find the smallest letter for all states in S_i.

To find the smallest letters (***minletter*** function), we simply iterate through all edges for all in states S_i store the value (letter) of all edges into a dictionary of lists (denote as *availableletters*, format $\{\ i : [letters]\ \}$ and i is the position). During the iteration, we also store and group all states the is reachable with every letter for all state in S_i (denote as *letterofedges*, format $\{\ i : \{letter : [reachablestates]\}\ \}$). After the iteration for all states,

we simply sort the list in descending order (from w to o to c) and pick the first letter as the smallest letter and return the letters which is denoted as k_i. Then, we create S_{i+1} which contain all states p which has edge value of k_i for all node in S_i and add S_{i+1} to state stack. The data for S_{i+1} can be easily find using the information from *letterofedge*. Once all letters (which are alphabet of set $\mathcal{V}$*config*) are obtained find up to $N-1$, we set the last letter k_N equal to $\vec{c}_{q_f}$ which closes all variable $x \in V$. The function ends by returning the smallest string obtainable from $\mathcal{L}(A)$ and accepted by A_G starting in state S_i.

Next, going back to Main Enumeration, we have a while loop which search for the next smallest string for k using the function ***nextString(k)***. This function takes a string k, iterating from second last letter to first letter to find a minimum letter a that is not equal to k_i using the function ***nextletter***. Using the data from *availableletter*, we can check the existence of the next smallest letter a compared to k_i by looking at the number of letter in the list of *availableletter* of position i. If there exist at least 2 elements in the list, this implies that the next smallest letter does exists. We remove k_i from the list and return a as the next smallest letter. The stack S_{i+1} is created using the information from *letterofedges* and use the function *minString* generate the rest of the letters using the stack S_{i+1}. When nothing was return from ***nextletter***, this implies that next smallest letter does not exists and remove S_i from stack.

The ***nextString*** will return the next smallest string from $\mathcal{L}(A)$ until no more string is available from the q_0. The strings return in this enumeration are the (V, s)-tuples obtained from $[\![A]\!](s)$.

4 Implementation

In this section, we will describe the details of the implementation of algorithm chronologically. Starting from preparing the vset-automaton, using spanner algebra, processing the polynomial delay evaluation algorithm and displaying the final results.

4.1 The Library

The library of functions is written in programming language Python and run with Python3. The reason for choosing Python was due to the ease of defining data types and flexibility storing different data type within python dictionary and list. The library consists of five scripts:

1. script1.py consists of functions from algorithms such as functionality test, polynomial delay evaluation algorithm (to construct A_G) and enumeration algorithm (for enumerating $\mathcal{L}(A_G)$), also has reading text files data to create the automaton.
2. script2.py is used for defining data structure for vset-automaton, along with its class methods and conversion of regex formula to vset-automaton through the use of a parser generator.
3. script3.py consists of algebraic operations for automata such as union, projection, string equality.
4. scriptgrph.py contains functions for displaying vset-automata in a graph and printing out the final results from enumeration algorithm
5. scriptlibrary.py is the script that contain functions that call all functions from the four scripts.

The user will only need to import scriptlibrary.py file in the python script in order to access all functions provided in the library. The library uses the following packages: sys, time, re, copy, os, texttable, arpeggio and graphviz. It is necessary to install all these packages in python in order to use the library.

4.2 Pre-processing

There are three possible inputs the library can accept and convert to a vset-automaton. The user can insert a text file containing a specific format of the automata, or input a regex formula or create the automaton object directly.

4.2.1 Automata class

The automata class is the main format for storing information of the vset-automaton. In the library, numbers are used to label each state in A but the label interchanges between integer and string data type depending with it uses within the library. Initially, states started with the integers data type. This is so that it is possible to add new states to the automaton by simply incrementing the number. However, in order to support labels that use alphabetical letters or symbols, it is necessary to convert its data type to a string. Although the states are numbers, we still use q as states in A and q_0 as starting state and q_f as terminal state in this report.

An object created from automata class is known as an automaton object which stores the following instance variables: These instance variable are called by **'object_name'**.*variable_name* in python script.

- ***.start***
  - Data type: An integer or a string
  - Purpose: Store the starting state of the automaton
- ***.end***
  - Data type: An integer or a string
  - Purpose: Store the terminal state of the automaton
- ***.last***
  - Data type: An integer
  - Purpose: Store the name of last node created, used only when building and modifying the states of the automaton.
- ***.states***
  - Data type: A list of strings
  - Purpose: Store all states in automaton.
- ***.varstates***
  - Data type: A list of strings
  - Purpose: Store the name of $x \in V$.
- continue next page...

- ***.varconfig***
  - Data type: A dictionary of lists
  - Format: $\{q : \vec{c}_q\}$ where $\vec{c}_q = [\vec{c}_q(x_0), ..., \vec{c}_q(x_{|V|})]$, for all $x_i \in V$ and x_i refer to variable x with integer i in ***.key***.
  - **Example** 8: Example of ***.varconfig*** $\{A : [w, w],\ B : [o, o],\ C : [c, c]\}$
  - Purpose: Store $\vec{c}_q$ for all states q.
- ***.key***
  - Data type: A dictionary (for referencing)
  - Format: $\{x : k\}$, where $x \in V$ and k is some integer.
  - Purpose: Each variable $x \in V$, refer to a pointer of the list $\vec{c}_q$ from ***.varconfig***. Using the pointer, we can extract the variable configuration of x from the list.
  - **Example 9**. Given that ***.varconfig*** $= \{D : [w, o, c]\}$ and ***.key*** $=$ $\{x : 0,\ y : 1,\ z : 2\}$. Using pointers from the list, we can extract the following:
    .varconfig$[\,D\,][\,0\,] = w$, ***.varconfig***$[\,D\,][\,1\,] = o$, ***.varconfig***$[\,D\,][\,2\,] = c$

    Variable x has a pointer of 0 in ***.key***, we see that the $\vec{c}_D(x) = w$.
  - For all $x \in V$, we assign a pointers for the list $\vec{c}_q$. Therefore, variables to pointer must be established before creation of $\vec{c}_q$ lists in ***.varconfig*** and order of the variable configurations must follow the assignment in return.
- ***.transitions***
  - Data type: Dictionary of lists with tuples
  - Format: $\{q : [(t_1^j,\ t_2^j)]\}$
    where q are states, with contents as list of tuples. Each tuple represent an edge from q, where $|q|_{edges} = j$ and $t^j = (t_1^j, t_2^j)$. Each tuple contain an destination t_1^j and a value t_2^j for q.
    t_1^j is either an integer or a string and t_2^j is a string.
  - To store all edge of the automaton for each state q.

Initially, ***.varconfig*** had the format: $\{q : \{x : \vec{c}_q(x)\}\}$, for all $x \in V$. However, for every state q, we will need a dictionary for all $x \in V$ which takes up memory space in python. Therefore, it was replaced with the current version. A future optimization would be remove referencing for $x \in V$ in ***.key*** and replace it with a reference to all possible patterns of $\vec{c}_q$ in $\mathcal{L}(A)$ (the set *Vconfig*). This method will remove redundant copies of $\vec{c}_q$ for the automaton object and save memory in the library.

When initialising a vset-automaton, we create an automaton object from class and insert data to all instance variables except for ***.last***, ***.key*** and ***.varconfig***.

In the automata class, there are several methods available. Methods are functions that only be call from an automaton object. We call the method using **'object_name'**.*method_name* in python script.

- ***def reset(self):***
  - This method resets all instances variables in automata object to empty. It is used when initialising an automata object or when we reuse an automata object. (located at line 40-48 in script2.py)
- ***def tostr(self):***
  - This method changes all integer values in automata class objects to string for automaton object. It is used to transform the automata object suitable for processing in the algorithm. (located at line 51-65 in script2.py)
- ***def rename(self):***
  - This method renames the existing states in the automaton object to a number but stored in as a string data type. It is used after each join operation between two automata objects. (located at line 67-94 in script2.py)
  - Let assume we have two automata objects A_1 and A_2 and after the join operation, we get A_J. Let $q(A_J)$ be states q of A_J, each $q(A_J)$ is labelled as: $(q(A_1),\ q(A_2))$, a combination of two states of A_1 and A_2 in a tuple with stored as a string data type.
  - In the situation when multiple join operations are to be executed, A_J will result in a very long string display tuples within tuples of states from previous automaton object, which are unnecessary. Therefore, by renaming the states, it will prevent this situation which is the purpose for this method.

- ***def toint(self):***
  - Similar to def rename(self), the method rename the existing states in automaton object to a number as a integer data type. It is used to convert automaton object suitable for processing some algebraic operations. (located at line 98-125 in script2.py)
- ***def printauto(self):***
  - The method simply print out all instance variables in automaton object to terminal (located at line 214-234 in script2.py)

These methods are used to transform the automaton object format for processing with spanners. Below, we display the methods that are used strictly for when building the vset-automaton from regex formula when enumerating through a parse tree. When the parser started to creation of the parse tree and encounter any of the syntax for regex formula, the parse tree will create a node named with the operation of the syntax, i.e *union* and link the node with children from the grammar, i.e. φ_1 and φ_2. It will continue to do so until it reach the leaf node which is used to call the initialisation of the automaton object when enumerating through the parse tree. From the leaf nodes, we enumerating through the parse tree and call methods to transform our regular expression to a λ-NFA [11].

Note that for every transition on the automata, we adapted this to have exactly one terminal state. While enumeration, it will continue to build our automaton until we reach the root node where we would have a complete conversion of the regex formula into vset-automaton. (More details on when we call these methods in the next section).

- ***def renumber(self,num):***
  - Add ***num*** value to all numbered variable instances in automaton object, only applied where all states in automaton object is of integer type. Used to rename all states in the automaton. (located at line 127-148 in script2.py)
  - This method is used when we want to rename all states in the automaton object by increasing all states with a number.
  - **Example** 10: By applying $A_1.renumber(5)$. It should increase all states with value of 5.

	A_1	A_1*.renumber(5)*
.start	0	5
.end	2	5
.last	2	7
.states	$[0, 1, 2]$	$[5, 6, 7]$
.transition	$\{0 : [(1, a)], 1 : [(2, a)].2 : [\,]\}$	$\{5 : [(6, a)], 6 : [(7, a)].7 : [\,]\}$

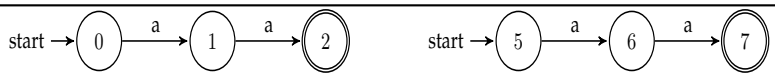
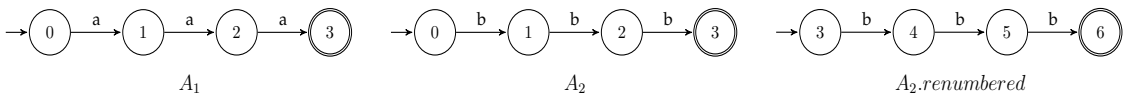
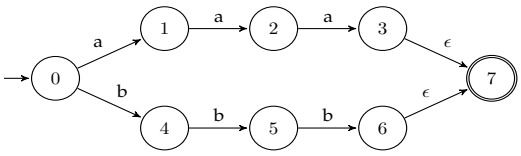


Figure 6: A_1 on the left and $A_1.renumber(5)$ on the right

- ***def addedge(self,start,dest,value):***
  - This method is used to add an edge to the automaton object. (located at line 150-162 in script2.py)
- ***def union(self,auto1):***
  - Used when converting regex formula to vset-automaton. This is not the union for spanner algebra.
  - In this method, we change all name of states in auto1 and change auto1 start and end state to ***self***.start (***self*** refer to the automaton object used to call the method). Lastly, we add a new end state and add an empty edge both old end states to new end state. Update self automaton object with all values in auto1. In order to
  - Below, we show an illustration of automata A_1 and A_2 union together by this method.



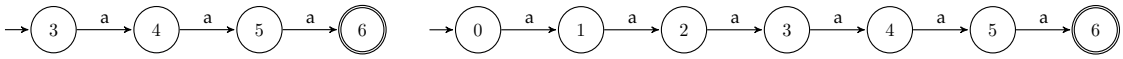
A_1 A_2 $A_2.renumbered$



$A_1 \cup A_2$

Figure 7: Showing from the top, automata A_1, A_2 and A_2 which is renumbered by 5. From the botton, the resulting automaton from the union of A_1 and A_2.

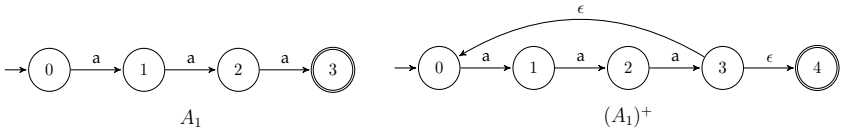
- ***def concat(self,auto1):***
  - Similar to union, rename all state in auto1 and only need to replace start state on auto1 with end state on ***self*** and set the end state on self automaton object to auto1 end state. (Code located at line 182-191 in script2.py)
  - Let A_1 and A_2 be the same as the previous example, we show an illustration of automata A_1 and A_2 concatenated together by this method.



$A_2.renumberwithA_1.end$ $\qquad$ $A_1 \cdot A_2$

Figure 8: Showing renumbered A_2 and concatenation of A_1 and A_2.

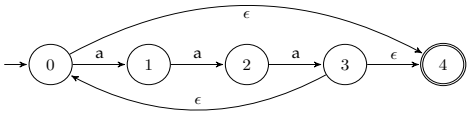
- ***def plus(self):***
  - Again only use when converting regex formula to vset automaton. Add an edge in ***self*** automaton from end state to start state. Then add a new end state and an empty edge from old end state to new end state. (Code located at line 196-199 in script2.py)
  - We show an illustration of automata A_1 and $(A_1)^+$.



A_1 $\qquad$ $(A_1)^+$

Figure 9: Showing A_1 and A_1 with plus

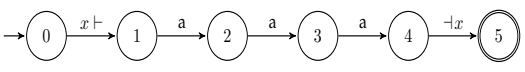
- ***def star(self):***
  - Same as ***def plus(self)***, but also an an empty edge from start state to new end state.
  - Let A_1 be the same as before, we show an illustration of automata $(A_1)^*$.



$(A_1)^+$

Figure 10: Showing previous A_1 with star

- ***def addvarconfig(self,alpha):***
  - For this operation, we create a new start state and end start. We add an edge from new start state to old start state with the open variable operation of letter alpha. Then we add an edge from old end state to new end start with the close variable operation of alpha. Update ***self.****varstates* with alpha. (Code located at line 207-212 in script2.py)
  - Let A_1 be the same as before, we show an illustration of automata $A_1.addvarconfig(alpha)$, with $alpha = x$.



(A_1) with variable x

Figure 11:Showing A_1 when adding variable x

4.2.2 Regex Formula

To read and convert regex formula, we used a parser generator called 'Arpeggio' [8] which allows us parse the regex formula with a written grammars to create a parse tree of the regex formula. Once the parse tree is created, we can efficiently build the automata object from enumerating the parse tree by creating functions which can be executed when the enumeration find a specific node in the parse tree. 'Arpeggio' is based on PEG [3] and it is written using Python statements. All information about the grammars and parsing information are written in script2.py file. The parser is activated when the main function of the script is called. Below we show the code of the main function in script2.py.

```
def main(argv):
  parser = ParaserPython(formula)
  parse_tree = parser.parse(argv)
  results = visit_parse_tree(parse_tree, formVisitor())
  result.tostr()
  return result

if __name__ == '__main__":
  main(sys.argv[1])
```

The input *argv* is the regex formula to be parse. First the function create a parser from the grammar. Then it will generate a parse tree from the regex formula. The parse tree is enumerated with the class *formVisitor()* which provide function when enumeration encounters specific nodes. To run the parser generater, we can either execute the script through the terminal and writing the regex formula next to the script name or we can import the script into another python script and call the main function directly.

A grammar contain series of rules to match when parsing the regex formula. The grammer used in the library is shown below.

```
def alphabet():    return _(r'([^\,\|\<\>\*\+])+')
def varconfig():   return "<", _(r'[a-zA-Z0-9]') ,":", expression,">"
def terminals():   return [plus, star,("(", expression, ")"), alphabet, varconfig]
def plus():      return alphabet,"+"
def star():      return alphabet,"*"
def concat():    return terminals, OneOrMore(",", terminals)
def union():     return terminals, OneOrMore("|", terminals)
def expression(): return [concat, union, varconfig, terminals]
def formula():     return OneOrMore(expression), EOF
```

Each rule in the grammar is labelled with a rule name. When a rule is accepted during parsing of regex formula, it will create a node denote n labelled with the rule name in the parse tree and the returned values for the rule are treated as children nodes for n. For example, the rule ***def union():*** will have children nodes terminals, ",", terminals. The nodes terminals will have its own children node from the rule and so on. The only terminal match rule is ***def alphabet():*** which will create a leaf node in the parse tree with a value that does not have any special characters used in the grammar.

1. With this grammar, it will first split the regex formula into expressions.
2. ***Expressions()*** can match either ***concat()***, ***union()***, ***varconfig()*** or ***terminals()***.
3. To match ***concat()*** and ***union()***, it should match at least two terminals with syntax for concatenation , and union | in-between.
4. To match ***varconfig()***, it need to match the format $<x$:expression> where x must match regular expression '[a-zA-Z0-9]'.
5. ***Terminals()*** can match either ***plus()***, ***star()***, ***expression()*** with brackets, ***alphabet()*** or ***varconfig()***.
6. To match ***plus()*** and ***star()***, it should match an ***alphabet()*** with symbol $+, *$ respectively.
7. Finally, an ***alphabet()*** return string x that matches regular expression that does not accept value which has any special characters used in the grammar.

Once a parse tree is successfully created from the regex formula, we want to construct our vset-automaton (automata object) while reading the information from the parse tree. We need to define some functions (we denote as visit functions) such that when a node labelled with a rule name from the grammar is visited, we will retrieve information from its children node to perform some automata construction operations. These

operations are the methods which we described in the previous section. Out of all the rules in the grammar, only ***expression()*** and ***formula()*** rule does not have a visit function. Please look at line 237-279 in script2.py for the code.

Once the enumeration is finished, it will return an automata object which is a complete conversion of regex formula to vset-automaton.

4.2.3 Reading File

One other way to generate an automata object is by reading in a .txt file containing information of the edges in the automata. Each edge has the format: startnode,endnode,value; States has to be an integer, the value can be any alphabet and including variable operations where + represent open operation and - is close operation. The output is an automata object. (see line 22-47 in script1.py for code)

4.3 Main Processing

For each function in this section, we will present the task to be completed, description of how the task was implemented initially, the benefits and disadvantages of the initial implementation, the final implementation of the task and possible future improvement and alteration to be considered.

4.3.1 Functionality Check

The task is to check whether the given vset-automata is functional. The algorithm for the functionality check is taken [4] and it is shown and described in section 3.3. The initial implementation of the functionality check implemented exactly as it is shown in section 3.3 except that instead of returning set O and C, we iterate through the sets, to get the variable configuration for all state and store each configurations in automata class variable ***.varconfig***. The initial implementation was successful. It clearly distinguish each process with in the function and obtain accurate results. However, the function was not optimized as there were many loops which can be made redundant. The pseudocode below show main loops of the initial implementation.

```
def funchk(automata):
  #Initialisation of O,C,Senn,ToDo
  ...(see line 73-96 in script1.py)...
  while todo:
    item = todo.pop()
    for delta(p,a) in automata.transition[item]:
      #for some a in \Sigma
      ...(see line 10-14 in Algorithm 1, section 3.2.1)
    for delta(p,a) in automata.transition[item]:
      if last letter a == '+':
        #if a[:-1] is in automata.varstates (by iterating in  automata.varstates)
          ...(see line 16-21 in Algorithm 1, section 3.2.1)
    for delta(p,a) in automata.transition[item]:
      if last letter of a == '+':
        #if a[:-1] is in automata.varstates (by iterating in automata.varstates)
          ...(see line 24-28 in Algorithm 1, section 3.2.1)
  #Store variable configurations for all states by iterate through all states.
```

Looking at the time complexity of the initial implementation. We have a *while* loop for iterating through the states with O(n), three *for* loops within *while* loop for iterating through its edges denote O($3m$), if the edge value is a variable operation, then we need to check if $x \in V$, iterating through V giving us O(v), outside of *while* loop, we loop through all states to find $\vec{c}_q$ give us O(n), so in total we have O($n \cdot m \cdot (2 \cdot m \cdot v) \cdot n$) = O($2(n \cdot m^2 \cdot v)$).

In the final implementation, we reduce the three *for* loop to one and have three if statement to check whether the edge value a is $a \in \Sigma$ or $a[-1] =$'+' or $a[-1] =$'−' ($a[-1]$ is last letter in a). With the $+$ and $-$ if statement, we also check whether $len(a) \geq 1$. We assume that there is variable $x \in V$ if the value a has length greater than 1 and last letter is either $+$ or $-$. $len(a) \geq 1$ condition is necessary because there might exists an edge that only accept the symbol $+$ or $-$, we need to ensure that variable is attached to $+$ or $-$. This new condition will removing the need to iterate through V (***.varstates***) to check if $a[:-1] \in V$. We also reduce the need to iterate through the state to work out the $\vec{c}_q$ by working out $\vec{c}_q$ with in the *while* loop and storing them in ***varconfig*** in

the automaton object. These changes have improved the time complexity to O($n * m$). The code below, shows that final implementation of functionality test.

```
def funchk(auto):
  #Initialisation (see line 73-96 in scripty1.py)
  seenlist = {str(auto.start)}  #Store seen nodes
  todolist = {str(auto.start)}  #Store nodes for
    processing
  while todolist:
    origin = todolist.pop() #Return an element from
     the right side
    for item in auto.transition[origin]:
      dest = str(item[0])
      letter = item[1][:-1]
      op = openlist[dest]
      oq = openlist[str(origin)]
      cp = closelist[dest]
      cq = closelist[str(origin)]
      if item[1][-1] == '+' and len(item[1]) > 1:
        if oq & {letter}:
          print ('Error from open: Variable state',
    letter,'have multiple open, not functional')
          sys.exit(1)
        if seenlist & {dest}:
          if op != (oq | {letter}) or cp != cq:
            print ('Error from opne: Obtained
    multiple variable configuration for one node')
            sys.exit(1)
        else:
          seenlist.add(dest)
          todolist.add(dest)
          openlist[dest] = (openlist[origin] | {
    letter})
          closelist[dest] = closelist[origin]
          auto.varconfig[dest] = copy.deepcopy(auto
    .varconfig[origin])
          auto.varconfig[dest][auto.key[letter]] =
    'o'



      ...continue
      continue...
      elif item[1][-1] == '-' and len(item[1]) > 1:
        if {letter} - oq or cq & {letter}:
          print ('Error from close: Either the
    variable state has not opened or its has
    already closed')
          sys.exit(1)
        if seenlist & {dest}:
          if op != oq  or cp != (cq | {letter}):
            print ('not function different states
    for one node')
            sys.exit(1)
        else:
          seenlist.add(dest)
          todolist.add(dest)
          openlist[dest] = openlist[origin]
          closelist[dest] = (closelist[origin] | {
    letter})
          auto.varconfig[dest] = copy.deepcopy(auto
    .varconfig[origin])
          auto.varconfig[dest][auto.key[letter]] =
    'c'
      else:
        if seenlist & {dest}:
          if op != oq  or cp != cq:
            print ('Error normal: Multiple variable
     states for one node')
            sys.exit(1)
        else:
          seenlist.add(dest)
          todolist.add(dest)
          openlist[dest] = openlist[origin]
          closelist[dest] = closelist[origin]
          auto.varconfig[dest] = copy.deepcopy(auto
    .varconfig[origin])
for confg in auto.varconfig[str(auto.end)]:
  if confg != 'c':
    print ('There exists variables in auto that are
     not closed')
    sys.exit(1)
```

A potential optimization for this function would be to use the variable configurations stored in the automaton object instead of using set *O* and *C* to check between the variable configurations. This will remove the need for set *O* and *C* and save up some memory space during the process but it will create complicated if statements when performing checks between the variable configurations due to ***.varconfig*** store only a list of variables states for each state. In order to distinguish between the variable states, one must refer to ***.key*** in the automaton object, to extract the relation between list positions and variable.

4.3.2 Conversion to epsilons

After a successful functionality check, we will convert all edges in the automata object with value of variable operations $\Gamma_V = \{x \vdash, \dashv x | x \in V\}$ to ϵ. Replacing the Γ_V values with ϵ will change the language of vset-automaton to $\mathcal{L}(A) \in (\Sigma \cup \{\epsilon\})$ instead $\mathcal{L}(A) \in (\Sigma \cup \Gamma_V \cup \{\epsilon\})$. The purpose of Γ_V is to recognise changes in $x \in V$ between states but since we have $\vec{c}_q$ for all states from the functionality check, Γ_V values are not longer necessary. Also, when constructing A_G with a string s, the edges with Γ_V values does not actually need to match with any character in s (see section 3.2 for details), so replacing these values with ϵ will not pose any problem. The code is located on line 54-67 in script1.py. This functionality is perform with a separate function in the library but it is possible to combine this functionality within the functionality test function as a future improvement for the library for better efficiency. However, we left this functionality as a separate function to

properly separate the tasks in order to get a clear structure in the library.

4.3.3 Spanner Algebra Operations

When automata objects have processed with ***def csymtonulllong(auto):***, we can perform algebraic operations on the automata objects. There are four spanner algebraic operation: Union, Projection, (Natural) Join and Selection (String equality selection), please read section 2.3 for its definitions and detailed explanation for spanner algebra. All spanner algebra are to be used before using the evaluation algorithm.

4.3.3.1 Union Operation

Below show the code for the union function for spanners (vset-automaton). The actual code is located at line 468-521 in script3.py. We use union for searching spans that could contain either search term e_1 or search term e_2 but not both, since they have the same variable $x \in V$.

```
def union(auto1,auto2):
  if set(auto1.varstates) == set(auto2.varstates):
    auto1.toint()
    auto2.toint()
    auto1.renumber(int(1))
    auto2.renumber(int(auto1.end+1))
    auto1.last = auto2.end+1
    auto1.addedge(auto1.end,auto1.last,'[epsi]')
    auto1.addedge(auto2.end,auto1.last,'[epsi]')
    auto1.addedge(0,1,'[epsi]')
    auto1.addedge(0,auto2.start,'[epsi]')
    templatew = []
    tempatec = []
    for item in auto1.varstates:
      templatew.append('w')
      templatec.append('c')
...continue...
```

```
...continue...
    auto1.varconfig[0] = templatew
    auto1.varconfig[auto1.last] = templatec
    auto1.states = []
    for key, item in auto2.transition.items():
      if not key in auto1.transition:
        auto1.transition[key] = []
      auto1.transition[key].extend(item)
      auto1.varconfig[key] = auto2.varconfig[key]
    temp = {}
    for i in range(auto1.last+1):
      auto1.states.append(str(i))
      temp[str(i)] = auto1.varconfig[i]
    auto1.varconfig = temp
    auto1.start = 0
    auto1.tostr()
  else:
    print('cannot union, different set V')
```

This function take two automata objects (auto1, auto2). In the code, we have a condition statements on line 2 which only runs the operation if both automata has the same $\mathbb{V}$ set, otherwise it will print out an error message to the terminal. This union function is similar to the union method for automata object discussed in section 4.2.1. The only difference is checking if $\mathbb{V}(P_1) := \mathbb{V}(P_2)$ on line 2, converting the states of automata into numbers a integer data type on line 3-4 and updating the variable configuration on line 12-31 and line 39-42. One future improvement would be to be able to use this function for $P_1(s) \cup P_2(s)$ since currently, our function is only used for $(P_1 \cup P_2)(s)$ since we do not do any more processing in the library after the evaluation algorithm other then printing out the results.

4.3.3.2 Projection Operation

The projection function in the library has two required inputs which are an automaton object and a list of variables for projection and output an automata object. Below show the code for the projection function. The actual code is located at line 13-52 in script3.py. We use projection to remove span of some variable $x \in V$ that we do not wish to display. For example, the spans of x contains extraction of locations and spans of y contains extractions times and we wish to display all location in the string, so we use projection function to only keep variable x.

```
def projection(automata,listofprojections):
auto = sc2.automata(0,0,0) #Initialise a new automata object
auto.reset()
auto = copy.deepcopy(automata)
auto.varstates = listofprojections
newkey = {}    #New key for new varstates
templist = [] #List of positions from old key for new varstates
for i in range(len(listofprojections)):
```

```
    newkey[str(listofprojections[i])] = i
for state in list(auto.key):
  if state in listofprojections:
    templist.append( auto.key[state] )
#Replace current varconfig from nodes with only new varstates
for key, var in auto.varconfig.items():
  tempvar = []  #Store new varconfig for node(key)
  #Get desire varconfig of new varstates using positions in templist
  for pos in templist:
    tempvar.append(var[pos])  #Get projected varconfigs from old automata data
  auto.varconfig[key] = copy.deepcopy(tempvar)
auto.key = newkey

return auto
```

To perform projection on the automata object, we simply need to remove all variable of automaton that are not in the list of projection variables. The projection function first create a copy of automata object input, so we do not modify the input and set *varstates* equal to the list of projection variables. Then, the function create a list of pointers in ascending order for the projection variables. This list of pointers is used to extract the variable configuration of the projection variables in *varconfig* and replace every $\vec{c}_q$ list with a new $\vec{c}_q$ list that only include project variables. Lastly, a new key is set for the updated $\vec{c}_q$ list and the function will return the edited automata object.

4.3.3.3 Natural Join Operation

This algebraic operation was one of the longest and hardest to implement. There were many elements to look out for during its implementation in the library and it was not as straight forward as it initially seem from the definition. A natural join operation between two automata is the combination of its states and have edges that retain feature of both automata. Intuitively, if we have a span of x in A_1 that extract dates and a span of y in A_1 that extract times. Joining these two automata will look all dates and times with variable x and y.

Example 11. Let A_1 and A_2 be vset-automata. Figure 12, we show the result of $A_J = (A_1 \bowtie A_2)$.

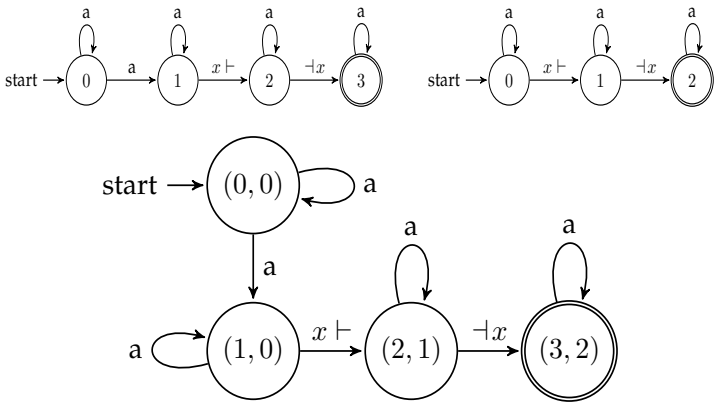


Figure 12: Showing automaton A_1, A_2 and A_J

As we can see, Figure 12 show A_1 on the top left, A_2 on the top right and A_J at the bottom. The states of A_J is a combination of a state in A_1 and a state in A_2 stored in a tuple. This will be the format of a state in A_J for the natural join function. A edge exists between (q_{n_1}, q_{m_1}) and (q_{n_2}, q_{m_2}) if it satisfies the following conditions:

1. An edge exists if the destination state (q_{n_2}, q_{m_2}) where q_{n_2} is a state in A_1 and q_{m_2} is a states in A_2, has the same variable configuration for all variable x that are in both $\mathcal{V}(A_1)$ and $\mathcal{V}(A_2)$. For vset-automata in the join operation, for every state (q_n, q_m) in A_J we have $\vec{c}_{q_n}(x) = \vec{c}_{q_m}(x)$, for all $x \in \{\mathcal{V}(A_1) \cap \mathcal{V}(A_2)\}$.
2. An edge exists if $q_{n_1} \in \delta(q_{n_2}, val_1)$ and $q_{m_1} \in \delta(q_{m_2}, val_2)$ such that $val_1 = val_2$. In other word, the transition from state to state in both automata must have the same edge value.

For every state in a vset-automata, we assume there is an empty edge from q_i to q_i, an empty edge that loop to itself. The following list show the condition for adding edges between states for the join operation.

1. If ($q_{n_1} = q_{n_2}$ and $q_{m_1} \neq q_{m_2}$) and (q_{n_2}, q_{m_2}) is a valid state. Then, we have an edge (q_{n_1}, q_{m_1}) and (q_{n_2}, q_{m_2}) if the edge between q_{m_1} and q_{m_2} is an empty edge. This satisfied the condition for having the

same edge value for both automata transitions because $q_{n_1} = q_{n_2}$. We can assume that there is an empty edge from q_{n_1} to q_{n_1}, thus both transitions have the empty word transition.

2. If ($q_{n_1} \neq q_{n_2}$ and $q_{m_1} = q_{m_2}$) and (q_{n_2}, q_{m_2}) is a valid state. Then, we have an edge (q_{n_1}, q_{m_1}) and (q_{n_2}, q_{m_2}) if the edge between q_{n_1} and q_{n_2} is an empty edge.
3. For the case when ($q_{n_1} \neq q_{n_2}$ and $q_{m_1} \neq q_{m_2}$), we must check if the edge from q_{n_1} to q_{n_2} have the same value as the edge from q_{m_1} and q_{m_2}. In other words, for $q_{n_1} \in \delta(q_{n_2}, val_1)$ and $q_{m_1} \in \delta(q_{m_2}, val_2)$, we have $val_1 = val_2$. If both transitions, q_{n_1} to q_{n_2} and q_{m_1} to q_{m_2} have the same edge value and (q_{n_2}, q_{m_2}) is a valid state. Then, the resulting edge will contain a value from the either edge.

There are two ways of implementing the join operation with these conditions. One method is the on demand approach. We have a stack or queue with the starting state (q_n, q_m) of A_J (which is a tuple of start states for A_1 and A_2), we iterate through the edges of q_n and q_m from A_1 and A_2 respectively and add edges to A_J if it satisfy the condition above. The destination states for the added edges push to the stack or queue and continue to get states from the stack or queue to complete the join operation. The alternative method would be to iterate through all states in A_1 and A_2 to get a list of all possible valid states for A_J. Using this list, we proceed in add edges between the valid states to complete the join operation.

The former method of using a stack or queue is chosen for implementation. Initially, the join operation is implemented using a set $todo$ which store all valid states (q_n, q_m) to be processed, a while loop that check whether the set is empty, within the while loop, the function iterate through the edges for q_n and q_m in A_1 and A_2 respectively looking for equal edges, then check whether the destination state is valid. However, there are several underlying important elements of the join operation that we need to look out for in the implementation.

1. First, dealing loops in the automata. When processing a state q, it is possible to get multiple destination states from the q and as a result, many redundant edges are created in A_J. To prevent this, when processing a state q within the while loop, a set labelled $seen$ is initialised. This set will store all states that has already been added with a edge from q. Before adding a new edge, the function will check is states exists in $seen$ and will not add any edge if state is in the set.
2. Secondly, there were instances where the same state is processed again. While set $todo$ will not store repeated states if the state already exists in the set. However, once a state is processed and discarded, discarded state were inserted in the set again through processing other states. Similarly from before, a set $done$ was added to the function to store all processed state to prevent preprocessing of the same state.
3. Checking variable configuration in the both automata is a delicate process. Recall that a pointer is used to read a variable configuration for a variable $x \in V$ and every $x \in V$ is assigned a pointer. When reading and comparing the variable configuration for some state q for a variable x, the pointer from the automata objects for variable x is not the same and an error will occur is the said variable x does not exists in one of the automaton. Therefore, a new reference dictionary which stores the pointer for both automata for variable x was created in the function and when a variable x does not exists in an automata, the pointer will be labelled as -1 to indicate x does not have a variable configuration in an automata.
4. The implementation needed to deal with reading multiple empty word edges. This was very important discovery during the testing of the implementation. The join function is set to read just single edges from (q_n, q_m) for both automata and does not goes beyond reading empty word edges through multiple states.
Example 12. Let assume that A_1 and A_2 are vset-automata.

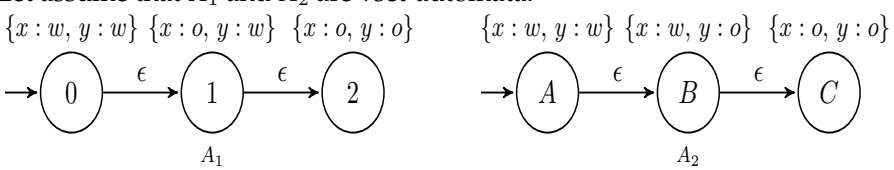


Figure 13: Showing A_1 and A_2 with variable configuration for each states

Starting with the valid state $(0, A)$, the function will only check through states $[(0, B), (B, 0), (1, B)]$. Also, looking at the variable configuration between the states, $(2, C)$ is a valid state and state A and 0 should be able to reach C and 2 respectively by going through empty word edges. Currently, the join function will never check whether state $(2, C)$ is valid since the function only read single edge from state A and states $[(0, B), (B, 0), (1, B)]$ are rejected due to having different order of opening operation for variable x and y. However, since both open x and open y are connected through empty word edges, it is not necessary to

strictly going through the order of opening for both automata. Creaing an empty edge from 0 to 2 and A to C would be valid.

This problem of read multiple empty word edges is solved by adding new empty edges to the automata object when more than one empty edges are connected between the states before joining the automata together. The functions ***def addepsilon():*** and ***def checkfunction():*** are sub-functions in the join function which are used perform this task. The ***def addepsilon():*** iterate all edges for all states in the automata and when it finds an empty edge, it will pass the start and end state of the edge to ***def checkfunction():***. The ***def checkfunction():*** checks for empty edges from the end passed end state for edges that does not end in start state and only add new empty edge if it does not exist in the start state. The code for this function is shown below.

```
def addepsilon(auto):
  for startnode, tuples in auto.transition.
    items():
    for tup in tuples:
      #If find an [epsi] value, we pass the
    start node and the end node of the edge
    to the subfunction.
      if tup[1] == '[epsi]':
        checkfunction(auto,startnode,tup[0])
```

```
def checkfunction(auto,start,search):
  for tup in auto.transition[search]:
    if tup[1] == '[epsi]' and tup[0] != start:
      #Only add the epsi edge if it does not
    exist in the start node.
      if (tup[0],'[epsi]') not in auto.transition
    [str(start)]:
        auto.transition[str(start)].append((tup
    [0],'[epsi]'))
      #Continue searching for possible paths
      checkfunction(auto,start,tup[0])
```

Example 13. Illustrating the use of ***def addepsilon():*** function

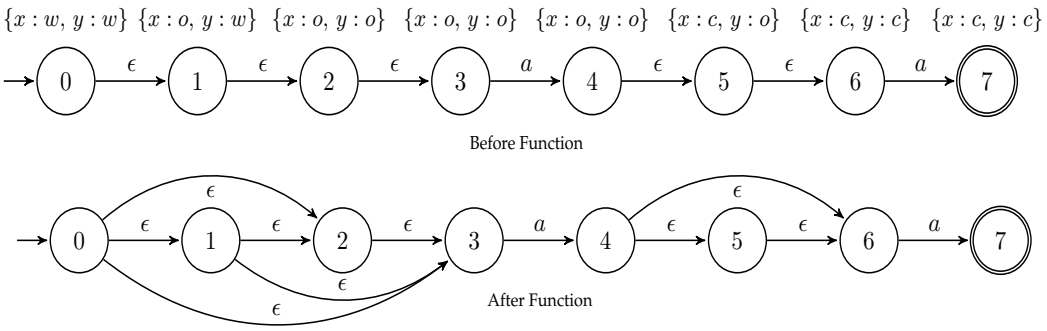


Figure 14: Showing the different before and after running the ***addepsilon()*** function.

An alternative method to read multiple empty word edges would be to check for empty word edges for all state q_n produced when processing state q_m. However, this will lead to excessive amount of iteration in the automata to check for empty word edges for every state produced which is insufficient.

5. The edge values are essentially regular expression and when constructing the A_G graph, we perform a regular expression match between the characters of the string with the value of the edge. In cases where the edge contain special character, we need to ensure that matching these special characters are properly expressed as a regular expression. In the join function, we added a extra condition for matching two regular expression. When the two value of the edges are not equal and not empty edges, then we assume that they are regular expressions and we use $(?=reg_1)(?=reg_2)$ to join the two regular expression together (Code located at line 226-247 in script3.py). $(?=)$ is called a positive lookahead in regular expression, when we have $p(?=q)(?=t)$ it will try to match q by looking ahead of the matched character p and check whether it matches. if it does not, it stay on the character p and try to match the next expression $(?=t)$. In our case, p is a empty word, so $(?=reg_1)(?=reg_2)$ will try to find a character that either matches reg_1 or reg_2.

Below, we show the code for the sub-function ***def joincreate():*** used to initialise the joined automata object, the sub-function ***def checklegal():*** which checks if the destination states is valid for the join automata and code for join function.

```
def joinver1(auto1,auto2):
  addepsilon(auto1)
  addepsilon(auto2)
  auto, keytemp = joincreate(auto1,auto2)
  template = list()
  for i in range(len(auto.varstates)):
    auto.key[str(auto.varstates[i])] = i
    template.append('w')
  isnotlv5(auto.transition,str(auto.end))
  todo = set([auto.start])#Set to be processed
  done = set([])  #Set had been processed,
  auto.varconfig[str(auto.start)] = template
  #Find new states and add edges
  while todo:
    currentnode = todo.pop()
    auto.states.append(str(currentnode))
    done.add(currentnode)
    seen = set([])  #Prevent repeat edges
    for edge1 in auto1.transition[currentnode
    [0]]:
      for edge2 in auto2.transition[currentnode
    [1]]:
        if edge1[1] == edge2[1]:
          dest = (edge1[0],edge2[0])
          val = edge1[1]
          if len(edge1[1]) == 1:
            if re.match('\W',edge1[1]):
              val = "\\"+str(edge1[1])
              #\\ match symbol
          checklegal(auto,keytemp,auto1,auto2,
    seen,dest,currentnode,val,todo,done)
        else:
          if edge1[1] != '[epsi]' and edge2[1] !=
     '[epsi]':
            dest = (edge1[0],edge2[0])
...continue..
```

```
...continue...
            val1 = str(edge1[1])
            val2 = str(edge2[1])
          if len(edge1[1]) == 1:
            if re.match('\W',edge1[1]):
              val1 = "\\"+str(edge1[1])
          if len(edge2[1]) == 1:
            if re.match('\W',edge2[1]):
              val2 = "\\"+str(edge2[1])
          #(?= )(?= ) join two exp
          if edge1[1][0:3] == "(?=" and edge2
    [1][0:3] != "(?=":
            value = edge1[1]+'(?='+val2+')'
          elif edge1[1][0:3] != "(?=" and edge2
    [1][0:3] == "(?=":
            value = '(?='+val1+')'+edge2[1]
          elif edge1[1][0:3] == "(?=" and edge2
    [1][0:3] == "(?=":
            value = edge1[1]+edge2[1]
          else:
            value = '(?='+val1+')(?='+val2+')'
          checklegal(auto,keytemp,auto1,auto2,
    seen,dest,currentnode,value,todo,done)
    #Change state1, keeping state2
    if edge1[1] == '[epsi]':
      dest = (edge1[0],currentnode[1])
      checklegal(auto,keytemp,auto1,auto2,seen,
    dest,currentnode,edge1[1],todo,done)

  for edge3 in auto2.transition[currentnode[1]]:
    #Changing node2, keeping state1
    if edge3[1] == '[epsi]':
      dest = (currentnode[0],edge3[0])
      checklegal(auto,keytemp,auto1,auto2,seen,
    dest,currentnode,edge3[1],todo,done)
return auto
```

Figure 15: Join function in library starting from left to right

```
def checklegal(auto,keytemp,auto1,auto2,seen,dest
    ,currentnode,endvalue,todo,done):
  fail = 0
  for variable in auto.varstates:
    #Only 1 pointer, valid state no conflicts
    if keytemp[variable][0] == -1 or keytemp[
    variable][1] == -1:
      fail = 0
    elif auto1.varconfig[str(dest[0])][keytemp[
    variable][0]] != auto2.varconfig[str(dest[1])
    ][keytemp[variable][1]]:
      fail = 1  #Different varconfig
      break
  if fail == 0 and (not seen & {dest}):
    seen.add(dest)
    auto.varconfig[str(dest)] = copy.deepcopy(
    auto.varconfig[str(auto.start)])
    isnotlv5(auto.transition,str(currentnode))
...continue...
```

```
...continue...
  for variable in auto.varstates:
    tup = keytemp[variable]
    if tup[0] != -1 and tup[1] == -1:
      auto.varconfig[str(dest)][auto.key[variable
    ]] = auto1.varconfig[str(dest[0])][tup[0]]
    elif tup[1] != -1 and tup[0] == -1: #Varstate
     does not exist in auto1, use auto2 varconfig
      auto.varconfig[str(dest)][auto.key[variable
    ]] = auto2.varconfig[str(dest[1])][tup[1]]
    else:
      auto.varconfig[str(dest)][auto.key[variable
    ]] = auto1.varconfig[str(dest[0])][tup[0]]
    end = (str(dest), endvalue)
    auto.transition[str(currentnode)].append(end)
    isnotlv5(auto.transition,str(dest))
    if not done & {dest}:
        todo.add(dest)
```

Figure 16: Sub function for join starting from left to right

4.3.3.4 String Equality Selection

Let us denote the automata object $P(s)$ as a string automaton over the alphabet $(\Sigma \cup \Gamma_V \cup \{\epsilon\})$, the function $\mu(x)$ is the span of variable $x \in V$ and $s_{\mu(x)}$ to the sub-string from s obtain from the span of variable $x \in V$. The string equality function in this library takes a string s and two variables $x, y \in V$ and create $P(s)$ such that $P(s)$ contain the set of all spans $\mu \in P(s)$ such that $s_{\mu(x)} = s_{\mu(y)}$. In other words, the string equality function create a string automaton $P(s)$ with variable x and y such that every sub-string obtained from the spans of variable x and y are identical to each other. A ref-word (generated on some path from q_0 to q_f) in $P(s)$ is valid, if $s_{\mu^r(x)} = s_{\mu^r(y)}$ and if $\forall x \in \mathbb{V}(A)$, $|r|_{x\vdash} = |r|_{\dashv x} = 1$ and $x \vdash$ occurs before $\dashv x$.

The task of string equality function is to find all possible equal sub-strings from the given string and build an automata object using the intervals of these sub-strings such that these equal sub-strings of some span $\mu(x)$ and $\mu(y)$ can be find as a valid ref-word in the automata. The initial implementation of string equality function has the following process:

1. To search for equal sub-strings, we used three *for* loop to achieve this.

```
...in stringequality function
for i in range(0,len(string)+2):
  for j in range(1,len(string)+2-i):
    for k in range(1,len(string)+2-i):
      if string[j-1:j+i-1] == string[k-1:k+i-1]:
        tup = (i,j,k)
        if count == 0:
          autostring, deststring, shortcut = createauto(tup,string,['x','y'])
        else:
          autostring,deststring,shortcut = combinationauto(autostring,deststring,shortcut,tup,
    string,['x','y'])
        count += 1
...continue...
```

   First loop will iterate the size of the span from length 0 to length of string+1. Within the first loop, we have a two *for* loops which iterate for every character in the string up to len(string)+1-i for every character from position j of string. These loops will iterate through almost all the character of the string twice (by *for* loop of j and k) for length up to size of the string.

2. When the function find a equal sub-string, a tuple is created to store the value of i, j and k which contains the information for the length of the span, the beginning of boundary indices for span of variable x and the beginning of boundary indices for span of variable y, respectively. This tuple is passed to ***createauto()*** function when count is zero indicating the first match sub-string find or ***combinationauto()*** function.

3. ***createauto()*** is a function that create an automaton object using the tuple. The function first initialised the automaton object and calculate the boundary indices for both variable x and y from the tuple. Next, we perform a iteration through the string, adding edges for every characters in the string to the automaton and variable operation when the iteration reaches the boundary indices for the variables. The resulting automaton will only have one path from starting state to terminal state and will only contain a single valid ref-word which has the sub-string within all the interval of opening and closing variable operation.

4. ***combinationauto()*** is a function that combine the previous automaton object A_1 created previously by either ***combinationauto()*** or ***createauto()*** with the new tuple that contain a matching between the sub-strings. In this function, it first create a small automaton object A_2 using ***createauto()*** with the tuple. To combine A_1 with A_2, we use the renumber method for automaton object in A_2 to rename all states. Lastly, we replace the start and terminal state for A_2 with that of A_1 and update all the edges and states of A_1 with A_2. This will result in having start state of A_1 extended with edges of the start states of A_2 joining both start states as one and the edges that terminate to a terminal state in A_2 will instead terminate to the terminal state in A_1. Successfully linking both automata together.

5. When all matching of substring is completed, we will obtain an automaton object $P(s)$. Next, the automaton object will need to be converted so that all states are in the string data type using the method ***.tostr()*** (see section 4.3.1 for details). Then, to work out all $\vec{c}_q$ for all q in automaton, we will run the functionality text with the automaton object. Finally, the automaton object is processed with *Conversion to epsilons* function to finish.

After the initial version was implemented, it was quickly discovered that this implementation was very slow and inefficient for use. This was because there were many stages within the function that repeatedly iterated through the automaton object which caused the processing to slow. Therefore, we carried out the following optimisation in the sub-function of the string equality function:

1. The final stages in the function, where we work out the variable configuration for all states in $P(s)$ during the creation of the automata in ***createauto()*** function. This is possible because the function builds the automata using the boundary indices for both variable x and y from the tuple. The indices serve as an indication for variable configuration changes between the states. Therefore, while iterating through the string, we add both edges and variable configuration for each state. Performing these task during the creation will reduce the number of iteration through the automata by one.

2. The renumbering function that is used to rename all states in automaton A_2 is one of the facters for slow processing. This is because every variable instances in the automaton object has to be iterated in order to rename every states. To remove this renaming, we pass an extra variable for every ***createauto()*** function which contain the last numeric value used for labelling a state. The ***createauto()*** function will continue labelling states by incrementing this value. This value is replaced with the latest value after the combination of the two automata.

3. The main information we want from the automaton is $s_{\mu(x)} = s_{\mu(y)}$ and represent these spans within the automaton. We are only interested with the contents within the spans, it does need to care about any character that is outside of the spans since the purpose of $P(s)$ is to find matching sub-strings. Therefore, there is not need for the automaton to transverse a specific order of character for s before the opening of variables and after the closing of variable and we simply add an edges that accept all characters in a loop.

   In the ***createauto()*** function, we add an edge $q_n \in \delta(q_n, (.))$ and an edge $q_n \in \delta(q_f, (.))$ where q_f is a terminal state with edge value '(.)' which is a regular expression that accept all characters if there exists the state q_n where both variables x, y are closed and before the end of characters iteration of the string. This reduces the number of edges and states created for the string automaton.

   Only characters after both variables are closed are omitted in the function due to time constraint of the project but there is a notable different between number of states and edges compared to not omitted. Potential future optimization would be to implement the loop before the opening of variables and also case like when one variable is closed but the other variable is waiting. The current function can only take two variables, so for future improvement would be the modify the function to accept multiple variables.

With these optimisation implemented, the string equality function is functional for small strings but unable to process long strings in a reasonable amount of time. This is due to the fact that a large number of string matches could be produced from the *for* loops. Therefore, in order to make this function workable for long strings, two restrictions are implemented to the function.

The first restrictions is the size of the match strings. Initially, the function should match all length from 0 to total length of string. However, this generate a massive number of valid match strings. As the number of successful match string increases, the size of the automata object grows until it is no longer able to store in the memory. Not to mention that match strings of length zero is basically matching empty string which serve no purpose in extracting this information. By restricting the size, we restrict the number of successful matches and able to narrow down the string we wish to extract in the text. The second restrictions is to add extra conditions beside the matching of strings before returning a successful match. For example, if we want to extract ip addresses from the text, we can add a condition such that both sub-strings should not contain any alphabet. Then, even if the sub-strings matches, we can filter out any undesired results from the matching. Details on the number of state generated in the string equality function can be find in section 5.

Three modes are implemented in the function. The first mode performs string equality selection with only restricting the size of sub-string, which will reduce the number of successful matching results hence reducing the size of the automaton. The second mode can restrict the size of sub-string and allows the use of custom conditions to filter undesired matching. Also, it will skip any new line characters in the string. The third mode is the same as the second mode but only matches string on separate lines in the text file. The third can be used

to find matching between records in the text and only sub-string that occur at least twice in the text. Other modes allow matching of the same sub-string with the same boundary indices.

The final version of the string equality function is able to process texts that is not possible in the initial implementation and the number of states produced by the automata object has decreased significantly. However, while the function allows us to process small text files but processing large strings is still not feasible. These problems is shown in section 5.

Future improvements for this function are to modify the function to be able to accept any number variables $x \in V$ as the current function only limits two. To improve the speed of the function, one might consier use multi-threading to split the processing time to build the automaton for each matched results or using a different method or algorithm to find matching sub-string in the text and lastly, one might consider using different data structure for storing information about the automaton so that can retrieve and store information quicker then our implementation.

4.3.4 Creating A_g graph

Once we are satisfied with the vset-automaton, we proceed in creating the Ag graph. The process of creation is as described in section 3.2. In this section, we will show part of the code for ***def generateAg():*** and briefly explain what it does.

```
def generateAg(auto,text):
  finalgraph = {}
  for i in range(len(text)):
    finalgraph[i] = {}
  #-1 reference to q0 of the graph.
  #'0' is just a placeholder, not node '0'
  finalgraph[-1] = {'0': set([])}
  tochecklist = set([str(auto.start)])
  nxsetnodes = set([])
  seenlist = set([str(auto.start)])


..continue...
```

```
...continue...
  while tochecklist:
    item = tochecklist.pop()
    #Started with auto.start as starting node
    seenlist.add(item)
    #Iterate avaiable edges for node
    for tup in auto.transition[item]:
      if tup[1] != '[epsi]':
        if re.match(tup[1],text[0]):
          nxsetnodes.add(item)
          finalgraph[-1]['0'].add(item)
      if tup[1] == '[epsi]' and ({str(tup[0])
      } not in seenlist):
          tochecklist.add(str(tup[0]))
```

Figure 17

The code in Figure 17 on the left show the initialisation of the variables. Recall, each state in A_G has format of a tuple. *finalgraph* is a dictionary which store connection between edges in A_G graph. It has the format of $\{i : \{q_{m_k}\}\}$ where i is the position of the string, q_{n_j} are state with position i and q_{m_k} are states which q_{n_j} connect to. Then, we see that $(i, q_{n_j}$ has an edge to $(i+1, q_{m_k})$. Position -1 refer to state q_0 for A_G. *tochecklist* is the set of states to check initially for building an edge from q_0 to $(0, q_n)$ for some state q_n. *nxsetnode* is the set of states generated for some position i and the states are used to find edges for position $i + 1$. *seenlist* store seen states.

The code in Figure 17 on the right contain codes to find initial valid nodes for position 0. It used a *while* loop to continue search through the edges of some state in *tochecklist*. It search for an edge that matches the regular expression value of the edge with the first character of the string. We add the start state of the edge to finalgraph and to *nxsetnodes* if matching successful. For edges with ϵ, we add end state of edge to *tochecklist* for further searching.

After obtaining initial states of $(0, q_n)$ for some state q_n, we iterate through the character of the string/text, skipping newline, return and tab command characters (because the library did not set to recognise these values) and processing all states in *nxsetnodes* searching for available edges to connect to. Below, show the code for this process.

```
...continue
  ext2 = ['\n','\r','\t']
  for i in range(len(text)):
    nexttodo = set([])
    if not text[i] in ext2:
      while nxsetnodes:
        extratodo = set([])
        currentnode = nxsetnodes.pop()
        for edge in auto.transition[
    currentnode]:
          if edge[1] != '[epsi]':
            if re.match(edge[1],text[i]):
              if i == len(text)-1:
                if edge[0] == str(auto.end):
                  ifnotlv3(finalgraph, i,
    currentnode)
                  finalgraph[i][currentnode].
    add(edge[0])
              else:
                  ifnotlv3(finalgraph, i,
    currentnode)
                  finalgraph[i][currentnode].
    add(edge[0])
              extratodo.add(edge[0])
          elif edge[1] == '[epsi]' and auto.
    varconfig[currentnode] == auto.varconfig[
    edge[0]]:
            ifnotlv3(finalgraph, i,
    currentnode)
            foundepsilon(auto,finalgraph,
    currentnode,edge[0],text,i,extratodo)
..continue
```

```
..continue within while nxsetnodes:
while extratodo:
  extranode = extratodo.pop()
  for edge in auto.transition[extranode]:
    if edge[1] == '[epsi]' and auto.varconfig
    [extranode] != auto.varconfig[edge[0]]:
      if i == len(text)-1:
        if edge[0] == str(auto.end):
          ifnotlv3(finalgraph, i, currentnode
    )
          finalgraph[i][currentnode].add(edge
    [0])
      else:
        ifnotlv3(finalgraph, i, currentnode)

        finalgraph[i][currentnode].add(edge
    [0])
      extratodo.add(edge[0])
    elif edge[1] == '[epsi]' and auto.
    varconfig[extranode] == auto.varconfig[
    edge[0]]:
      if edge[0] == str(auto.end):
        finalgraph[i][currentnode].add(edge
    [0])
      else:
        extratodo.add(edge[0])

if currentnode in finalgraph[i]:
  nexttodo = nexttodo | finalgraph[i][
    currentnode]
#Out of while nxsetnodes:
nxsetnodes = nxsetnodes | nexttodo
...continue
```

Figure 18

The code in Figure 18 on the left is used to search for edges that matches the current character *text[i]* with the regular expression value of the edge (line 35-37). If matching is successful, we add the end state of the edge to *finalgraph* under the processing state (label *currentnode*. If we are matching the last character, we only add the end state if it is a terminal state. We add end state to *extratodo* set for all successful matching. When we encounter a empty edge with no variable configuration changes, we pass the start and end state of the edge to *foundepsilon* function which is used to continuing iterating states with empty edges to match the character.

After iterating through the edges of processing state, we process the states in *extratodo* set. This set contain end states all successful matching of processing state from before. We iterate through the edges from states in set to find empty edges where $\vec{c}_{q_n} = \vec{c}_{q_m}$ and add the end state of these empty edges to processing states. Similar to before, for matching last character, we only add end states which are terminal. For all empty edges regardless of $\vec{c}_{q_n}$, we add end state to *extratodo* set, since it is possible to transverse through multiple empty edges. Before processing the next state in *nxsetnodes*, store the set of successful states added to *finalgraph* for this state to *nexttodo* set which will be used to replace *nxsetnodes* when *nexsetnodes* is empty.

Finally, we show the pruning of A_G, only taking states that can reach the starting state and terminal states and ***def foundepsilon():*** function which was used before.

```
...continue...
#Pruning
tokeepnodes = set([str(auto.end)])
for i in range(len(text)-1,-2,-1):
  updatetokeep = set([])
  list0 = list(finalgraph[i].keys())
  for key in list0:
    list1 = list(finalgraph[i][key])
    for item in list1:
      if {str(item)} & tokeepnodes:
        updatetokeep.add(key)
      else:
        finalgraph[i][key].remove(item)
    if len(finalgraph[i][key]) == 0:
      del finalgraph[i][key]
  tokeepnodes = set([])
  tokeepnodes = tokeepnodes | updatetokeep

return finalgraph
```

```
def foundepsilon(auto,finalgraph,currentnode,
    edgenode,text,letterpos,extratodo):
  for edge in auto.transition[edgenode]:
    if edge[1] != '[epsi]':
      if re.match(edge[1],text[letterpos]):
        if letterpos == len(text)-1:
          if edge[0] == str(auto.end):
            finalgraph[letterpos][currentnode].
    add(edge[0])
        else:
          finalgraph[letterpos][currentnode].add(
    edge[0])
        extratodo.add(edge[0])
    elif edge[1] == '[epsi]' and auto.varconfig[
    edgenode] == auto.varconfig[edge[0]]:
      foundepsilon(auto,finalgraph,currentnode,
    edge[0],text,letterpos,extratodo)
    #If changes to variable configuration, ignore
```

The ***foundepsilon()*** function is same as the process for searching edges that matches the current character *text[i]* with the regular expression value of the edge. However, the difference is that when the iteration through the edges encounter a empty edge with $\vec{c}_{q_n} = \vec{c}_{q_m}$, it will call the function again replacing the *edgenode* input variable with end state of the empty edge.

The pruning of the graph is done by find all states that is reachable by terminal state. We want to keep all possible paths that are connected to the terminal state. We initialise the set *tokeepnode* which store states that has a connection to terminal state from the previous i during the iteration. Initially has the terminal state. Starting from the last position of the string, we search through all states in position i, such that for state $\{q_{n_j} : \{q_{m_k}\}\}$, state q_{n_j} contains q_{m_k} that has states in *tokeepnode* and store q_{n_j} to *updatetokeep* set. We also remove any states that is not in the *tokeepnode*. Once we prune the state for the position i, we replace *tokeepnode* with *updatetokeep* and repeat the pruning for each position. This will keep all states that has a path to terminal state and since all states where initially created from start state in A_G, all state can reach both start and terminal state and the A_G is completed.

4.3.5 Enumeration Algorithm

The enumeration algorithm was implemented exactly as it was described in section 3.5. No major changes were made to the algorithm. We enumerate through A_G finding the smallest string first, then all subsequent smallest strings by choosing the smallest letter available from the states in each stack. The important thing to note are two variables to help finding the smallest letter. One is named *availableletters* which is a dictionary for storing all possible letter available for each stack since each stack only store states for the selected minimum letter. It is used for easy retrieval of next minimum letters for each stack. Another dictionary named *letterofedges* is used to store all state reachable for some letter in each stack. Once a minletter or nextletter is selected, we can retrieve all states which that end with an edges that has this letter and store it in the stack. (see code at line 352-458 in script1.py)

4.4 Post-processing

4.4.1 Display Results

After obtaining the (V, s)-relation for the automaton, we can use the following functions to display our results. These functions are located in the scriptgrph,py file.

1. ***def printresults(listofoutputs)***: This function takes a list of outputs and simply display them on the screen. The *listofoutputs* is a list of sequences of all possible variable configurations $\vec{c}_{q_0}, ..., \vec{c}_{q_N}$, where $N = |s|$ obtained from the enumeration algorithm.
2. ***def printresultsv2(listofoutputs,auto,string,showstring=0,showconfig=1,showposstr=0,showspan=0)***: This function takes the list of outputs, automaton object and string to print the results in a table format. The

table will by default show the number of results and the variable configurations for each character of the string. With this function, it has the option to display ref-word (showstring), the sub-string between the span (showposstr) and the boundary indices of the spans (showspan). The function uses the package texttable to draw the table in python.

3. ***def printspan(outputs,auto)***: This function will print the span for all variable for an output of enumeration algorithm. It is used to print out the results as soon as it is computed for the enumeration algorithm. The user can toggle this print function on and off from the scriptlibrary.py functions.

It is also possible to display the automaton object as graph. There are three functions available to print out a graph. To draw and display the graph, we use the package graphviz in python.

1. ***def printgraph(auto,name)***: The function takes an automaton object and a name for the graph. It will draw the nodes and edges for the automaton and output the graph into a pdf file with the input name to the folder which contain the running script. It will also open a window displaying the pdf file.

2. ***def printrawgraph(graph,end,name)***: This function is similar to the previous, but it takes only a dictionary which contains the edges for the automaton along with a end node in order to draw the terminal state properly in the graph. It will create and display the pdf file of the graph.

3. ***def printgraphconfig(auto,finallist,name)***: This function takes an automata object and a list of variable configuration for each state. It will draw the automaton along with its variable configuration changes for each states in the graph. It will create and display the pdf file of the graph.

4.4.2 Other functions

There are three other functions in the framework.

1. ***def finalauto(auto,graph)***: This function is used to create transform automaton from evaluation algorithm to a proper format to display as a graph.

2. ***def concat(auto1,auto2)***: This function is used to concatenate between two automata objects. This has a similar process compare to union function and concat method in automaton object. The function convert all states in automaton to integer and we renumber auto2 to one with terminal value of auto1. Then we update all value from auto1 with auto2, join the terminal state of auto1 and start state of auto2 together and denote a new terminal state for auto1. Functionality test need to be run for the resulting automaton.

3. ***def alpha(listings,varstate)***: This function takes a list of strings and a variable x and create a vset-automaton such that the all value in the list is with in the interval of open x and close x and accept any characters outside the interval.

The code below shows all functions of the three scripts that can be call from scriptlibrary.py. Importing scriptlibrary.py in some python script will allows the user to use the library.

```
def callreadauto(fname):
  auto = sc1.readauto(fname)
  return auto

def regextoauto(reg):
  auto = sc2.main(reg)
  return auto

def readlogfile(name):
  f = open(name, 'r')
  string = f.read()
  f.close()
  return string

def initialprocess(auto):
  sc1.funchk(auto)
  sc1.csymtonulllong(auto)

def endprocess(auto,string,output=0,a=1,b=1,c=1,d
    =1,printnow=0):
  finalgraph = sc1.generateAg(auto,string)
  if not finalgraph[-1]:
    print('No results')
    sys.exit(1)
  if output == 1:
    outputgraph = sg.finalauto(auto,finalgraph)
    sg.printgraph(outputgraph,'outputgraph')
  outputs = sc1.calcresults(finalgraph, len(
    string), auto.varconfig,auto,printnow)
  sg.printresultsv2(outputs,auto,string,a,b,c,d)

def autoprocess(auto,string,output=0,printnow=0):
  sc1.funchk(auto)
  sc1.csymtonulllong(auto)
  finalgraph = sc1.generateAg(auto,string)
  if not finalgraph[-1]:
    print('No results')
    sys.exit(1)
  if output == 1:
    outputgraph = sg.finalauto(auto,finalgraph)
    sg.printgraph(outputgraph,'outputgraph')
  outputs = sc1.calcresults(finalgraph, len(
    string), auto.varconfig,auto,printnow)
  sg.printresultsv2(outputs,auto,string,1,1,1,1)

def autostringequ(auto,string,mode,start=1,end
    =-1,condits=-1,output=0,printnow=0):
  sc1.funchk(auto)
  sc1.csymtonulllong(auto)
  stri, auto2 = sc3.stringequality(string,mode,
    start,end,condits)
  auto3 = sc3.joinver1(auto,auto2)
  auto3.rename()
  print('ok')
  finalgraph = sc1.generateAg(auto3,stri)
  print('ok')
  if not finalgraph[-1]:
    print('No results')
    sys.exit(1)
  if output == 1:
    outputgraph = sg.finalauto(auto,finalgraph)
    sg.printgraph(outputgraph,'outputgraph')
  outputs = sc1.calcresults(finalgraph, len(
    string), auto3.varconfig,auto,printnow)
  sg.printresultsv2(outputs,auto,string,1,1,1,1)
```

```
def initauto(a,b,c):
  auto = sc2.automata(a,b,c)
  return auto

def callfunck(auto):
  sc1.funchk(auto)

def callcepsilon(auto):
  sc1.csymtonulllong(auto)

def callprojection(automata,listofprojections,
    before=0):
  auto = sc3.projection(automata,
    listofprojections,before)
  return auto

def calljoin(auto1,auto2):
  auto = sc3.joinver1(auto1,auto2)
  return auto

def callrename(auto):
  auto.rename()

def callgenAg(auto,string):
  finalgraph = sc1.generateAg(auto,string)
  return finalgraph

def callfinalauto(auto,finalgraph):
  outputgraph = sg.finalauto(auto,finalgraph)
  return outputgraph

def callcalcresults(finalgraph, length, varconfig
    ):
  outputs = sc1.calcresults(finalgraph, length,
    varconfig)
  return outputs

def calstringeq(string,mode,start=1,end=-1,
    condits=-1):
  stri, auto = sc3.stringequality(string,mode,
    start,end,condits)
  return stri, auto

def callunion(auto1,auto2):
  sc3.union(auto1,auto2)

def callconcat(auto1,auto2):
  sc3.concat(auto1,auto2)

def callalpha(listings,varstates):
  auto = sc3.alpha(listings,varstates)
  return auto

def callprintgraph(auto,name):
  sg.printgraph(auto,name)

def callprintrawgraph(graph,end,name):
  sg.printrawgraph(graph,end,name)

def callprintgraphconfig(auto,finallist,name):
  sg.printgraphconfig(auto,finallist,name)

def callprintresultsv2(outputs,auto,string,a=0,b
    =0,c=1,d=0):
  sg.printresultsv2(outputs,auto,string,a,b,c,d)
```

5 Results and Testings

In this section, we will demonstrate that library and the evaluation algorithm is implemented correctly by query a text with document spanners with the use of library functions. Next, we will investigate the performance of the algorithm to see if it is feasible for practical uses. We also investigate how length of text affect string equality function. Lastly, we will conclude if the algorithm is feasible for practical uses.

5.1 Testing the library

To use the library, we import scriptlibrary.py in python script. In this section, we show three examples. The first examples will demonstrate the process of query a text with regular spanners using functions in the library. The second examples will demonstrate the process of query a text with core spanners. Lastly, the third example will go through the process of using the library to query log files. These tests are to show that our library is working.

5.1.1 Testing Algorithm without String Equality Selection

In this section, we will use a generic example to show the process of query a string using document spanners. First to create a spanner using the library, we can either insert a pre-made text file where all the edges for the automaton are written (see section 4.2.3 for format), write the automaton directly in the main script or write a regex formula and convert it to an automaton.
Let denote a vset-automaton A_1 to be Figure 19:

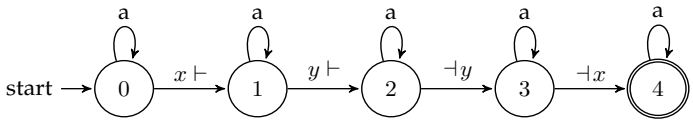


Figure 19

The Figure 20 show codes of which A_1 could be created in the script.

```
import scriptgrph as sp
import time, sys, copy

#Read File
automata1 = sp.callreadauto('test.txt')
#Or Read Regex
regex = 'a*,<x:a*,<y:a*>,a*>,a*'
automata1 = sp.regextoauto(regex)
#Or insert manally
automata1 = sp.initauto(0,0,0)
automata1.reset()
automata1.states = ['0','1','2','3','4']
automata1.varstates = ['x','y']
automata1.transition['0'] = [('0','a'),('1','x+')]
automata1.transition['1'] = [('1','a'),('2','y+')]
automata1.transition['2'] = [('2','a'),('3','y-')]
automata1.transition['3'] = [('3','a'),('4','x-')]
automata1.transition['4'] = [('4','a')]
automata1.start = '0'
automata1.end = '4'
```

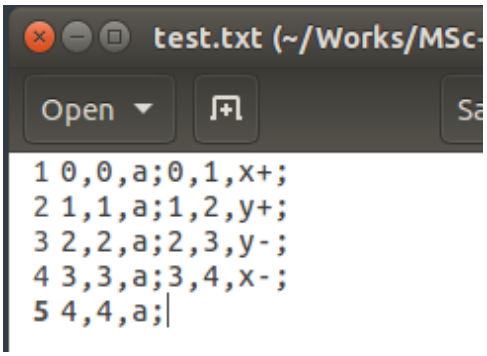


Figure 20

Next, we call ***initialprocess()*** function in the library, which will run ***funchk()*** and ***csymtonullong()*** which will check the functionality of the automaton, generate $\vec{c}_q$ for all states and convert variable operation to '[epsi]' (stand for ϵ in library). We can see from the A_1 in Figure 12 that the automaton is functional as there are not multiple opening or closing of a variable $x \in V$ and the variable configuration for each state is

$$\vec{c}_0 = [w, w] \qquad \vec{c}_1 = [o, w] \qquad \vec{c}_2 = [o, o] \qquad \vec{c}_3 = [o, c] \qquad \vec{c}_4 = [c, c]$$

The Figure 21 shows the data of the automaton after the function.

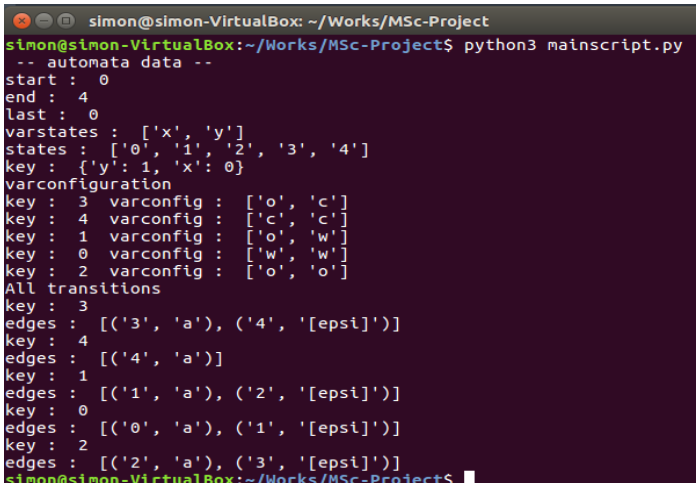


Figure 21

We can see that the variable configuration we obtain from the script is correct.
In the case where automaton is not functional, it will return one of the four error show in the Figure 22.

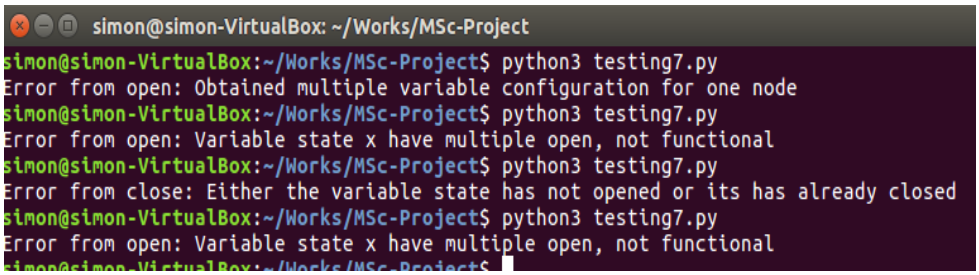


Figure 22

After the library checked the automaton is functional, this automaton can now be used for any of the spanner algebraic operation, union, projection, join and string-equality. However, in this section we will continue on with the next part of the process, which is evaluating the automaton (regular spanner in this case since we did not perform string equality). The function ***endprocess()*** will run our polynomial evaluation algorithm and extract all the relations over the spans of string. To run the function ***endprocess()*** inputting the automaton, a string and value indication whether to display the A_G. (see section 4.4.2 for the function code). Let the string be aa. We insert the automaton and the string into the function. Figure 23 shows our output of the spans after a successful evaluation of the spanner.

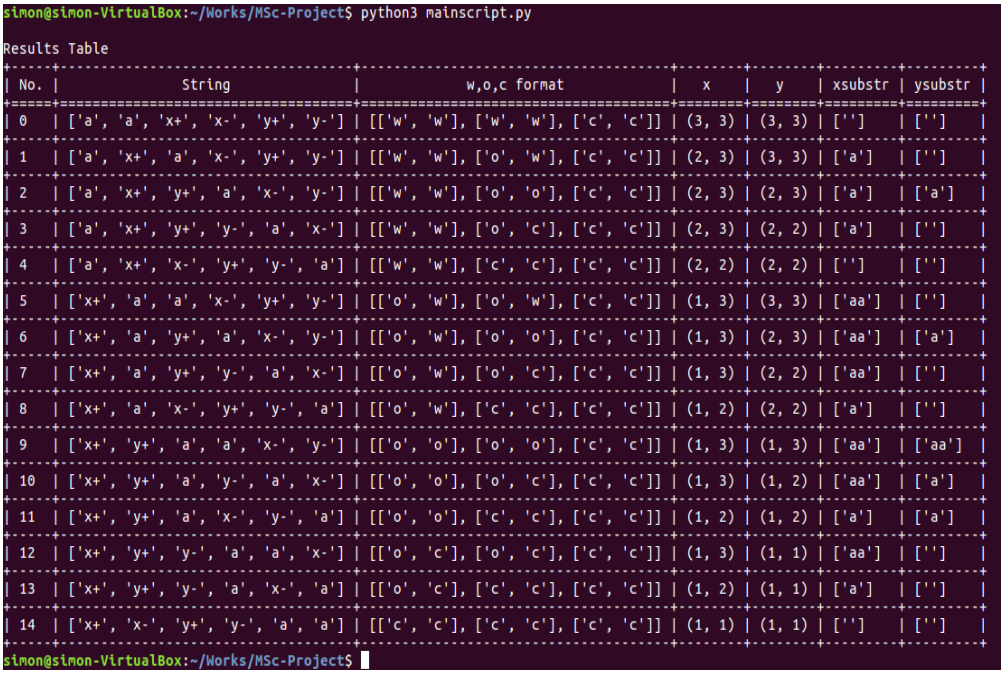


Figure 23

Looking thorough each record in the table, we see that the variable configuration follows the order of variable configuration from state 0 to terminal state. There does not exists a case where variable y is open while x is waiting or y is open but x is closed. Please note the the column showing the ref-words of each record is not entirely correct because of the way it is printed but the variable configuration format and the span of x and y are absolutely correct.

5.1.2 Testing Algorithm with String Equality

This section use the example in the previous section and continue from where we know that A_1 is functional and can be used for algebraic operations. Now we want to test and show some evidence that the string equality functional actually find all substrings that are identical. The quickest way of proving the string equality functional works is by either looking through the automaton $P(s)$ that was produced for the string and check by eye that every path from start state to terminal state contain spans of x and y that has the same substring. Another way would be to perform an algebraic join operation with A_1, evaluate the joined automaton and check if all relations of spans of x and y spans the same substring. We will show both of these evidences below. First to perform a string equality, we call the function ***calstringeq()*** which takes a string, list of variables, a indicator of which mode to use and other optional variables (see section for the function, see section for detail of string equality implementation) Figure 24 show automaton $P(s)$.

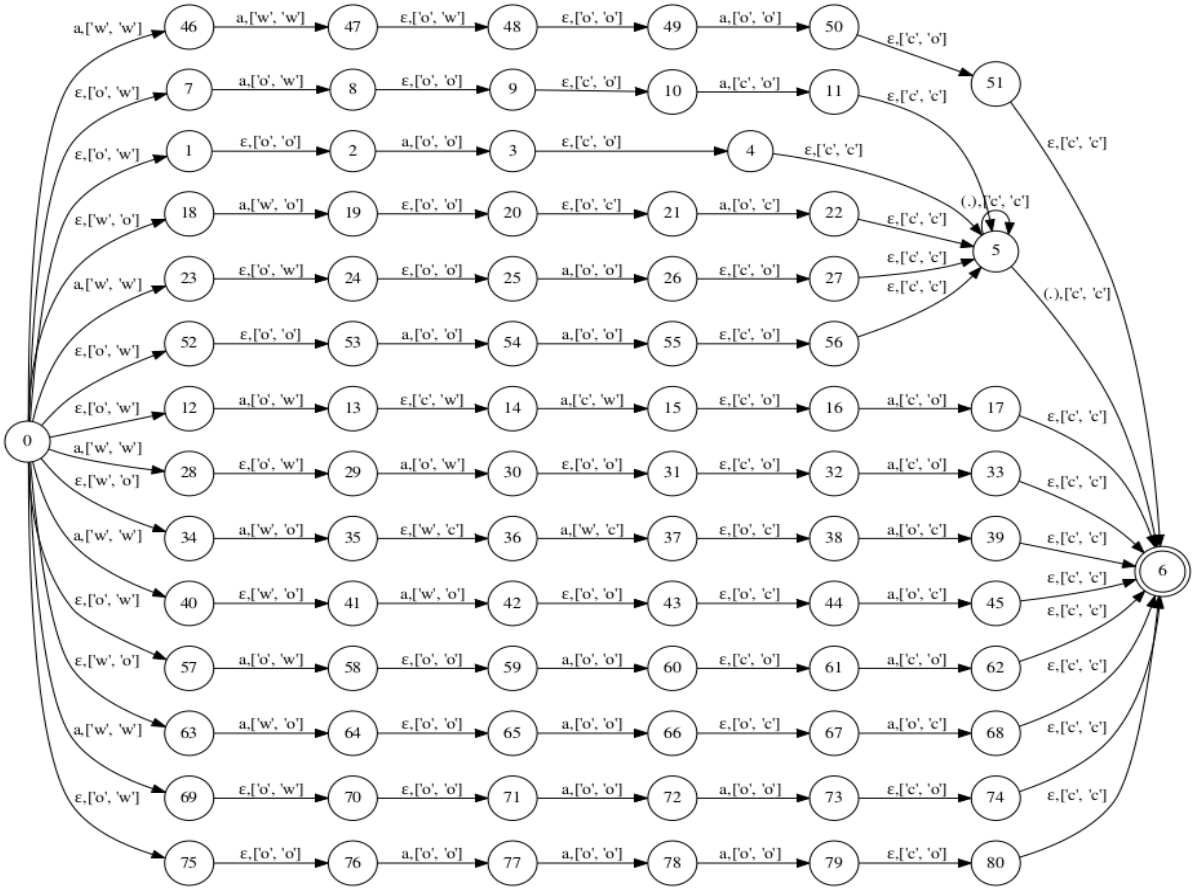


Figure 24

Next, we call the function ***calljoin()*** from the library for $A_1 \bowtie P(s)$. Finally, we run the function ***endprocess()*** to get the relations of spans for $A_1 \bowtie P(s)$ which is shown below.

```
simon@simon-VirtualBox: ~/Works/MSc-Project
simon@simon-VirtualBox:~/Works/MSc-Project$ python3 mainscript.py
count: 14
autolast 80

Results Table
+-----+-----------------------------------------+-----------------------------------------------+--------+--------+-----------+
| No. |                String                   |                 w,o,c format                  |   x    |   y    | substring |
+=====+=========================================+===============================================+========+========+===========+
| 0   | ['a', 'a', 'x+', 'y+', 'a', 'x-', 'y-'] | [['w', 'w'], ['w', 'w'], ['o', 'o'], ['c', 'c']] | (3, 4) | (3, 4) | ['a']     |
+-----+-----------------------------------------+-----------------------------------------------+--------+--------+-----------+
| 1   | ['a', 'x+', 'y+', 'a', 'a', 'x-', 'y-'] | [['w', 'w'], ['o', 'o'], ['o', 'o'], ['c', 'c']] | (2, 4) | (2, 4) | ['aa']    |
+-----+-----------------------------------------+-----------------------------------------------+--------+--------+-----------+
| 2   | ['a', 'x+', 'y+', 'a', 'x-', 'y-', 'a'] | [['w', 'w'], ['o', 'o'], ['c', 'c'], ['c', 'c']] | (2, 3) | (2, 3) | ['a']     |
+-----+-----------------------------------------+-----------------------------------------------+--------+--------+-----------+
| 3   | ['x+', 'y+', 'a', 'a', 'a', 'x-', 'y-'] | [['o', 'o'], ['o', 'o'], ['o', 'o'], ['c', 'c']] | (1, 4) | (1, 4) | ['aaa']   |
+-----+-----------------------------------------+-----------------------------------------------+--------+--------+-----------+
| 4   | ['x+', 'y+', 'a', 'a', 'x-', 'y-', 'a'] | [['o', 'o'], ['o', 'o'], ['c', 'c'], ['c', 'c']] | (1, 3) | (1, 3) | ['aa']    |
+-----+-----------------------------------------+-----------------------------------------------+--------+--------+-----------+
| 5   | ['x+', 'y+', 'a', 'x-', 'y-', 'a', 'a'] | [['o', 'o'], ['c', 'c'], ['c', 'c'], ['c', 'c']] | (1, 2) | (1, 2) | ['a']     |
+-----+-----------------------------------------+-----------------------------------------------+--------+--------+-----------+
simon@simon-VirtualBox:~/Works/MSc-Project$
```

Figure 25

As you can see from the Figure 25, the results are correct since all spans produce for each record has the same substring. Looking closely at the results, we see the variable configuration for x and y of each state is the same which proved that the automaton has successfully join with the string automaton, since the variable operation for y is y opened after x and closes before x. The boundary indices for y can never exceed x. Therefore, in order to have equal substring for x and y, they both must have the same variable configuration for letter in the string. We have proved the results are correct.

5.1.3 Example with Log Files

In this section, we will demonstrate how the user can use the library extracting information from a log file. Basically performing log file analysis. In this example, we have a log file of 1195 characters and we wish to find all IP addresses in the log file. Below we show a screen-shot of the log file.

```
access_log2                                                    access_log
 1 212.92.37.62 - - [08/Mar/2004:08:26:41 -0800] "GET / HTTP/1.1" 200 3169
 2 64.242.88.10 - - [07/Mar/2004:16:11:58 -0800] "GET /twiki/bin/view/TWiki/WikiSyntax
   HTTP/1.1" 200 7352
 3 10.0.0.153 - - [08/Mar/2004:09:02:32 -0800] "GET /cgi-bin/mailgraph.cgi/
   mailgraph_1_err.png HTTP/1.1" 200 7182
 4 212.92.37.62 - - [08/Mar/2004:08:27:23 -0800] "GET /twiki/bin/view/Main/
   SpamAssassinAndPostFix HTTP/1.1" 200 4034
 5 10.0.0.153 - - [08/Mar/2004:09:02:32 -0800] "GET /cgi-bin/mailgraph.cgi/
   mailgraph_2_err.png HTTP/1.1" 200 6805
 6 64.242.88.10 - - [07/Mar/2004:16:20:55 -0800] "GET /twiki/bin/view/Main/DCCAndPostFix
   HTTP/1.1" 200 5253
 7 195.246.13.119 - - [09/Mar/2004:01:51:17 -0800] "GET /twiki/bin/view/Main/DCCAndPostFix
   HTTP/1.1" 200 5253
 8 195.246.13.119 - - [09/Mar/2004:01:51:41 -0800] "GET /twiki/bin/edit/Main/
   RazorAndPostFix?topicparent=Main.WebHome HTTP/1.1" 401 12851
 9 195.246.13.119 - - [09/Mar/2004:01:51:45 -0800] "GET /twiki/pub/TWiki/TWikiDocGraphics/
   help.gif HTTP/1.1" 200 130
10 207.195.59.160 - - [09/Mar/2004:08:08:54 -0800] "GET /twiki/pub/TWiki/TWikiLogos/
   twikiRobot46x50.gif HTTP/1.1" 304 -
11 207.195.59.160 - - [09/Mar/2004:08:08:57 -0800] "GET /twiki/bin/view/Main/
   SpamAssassinTaggingOnly HTTP/1.1" 200 5691
```

Figure 26

The IP addresses to be find are 212.92.37.62, 64.242.88.10, 10.0.0.153, 195.246.13.119 and 207.195.59.160. To find IP addresses we need to define a regex formula which variable x where $x \in V$ such that the regex formula extracts spans x that represent IP addresses. Then, this regex formula is then converted into a vset-automaton A using the library functions.

```
regex = '(.)*,<x:[0-9],[0-9]*,.,[0-9],[0-9]*,.,[0-9],[0-9]*,.,[0-9],[0-9]*>,(.)*'
automata1 = sp.regextoauto(regex)
```

Figure 27

After conversion, we run the functionality test on the vset-automaton to check if the automaton is functional. If it is functional, it will return the automaton after working out all variable configuration for all states in the automaton. We also convert all variable operations of the automaton to ϵ. We use the function *initialprocess* to perform this. Now, we use string equality function with the log file to generate a string automata $P(s)$. We want this string automata because it extract all intervals in the string that have the same substring, these substring includes IP addresses as well but along with all valid matching substring in the log file. We limited the size of the substring from 10 to 12 and add in a extra conditions that only accepts numbers and dots for a successful match.

```
condits = [(lambda s,i,j: re.match(r'^\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}$',s[j-1:j+i-1]))]
string, automata = sp.calstringeq(string,1,10,13,condits)
```

Figure 28

Next, we use the natural join function to join A and $P(s)$ to get automaton $A(s)$. We join the automata together in order to find the IP addresses that is hidden in $P(s)$. Lastly, we use the evaluation algorithm and enumeration algorithm to obtain our relations of spans of IP addresses. Below we show a screenshot for part of results, since the output was too large.

simon@simon-VirtualBox: ~/Final/MSc-Project/FinalVersion

No.	x	y	substring
0	(1082, 1094)	(1082, 1094)	['7.195.59.160']
1	(1082, 1093)	(1082, 1093)	['7.195.59.16']
2	(1082, 1092)	(1082, 1092)	['7.195.59.1']
3	(1081, 1093)	(1081, 1093)	['07.195.59.16']
4	(1081, 1092)	(1081, 1092)	['07.195.59.1']
5	(1080, 1092)	(1080, 1092)	['207.195.59.1']
6	(1082, 1094)	(966, 978)	['7.195.59.160']
7	(1082, 1093)	(966, 977)	['7.195.59.16']
8	(1082, 1092)	(966, 976)	['7.195.59.1']
9	(966, 978)	(1082, 1094)	['7.195.59.160']
10	(966, 977)	(1082, 1093)	['7.195.59.16']

Figure 29

As you can see from Figure 29, the output contains parts of the IP addresses. This is because our automaton was set to accepting one number and then countless many numbers. However, the important thing to note is that we have extracted IP addresses from the log file and for future improvement, we could set analysing the results obtain from the string.

5.2 Performance Tests

In this section, we investigate the performance of the algorithm and the string equality function.

5.2.1 Size of string with time taken for algorithm results

In this section, we investigate the size of the string with the number of outputs from the algorithm and measure the time taken for the algorithm to complete. The string used is the alphabet a. We increase the size of string by incrementing the letter a to string. The automaton A_1 and A_2 used for to obtain the set of results is:

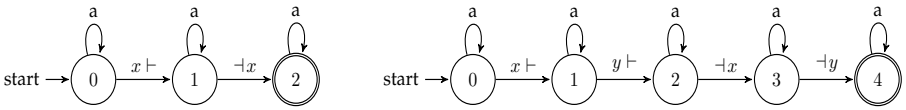


Figure 30: A_1 is on the left and A_2 is on the right

This automaton should return all possible combination the interval could take with the string. Table 1 show the results obtained with A_1 and A_2 and the string a.

Table 1: Time taken for evaluation algorithm to generate all spans with various size of string n.

n	1	2	3	5	10	20	50
Total Outputs for A_1	3	6	10	21	60	231	1326
Total Outputs for A_2	5	15	35	126	1001	10626	316251
Time Taken for A_1 (sec)	0.0006	0.0008	0.001	0.001	0.02	0.010	0.136
Time Taken for A_2 (sec)	0.0006	0.001	0.001	0.003	0.021	0.301	25.633
n	100	150	200	300	400	500	
Total Outputs for A_1	5151	11476	31626	45451	80601	125751	
Total Outputs for A_2	-	-	-	-	-	-	
Time Taken (sec) for A_1	1.016	3.637	20301	28.043	85.331	221.686	
Time Taken (sec) for A_2	-	-	-	-	-	-	

As you can see from Table 1, processing 500 characters with a single variable x iterating through the string will take roughly 3.5 minutes with total spans of 80601. However, for 100 character with two variables that accepts all possible combination from the string, it was too much for the program to handle. What we can obtain from these results is for long strings if number of resulting spans is large, the time taken to evaluate will increase. Please note that these automata will produce every possible pattern of spans available for the string. This table can be consider to be the worst case scenario for automaton with one variable x and for automaton with two variables x, y.

5.2.2 String Equality Investigation

The string equality function in the library find two intervals that has identical substring for a given string. For each successful matching, the substring create an automaton from the spans of the matching and the number of state created is $|s|$ + number of variable + 1, and joining two automata together for different matching, the total number of state is increase by $|s|$ + number of variable - 1, since start and terminal state is the same. The calculation for the total number of states in the string automaton is (number of successful matches)*($|s|$ + number of variable - 1) + 2, assuming the function is not optimised in the library.

For this string equality function, we want to compare three situations. The first situation is when the size of the substring is not limited when running the function in order to demonstrate what will happen when performing string equality selection with no restriction. The second situation is when we restrict the size of the substring. The third situation is when the function receive external conditions from the user. These conditions will be performed after finding a successful matching in other to reduce necessary results.

Given a part of a log file (with 435 characters in total). We wish to find all IP addresses the occurs in the file. To do this, we need to run the string equality function in order to get the spans for all occurrences of IP addresses.

Under the first situation, we just need to insert the string and with variable $\{x, y\}$. In the second situation, we limit the length of the substring to 7-15 character, which is the length of all IP addresses. In the third situation, we restrict the length and also add the conditions such that the function only matching substring of IP addresses.

Table 2 below shows the test results for these situations. The function is ran under mode 1, skipping newlines.

Table 2: Time taken for evaluation algorithm to generate all spans with various size of string n.

n	Without Restriction	Of length 7-15	Only IP addresses
Total matched substring	-	5 229	6
Total states created	-	1 322 049	1 236
Time Taken for function (sec)	-	26.335	0.916
Time Taken for Join (sec)	-	61.608	0.086
Time Taken for A_G (sec)	-	133.281	0.029
Time Taken for Results (sec)	-	0.123	0.046
Total Time Taken (sec)	-	195.112	1.077

We can see from Table 2, when the string equality is not restricted, the program will not compute. With length restriction, we have reduced the number of matches enough for to be able to run the algorithm. With extra conditions added, we have further reduced the number of match states hence reduce the time taken to finish.

We can conclude that without the length restriction, the string equality function will not work at all. Therefore, it is necessary to restrict the size of the substring for the matching in order for the function to work. Adding extra conditions to further restrict the result is welcomed. However currently, these conditions has to be inputed from the user. For future improvement, we would like the library to automatically extract conditions from the vset-automaton to use for the string equality function. Also from Table 2, we see that the time taken join operation to be completed was quite long. This is due to the number of states we have in the string automaton which the join operation need to search and process. For future improvement, we would like to investigate this further and we could implement an efficient algorithm for searching through automata in order to reduce the running time.

5.3 Results and Testings Analysis

Judging from the information gathered in the section 5, we see that the implemented evaluation algorithm is successfully implemented and produced correct results from our spanners. However, when size of string and number of spans are large, the evaluation algorithm will process very slowly which is evidenced by Table 1. However, from Table 2, we see that the size of the automaton greatly effect the evaluation algorithm looking at column two of the table with the results from length 7-15. If we limit the number of substrings matches for log file analysis to reduce the number of states in the string automaton, then it is possible to iterate through large amount of text very quickly which was evidence by the example in section 5.1.3 and Table 2. This implies that the number of automaton states has the most effect on the speed of the evaluation algorithm then an the number of output produced.

From this, we can conclude that the evaluation algorithm and specifically the implementation is not yet feasible for practical application. However, now that we see the problem lies on the size of the input of the spanner for the evaluation algorithm, it is possible to come up various method of improvements such as reducing the size of the input by removing empty word transitions or coming up with a way to divide processing time with multi-threadings for evaluation algorithm or coming up with a different method for implementing the evaluation algorithm. Since the implemented evaluation algorithm does not produce any incorrect result, it serves as evidence that the evaluation algorithm is correct and works for document spanners. In conclusion, the evaluation algorithm and the library is great for specific extraction with many limitations but slow when doing a broad search the require to process large spanner. The evaluation algorithm may have polynomial delay in theory but in practice it will takes some time to process if it is not implemented efficiently. Given more time on researching and developing difference method of implementation, we see a possible future of a practical program using this evaluation algorithm for log file analysis.

6 Conclusion

In conclusion, in terms of the main objective of this project, we have achieved our goal of creating the first implementation of the polynomial delay evaluation algorithm to determine whether the algorithm is implementable and have produced correct results from using document spanners to query texts. We discussed whether the algorithm is feasible to use for practical applications and concluded that while our current implementation may not be feasible for practical applications, the fault does not entirely lies on the algorithm but with the efficient of the implementation of document spanners as well. Therefore, given more time and research into investigate efficient implementation for document spanners that use the evaluation algorithm, there is high potential that to use the evaluation algorithm for practical applications.

In the process of achieving these objectives, we have encounter many problem and issues during the implementation and we find that the implementation was more difficult and time consuming then expected. This mainly due to the fact there is huge difference between defining the theory of a functionality and implementing the theory for practical purposes. For instance, string equality selection and join operations were defined on a few sentences but in the implementation, different elements are needed to be considered in order to make it work with spanners. These difficulties are all highlighted in section 4.

The future directions one can take from the project would be to reimplement the document spanner library in an another programming language other then python for better memory management, investigating efficient data structure to store the automaton and efficient methods to enumerate the automaton, continuing building up the library by further optimising the functions through use of multi-threading dividing the processing time and considering adding extensions to process the results obtained from the spanners.

In the end, we were able to create a prototype that can be worked toward to use of document spanner with the polynomial delay evaluation algorithm for log file analysis. We conclude that this project was a successful with potential improvements. The ultimate goal of bring the expressive power of relational queries to searching all kinds of texts using document spanners is still a distance goal. However, the first step have been made. With more time and research, it is possible to make use this algorithm for practical purposes. We conclude that the first prototype for using the polynomial delay evaluation algorithm has been completed.

References

[1] Laura Chiticariu, Rajasekar Krishnamurthy, Yunyao Li, Sriram Raghavan, Frederick R Reiss, and Shivakumar Vaithyanathan. Systemt: an algebraic approach to declarative information extraction. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics*, pages 128–137. Association for Computational Linguistics, 2010.

[2] Ronald Fagin, Benny Kimelfeld, Frederick Reiss, and Stijn Vansummeren. Spanners: a formal framework for information extraction. In *Proceedings of the 32nd ACM SIGMOD-SIGACT-SIGAI symposium on Principles of database systems*, pages 37–48. ACM, 2013.

[3] Bryan Ford. Parsing expression grammars: a recognition-based syntactic foundation. In *ACM SIGPLAN Notices*, volume 39, pages 111–122. ACM, 2004.

[4] Dominik D. Freydenberger. A logic for document spanners. Schloss Dagstuhl–Leibniz Center for Informatics, 2017.

[5] Dominik D. Freydenberger and Mario Holldack. Document spanners: From expressive power to decision problems. *Theory of Computing Systems*, pages 1–45, 2017.

[6] Dominik D. Freydenberger, Benny Kimelfeld, and Liat Peterfreund. Joining extractions of regular expressions. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 137–149. ACM, 2018.

[7] Ralph Grishman and Beth Sundheim. Message understanding conference-6: A brief history. In *COLING 1996 Volume 1: The 16th International Conference on Computational Linguistics*, volume 1, 1996.

[8] Dejanović I., Milosavljević G., and Vaderna R. Arpeggio: A flexible peg parser for python. Internet: http://www.igordejanovic.net/Arpeggio/, [accessed 06-September-2018].

[9] Benny Kimelfeld. Database principles in information extraction. In *Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 156–163. ACM, 2014.

[10] Francisco Maturana, Cristian Riveros, and Domagoj Vrgoc. Document spanners for extracting incomplete information: Expressiveness and complexity. *CoRR*, abs/1707.00827, 2017.

[11] Sheng Yu. Handbook of formal languages vol 1, chapter 2. pages 71–74.