# Towards Practically Feasible Answering of Regular Path Queries in LAV Data Integration

by

**Manuel Tamashiro**
BSc, University of Victoria, 2005

A Dissertation Submitted in Partial Fulfillment of the
Requirements for the Degree of

**Master of Science**

in the Department of Computer Science

© Manuel Tamashiro, 2007

University of Victoria

*All rights reserved. This dissertation may not be reproduced in whole or in part by photocopy or other means, without the permission of the author.*

**Towards Practically Feasible Answering of Regular Path Queries in LAV Data Integration**

by

**Manuel Tamashiro**
BSc, University of Victoria, 2005

## Supervisory Committee

---

Dr. A. Thomo, Co-Supervisor (Department of Computer Science)

---

Dr. V. Srinivasan, Co-Supervisor (Department of Computer Science)

---

Dr. U. Stege, Member (Department of Computer Science)

---

Dr. L. Cai, Outside Member (Department of Electrical and Computer Engineering)

**Supervisory Committee**

Dr. A. Thomo, Co-Supervisor (Department of Computer Science)

Dr. V. Srinivasan, Co-Supervisor (Department of Computer Science)

Dr. U. Stege, Member (Department of Computer Science)

Dr. L. Cai, Outside Member (Department of Electrical and Computer Engineering)

# Abstract

Regular path queries (RPQ's) are given by means of regular expressions and ask for matching patterns on labeled graphs. RPQ's have recently received great attention in the context of semistructured data, which are data whose structure is irregular, partially known, or subject to frequent changes. One of the most important problems in databases today is the integration of semistructured data from multiple sources modeled as views. In this setting, the database is not available, and given a user query, the system has to answer based solely on the information provided by the views. The problem is computationally hard, and the well-known algorithm for solving it runs in 2EXPTIME. In this paper, we provide practical evidence that this algorithm performs poorly on the average as well. Then, we propose automata-theoretic techniques which make the view-based answering of RPQ's more feasible in practice.

# Table of Contents

# List of Tables

# List of Figures

## Acknowledgements

All my gratitude to my parents, my sisters, and my *ojichan*; whose effort, love and support made me accomplish this goal. I am also thankful to Alex and Venkatesh, who guided me with patience throughout this research.

# Chapter 1

# Introduction

## 1.1 Regular Path Queries and LAV Data Integration

Regular path queries (RPQ's) are in essence regular expressions over a fixed database alphabet. They have received a great deal of attention in the recent years due to the well-known semistructured data model. Semistructured data is data whose structure is irregular, partially known, or subject to frequent changes (1). They are commonly found in a multitude of applications in areas such as communication and traffic networks, web information systems, digital libraries, biological data management, etc. Semistructured data are formalized as edge labeled graphs and the basic querying mechanism over such graphs is the one that finds all the pairs of nodes connected by a path spelling a word in a given RPQ (cf. (12, 1, 4, 3, 5, 7, 8)). For example, the RPQ

$$Q = AirCanada^* \cdot (Lufthansa + \epsilon)$$

asks for all the pairs of cities connected by (possibly multihop) Air Canada routes, followed by a last optional segment serviced by the partner company Lufthansa. We can observe that evaluating RPQ's on semistructured databases amounts to [regular expression] pattern matching on graphs as opposed to strings.

Now, suppose that we do not have a database available. Rather, what we have is a set of views on the possible data. These views represent partial information about the database and are expressed by regular expressions as well. For example, we could be given two views with definitions $V_1 = AirCanada \cdot AirCanada$ and $V_2 = Lufthansa$. Notably, the view definitions are nothing else but regular path queries. Additionally, for each view, we are given a set of pairs that represent the answer to these views (considering them as RPQ's).

This is the classical scenario in LAV ("local-as-view") data integration (cf. (6, 4, 3, 9, 5, 2, 8)). The basic problem in this setting is to be able to answer a given query using only the available view information. This is a very important problem which emerges in a variety of situations both commercial (when two similar companies provide partial access to their data) and scientific (combining research results from different bioinformatics repositories). Data integration appears with increasing frequency as the volume and the need to share existing data explodes.

## 1.2 View Based Rewriting

Answering queries using views is typically achieved by reformulating the query in terms of the view definitions and then evaluating it on the provided view data. For example, the above query $Q$ can be reformulated (or rewritten) as $Q' = V_1^* \cdot (V_2 + \epsilon)$.

Then, if there are pairs $(a, b)$ and $(b, c)$ associated with views $V_1$ and $V_2$ respectively, we produce $(a, c)$ (among other pairs) as an answer to $Q$. Of course, if we have a database in the classical sense, then we would be able to also produce pairs of nodes connected by paths with an *odd* number of Air Canada segments followed by an optional Lufthansa segment. However, for the given views this is not possible.

The most important cornerstone in the rewriting of RPQ's using views is the work by Calvanese, De Giacomo, Lenzerini, and Vardi (3), which shows that the rewriting is indeed possible by giving an algorithm for computing it. The complexity of computing the (maximal) view-based rewriting of a regular path query $Q$ is shown to be in 2EXPTIME (see (14) for the definition of this class) and this bound is also shown to be tight ((3)). Also, in ((3)) it is shown that the size of the automaton for $Q'$ can be doubly exponential in the size of the query $Q$ as measured by the size of a simple NFA for $Q$.

It should be clear what the inherent problem complexity of $2^{2^n}$ (tight) faces us with in practice. If $n$, the query size, is just 6 for example, then only printing a doubly exponential rewriting would need about $2^{2^6} \approx 18 \cdot 10^{18}$ instructions that is $18 \cdot 10^{18}/(30,000 \cdot 10^6 \cdot 60 \cdot 60 \cdot 24 \cdot 365) \approx 19$ years for a modern Intel processor working at about $30,000$ millions of instructions per second.

This illustrates that obtaining a view-based rewriting is computationally hard except for very small query instances. However, it is possible to argue that the analysis in (3) is worst-case and hence it might take only reasonable amount of time to compute rewritings on the average. Unfortunately, our experimental results

indicate that this is not the case (see Section 6). Experimentally, we were unable to compute[1] the view-based rewriting, in reasonable time and space, for about one third of the time while working on "randomly generated" instances. This gives us evidence that computing rewritings is indeed hard on the average as well. We believe that this observation is an important contribution of our paper given the importance of the database problem being studied.

In order to make feasible the answering of RPQ's using views, we examine each step in the algorithm of (3). Then, we show that we can in fact avoid the most expensive step in the algorithm by evaluating instead the complement of the rewriting on the view data. The complement is in the form of an NFA as opposed to a DFA for the rewriting (if the latter is fully computed). This might suggest that the evaluation on the view data would be slower compared to the evaluation of the DFA for the rewriting. Of course, this is relevant only for the cases when the rewriting can be computed in reasonable time and space. Interestingly, we show that even in such cases, by using a bitvector implementation of NFA's, reminiscent of the implementation of r-AFA's in (13), we can achieve similar performance and sometimes even better. This is attributed to hardware parallelism and better cache utilization.

Surprisingly, we also found that a seemingly inexpensive polynomial step in the algorithm of (3) was a serious performance bottleneck. In order to overcome it, we show a simple optimization which gives more than six fold speedup.

In short, we show that by employing our simple techniques, the hard problem of answering regular path queries using views becomes practically more feasible. This

[1]Using the method of (3)

paper, although simple, is the first to shed light on the practical feasibility of the very basic and important problem of answering RPQ's in LAV data integration systems and to provide positive results in this direction.

The rest of the thesis is organized as follows. In Chapter 2, we formally define semistructured databases, regular path queries, and their semantics. In Chapter 3, we discuss the query answering in LAV information integration systems. In Chapter 4, we examine the algorithm of (3) for obtaining maximal view-based rewritings. Then, in Chapter 5 we present our optimization techniques. We show our experimental evaluations in Chapter 6. Finally, Chapter 7 concludes the thesis. There is an additional Chapter 8 containing the source code for the implementation of the experiments.

# Chapter 2

# Semistructured Databases and Regular Path Queries

## 2.1 Database Model

We consider a database to be an edge labeled graph. This graph model is typical in semistructured data, where the nodes of the database graph represent the objects and the edges represent the attributes of the objects, or relationships between the objects.

Formally, let $\Delta$ be a finite alphabet. We shall call $\Delta$ the *database alphabet.* Elements of $\Delta$ will be denoted $R, S, \ldots$. As usual, $\Delta^*$ denotes the set of all finite words over $\Delta$. Words will be denoted by $u, w, \ldots$. We also assume that we have a universe of objects, and objects will be denoted $a, b, c, \ldots$. A *database DB* over $\Delta$ is a subset of $N \times \Delta \times N$, where $N$ is a finite set of objects, that we usually will call nodes. We view a database as a directed labeled graph, and interpret a triple $(a, R, b)$ as a directed edge from object $a$ to object $b$, labeled with $R$. If there is a

path labeled $R_1, R_2, \ldots, R_k$ from $a$ to $b$, we write $a \xrightarrow{R_1 R_2 \ldots R_k} b$.

**Example 1.** We show in Fig. 2.1 a database with information about an online store, which sells books and software products. A book has an author and covers some software product(s). A software product has a company and possibly other software subproducts. A company might recommend some books for its products. The database is semistructured because the schemas of its objects are not rigid. For example, a company can only optionally recommend books, or we might be missing information about what products a book might cover. ■

## 2.2 Query Model

A *(user) query* $Q$ is a regular language over $\Delta$. For the ease of notation, we will blur the distinction between regular languages and regular expressions that represent them. Let $Q$ be a query and $DB$ a database. Then, the *answer* to $Q$ on $DB$ is defined as

$$ans(Q, DB) \quad = \quad \{(a, b) : a \xrightarrow{w} b \text{ in } DB \text{ for some } w \in Q\}.$$

**Example 2.** Suppose that the user would like to know for each software product, all the books that might have some useful information about the product. For this, the user can give the regular path query $Q = \mathit{covers} \cdot \mathit{software}^*$ . This query, on the

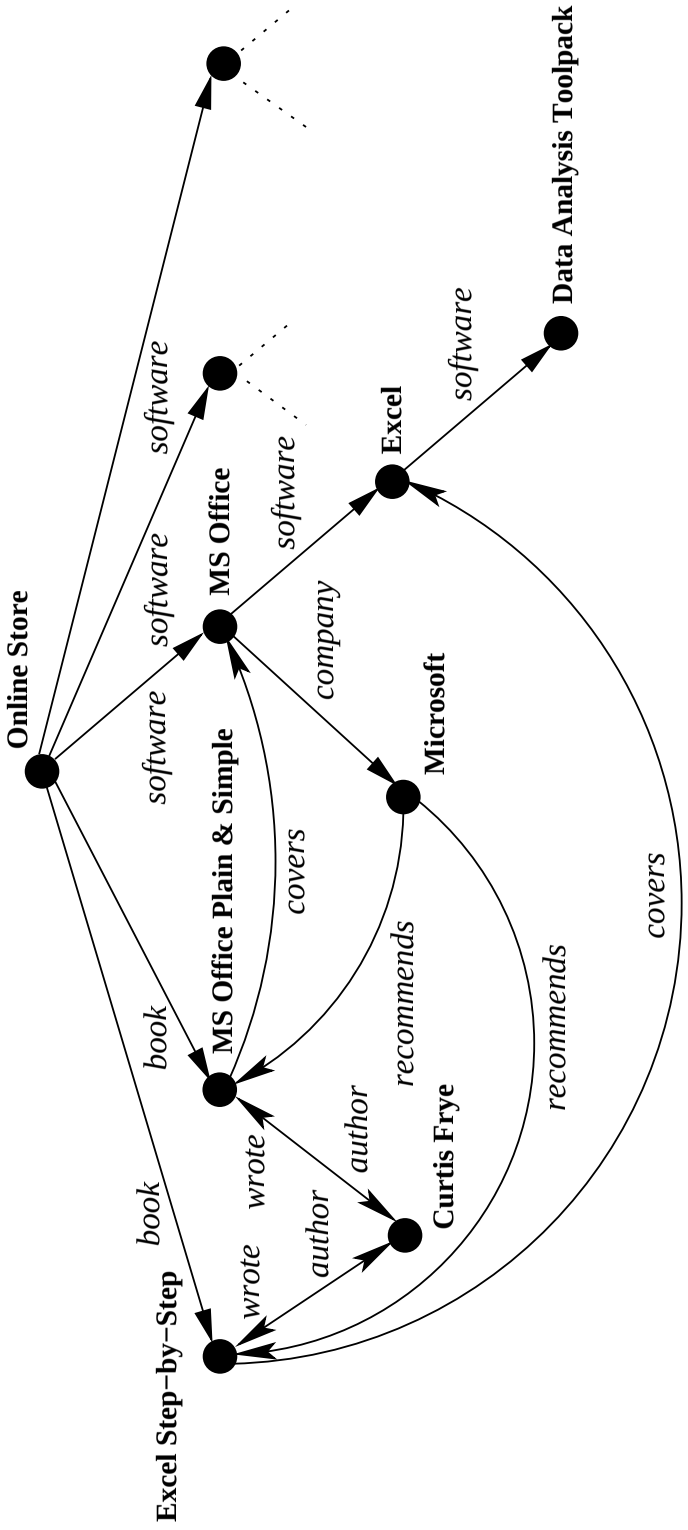



**Figure 2.1:** A graph database.

database $DB$ in Fig. 2.1, will have as answer

$$\begin{aligned} ans(Q, DB) \;=\; \{&(\text{MS Office Plain \& Simple},\ \text{MS Office}),\\ &(\text{MS Office Plain \& Simple},\ \text{Excel}),\\ &(\text{MS Office Plain \& Simple},\ \text{Data Analysis Toolpack}),\\ &(\text{Excel Step-by-Step},\ \text{Excel}),\\ &(\text{Excel Step-by-Step},\ \text{Data Analysis Toolpack})\} \end{aligned}$$

■

## 2.3 Answering RPQ's on Databases

The well-known method for answering RPQ's on a given database (cf. (1)) is as follows. In essence, we create state-object pairs from the query automaton and the database. For this, let $\mathcal{A}$ be an NFA that accepts an RPQ $Q$. Starting from an object $a$ of a database $DB$, we first create the pair $(p_0, a)$, where $p_0$ is the initial state in $\mathcal{A}$. Then, we create all the pairs $(p, b)$ such that there exist a transition from $p_0$ to $p$ in $\mathcal{A}$, and an edge from $a$ to $b$ in $DB$, and furthermore the labels of the transition and the edge match. In the same way, we continue to create new pairs from existing ones, until we are not anymore able to do so. In essence, what is happening is a lazy construction of a Cartesian product graph of the query automaton with the database graph. Of course, only a small (hopefully) part of the Cartesian product is really constructed depending on the selectivity of the query.

After obtaining the above Cartesian product graph, producing query answers

becomes a question of computing reachability of nodes $(p, b)$, where $p$ is a final state, from $(p_0, a)$, where $p_0$ is the initial state. Namely, if $(p, b)$ is reachable from $(p_0, a)$, then $(a, b)$ is a tuple in the query answer.

# Chapter 3

# Views in Information Integration Systems

## 3.1 View Graphs and Possible Databases

Let $V_1, \ldots, V_n$ be languages (queries) on alphabet $\Delta$. We will call them *views* and associate with each $V_i$ a view name $v_i$.

We call the set $\Omega = \{v_1, \ldots, v_n\}$ the *outer alphabet*, or *view alphabet*. For each $v_i \in \Omega$, we set $def(v_i) = V_i$. The substitution $def$ associates with each view name $v_i$ in $\Omega$ alphabet the language $V_i$. The substitution $def$ is applied to words, languages, and regular expressions in the usual way (see e.g. (16)).

A *view graph* is database $\mathcal{V}$ over $\Omega$. In other words, a view graph is a database where the edges are labeled with symbols from $\Omega$. View graphs can also be queried by regular path queries over $\Omega$.

In a LAV ("local-as-view") information integration system (9), we have the "global schema" $\Delta$, the "source schema" $\Omega$, and the "assertion" $def : \Omega \to 2^{\Delta^*}$. The only extensional data available is a view graph $\mathcal{V}$ over $\Omega$ (see also (4, 5, 8)).

Although the user can directly query a view graph $\mathcal{V}$ over $\Omega$, the assumption in

LAV data integration is that what is convenient for the user is to pose queries on $\Delta$, and the system has to answer based solely on the information provided by the views. In order to do this, the system has to reason with respect to the set of *possible databases* over $\Delta$ that $\mathcal{V}$ could represent. Under the *sound view* assumption, a view graph $\mathcal{V}$ defines a set $poss(\mathcal{V})$ of databases as follows:

$$poss(\mathcal{V}) \;=\; \{DB : \mathcal{V} \subseteq \bigcup_{i\in\{1,\ldots,n\}} \{(a, v_i, b) : (a, b) \in ans(V_i, DB)\}\}.$$

(Recall that $V_i = def(v_i)$.) The above definition reflects the intuition about the connection between an edge $(a, v_i, b)$ in $\mathcal{V}$ with some path from $a$ to $b$ in the possible $DB$'s, labeled by some word in $V_i$.

**Example 3.** Consider the view graph in Fig. 3.1 [top], and view definitions $V_1 = def(v_1) = RS^*$, $V_2 = def(v_2) = S^*R$, and $V_3 = def(v_3) = S^+$. Then, a possible database is shown in the same figure [bottom]. Observe that the views are sound only. They are not required to be complete. For example, we do not have a $v_2$-edge from $f$ to $b$ in the view graph. In fact, we do not even have a $f$ object in the view graph. We remark that view soundness is usually the only "luxury" that we have in information integration systems, where the information is often incomplete. ■

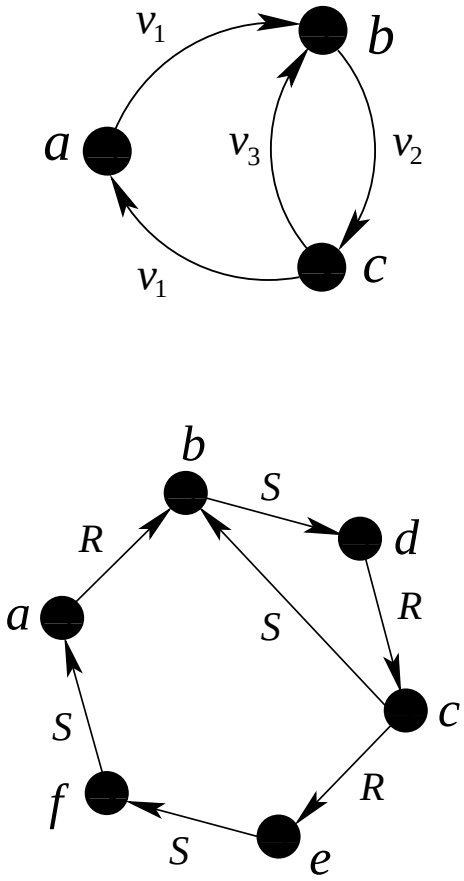



**Figure 3.1:** A view graph and a possible database.

## 3.2 Querying a View Graph

The meaning of querying a view graph through the global schema $\Delta$ is defined as follows. Let $Q$ be a query over $\Delta$. Then

$$\text{ANS}(Q, \mathcal{V}) = \bigcap_{DB \in poss(\mathcal{V})} ans(Q, DB).$$

There are two approaches for computing $\text{ANS}(Q, \mathcal{V})$. The first one is to use an *exponential* procedure in the size of the data (i.e. $\mathcal{V}$) in order to completely compute $\text{ANS}(Q, \mathcal{V})$ (see (4)). There is little that one can better hope for, since in the same paper it has been proven that to decide whether a tuple belongs to $\text{ANS}(Q, \mathcal{V})$ is co-NP complete (see (14) for the definition of this class) with respect to the size of data.

The second approach is to compute first a view-based rewriting $Q'$ for $Q$, as in (3). Such rewritings are regular path queries on $\Omega$. Then, we can approximate $\text{ANS}(Q, \mathcal{V})$ by $ans(Q', \mathcal{V})$, which can be computed in polynomial time with respect to the size of data ($\mathcal{V}$). In general, for a view-based rewriting $Q'$ computed by the algorithm of (3), we have that

$$ans(Q', \mathcal{V}) \subseteq \text{ANS}(Q, \mathcal{V}),$$

with equality when the rewriting is exact ((4)). In the rest of the paper, we will assume that the data-integration system follows the second approach.

# Chapter 4

# Maximal View-Based Rewritings

## 4.1 Definition

Our proposed techniques enhance the computation and use of maximal view-based rewritings given in (3). Thus, we first examine these maximal view-based rewritings and the method of (3) for their computation.

Formally, for a given query $Q$, the maximal view-based rewriting $Q'$, is the set of *all* words on $\Omega$ such that their substitution through $def$ is contained in the query language $Q$, i.e.

$$Q' = \{w : w \in \Omega^* \text{ and } def(w) \subseteq Q\}.$$

Interestingly, as shown in (3), the above set is a regular language on $\Omega$ and the algorithm of (3) for computing an automaton for this language is described on the next section.

## 4.2 Algorithm

**Algorithm 1**

1. *Construct a DFA* $\mathcal{A} = (\Delta, S, s_0, \tau_{\mathcal{A}}, F)$ *such that* $Q = L(\mathcal{A})$.

2. *Construct automaton* $\mathcal{B} = (\Omega, S, s_0, \tau_{\mathcal{B}}, S - F)$, *where* $(s_i, v_a, s_j) \in \tau_{\mathcal{B}}$ *iff there exists* $w \in V_a$ *such that* $(s_i, w, s_j) \in \tau_{\mathcal{A}}^*$.

3. *The rewriting* $Q'$ *is the* $\Omega$ *language accepted by an automaton* $\mathcal{C}$ *obtained by complementing automaton* $\mathcal{B}$.

■

Step 2 can also be expressed equivalently as: Consider each pair of states $(s_i, s_j)$. If in $\mathcal{A}$ there is a path from $s_i$ to $s_j$, which spells a word in some view language $V_a$, then insert a corresponding $v_a$-transition from $s_i$ to $s_j$ in $\mathcal{B}$.

Observe that, if $\mathcal{B}$ accepts an $\Omega$-word $v_1 \cdots v_m$, then there exist $m$ $\Delta$-words $w_1$, $\ldots$, $w_m$ such that $w_i \in V_i$ for $i = 1, \ldots, m$ and such that the $\Delta$-word $w_1 \ldots w_m$ is rejected by $\mathcal{A}$. On the other hand, if there exists a $\Delta$-word $w_1 \ldots w_m$ that is rejected by $\mathcal{A}$ such that $w_i \in V_i$ for $i = 1, \ldots, m$, then the $\Omega$-word $v_1 \cdots v_m$ is accepted by $\mathcal{B}$. That is, $\mathcal{B}$ accepts an $\Omega$-word $v_1 \cdots v_m$ if and only if there is a $\Delta$-word in $def(v_1 \cdots v_m)$ that is rejected by $\mathcal{A}$. Hence, $\mathcal{C}$ being the complement of $\mathcal{B}$ accepts an $\Omega$-word if and only if all $\Delta$-words $w = w_1 \ldots w_m$ such that $w_i \in V_i$ for $i = 1, \ldots, m$, are accepted by $\mathcal{A}$.

## 4.3 Examples

**Example 4.** Let query be $Q = (RS^*)^2 + S^+$ and the view definitions be as in Example 3, i.e. $V_1 = def(v_1) = RS^*$, $V_2 = def(v_2) = S^*R$, and $V_3 = def(v_3) = S^+$. The DFA $\mathcal{A}$ for the query $Q$ is shown in Fig. 4.1[top] and the corresponding automaton $\mathcal{B}$ is shown in in Fig. 4.1[middle]. The resulting complement automaton $\mathcal{C}$ is shown in Fig. 4.1[bottom]. Note that the "trap" and unreachable states have been removed for clarity.

As mentioned in the previous section, the view-based rewriting $Q'$ represented by automaton $\mathcal{C}$ is evaluated on a view graph $\mathcal{V}$ obtaining $ans(Q', \mathcal{V})$ which is an approximation of $\textsc{ans}(Q, \mathcal{V})$.

**Example 5.** Consider the rewriting $Q'$ represented by the automaton $\mathcal{C}$ in Fig. 4.1 [bottom], and the view graph $\mathcal{V}$ in Fig. 3.1 [left]. It is easy to see that $ans(Q', \mathcal{V}) = \{(a, b), (a, c), (c, b)\}$. ■

Assuming that the user query is given by means of a regular expression, (3) showed, using the algorithm above, that the complexity of computing the maximal view-based rewriting is in 2EXPTIME. Moreover, this bound was shown to be tight by constructing a query instance $Q$, whose rewriting has a doubly exponential size compared to the size of a simple NFA for $Q$.

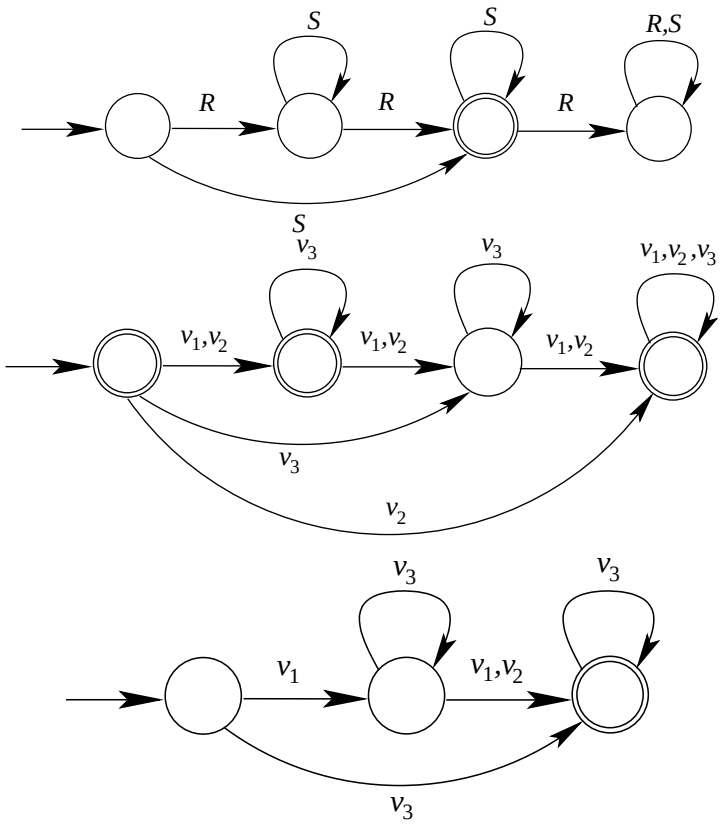



**Figure 4.1:** Automaton $\mathcal{A}$ [top], Automata $\mathcal{B}$ [middle] and Automata $\mathcal{C}$ [bottom]

| | |
|---|---|
| $a, b, c, \ldots$ | Database objects |
| $R, S, \ldots$ | Database symbols |
| $w$ | Words over a given alphabet |
| $DB$ | Database |
| $Q$ | Query |
| $V_1, V_2, \ldots$ | View languages |
| $v_1, v_2, \ldots$ | View symbols |
| $\Delta$ | Database alphabet |
| $\Omega$ | View alphabet |
| $\mathcal{V}$ | Viewgraph |

**Table 4.1:** Table of symbols.

For the convenience of the reader we summarize in Tab. 4.3 the terminology used in this thesis.

# Chapter 5

# Our Optimization Techniques

The above 2EXPTIME bound is somewhat discouraging because it tells us that to obtain a view-based rewriting is computationally hard except for small query instances. While the first determinization [for obtaining automaton $\mathcal{A}$] is in practice quite tolerable for typical user queries, the second determinization [for obtaining automaton $\mathcal{C}$ by complementing $\mathcal{B}$] is often prohibitively expensive. However, it is possible to argue that the analysis in (3) is worst-case and hence the algorithm might take only reasonable amount of time on "typical" instances (or on the average). Our experimental results indicate that this is not the case (please see Section 6). Experimentally, we were unable to compute automaton $\mathcal{C}$, in reasonable time and space, for about one third of the time while working on "randomly generated" instances. This gives us evidence that the algorithm indeed does poorly on the average and needs to be cleverly implemented if we would like to make it work on "large" instances. We believe that this observation is an important contribution of our paper given the fundamental importance of the database problem being studied.

In this chapter, we first describe how to optimize the construction of automaton $\mathcal{B}$ in step 2, which is a significant bottleneck when the number of views is considerable. Then, we deal with step 3 of the algorithm, and propose a technique which essentially eliminates this step.

## 5.1 Computing Automaton $\mathcal{B}$ Efficiently

We present an optimization technique for the step 2 of the above algorithm for computing automaton $\mathcal{B}$. In our experiments we observed that this step, although a polynomial one, is very time consuming, if implemented in the straightforward manner.

Taking a closer look at step 2, let $s_i$ and $s_j$ be two arbitrary states in automaton $\mathcal{A}$. Now consider automaton $\mathcal{A}_{ij}$, which is obtained by keeping all the states and transitions in $\mathcal{A}$, but making state $s_i$ and $s_j$ initial and final respectively. All the other states in $\mathcal{A}_{ij}$ are neither initial nor final.

In step 2 of the algorithm, we want to determine whether there should be transition $v_a$ between states $s_i$ and $s_j$ in $\mathcal{B}$. It is easy to see that this is in fact achieved by testing for the emptiness of the intersection $L(\mathcal{A}_{ij} \cap V_a)$. Namely, we insert a transition $(s_i, v_a, s_j)$ in $\mathcal{B}$ iff $L(\mathcal{A}_{ij} \cap V_a) \neq \emptyset$.

However, the automata $\mathcal{A}_{ij}$ for different $i$'s and $j$'s have the same states and transitions [namely those of automaton $\mathcal{A}$]. Only their initial and final states are different. Thus, we construct only one Cartesian product $\mathcal{A} \times V_a$ for a given view $V_a$. Then, we test emptiness on this Cartesian product automaton for $|\mathcal{A}|^2$ different combinations of [one] initial and [one] final states. Although asymptotically there is

no gain in doing this, experimentally, we found that for typical queries and views, the speedup achieved by this optimization is often more than 6-fold. This is explained by a better utilization of the CPU cache because there is only one Cartesian product automaton to be constructed and examined.

## 5.2 Answer Computation Through Input-Aware Determinization

In this subsection, we describe how to essentially eliminate step 3 of the algorithm of (3). These ideas were inspired by some techniques used in the study of alternating finite automata (AFA). For a good source, see the survey on regular languages by Yu (16).

Recall that the "riskier penalty" in the algorithm of (3) is the computation of automaton $\mathcal{C}$ in step 3 by complementing the automaton $\mathcal{B}$ obtained in step 2. $\mathcal{C}$ might be doubly exponential in the size of the query. Once $\mathcal{C}$ is computed, the final step is to compute $ans(Q', \mathcal{V})$ by constructing the Cartesian product of the automaton $\mathcal{C}$ and a viewgraph $\mathcal{V}$. We ask if it still possible to compute $ans(Q', \mathcal{V})$ directly without first computing the DFA for $L(\overline{\mathcal{B}})$? We achieve this by merging the underlying determinization procedure of step 3 and the subsequent computation of the Cartesian product graph into a single step. We illustrate this using an example.

**Example 6.** Consider the NFA $\mathcal{B}$ and the viewgraph $\mathcal{V}$ shown in Fig. 5.1 [top] and in Fig. 5.1 [middle] respectively. We will build a lazy Cartesian product graph, whose nodes are object–bitvector pairs and edges are labeled with $\Omega$ symbols.

We start with the pair $(a, 100)$, where 100 is an abbreviation for $(1, 0, 0)$. This bitvector says that automaton $\mathcal{B}$ is now in state $s_0$. Next, we construct the pair $(b, 110)$ and put a $v_1$–edge from $(a, 100)$ to $(b, 110)$. This is because when reading symbol $v_1$, we hop to object $b$ in $\mathcal{V}$, and in states $s_0$ and $s_1$ in $\mathcal{B}$. Continuing in this way, we obtain the Cartesian product graph shown in Fig. 5.1 [bottom].

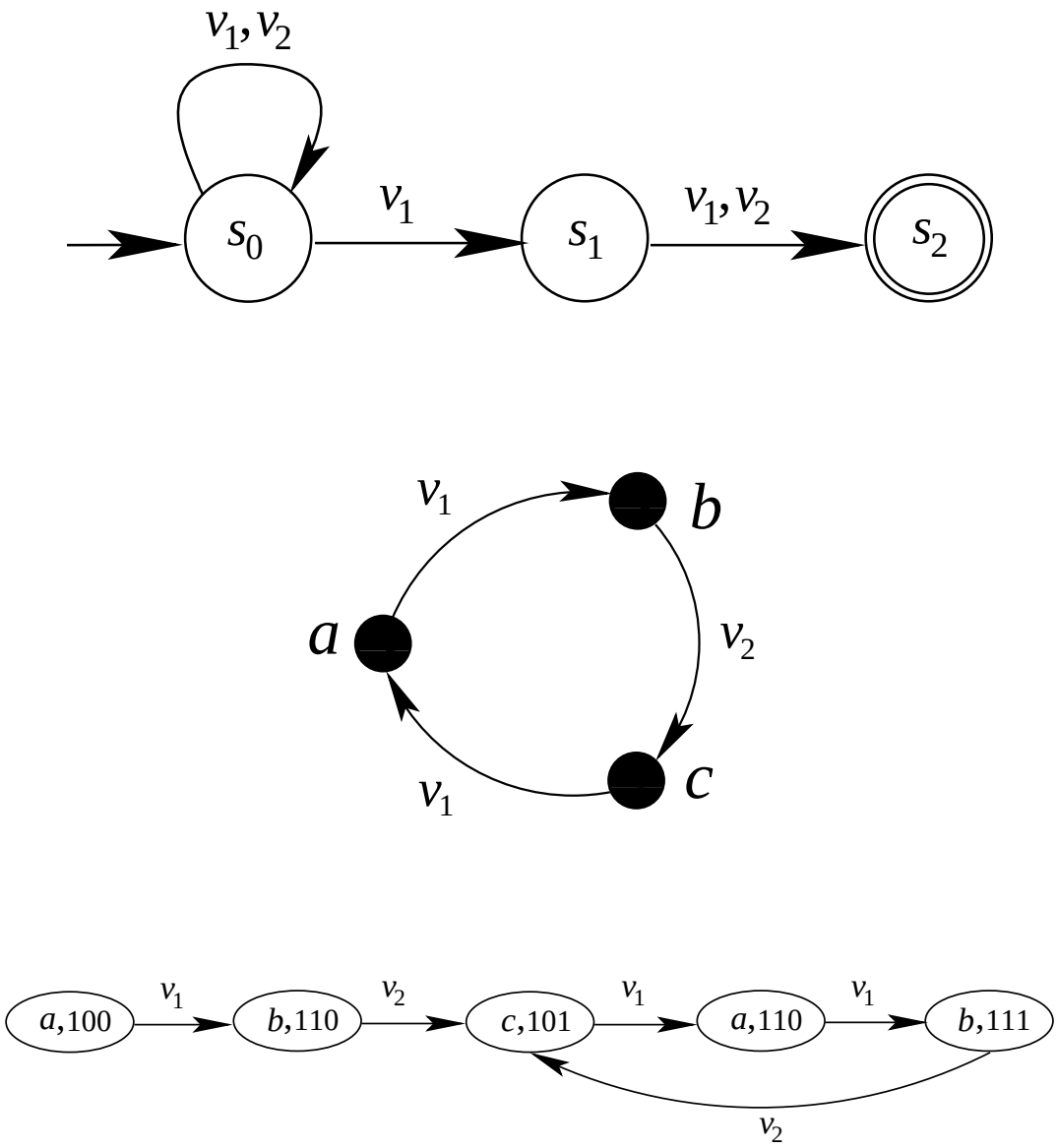



**Figure 5.1:** Automaton $\mathcal{B}$ [top], viewgraph $\mathcal{V}$ [middle], and Cartesian product graph [bottom].

Building of the above bitvectors is reminiscent of the classical subset construction for converting an NFA into a DFA. In fact, each bitvector corresponds to a state in a DFA for $\mathcal{B}$. However, we only build those bitvectors, which are asked for by the input viewgraph. Thus, observe that in this example only 4 bitvectors are needed, namely 100, 110, 101, and 111. On the other hand, the minimum size DFA corresponding to $\mathcal{B}$ has eight states.

Now, once the Cartesian product graph is constructed, it is easily seen that $b$ is reachable from $a$ using a string not in $L(\mathcal{B})$ but $c$ is not. ■

In general, for a $\mathcal{B}$ automaton with set $S$ of states, we use bitvectors of size $|S|$ to keep track of the states that $\mathcal{B}$ can be when reaching some object of the viewgraph. As illustrated by the above example, the nodes of the (lazy) Cartesian product graph are of the form $(a, u)$ where $a$ is an object in the viewgraph and $u$ is a bitvector of size $|S|$. Since the input is a graph as opposed to a string, there can be different bitvectors associated with the same given object (for instance with objects $a$ and $b$ in the example).

We want to stress that we build the Cartesian product graph starting from all the viewgraph objects. In the above example, for clarity we showed the Cartesian product constructed starting from one object only. However, these Cartesian products overlap, and thus, in order to not generate the same object–bitvector pair twice, we maintain a hashtable of the pairs generated so far. In fact, even for a single Cartesian product, the same pair might be needed more than once, and the hashtable is necessary for this case as well in order for the method to terminate.

The edge labels in the Cartesian product graph are of no importance when it comes to generating the query answers. The only thing that matters in this graph is pure reachability. Namely, we produce a pair $(a, b)$ as an answer, if there exists a path [in the Cartesian product graph] from $(a, u_0)$ to $(b, w)$, where $u_0$ is the initial bitvector $10\ldots0$, and $w$ is a bitvector having no bit set to 1 for any final state in $\mathcal{B}$.

Formally, our algorithm is as follows.

**Algorithm 2**

**Input:** Automaton $\mathcal{B}$ and a viewgraph $\mathcal{V}$.

**Output:** $ans(Q', \mathcal{V})$, where $Q' = L(\overline{\mathcal{B}})$.

**Method:**

1. Denote by $u_0$ the bitvector $10\ldots0$ corresponding to the initial state $s_0$ in $\mathcal{B}$.
2. Initialize
   (a) A processing queue $P = \{(a, u_0) : a \text{ object in } \mathcal{V}\}$.
   (b) A hashtable $H = \emptyset$.
   (c) A Cartesian product graph $\mathcal{G} = \emptyset$.
3. Repeat (a), (b), and (c) until queue $P$ becomes empty.
   (a) Dequeue a pair $(a, u)$ from $P$.
   Lookup $(a, u)$ in $H$.

If $(a, u)$ is not yet in $H$, then insert it in $H$ and $\mathcal{G}$.

Otherwise, discard $(a, u)$ as we have already dealt with it.

(b) For each outgoing edge from $a$ to $b$ in $\mathcal{V}$, labeled by some symbol, say $v_{ab}$, compute the "next" bitvector $w$ by procedure

$$w = Next(u, v_{ab}).$$

[We discuss this procedure soon.]

(c) If $w$ is different from the all zero's vector, then insert $(b, w)$ in $P$.

Also, insert edge $((a, u), v_{ab}, (b, w))$ in $\mathcal{G}$

4. Finally, set

$$ans(Q', \mathcal{V}) = \{(a, b) \ : \ \text{there exists a path in } \mathcal{G} \text{ from } (a, u_0) \text{ to } (b, w) \text{ such that } w \text{ has no bit set to 1 for any final state in } \mathcal{B}\}.$$

**Implementation of** $Next(u, v)$

We optimize the amount of time taken to compute adjacent bitvectors using a technique inspired by (13).

Normally, each entry in the transition table of $\mathcal{B}$ is just a list of next possible states of the NFA given the current state and input symbol. Instead of storing this list, we store a bitvector $\alpha$ of $|S|$ bits, that is the characteristic vector of this list of states. Using the various values of $\alpha$ in the transition table, given an object-vector pair of the form $(a, u)$ and an input symbol $v$, we can compute $Next(u, v)$ in only

$O(n)$ time using a sequence of bitwise-OR operations [compared to the naive method of updating vectors that takes $O(n^2)$ in the worst case]. In particular, without loss of generality, suppose the set of indices in $u$ which have a 1 is exactly $\{i_1, i_2 \ldots, i_k\}$. Then it is easy to see that

$$Next(u, v) = \alpha_{i_1,v} \vee \alpha_{i_2,v} \vee \ldots \vee \alpha_{i_k,v}$$

where $\alpha_{i_j,v}$ is the bitvector $\alpha$ in the transition table corresponding to the state $q_{i_j}$ and the input symbol $v$. In the next section, we show that our ideas give substantial improvement in running time making it possible to solve the problem of view-based answering on much larger instances compared to the naive implementation.

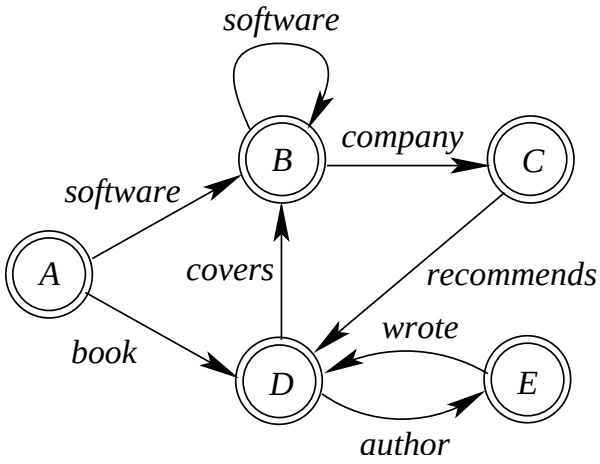
# Chapter 6

# Experimental Results

We conducted some simple experiments in order to assess the improvements offered by answering the query using automaton $\mathcal{B}$ over answering using automaton $\mathcal{C}$.

First, we give some details on how we generated queries, views, and viewgraphs. For this we used a simple DataGuide (cf. (1)). DataGuides are essentially finite state automata capturing all the words spelled out by the database paths. In general, DataGuides are compact representations of graph databases. They are small automata presented to the user in order to guide him in writing queries. Each word in a DataGuide could possibly represent many paths that spell that word in a database. For example a DataGuide, capturing databases such as the one shown in Fig. 2.1, contains a word *software·company·recommends*. Certainly, there are many such paths in databases about online stores.

In our experiments, we used the DataGuide given in Fig. 6.1, where all the states are both initial and final.




$$
\begin{array}{lcl}
A & \rightarrow & software \cdot B \mid book \cdot D \mid \epsilon \\
B & \rightarrow & software \cdot B \mid company \cdot C \mid \epsilon \\
C & \rightarrow & recommends \cdot D \mid \epsilon \\
D & \rightarrow & covers \cdot B \mid author \cdot E \mid \epsilon \\
E & \rightarrow & wrote \cdot D \mid \epsilon
\end{array}
$$

**Figure 6.1:** [Top] DataGuide corresponding to the database in Fig. 2.1. [Bottom] Grammar for the given DataGuide.

## 6.1 Database Generation

A database is represented as a set of *object-symbol-object* edges that are generated based on the dataguide of Fig. 6.1 [top].

In order to generate such a triplet, we first randomly select a state from the dataguide and an outgoing transition from that state. For example, suppose that $B$ and $(B, company, C)$ are the chosen state and transition respectively.

Then, for each of the two states of the chosen transition we generate a random number. These numbers are paired-up with the states of the transition. Each pair will correspond to a database object. For example, for the above-chosen transition we could generate two database objects ($B$,3) and ($C$,1). Therefore, the generated database edge is (($B$,3)-*company*-($C$,1)).

This procedure guarantees, that every path of the generated database, spells a word accepted by the dataguide. For example see Figure 6.2.

## 6.2 Views and Rewriting NFA Generation

For generating view language definitions, we randomly generated partial derivations using the above grammar. Such a partial derivation is for example $B \rightarrow company \cdot recommends \cdot D$. By randomly selecting such partial derivations, we created new right linear grammars. We kept only those grammars generating non-empty languages. Clearly, the grammars generated in this way capture sublanguages of the DataGuide. By this random procedure, we created 50 test sets of 40 views definitions each.

Then, for each set of views, we created random queries as follows. Let $\mathbf{V} =$

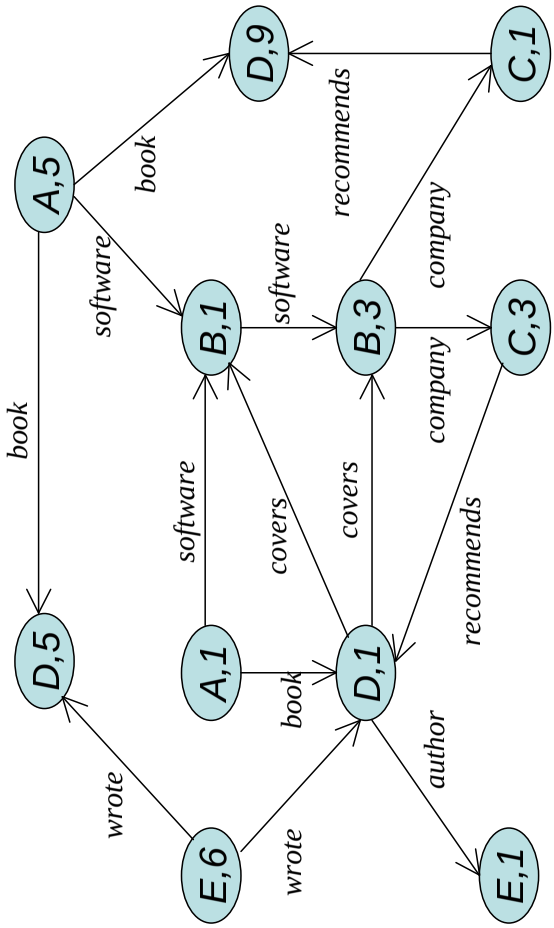



**Figure 6.2:** Example of database generation.

$\{V_1, \ldots, V_{40}\}$ be a view set. Then, the outer-alphabet is $\Omega = \{v_1, \ldots, v_{40}\}$. First, we randomly created a regular expression on $\Omega$ of length not more than 10. For instance such a regular expression could be $re = v_1 \cdot v_{13}^* + v_{40}$. Next, we set $Q = def(re)$, which is a language on $\Delta$, and computed its view-based rewriting using set $\mathbf{V}$ of views.

We could certainly generate queries in a similar fashion as for generating view languages i.e. directly from the DataGuide. However, doing so generates many cases when the rewriting is empty, and the experiments would be uninteresting. On the other hand, generating queries as above guarantees that the rewritings will not be empty.

Regarding the generation of view graphs, we first randomly generated databases from the Data-Guide, and then evaluated on these databases each of the generated views. In this way, we obtained an "answer" for each view. For instance, we could have $\{(a, b), (b, c), \ldots\}$ as the answer for $V_1$ in some randomly generated database. Then, we inserted edges $(a, v_1, b)$, $(b, v_1, c)$, ... in the the corresponding viewgraph. For each of the 50 sets of views, we randomly generated as above a viewgraph of more than 10,000 nodes.

## 6.3 Automaton $\mathcal{B}$ and Viewgraph Evaluation

Then, we computed automaton $\mathcal{B}$ for each set of views, and evaluated it [as described in Section 5] on the corresponding viewgraph. Also, we tried to compute automaton $\mathcal{C}$ accepting $L(\overline{\mathcal{B}})$. This was done by determinizing automaton $\mathcal{B}$. We used GRAIL+, which is a well-engineered automata package. As already mentioned, computing $\mathcal{C}$ was not always possible. Out of our 50 cases, computing $\mathcal{C}$ timed out in 15 of them.

| ID | B-NFA Size | C-DFA-size | C-DFA-time | C-DFA-V-time | C-DFA-V TTime | B-BitNFA-V-time | B-BitNFA-V-size | Ratio |
|---|---|---|---|---|---|---|---|---|
| 1 | 11 | 35 | 2 | 317 | 319 | 348 | 27887 | 1.1 |
| 2 | 9 | 16 | 1 | 396 | 397 | 390 | 23967 | 1 |
| 3 | 12 | 67 | 7 | 330 | 337 | 396 | 31875 | 1.2 |
| 4 | 11 | 34 | 3 | 410 | 413 | 417 | 29003 | 1 |
| 5 | 9 | 40 | 1 | 407 | 409 | 419 | 26172 | 1 |
| 6 | 12 | 74 | 5 | 489 | 494 | 530 | 31766 | 1.1 |
| 7 | 13 | 57 | 7 | 573 | 580 | 651 | 32501 | 1.1 |
| 8 | 15 | 83 | 11 | 630 | 641 | 678 | 35332 | 1.1 |
| 9 | 23 | 462 | 393 | 454 | 847 | 773 | 40899 | 0.9 |
| 10 | 12 | 69 | 8 | 805 | 813 | 887 | 37850 | 1.1 |
| 11 | 14 | 114 | 8 | 703 | 711 | 901 | 39252 | 1.3 |
| 12 | 17 | 166 | 29 | 540 | 569 | 911 | 44261 | 1.6 |
| 13 | 13 | 72 | 9 | 905 | 914 | 1037 | 38095 | 1.1 |
| 14 | 13 | 221 | 19 | 642 | 661 | 1159 | 50413 | 1.8 |
| 15 | 16 | 513 | 87 | 609 | 696 | 1180 | 48698 | 1.7 |
| 16 | 20 | 319 | 153 | 743 | 896 | 1247 | 47582 | 1.4 |
| 17 | 12 | 82 | 8 | 1067 | 1074 | 1457 | 47051 | 1.4 |
| 18 | 35 | 1442 | 2148 | 824 | 2972 | 1505 | 52686 | 0.5 |
| 19 | 33 | 3316 | 4126 | 592 | 4718 | 1593 | 61860 | 0.3 |
| 20 | 16 | 266 | 61 | 1058 | 1119 | 1867 | 56361 | 1.7 |
| 21 | 21 | 552 | 296 | 859 | 1154 | 1879 | 58879 | 1.6 |
| 22 | 35 | 723 | 710 | 1106 | 1816 | 2074 | 55939 | 1.1 |
| 23 | 31 | 831 | 461 | 913 | 1374 | 2113 | 61329 | 1.5 |
| 24 | 21 | 1316 | 526 | 867 | 1392 | 2121 | 66651 | 1.5 |
| 25 | 20 | 1098 | 379 | 1046 | 1425 | 2206 | 65004 | 1.5 |
| 26 | 18 | 238 | 60 | 1372 | 1432 | 2846 | 63561 | 2 |
| 27 | 18 | 523 | 104 | 1056 | 1160 | 3061 | 74273 | 2.6 |
| 28 | 20 | 550 | 177 | 1083 | 1260 | 3403 | 83515 | 2.7 |
| 29 | 26 | 2001 | 855 | 1245 | 2099 | 3512 | 80085 | 1.7 |
| 30 | 33 | 3578 | 2197 | 1106 | 3303 | 3599 | 83338 | 1.1 |
| 31 | 38 | 3492 | 2937 | 1628 | 4565 | 3666 | 76546 | 0.8 |
| 32 | 35 | 1720 | 1210 | 959 | 2169 | 3674 | 84318 | 1.7 |
| 33 | 28 | 2894 | 2515 | 1330 | 3845 | 4477 | 102625 | 1.2 |
| | | | | | | | | **1.3** |
| 34 | 49 | N/P | N/P | N/A | N/A | 697 | 103251 | |
| 35 | 42 | N/P | N/P | N/A | N/A | 820 | 101291 | |
| 36 | 44 | N/P | N/P | N/A | N/A | 892 | 92852 | |
| 37 | 53 | N/P | N/P | N/A | N/A | 1224 | 50903 | |
| 38 | 41 | N/P | N/P | N/A | N/A | 1554 | 56048 | |
| 39 | 52 | N/P | N/P | N/A | N/A | 1754 | 44805 | |
| 40 | 48 | N/P | N/P | N/A | N/A | 2033 | 66406 | |
| 41 | 53 | N/P | N/P | N/A | N/A | 2239 | 66052 | |
| 42 | 43 | N/P | N/P | N/A | N/A | 2270 | 76941 | |
| 43 | 42 | N/P | N/P | N/A | N/A | 2549 | 85026 | |
| 44 | 48 | N/P | N/P | N/A | N/A | 3358 | 80330 | |
| 45 | 44 | N/P | N/P | N/A | N/A | 3468 | 83515 | |
| 46 | 30 | N/P | N/P | N/A | N/A | 3542 | 86632 | |
| 47 | 42 | N/P | N/P | N/A | N/A | 3816 | 81133 | |
| 48 | 40 | N/P | N/P | N/A | N/A | 3890 | 84563 | |
| 49 | 45 | N/P | N/P | N/A | N/A | 4872 | 103183 | |
| 50 | 47 | N/P | N/P | N/A | N/A | 6185 | 123831 | |

**Table 6.1:** Table of results.

We used a big timeout of 4 hours. Whenever we were able to obtain a DFA $\mathcal{C}$, we evaluated it on the corresponding viewgraph. For these case, we compared the times of evaluating $\mathcal{B}$ versus evaluating $\mathcal{C}$ on the viewgraphs.

In all the test cases, we computed automaton $\mathcal{B}$ using the technique described in Subsection 5.1. It was this technique that made possible the computation of $\mathcal{B}$ in a reasonable amount of time for each test case (of 40 views each). As mentioned in Subsection 5.1, using our technique we were able to achieve a speedup of more that six-fold in computing $\mathcal{B}$. Due to space constraints, we do not show the times for computing the $\mathcal{B}$ automata. These times range between 10 to 15 minutes.

## 6.4 Results

We have tabulated our time and size results in Tab. 6.1. The results were obtained using a modern Sun-Blade-1000 machine with 1GB of RAM. In the following, we describe the column headers of our result table.

**ID:** ID of test set.

**B-NFA-size:** Size of automaton (NFA) $\mathcal{B}$.

**C-DFA-size:** Size of automaton (DFA) $\mathcal{C}$.

**C-DFA-time:** Time (in secs) to compute automaton (DFA) $\mathcal{C}$.

**C-DFA-V-time:** Time (in secs) to evaluate automaton (DFA) $\mathcal{C}$ on the corresponding viewgraph.

**C-DFA-V-TTime:** Total time (in secs) to compute and then evaluate automaton

(DFA) $\mathcal{C}$ on the corresponding viewgraph. [This is the sum of the above two times.]

**B-BitNFA-V-time:** Time (in secs) to bitwise evaluate automaton (NFA) $\mathcal{B}$ on the corresponding viewgraph.

**B-BitNFA-V-size:** Size of the input-aware Cartesian product of automaton (NFA) $\mathcal{B}$ with the corresponding viewgraph.

**Ratio:** Ratio of the time to obtain the answers using bitwise evaluation of automaton (NFA) $\mathcal{B}$ to the time to obtain the answers using automaton (DFA) $\mathcal{C}$ whenever possible. The last number of 1.3 in this column is the average of the column.

We have sorted the results in ascending order of the **B-BitNFA-V-time**. The first part of the table contains the results for the cases when the computation of automaton $\mathcal{C}$ succeeded. The second part of the table contains the results for the cases when the computation of automaton $\mathcal{C}$ failed. As such, the second part of the table has results which relate to the use of automaton $\mathcal{B}$ only. The shaded area of this part of the table is marked by N/P (Not Possible) or (N/A) (Not Applicable) as appropriate.

Based on this table of experimental results, we are able to draw the following natural conclusions.

1. Computing in full the view-based rewriting represented by automaton $\mathcal{C}$ is hard and fails in a considerable number of cases (30% of them). Hence, one should not pursue this route for producing view-based query answers.

2. Even when constructing $\mathcal{C}$ is possible, the performance advantage offered by

the determinism of $\mathcal{C}$ over automaton $\mathcal{B}$ is small. In average, bitwise evaluation of $\mathcal{B}$ is only 1.3 times slower on the average, while in some cases it can be even faster. This is due to the smaller footprint of $\mathcal{B}$ having so a better hardware cache utilization.

3. For all the test cases, the size of the input-aware bitwise Cartesian product of automaton $\mathcal{B}$ with the corresponding viewgraph $\mathcal{V}$ is very far from the worst case of $2^{|\mathcal{B}|} \cdot |\mathcal{V}|$.

From all the above, one can see that by employing our techniques, the view-based answering of RPQ's becomes feasible in practice.

# Chapter 7

# Conclusions

In this paper, we examined the well-known problem of answering regular path queries (RPQ) using views. This problem is particularly important in applications using semistructured data. This paper makes two very useful contributions towards a better understanding of the important algorithm of (3). Firstly, it shows experimental evidence that the algorithm, known to have worst-case running time of 2EXPTIME, also takes lot of time on the average. Secondly, it applies some simple automata-theoretic techniques to optimize the implementation of the various steps of the algorithm aimed towards speeding up the algorithm on large instances. We show, through experimental data, that this leads to significant improvement of running-time on large instances. In particular, we would like to emphasize the usefulness of the "input-aware lazy determinization" that we have used in this paper. We hope that this paper will lead to further study of this very important problem.

# Bibliography

[1] Abiteboul S., P. Buneman, and D. Suciu. *Data on the Web : From Relations to Semistructured Data and XML.* Morgan Kaufmann Publishers. San Francisco, CA., 1999.

[2] Bravo L., and L. Bertossi. Disjunctive Deductive Databases for Computing Certain and Consistent Answers to Queries from Mediated Data Integration Systems. *Journal of Applied Logic* 3(1): 329–367, 2005.

[3] Calvanese D., G. Giacomo, M. Lenzerini and M. Y. Vardi. Rewriting of Regular Expressions and Regular Path Queries. *J. Comput. Syst. Sci.* 64 (3) : 443–465, 2002.

[4] Calvanese D., G. Giacomo, M. Lenzerini and M. Y. Vardi. Answering Regular Path Queries Using Views. *Proc. ICDE '00.*

[5] Calvanese D., G. Giacomo, M. Lenzerini, and M. Y. Vardi. View-based Query Processing: On the Relationship between Rewriting, Answering and Losslessness. *Proc. of ICDT '05.*

[6] Grahne G., and A. O. Mendelzon Tableau Techniques for Querying Information Sources through Global Schemas. *Proc. ICDT '99.*

[7] Grahne G., and A. Thomo. Algebraic Rewritings for Optimizing Regular Path Queries. *Proc. ICDT '01.*

[8] Grahne G., A. Thomo, and W. Wadge. Preferentially Annotated Regular Path Queries. *Proc. of ICDT'07.*

[9] Lenzerini M. Data Integration: A Theoretical Perspective. *Proc. of PODS'02.*

[10] Levy A. Y., Mendelzon A. O., Sagiv Y., Srivastava D. Answering Queries Using Views. *Proc. PODS '95*, pp. 95-104

[11] Mendelzon A. O., and P. T. Wood, Finding Regular Simple Paths in Graph Databases. *SIAM J. Comp.* 24 (6) : 1235–1258, 1995.

[12] Mendelzon A. O. G. A. Mihaila and T. Milo. Querying the World Wide Web. *Int. J. Dig. Lib.* 1 (1) : 57–67, 1997.

[13] Salomaa K., X. Wu, S. Yu. Efficient Implementation of Regular Languages Using Reversed Alternating Finite Automata. *Theor. Comput. Sci.* 231 (1) : 103–111, 2000.

[14] Sipser M. Introduction To The Theory Of Computation *Thomson Course Technology*, 2005.

[15] Ullman J. D. Information Integration Using Logical Views. *Proc. ICDT '97*, pp. 19-40.

[16] S. Yu. Regular Languages. In: *Handbook of Formal Languages*, pp. 41–110. G. Rozenberg and A. Salomaa (Eds.). Springer Verlag 1997.

# Chapter 8

# Appendix

## 8.1 Data Guide

```
1-a4|b2|c3|e
2-a2|d4|e
3-b1|c2|a4|e
4-c2|a3|b1|e
```

## 8.2 Database Generator

```
#include    <fstream.h>
#include    <iostream.h>
#include    <stdlib.h>
#include    <string.h>
/*******************************************************************************************
Database Generator

- This Program takes no parameters. It takes the dataguide file and generates the
  database from it.

- compilation:
  %gcc db_viewgraph.C -o db_viewgraph.out    OR
  %CC db_viewgraph.C -o db_viewgraph.out
*******************************************************************************************/

int
get_state_length(char *str, int pos){
   int i = 0;
   while((str[pos]>='0')&&(str[pos]<='9')){
        pos++;
           i++;
   }
   return i;
}


int
get_symbol_length(char *str, int pos){
   int i = 0;
   while((str[pos]>='a')&&(str[pos]<='z')){
        pos++;
           i++;
   }
   return i;
}

//it counts the number of rules present per line
int
get_number_of_options(char *str){
     int i =0;
     int count =0;
     while((str[i]!='\0')&&(str[i]!='\n')){
```

```
          if(str[i] == '|') count++;
          i++;
     }
     return count;
}

//returns the state number appearing in some position
char *
get_state(char* str, int index){

     int number_digits = get_state_length(str,index);


     char* str_state;
     str_state = (char*)malloc(20*sizeof(char));

     for(int j = 0; j < number_digits; j++){
          str_state[j] = str[index];
          index++;

     }
     str_state[number_digits] = 0;


     return str_state;

}

//append an string to another string on a specified position
void
append(char* derived,char* source, int* count){
     int i= 0;
     while(source[i]!='\0'){
          derived[(*count)] = source[i];
          i++;
          (*count)++;
     }

}
// this function is weird. I should be able to do this with strcpy given
// I will revise this later

void
string_copy(char* derived , char* source){
     int i= 0;
     while(source[i]!='\0'){
          derived[i] = source[i];
          i++;
     }
     derived[i] = '\0';
}

// find the nth index of the separator "|" on a string
int
index_of_or(char* str, int rule_pos){
     int i = 0;

     while(str[i]!='-') i++;

     for(int times = 0; times< rule_pos; times++){
          i++;
          while(str[i]!='|') i++;
     }
     return i;
}

/*
     Generates the random rule based on the following:
     - generates a random number from 0 10 100
     - if the number is less than 50 then is termnation (no more rules to apply)
     - otherwise, match is value with a partition given by the number of rules
       given as a paramenter
*/
int
get_random_rule(int n_rules){
     int randis = rand()%100;
     if(randis<50)  return -1;
     else{
          int slice = (int)(50/n_rules);
          for(int i = 0; i< n_rules; i++){
               int low_bound = 50 + i*slice;
               int high_bound = 50 + (i+1)*slice;
               if((randis>low_bound)&&(randis<high_bound))
                    return i;
           }
     }
     return -1;
}
/*
     Generates the label of the new state based on the concept of pool_label and random
     labelling
```

```
     Parameters:
          temp_state: represents the state to which it is moving to.
          label_pool: contains all the labes already used
          label pool count: just keeps track of the number of element inside the pool
          input label count: set the number of labels to add
     returns the new numbering to be applies to the database
*/
int
get_new_state_label(int temp_state, int*** label_pool, int *label_pool_count, int input_label_size){
     // find a random number to apply
     int input_label = rand()%input_label_size;
     int found = 0;
     int i =0;

     while((!found)&&(i<(*label_pool_count))){
          if(((*label_pool)[i][0]==temp_state)&&((*label_pool)[i][1]==input_label))
               return i;
          i++;
     }

     if (!found){
          (*label_pool)[(*label_pool_count)][0] = temp_state;
          (*label_pool)[(*label_pool_count)][1] = input_label;
          (*label_pool_count)++;
     }

     return i;


}

/*
   Generates a random Database based on the dataguide provided.
   parameters:
         int dataguide_line_count: number of lines present on the dataquide
         char** dataguide: double array representing the grammar to be derived.
         DB_index: counts the number of lines for the derived subgrammar
   returns:
         A double array representing the new derived Database

*/
char**
gen_DB(int dataguide_line_count, char** dataguide, int* DB_index){
     char derived_line[50];  //this is the new line for the subgrammar

     //this one decides total number of objects on the database
     int label_alpha_size = 60000;
     *DB_index=0;

     // Initializing the array of values...
     char **DB = (char **)malloc(200000 * sizeof(char *));
     for(int i = 0; i < 200000; i++)
          DB[i] = (char *)malloc(50 * sizeof(char));

     // Initializing the array for the label pool
     int label_pool_count = 0;
     int ** label_pool = (int **)malloc(200000 * sizeof(int *));
     for(int j = 0; j < 200000; j++)
          label_pool[j] = (int *)malloc(2 * sizeof(int));

     //Just make it 50 times
     for(int l =0; l < 70000; l++){
          int i = 0;
          int apply_number = 0;
          char source_state[50] = "";
          //choose randomly a line to start
          int line_selection_index = rand()%dataguide_line_count;
          string_copy(source_state,get_state(dataguide[line_selection_index],i));
          // start now generating the random lines on the program
          int done = 0;  // flag that decides whether more rules are applied
          int index = 0; // it is just a pivot to navigate through the a dataguide line
          int new_sate_label = 0; // this is the new label system that uses the pool of labels
          int derived_count = 0;  // counts the lenght of the new line
          int rule_to_apply; // from one line, this rule is chosen randomly
          char input_symbol;// symbol that is used for the transitions
          char sink_state[50]; // state to which the transition function goes to
          int new_state_label; //state number generated using the label pool
          int temp_state;

          //just for the first element make sure it also gets the label
          temp_state = atoi(source_state)-1;
          new_state_label = get_new_state_label(temp_state, &label_pool, &label_pool_count, label_alpha_size);
          sprintf(source_state,"%d",new_state_label);

          // giving a limit for generation of ten at most
          while((!done)&&(apply_number<40)){
               int number_of_rules2 = get_number_of_options(dataguide[line_selection_index]);
               rule_to_apply = get_random_rule(number_of_rules2);
               if((rule_to_apply==-1)&&(apply_number>0)){done = 1;}
               else{
                    derived_count = 0;
```

```
                    append(derived_line,source_state,&derived_count);

                    //finds the corresponding rule and break its elements '
                    index = index_of_or(dataguide[line_selection_index],rule_to_apply);

                    // input symbol and destination state
                    input_symbol = dataguide[line_selection_index][index+1];
                    string_copy(sink_state,get_state(dataguide[line_selection_index],index+2));
                    derived_line[derived_count] = ' ';
                    derived_count++;
                    derived_line[derived_count] = input_symbol;
                    derived_count++;
                    derived_line[derived_count] = ' ' ;
                    derived_count++;

                    temp_state = atoi(sink_state)-1;

                    /* This is where the fun starts.  The following is to indicate the numbering of
                       the state to be used.  We will use a
                       function to return the new number label
                    */
                    new_state_label = get_new_state_label(temp_state, &label_pool,
                                      &label_pool_count, label_alpha_size);
                    sprintf(source_state,"%d",new_state_label);
                    append(derived_line,source_state,&derived_count);

                    /* This is the line to go to for the next rule.
                       this is again assuming that the lines appear in the same order as the
                       states are labeled
                    */
                    line_selection_index = temp_state;
                    derived_line[derived_count] = '\0';
                    string_copy(DB[*DB_index],derived_line);
                    free(derived_line);
                    (*DB_index)++;
                    apply_number++;
                }// end of else
            }// end of while
    }
    return DB;
}


int
main(int argc, char** argv)
{
       char** DB;
       int DB_index = 0;

        //new seed for every new iteration
        srand( (unsigned)time( NULL ) );
        // Initializing the array of values...
        char **dataguide = (char **)malloc(50 * sizeof(char *));
        for(int i =  0; i < 50; i++)
          dataguide[i] = (char *)malloc(50 * sizeof(char));

        //This part reads/writes the files and puts them into fm
        fstream      f_argin;
        fstream  f_argout;
        //keeps count on the number of lines
        int dataguide_line_count = 0;

        f_argin.open("grammar_1", ios::in);
        //populate dataguide
        while(!f_argin.eof()){
               f_argin.getline(dataguide[dataguide_line_count], 90);
               dataguide_line_count++;
        }

        //printing the dataguide
        for(int r =0; r < dataguide_line_count; r++){
           cout<<dataguide[r]<<"\n";
        }
        f_argin.close();
        int DB_count;
        DB_count = 0;
        DB = gen_DB(dataguide_line_count,dataguide, &DB_count);

        char file_name[100];
        sprintf(file_name, "database_large");
        f_argout.open(file_name, ios::out);
        //writing on the File the database lines
        for(int h=0;h<DB_count;h++){
           f_argout << (DB[h]) << "\n";
        }
        f_argout.close();
        //Free memory from the DB
        for (int a=0; a<DB_count; a++)
             free(DB[a]);
        free(DB);
```

```
        return 1;
}
```

## 8.3 Views Generator

```
#include    "include.h"
#include    "lexical.h"
#include    <strstream.h>
#include    <fstream.h>
#include    <iostream.h>
#include    <stdlib.h>
#include    <string.h>
/*********************************************************************************************
Views Generator

- This Program takes 1 parameter, which is the future name of the view created.
  It has to be a number (integer).
  It takes the dataguide file and generates the
  view from it.
- compilation:
  %CC db_viewgraph.C -o db_viewgraph.out
*********************************************************************************************/
int
get_state_length(char *str, int pos){
   int i = 0;
   while((str[pos]>='0')&&(str[pos]<='9')){
       pos++;
          i++;
   }
   return i;
}
int
get_symbol_length(char *str, int pos){
   int i = 0;
   while((str[pos]>='a')&&(str[pos]<='z')){
       pos++;
          i++;
   }
   return i;
}

//it counts the number of rules present per line
int
get_number_of_options(char *str){
    int i =0;
    int count =0;
    while((str[i]!='\0')&&(str[i]!='\n')){
        if(str[i] == '|') count++;
        i++;
    }
    return count;
}

//returns the state number appearing in some position
char *
get_state(char* str, int index){
    int number_digits = get_state_length(str,index);
    char str_state[20];
    for(int j = 0; j < number_digits; j++){
        str_state[j] = str[index];
        index++;
    }
    str_state[number_digits] = 0;
    return str_state;
}

//append an string to another string on a specified position
void
append(char* derived,char* source, int* count){
    int i= 0;
    while(source[i]!='\0'){
        derived[(*count)] = source[i];
        i++;
        (*count)++;
    }
}
// this function is weird. I should be able to do this with strcpy given
// I will revise this later
void
string_copy(char* derived , char* source){
    int i= 0;
    while(source[i]!='\0'){
        derived[i] = source[i];
        i++;
    }
    derived[i] = '\0';
}
```

```
// find the nth index of the separator "|" on a string
int
index_of_or(char* str, int rule_pos){
    int i = 0;
    while(str[i]!='-') i++;
    for(int times = 0; times< rule_pos; times++){
        i++;
        while(str[i]!='|') i++;
    }
    return i;
}
/*
    Generates the random rule based on the following:
    - generates a random number from 0 10 100
    - if the number is less than 50 then is termnation (no more rules to apply)
    - otherwise, match is value with a partition given by the number of rules
      given as a paramenter
*/
int
get_random_rule(int n_rules){
    int randis = rand()%100;
    if(randis<50)  return -1;
    else{
         int slice = (int)(50/n_rules);
         for(int i = 0; i< n_rules; i++){
            int low_bound = 50 + i*slice;
            int high_bound = 50 + (i+1)*slice;
            if((randis>low_bound)&&(randis<high_bound))
                return i;
         }
    }
    return -1;
}

/*
   Generates a random subgrammar based on the dataguide provided.
   parameters:
         int dataguide_line_count: number of lines present on the dataquide
         char** dataguide: double array representing the grammar to be derived.
         subgrammar_index: counts the number of lines for the derived subgrammar
   returns:
         A double array representing the new derived subgrammar

*/
char**
gen_subgrammar(int dataguide_line_count, char** dataguide, int* subgrammar_index){

    // Initializing the array of values...
    char **subgrammar = (char **)malloc(50 * sizeof(char *));
    for(int i = 0; i < 50; i++)
        subgrammar[i] = (char *)malloc(50 * sizeof(char));
    for(int l =0; l < dataguide_line_count; l++){
        // Initializing the strings
        char source_state[50] = "";
        char sink_state[50] = "";
        char string_symbol[50] = "";

        int number_symbols;
        int number_digits;
        int i = 0;

        char input_symbol;
        int number_of_options;

        string_copy(source_state,get_state(dataguide[l],i));

        int number_of_rules = get_number_of_options(dataguide[l]);

        //for each rule create a new derivated rule for the subgrammar
        for(int n=0;n<number_of_rules;n++){
            char derived_line[50]; //this is the new line for the subgrammar
            int derived_count = 0;  // counts the lenght of the new line
            int done = 0;           // flag that decides whether more rules are applied
            int apply_number = 0;
            append(derived_line,source_state,&derived_count);
            derived_line[derived_count] = '-';
            derived_count++;

            int index = index_of_or(dataguide[l],n);
            input_symbol = dataguide[l][index+1];
            string_copy(sink_state,get_state(dataguide[l],index+2));
            derived_line[derived_count] = input_symbol;
            derived_count++;
            int sink_state_line = atoi(sink_state)-1;

            /* rule to apply is the step that put collects that randomly decides
               the rule to be applied for the subgrammar
            */
            int rule_to_apply;
            while((!done)&&(apply_number<10)){
                rule_to_apply = get_random_rule(number_of_rules);
```

```
                if(rule_to_apply==-1){
                    done = 1;
                }
                else{
                    //finds the corresponding rule and break its elements
                    index = index_of_or(dataguide[sink_state_line],rule_to_apply);
                    // input symbol and destination state (sink)
                    input_symbol = dataguide[sink_state_line][index+1];
                    string_copy(sink_state,get_state(dataguide[sink_state_line],index+2));
                    derived_line[derived_count] = input_symbol;
                    derived_count++;
                     /* This is the line to go to for the next rule.
                        this is again assuming that the lines appear in the same order as the
                        states are labeled
                     */
                    sink_state_line = atoi(sink_state)-1;
                    apply_number++;
                }
            }
            //finishing the new derived line putting the sink state, termination symbol
            // and termination character
            append(derived_line,sink_state,&derived_count);
            append(derived_line,"|e",&derived_count);
            derived_line[derived_count] = '\0';\
            string_copy(subgrammar[*subgrammar_index],derived_line);
            (*subgrammar_index)++;
        }
     }
     return subgrammar;
}


/*
  This function converts any grammar into a function.
  Recall that this function also converts rules that contains not only an input symbol, but also
  an input string to the corresponding state or series of states.

  The final result is always a NFA

  parameters:
       char** grammar:  a double array representing the grammar to be converted
       int grammar_index: the number of lines in such array.

  returns:
       fm<char>* result_fm: a NFA from the given grammar.
*/
fm<char>*
gen_fm_from_grammar(char** grammar, int grammar_index)
{
    fm<char> *result_fm = new fm<char>();
    //variables for the convention
    set<state> starts;
    set<state> finals;
    // we need a pool of states for input strings
    int state_pool = grammar_index + 1;
    for(int line =0; line < grammar_index; line++){
        char string_number[50];
        char string_symbol[50];

        int number_symbols;
        int number_digits;
        int i = 0;
        //Get the state number for the left side
        number_digits = get_state_length(grammar[line],i);
        for(int j = 0; j < number_digits; j++){
            i++;
            string_number[j] = grammar[line][j];
        }
        string_number[number_digits] = '\0';
        state *original_source = new state(atoi(string_number));
        state *s1 = original_source;
        /*
          making sure that the start state is the first row
        */
        if(line == 0) starts += *s1;
        i++;
        while((grammar[line][i]!='\0')&&(grammar[line][i]!='\n')){
             char input_symbol = grammar[line][i];
             if((input_symbol<'a')||(input_symbol>'z')){
                i++;
             }
             else{
                 number_symbols = get_symbol_length(grammar[line],i);
                 int sink_start_pos = i;
                 for(int j = 0; j < number_symbols; j++){
                      i++;
                      string_symbol[j] = grammar[line][sink_start_pos + j];
                 }
                 string_symbol[number_symbols] = '\0';
                 if(number_symbols>1){
                    int source_label = s1->value();
```

```
                    int sink_label    = state_pool;
                    for(int letter_num = 0; letter_num<number_symbols-1; letter_num++){
                         sink_label    += letter_num;
                         state *s2temp = new state(sink_label);
                         state *s1temp = new state(source_label);
                         inst<char> *t = new inst<char>();
                         t->assign(*s1temp,string_symbol[letter_num],*s2temp);
                         result_fm->add_instruction(*t);
                         source_label = sink_label;
                    }
                    s1 = new state(source_label);
                    state_pool++;
                }
                else if(string_symbol[number_symbols-1] ==’e’){
                     finals += *original_source;
                     continue;
                }
                /*
                   This Code is to find the corresponding state number
                    even is more that one digit.
               */
               number_digits = get_state_length(grammar[line],i);
               sink_start_pos = i;
               for(int k = 0; k < number_digits; k++){
                   i++;
                   string_number[k] = grammar[line][sink_start_pos + k];
               }
               string_number[number_digits] = ’\0’;
               int state_num = atoi(string_number);
               state *s2 = new state(state_num);
               // Once source and sink state have been assigned and created
               // A transition function will be created and added to the automata
               inst<char> *t = new inst<char>();
               t->assign(*s1,string_symbol[number_symbols-1],*s2);
               result_fm->add_instruction(*t);
            }
        }
}
    result_fm->set_starts(starts);
    result_fm->set_finals(finals);

    return result_fm;
}


int
main(int argc, char** argv)
{
    char** subgrammar;
    int subgrammar_index = 0;
    fm<char>* view = new fm<char>;
    // Initializing the array of values...
    char **dataguide = (char **)malloc(50 * sizeof(char *));
    for(int i =  0; i < 50; i++)
        dataguide[i] = (char *)malloc(50 * sizeof(char));
    //This part reads the files and puts them into fm
    fstream    f_arg;
    fstream f_view;

    //keeps count on the number of lines
    int dataguide_line_count = 0;

    int g = atoi(args[1]);
    char[50] temp;
    sprintf(temp,"views\%i",g );
    f_arg.open("grammar_1", ios::in);
    f_view.open(temp,ios::out);

    //populate dataguide
    while(!f_arg.eof()){
       f_arg.getline(dataguide[dataguide_line_count], 90);
       dataguide_line_count++;
    }
    // do this for each line
    f_arg.close();

    int subgrammar_count = 0;;
    subgrammar = gen_subgrammar(dataguide_line_count,dataguide, &subgrammar_count);

    view = gen_fm_from_grammar(subgrammar,subgrammar_count);
    f_view << (*view);
    f_view.close();

    //Free memory from the subgrammar
    for (int t=0; t<subgrammar_count; t++){
        free(subgrammar[t]);
    }
    free(subgrammar);
    return 1;
}
```

## 8.4 Regular Expression Generator

```
#include    "include.h"
#include    "lexical.h"
#include    <strstream.h>
#include    <fstream.h>
#include    <iostream.h>
#include    <stdlib.h>
#include    <string.h>

/*******************************************************************************
regular expression generator

- This Program takes no parameters. It generates random regular expression and stores
  them in a text file.

- compilation:
  %gcc reg_exp_generator.C -o reg_exp_generator.out    OR
  %CC reg_exp_generator.C -o reg_exp_generator.out
*******************************************************************************/



int MAX_QUERIES = 40;
int MAX_ITERATIONS = 1000;
int MAX_REG_EXP = 100;

/*
this function is weird. I should be able to do this with strcpy given
I will revise this later
*/
void
string_copy(char* derived , char* source){
    int i= 0;
    while(source[i]!='\0'){
        derived[i] = source[i];
        i++;
    }
    derived[i] = '\0';
}

int
strlen(char* input){
    int i = 0;
    while(input[i]!='\0')
        i++;
    return i;
}

/*
    Rules:
        0:    S -> S
        1:    S -> SS+
        2:    S -> SS.
        3:    S -> S*
    Possible other values of S:  0...39
*/

char*
generate_random_regexp()
{
    char* result = (char*)malloc(MAX_REG_EXP * sizeof(char));
    char* result_temp = (char*)malloc(MAX_REG_EXP * sizeof(char));
    // at least one time the rules will be applied
    int total_rules = rand()%2 + 1;
     int rand_number;
    int pivot;
    char symbol;
    char *temp = "";
    result[0] = 'S';
    result[1] = '\0';
    int r_len =1;
    int u;

    for(int i=0; i< total_rules;i++){
        pivot=0;
        while(result[pivot]!='\0'){
            if(result[pivot]=='S'){
                rand_number = rand()%3+1;
                switch(rand_number){
                    case 1: for(u=r_len; u>pivot; u--)  result[u+4] = result[u];
                             result[pivot+1] = ' ';
                             result[pivot+2] = 'S';
                             result[pivot+3] = ' ';
                             result[pivot+4] = '+';
                             pivot+=4;
                             r_len+=4;
                             break;
                    case 2: for(u=r_len; u>pivot; u--)  result[u+4] = result[u];
                             result[pivot+1] = ' ';
```

```
                            result[pivot+2] = 'S';
                            result[pivot+3] = ' ';
                            result[pivot+4] = '.';
                            pivot+=4;
                            r_len+=4;
                            break;
                    case 3: for(u=r_len; u>pivot; u--)  result[u+2] = result[u];
                            result[pivot+1] = ' ';
                            result[pivot+2] = '*';
                            pivot+=2;
                            r_len+=2;
                            break;
                }// end of switch
            }// end of if
            pivot++;
        }//end of while
    }// end of for loop

    // now assign the values for each S
    pivot = 0;
    while(result[pivot]!='\0'){
        if(result[pivot]=='S'){
            rand_number=rand()%MAX_QUERIES;
            sprintf(temp,"%i",rand_number);
            if(rand_number>9){
                for(int k=r_len;k>pivot;k--)
                    result[k+1] = result[k];
                result[pivot] = temp[0];
                result[pivot+1] = temp[1];
                r_len++;
            }
            else
            result[pivot]=temp[0];
        }
        pivot++;
    }
    return result;
}


int
main(int argc, char** argv)
{
    char * reg_exp="";

    fstream    f_argin;
    fstream    f_regexp;

    f_regexp.open("queries_reg_exp_small_test", ios::out);

    for(int g=0; g<MAX_ITERATIONS; g++){
        reg_exp = generate_random_regexp();
        cout << reg_exp << " " << strlen(reg_exp)<<"\n";
        if(g==(MAX_ITERATIONS-1))
            f_regexp << reg_exp;
        else
            f_regexp << reg_exp << "\n";
    }

    f_regexp.close();
    return 1;
}
```

## 8.5 Query Generator (A Automaton)

```
#include    "include.h"
#include    "lexical.h"
#include    <strstream.h>
#include    <fstream.h>
#include    <iostream.h>
#include    <stdlib.h>
#include    <string.h>

/*********************************************************************************************
query generator

- This Program takes 1 parameters. Such is the file name to be use for the resulting
query.  It also represents the postion of the regular expression to be use to generate
the query.  The output is a finite state machine written to a . It also takes 40 views to
be evaluated on that whose names are from parameter*40 = 120 to parameter*40 + 39.

- compilation:
  %CC query_generator.C -o query_generator.out

It requires C++ Grail libraries.
*********************************************************************************************/
int  MAX_QUERIES = 40;;
```

```
int MAX_REG_EXP = 100;
void
string_copy(char* derived , char* source){
    int i= 0;
    while(source[i]!='\0'){
        derived[i] = source[i];
        i++;
    }
    derived[i] = '\0';
}

int
strlen(char* input){
    int i = 0;
    while(input[i]!='\0')
        i++;
    return i;
}

/*
    This Function takes regular expression and generates
    the finite state machine that accepts its language.
*/
fm<char>
eval_reg_exp(char* temp, fm<char>** views)
{
    queue<int> S;
    int value;
    char* s_value = (char*)malloc(2*sizeof(char));
    char* temp2 = "";
    fm<char> fm_value1;
    fm<char> fm_value2;

    fm<char>* fm_array = (fm<char>*)malloc(500*sizeof(fm<char>));
    int* fm_index = (int*)malloc(500*sizeof(int));
    int index_count = 0;

    for(int i=0;i<= strlen(temp); i++){
        if((temp[i]==' ')||(temp[i]=='\0')){
            switch(temp[i-1]){
                case '*':  value = S.pop();
                           if(value>999) fm_value1 = fm_array[value-1000];
                           else fm_value1 = *views[value];
                           fm_value1.star();
                           fm_array[index_count] = fm_value1;
                           S.push(index_count + 1000);
                           index_count++;
                           break;
                case '+': value = S.pop();
                           if(value>999) fm_value1 = fm_array[value-1000];
                           else fm_value1 = *views[value];
                           value = S.pop();
                           if(value>999) fm_value2 = fm_array[value-1000];
                           else fm_value2 = *views[value];
                           fm_value1+=fm_value2;
                           fm_array[index_count] = fm_value1;
                           S.push(index_count + 1000);
                               index_count++;
                           break;
                case '.':  value = S.pop();
                           if(value>999) fm_value1 = fm_array[value-1000];
                           else fm_value1 = *views[value];
                           value = S.pop();
                           if(value>999) fm_value2 = fm_array[value-1000];
                           else fm_value2 = *views[value];
                           fm_value1^=fm_value2;
                           fm_array[index_count] = fm_value1;
                           S.push(index_count + 1000);
                               index_count++;
                           break;
                default: if(i==1) {
                                                    sprintf(temp2,"%c",temp[i-1]) ;
                             value = atoi(temp2);
                             S.push(value);
                         }
                            else{
                             if(temp[i-2]!=' '){
                                 s_value[1] = temp[i-1];
                                 s_value[0] = temp[i-2];
                                 S.push(atoi(s_value));
                             }
                             else{
                                 sprintf(temp2,"%c",temp[i-1]) ;
                                 value = atoi(temp2);
                                 S.push(value);
                             }
                         }
                         break;
            }
        }
    }
```

```
    fm<char> result = fm_array[index_count-1];
    result.reachable_fm();
    return result;
}


int
main(int argc, char** argv)
{
    char reg_exp[200];
    fm<char> result;

    fstream    f_argin;
    fstream    f_argout;

    fstream    f_regexp;
    char filename[40];

    //Obtain the number of iteration
    int g = atoi(argv[1]);

    // regular expressions
    fm<char>**fms = (fm<char>**)malloc(MAX_QUERIES * sizeof(fm<char>*));
    f_regexp.open("queries_reg_exp_small", ios::in);

    for(int b=0; b<g; b++)
        f_regexp.getline(reg_exp, 200);

    for(int i=0; i<MAX_QUERIES; i++){
        int view_number = MAX_QUERIES*g + i;
        sprintf(filename, "views/%i",view_number);
        f_argin.open(filename, ios::in);
        fms[i] = new fm<char>;
        f_argin>>(*fms[i]);
        f_argin.close();
        fms[i]->reachable_fm();
    }

    f_regexp.getline(reg_exp, 200);
    result = eval_reg_exp(reg_exp, fms);
    sprintf(filename, "queries_small2/%i",g);
    f_argout.open(filename, ios::out);
    f_argout << result;
    f_argout.close();
    f_regexp.close();

    for(int u=0; u<MAX_QUERIES; u++){
        free(fms[u]);
    }
    free(fms);
        cout<< reg_exp<<"\n";
    cout << "Done for reg_exp: "<<g << "\n";
    return 1;
}
```

## 8.6 Rewriting Generator (B Automaton)

```
#include    "include.h"
#include     "lexical.h"
#include    <strstream.h>
#include    <fstream.h>
#include    <iostream.h>
#include    <stdlib.h>
#include    <string.h>

/*********************************************************************************************
Rewriting Generator

- This Program takes one parameter.  This parameter represents the set of views to be used for
  generating the rewriting.
- Assumptions: At this point:
  1.- Views have been already generated.
  2.- Queries have been already generated.
- Views and Queries are already stored in separated directories (look at the program for more
  info).
- The numbering:
     For the numbering the parameter is consider:
     e.g.
     % rewriting_generator.out 3
     that means:
     a.  The views to use are 40 (predefined): whose names are from 3*40 = 120 to 159
     b.  The query to use is the one shown as parameter: 3 (it is generated specifically
         for this set of views).
     c.  The output (rewriting NFA) will be stored in a specific location under the filename
         that is shown as parameter (in this case 3)
compilation:
     %CC - compat rewriting_generator.C -o rewriting_generator.out
It requires CC to compiled it and grail libraries.
```

```
*******************************************************************************/
int  MAX_QUERIES =40;
/*
     this function takes the following parameters:
    - map which is a 2 dimensional array made up by pairs of states.
     - count: number of pairs present in the array
     - num1 and num2: pair to be found
     it returns  the value stored in the location row = num1, col= num2
*/
int
get_state_from_map(int*** map, int *count, state num1, state num2)
{
     if ((*map)[num1.value()][num2.value()] == -1){
          (*map)[num1.value()][num2.value()] = *count;
          (*count)++;
     }

     return (*map)[num1.value()][num2.value()];
}

/*
     Produces the cartesian product of two automatas
     parameters:
     - query, view: two automatas
     - map_count: # of pairs in the map for storing pair of states
     - map: the actual container for pairs
*/
fm<char>*
cartesian_product(fm<char> query, fm<char> view, int *map_count, int*** map)
{
     state q_source, q_sink;
     state  v_source, v_sink;
     state  c_source, c_sink;
     char symbol;
     fm<char> *c_fm= new fm<char>();
     inst<char> q_instr;
     inst<char> v_instr;
     inst<char> c_instr;
     int new_source;
     int new_sink;
     (*map_count) = 0;
     for(int q_arc=0; q_arc<query.number_of_instructions();q_arc++){
          q_instr = query[q_arc];
          q_source = q_instr.get_source();
          q_sink = q_instr.get_sink();
          symbol = q_instr.get_label();

          for(int v_line=0; v_line<view.number_of_instructions(); v_line++){
               v_instr = view[v_line];
               if(symbol==v_instr.get_label()){
                    v_source= v_instr.get_source();
                    v_sink= v_instr.get_sink();;
                    c_source= get_state_from_map(map, map_count, q_source, v_source);
                    c_sink= get_state_from_map(map, map_count, q_sink, v_sink);
                    c_instr.assign(c_source,symbol ,c_sink);
                   c_fm->add_instruction(c_instr);
               }
          }
     }
     return c_fm;
}

/*
     This function simply initializes a two dimensional array.
     parameters:
     - map: two dimensional array made up by pair of states
     - rows: number of rows on the matrix
     - columns: same.
*/
void
init_map(int ***map, int rows, int columns)
{
     (*map) = (int**)malloc(rows*sizeof(int*));

     for(int m=0; m<rows; m++)
          (*map)[m] = (int*)malloc(columns*sizeof(int));

     for(int i=0; i<rows; i++)
          for(int j=0; j<columns; j++)
               (*map)[i][j] = -1;
}
int
main(int argc, char** argv)
{
     fm<char>  query;
     fm<char> view;
     fm<char> *c_fm;
     fm<char> *result_fm;;
     fm<char> *temp_fm;

     int**  map;
```

```
int map_count;
fstream     f_query;
fstream     f_view;
fstream     f_rewriting;

char file_name[100];
char temp[100];

state s1,s2;

set<state> c_starts;
set<state> c_finals;;
set<state> v_starts;
set<state> v_finals;

inst<char> t;

// this is the number of rewriting that will be performed
int g= atoi(argv[1]);

//RETRIEVAL PHASE
//----------------
sprintf(file_name,"queries_small2/%i",g);
f_query.open(file_name, ios::in);
query.add_file(f_query);
f_query.close();
query.complement();
int map_line_max = query.max_state().value() + 1;
char letter;
int new_state;
int myInteger;
result_fm = new fm<char>(query);
result_fm->clear_arcs();
for(int f =0;f<MAX_QUERIES;f++){
     // The view is loaded into a nfa based on grail implementation
     myInteger = g*MAX_QUERIES + f;
     sprintf(file_name, "views/%i", myInteger);
     f_view.open(file_name, ios::in);
     view.add_file(f_view);
     f_view.close();
     init_map(&map,map_line_max,100);
     map_count =0;

     c_fm = cartesian_product(query, view,&map_count ,&map);
     // Simply chaging the intial and final states of the cartesian product
     // according with every pair of the query and find non emptyness
     // to ensure non-empty intersection.
     for(int i=0; i<query.number_of_states(); i++){
          for(int j=0; j<query.number_of_states(); j++){
               temp_fm = new fm<char>(*c_fm);
               v_starts = view.starts(v_starts);
               c_starts.clear();
               for(int h=0; h<v_starts.size(); h++){
                    new_state = map[i][v_starts[h].value()];
                    if(new_state>-1){
                         s1 = new_state;
                        c_starts += s1;
                    }
               }
               temp_fm->set_starts(c_starts);
               v_finals = view.finals(v_finals);
               c_finals.clear();
               for(int a=0; a<v_finals.size();a++){
                    new_state = map[j][v_finals[a].value()];
                    if(new_state>-1){
                         s2 = new_state;
                             c_finals += s2;
                    }
               }
               temp_fm->set_finals(c_finals);
               temp_fm->reachable_fm();
               if(temp_fm->number_of_final_states()){
                    s1 = i;
                    s2 = j;
                    letter = 'A' + f;
                    if (letter>90) letter = 'a' + letter - 90;
                    t.assign(s1,letter,s2);
                    result_fm->add_instruction(t);
               }
               free(temp_fm);
          }
     }
     free(c_fm);
}
//reachability to get rid of trash states.
result_fm->reachable_fm();
sprintf(file_name, "rewritings_small/%i", g);
f_rewriting.open(file_name, ios::out);
f_rewriting << (*result_fm);
f_rewriting.close();
cout << "Done for Query: " << g << "\n";
```

```
     // freeing memory
     for(int y=0; y<map_line_max; y++)
          free(map[y]);
     free(map);
     free(result_fm);
     return 1;
}
```

## 8.7 ViewGraph Generator

```
#include    <strstream.h>
#include    <fstream.h>
#include    <iostream.h>
#include    <stdlib.h>
#include    <map>
/*********************************************************************************************
viewgraph Generator

- This Program takes one parameter.  This parameter represents the set of views to be used for
  generating the viewgraph.
- Assumptions: At this point:
  1.- Views have been already generated.
  2.- Database have been already generated.
- Views are already stored in separated directories (look at the program for more
  info).
- The numbering:
     For the numbering the parameter is consider:
     e.g.
     % viewgraph_generator.out 3
     that means:
     a.  The views to use are 40 (predefined): whose names are from 3*40 = 120 to 159
     c.  The output (Viewgraph) will be stored in a specific location under the filename
         that is shown as parameter (in this case 3)
compilation:
     %CC rewriting_generator.C -o rewriting_generator.out
 %g++ rewriting_generator.C -o rewriting_generator.out
*********************************************************************************************/
using namespace std;

int  MAX_QUERIES = 40;
int  MAX_ITERATION = 1000;

int
get_state_length(char *str, int pos){
   int i = 0;
   while((str[pos]>='0')&&(str[pos]<='9')){
      pos++;
      i++;
   }
   return i;
}

int
get_symbol_length(char *str, int pos){
   int i = 0;
   while((str[pos]>='a')&&(str[pos]<='z')){
      pos++;
      i++;
   }
   return i;
}


//returns the state number appearing in some position
char *
get_state(char* str, int index){
        int number_digits = get_state_length(str,index);
     char str_state[20];

     for(int j = 0; j < number_digits; j++){
          str_state[j] = str[index];
          index++;
     }
     str_state[number_digits] = 0;
     return str_state;

}

//append an string to another string on a specified position
void
append(char* derived,char* source, int* count){
     int i= 0;
     while(source[i]!='\0'){
          derived[(*count)] = source[i];
          i++;
          (*count)++;
     }
     derived[*count] = '\0';
```

```
}
// this function is weird. I should be able to do this with strcpy given
// I will revise this later

void
string_copy(char* derived , char* source){
     int i= 0;
     while(source[i]!='\0'){
          derived[i] = source[i];
          i++;
     }
     derived[i] = '\0';
}

int
get_source_state(char *pool)
{
     char* c_state = get_state(pool,0);
     int state = atoi(c_state);
     return state;
}

//this function finds the sink state on a object-edge-object table
// this state is located on the right hand side from the same line
int
get_sink_state(char *pool)
{
     int found = 0;
     int i=0;
     while(found<2){
          if(pool[i]== ' ') found++;
          i++;
     }
     char* c_state = get_state(pool,i);
     int state = atoi(c_state);
     return state;
}
int
is_final_info(char* temp)
{
     int i =0;
     while(temp[i]!=0){
          if(temp[i]=='F')
               return 1;
          i++;
     }
     return 0;
}

int**
get_pairs(char **matrix, int count)
{
     int** answer = (int**)malloc(count*sizeof(int*));
     for(int i=0; i<count; i++){
          answer[i] = (int*)malloc(2*sizeof(int));
     }
     for(int j=0; j<count; j++){
          answer[j][0] = get_source_state(matrix[j]);
          answer[j][1] = get_sink_state(matrix[j]);
     }
     return answer;
}

// This two functions are to fill the matrix.
char*
get_symbols(char **matrix, int count)
{
     char* answer = (char*)malloc(count*sizeof(char));
     int h;
     for(int i =0; i<count;i++){
          h=0;
          while(matrix[i][h]!=' ')
               h++;
          answer[i]=matrix[i][h+1];
     }
     return answer;
}

int
is_there(int key, map<int,int> m)
{
     map<int,int>::iterator it;
     it = m.find(key);
     if(it != m.end())
          return 1;
     return 0;
}

int
is_final(int num, int*nums, int nums_count)
```

```
{
     for(int i=0; i<nums_count;i++){
          if(nums[i]==num)
               return 1;
     }
     return 0;
}

/*
     prerequisite for this function to work:
      - fma should already been sorted.
*/
int
get_state_count(char** fma, int count)
{
     int answer= 1;
     int dummy = 0;
     int pivot = get_source_state(fma[0]);
     while(dummy<count){
          int cur_state = get_source_state(fma[dummy]);
          if( cur_state!=pivot){
               pivot = cur_state;
               answer++;
          }
          dummy++;
     }
     return answer;
}

/*this function takes the following parameters:
     char** DB :            database array
     int      DB_count :    # of lines on the database
     char **  view :          view array
     int view_count :       # of lines on the view
     int* final states:     contains the final states for the view
     int  final count:      #of final states on the view
     int VG_count:          number of lines on the viewGraph
                            (this number will be updated by the algorithm)
     int view_index:        numeric index for the view being used.
*/
void
gen_ViewGraph(char **DB, int DB_count,  char **view, int view_count,
              int * final_states, int final_count, char*** ViewGraph,
                   int *VG_count, int view_index)
{
     char temp[50];
     int line_pos;
     //-------------------------------------------------------------------------------------------
     int** DB_pairs = get_pairs(DB, DB_count);
     int** view_pairs = get_pairs(view, view_count);
     char * DB_symbols = get_symbols(DB, DB_count);
     char * view_symbols = get_symbols(view, view_count);
     //-------------------------------------------------------------------------------------------
     map<int, int> queue;
     map<int, int>reach;
     map<int, int> out;
     map<int, int>::iterator it;
     //-------------------------------------------------------------------------------------------
     int db_source_state;
     int db_sink_state;
     int view_source_state;
     int view_sink_state;
     char symbol;

     //-------------------- ALGORITHM STARTS HERE --------------------------------
     int DB_state_count = get_state_count(DB,DB_count);
     for(int p=0; p<DB_state_count; p++){
          int first_time=1;
          // just clearing the information
          queue.clear();
          reach.clear();
          queue[p]=view_pairs[0][0];
          while(!queue.empty()){
               db_source_state = (*queue.begin()).first;
               view_source_state = (*queue.begin()).second;
               reach[db_source_state]=view_source_state;
               if((is_final(view_source_state,final_states, final_count))&&(!first_time)){
                    out[p] = db_source_state ;
               }
               first_time=0;
               queue.erase(queue.begin());
               //I already have the state and object to find their transitions
               // Assuming the database is already sorted
               int i =0;
               while((i<DB_count)&&(DB_pairs[i][0]!=p)) i++;
               while((i<DB_count)&&(DB_pairs[i][0]==p)){
                    symbol = DB_symbols[i];
                    db_sink_state = DB_pairs[i][1];
                    for(int m=0;m<view_count;m++){
                         if((view_pairs[m][0]==view_source_state)&&(view_symbols[m]==symbol)){
                              view_sink_state=view_pairs[m][1];
```

```
                              //There is a match, make sure is not already there
                              if((!is_there(db_sink_state,queue))&&
                                 (!is_there(db_sink_state,reach))){
                                   queue[db_sink_state] = view_sink_state;
                              }
                         }
                    }
                    i++;
               }
          }
     }

     // this is to add to the final output the states that reached final together with its originator
     for( it = out.begin(); it != out.end(); it++ ){
          line_pos=0;
          sprintf(temp,"%d",(*it).first);
          append((*ViewGraph)[*VG_count],temp,&line_pos);
          append((*ViewGraph)[*VG_count]," ",&line_pos);
          char letter = 'A' + view_index;;
          if (letter>90) letter = ('a' - 1) + letter - 90;
          sprintf(temp,"%c",letter);
          append((*ViewGraph)[*VG_count],temp,&line_pos);
          append((*ViewGraph)[*VG_count]," ",&line_pos);
          sprintf(temp,"%d",(*it).second);
          append((*ViewGraph)[*VG_count],temp,&line_pos);
          (*VG_count)++;
     }

     //Clean the mess
     for (int c1=0; c1<DB_count; c1++)
             free(DB_pairs[c1]);
     free(DB_pairs);

     for (int c2=0; c2<view_count; c2++)
        free(view_pairs[c2]);
     free(view_pairs);

     free(DB_symbols);
     free(view_symbols);
}
int
main(int argc, char** argv)
{
     char** DB;
     char** View;
     char** ViewGraph;
     //keeps count on the number of lines
     int DB_line_count = 0;
     int View_line_count = 0;
     int VG_count = 0;

     // Initializing the array of values...
     DB = (char **)malloc(10000 * sizeof(char *));
     for(int k =  0; k < 10000; k++)
          DB[k] = (char *)malloc(50 * sizeof(char));

     //This part reads/writes the files and puts them into their respective arrays
     fstream      f_DBin;
     fstream     f_Viewin;
     fstream       f_argout;

     //more variables
     char * temp;
     int * final_states;
     int final_states_count;
     char file_name[100];

     //populate database array
     f_DBin.open("mini_database_1000", ios::in);
     while(!f_DBin.eof()){
          f_DBin.getline(DB[DB_line_count], 90);
          DB_line_count++;
     }
     f_DBin.close();
     int g = atoi(argv[1]);
     ViewGraph = (char **)malloc(50000 * sizeof(char *));

     for(int k =  0; k < 50000; k++)
          ViewGraph[k] = (char *)malloc(50 * sizeof(char));
     VG_count=0;

     for(int view_index =0; view_index<MAX_QUERIES; view_index++){
          View_line_count = 0;
          // Initializing the array of values...
          View = (char **)malloc(500 * sizeof(char *));
          for(int i =  0; i < 500; i++)
               View[i] = (char *)malloc(50 * sizeof(char));

          temp = (char*)malloc(90*sizeof(char));
          final_states = (int*)malloc(1000*sizeof(int));
          final_states_count=0;
```

```
          //populate View array. Notice here that there is a specific format for populating the view
          int myInteger = g*MAX_QUERIES + view_index;
          sprintf(file_name, "views/%i", myInteger);
          f_Viewin.open(file_name, ios::in);

          f_Viewin.getline(temp, 90);
          f_Viewin.getline(temp, 90);

          while(!is_final_info(temp)){
               string_copy(View[View_line_count],temp);
               View_line_count++;
               f_Viewin.getline(temp, 90);
          }

          final_states = (int*)malloc(View_line_count*sizeof(int));
          final_states[final_states_count] = get_source_state(temp);
          final_states_count++;

          while(!f_Viewin.eof()){
               f_Viewin.getline(temp, 90);
               final_states[final_states_count] = get_source_state(temp);
               final_states_count++;
          }
          f_Viewin.close();
          gen_ViewGraph(DB, DB_line_count, View, View_line_count,final_states,final_states_count,
                                    &ViewGraph,&VG_count,view_index);
          //Free memory from the view
          for (int b=0; b<500; b++)
                 free(View[b]);
          free(View);
     }
     sprintf(file_name, "mini_viewgraphs_1000/%i",g);
     f_argout.open(file_name, ios::out);

     //writing on the File the database lines
     for(int h=0;h<VG_count-1;h++){
          f_argout << (ViewGraph[h]) << "\n";
     }
     f_argout << (ViewGraph[VG_count-1]);
     f_argout.close();

     //Free memory from the ViewGraph
    for (int c=0; c<50000; c++)
       free(ViewGraph[c]);
    free(ViewGraph);
    //Free memory from the DB
    for (int a=0; a<50000; a++)
       free(DB[a]);
    free(DB);
    return 1;
}
```

## 8.8 Second Complement (C Automaton)

```
#include    "include.h"
#include     "lexical.h"
#include     <strstream.h>
#include    <fstream.h>
#include    <iostream.h>
#include    <stdlib.h>
#include    <string.h>
#include      <time.h>
/*******************************************************************************
Second complement

- This program takes as parameter the name of the file belonging to the rewriting to be
  used to get the second complementation and stores is into another folder but with the same
  filename. it also records the time into in a file.

- to run it:  %second_complement.out 3
- to compile it:  CC -compat second_complement.C -o second_complement.out
*******************************************************************************/
int
main(int argc, char** argv)
{
     int first_complement;
     int second_complement;

     time_t t1,t2;
     double  duration;
     //This part reads/writes the files and puts them into fm
     fstream  f_argin;
     fstream  f_argout;
     char file_name[100];
     char seconds[100];
     //Obtain the number of iteration
```

```
     int g = atoi(argv[1]);

     //REWRITING PHASE
     //----------------
     sprintf(file_name, "rewritings_small/%i",g);
     f_argin.open(file_name, ios::in);
     fm<char> query;
     f_argin>>query;

     first_complement = query.number_of_states();

     time(&t1);
     query.complement();
     time(&t2);
     sprintf (seconds,"%d\n",(t2-t1) );

     query.reachable_fm();
     second_complement = query.number_of_states();

     f_argout.open("complement_time_test_praga", ios::out|ios::ate|ios::app);
     f_argout <<g<<"\t"<<first_complement<<"\t"<<second_complement<<"\t"<<seconds<<"\n";
     cout <<g<<"\t"<<first_complement<<"\t"<<second_complement<<"\t"<<seconds<<"\n";
     f_argout.close();

     fstream f_rewriting;
     sprintf(file_name, "second_complement_small/%i",g);
     f_rewriting.open(file_name, ios::out);
     f_rewriting<<query;

     f_rewriting.close();

    return 1;
}
```

## 8.9 NFA vs ViewGraph

```
#include    <fstream.h>
#include    <iostream.h>
#include    <stdlib.h>
#include    <map>
#include    <set>
#include    <math.h>
#include    <queue>
#include    <string>
#include    <time.h>


/*****************************************************************************************
AFA vs ViewGraph evaluation

- This Program takes one parameter.  This parameter is the name of the file that contains
  the Viewgraph, and the AFA.  These are two separate files, but they should have the same
  name.  e.g.
           % afa_vd_viewgraph.out 0
- 0 in this case is the name of the viewgraph file and also the name of the corresponding
  afa file.  Both are in different directories. it also stores the time to process it into
  a file

compilation:
     %gcc afa_vs_viewgraph.C -o afa_vs_viewgraph.out  OR
     %CC afa_vs_viewgraph.C -o afa_vs_viewgraph.out
*****************************************************************************************/

using namespace std;

int MAX_QUERIES = 40;

/*
This function returns the lenght of a state. Use only by the function get_state
*/
int
get_state_length(char *str, int pos){
   int i = 0;
   while((str[pos]>='0')&&(str[pos]<='9')){
        pos++;
           i++;
   }
   return i;
}
int
get_symbol_length(char *str, int pos){
   int i = 0;
   while((str[pos]>='a')&&(str[pos]<='z')){
        pos++;
           i++;
   }
   return i;
```

```
}
/*
returns the state number appearing in some position
*/
char *
get_state(char* str, int index){
        int number_digits = get_state_length(str,index);
     char* str_state = (char*)malloc(20*sizeof(char));

     for(int j = 0; j < number_digits; j++){
          str_state[j] = str[index];
          index++;
     }
     str_state[number_digits] = 0;
     return str_state;

}

/*
append an string to another string on a specified position
*/
void
append(char* derived,char* source, int* count){
     int i= 0;
     while(source[i]!='\0'){
          derived[(*count)] = source[i];
          i++;
          (*count)++;
     }
     derived[*count] = '\0';
}

void
string_copy(char* derived , char* source){
     int i= 0;
     while(source[i]!='\0'){
          derived[i] = source[i];
          i++;
     }
     derived[i] = '\0';
}

/*
     this function finds the source state on a object-edge-object table
*/
int
get_source_state(char *pool)
{
     char* c_state = get_state(pool,0);
     int state = atoi(c_state);
     return state;
}

/*
     this function finds the sink state on a object-edge-object table
*/
int
get_sink_state(char *pool)
{
     int found = 0;
     int i=0;
        while(found<2){
          if(pool[i]== ' ') found++;
          i++;
     }
    char* c_state = get_state(pool,i);
     int state = atoi(c_state);
     return state;
}

/*
Determines if a line is a Final state or not (checking on the File)
*/
int
is_final_info(char* temp)
{
     int i =0;
     while(temp[i]!=0){
          if((temp[i]=='(')&&(temp[i+1]=='F'))
               return 1;
          if(temp[i]!=0) i++;
     }
     return 0;
}
/*
     This function get just the source and sink states for every transition
     input - matrix: it contains all the transition
             - count:  number of lines in the matrix
     output: another matrix
*/
int**
```

```
get_pairs(char **matrix, int count)
{
     int** answer = (int**)malloc(count*sizeof(int*));
     for(int i=0; i<count; i++){
          answer[i] = (int*)malloc(2*sizeof(int));
     }

     for(int j=0; j<count; j++){
          answer[j][0] = get_source_state(matrix[j]);
          answer[j][1] = get_sink_state(matrix[j]);
     }
     return answer;
}
// This two functions are to fill the matrix.
char*
get_symbols(char **matrix, int count)
{
     char* answer = (char*)malloc(count*sizeof(char));
     int h;
     for(int i =0; i<count;i++){
          h=0;
          while(matrix[i][h]!=' ')
               h++;
          answer[i]=matrix[i][h+1];
     }
     return answer;
}
//  Just to see if the a key is already present in a map
int
is_there(int key, map<int,int> m)
{
     map<int,int>::iterator it;
     it = m.find(key);
     if(it != m.end())
          return 1;
     return 0;
}
int
is_there(string key, map<string,int> m)
{
     map<string,int>::iterator it;
     it = m.find(key);
     if(it != m.end())
          return 1;
     return 0;
}

int
is_there(int key, int* array, int size)
{
     for(int i=0; i<size; i++){
          if(array[i]==key)
               return 1;
     }
     return 0;
}
// Check if a intenger is present in a vector with nums_count elements
int
is_final(int num, int*nums, int nums_count)
{
     for(int i=0; i<nums_count;i++){
          if(nums[i]==num)
               return 1;
     }
     return 0;
}
/*
     this function returns the number of states present
     input: - a matrix of transition
            - number of transition in such matrix
     output: max State.
*/
int
get_state_count(char** fma, int count)
{
     int max = 0;
     for(int i=0; i<count; i++){
          if(get_source_state(fma[i])>max){
               max = get_source_state(fma[i]);
          }
     }
     for(int g=0; g<count; g++){
          if(get_sink_state(fma[g])>max){
               max = get_sink_state(fma[g]);
          }
     }
     return max+1;
}
char get_symbol(char *line){
     int h =0;
     while(line[h]!=' ')
```

```
          h++;
     return line[h+1];
}
char**
init_matrix(int row_count, int col_count){
     char** result = (char**)malloc((row_count)*sizeof(char*));
     for(int g=0; g<row_count; g++)
          result[g] = (char*)malloc((col_count)*sizeof(char));
     return result;
}
int**
init_table(int row_count, int col_count){
     int** result = (int**)malloc((row_count)*sizeof(int*));
     for(int g=0; g<row_count; g++)
          result[g] = (int*)malloc((col_count)*sizeof(int));
     return result;
}
void
free_matrix(char **matrix, int count){
     for(int i=0; i<count; i++){
          free(matrix[i]);
     }
     free(matrix);
}
void
print_table(int** matrix, int count){
     cout << "\n";
     for(int i=0; i<count; i++){
          int g=0;
          while(matrix[i][g]!=-99){
               cout<<matrix[i][g]<< " ";
               g++;
          }
          cout << "\n";
     }
}
void
print_matrix(char** matrix, int count){
     cout << "\n";
     for(int i=0; i<count; i++){
          cout<<  i << " :"<<matrix[i]<< "\n";
     }
}
/*
     Merge Sort implemented
     input: - ordered vector left
             - ordered vector right
             - size of the left vector
             - size of the right vector
     output: the ordered merged of these two vectors
*/
char**
merge(char **left,char **right, int l_size, int r_size){
      char** result = init_matrix(l_size+r_size,MAX_QUERIES+2);
    int l_pivot = 0;
    int r_pivot = 0;
    int u_pivot = 0;

    while((l_pivot<l_size) && (r_pivot<r_size)){
          int state_left = get_source_state(left[l_pivot]);
          int state_right = get_source_state(right[r_pivot]);
          char symbol_left = get_symbol(left[l_pivot]);
          char symbol_right = get_symbol(right[r_pivot]);
          if((state_left<=state_right)||
          ((state_left==state_right)&&(symbol_left<symbol_right))){
              string_copy(result[u_pivot],left[l_pivot]);
              u_pivot++;
              l_pivot++;
            }
        else{
            string_copy(result[u_pivot],right[r_pivot]);
            u_pivot++;
            r_pivot++;
          }
      }
      for(int k=l_pivot; k<l_size; k++){
         string_copy(result[u_pivot],left[k]);
         u_pivot++;
      }
      for(int t=r_pivot; t<r_size; t++){
           string_copy(result[u_pivot],right[t]);
           u_pivot++;
      }
      return result;
}
/*
     Merge Sort Implemented
     input: - input: list to order (matrix of transitions)
             - size: number of transitions to check

     output: the ordered matrix of transtions (by source state only)
```

```
*/
char**
merge_sort(char ** input, int size)
{
     char ** list;
     int middle;
     char** left = init_matrix(size,MAX_QUERIES+2);
     char** right = init_matrix(size,MAX_QUERIES+2);
     if (size == 1) return input;
     middle = size / 2;
     for (int i=0; i< middle; i++){
        string_copy(left[i],input[i]);
     }
     for (int j=middle; j< size; j++){
        string_copy(right[j-middle],input[j]);
     }
     left = merge_sort(left,middle);
     right = merge_sort(right,size-middle);
     list = merge(left, right, middle, size-middle);
     //Here use all the destructors
     free_matrix(left,middle);
     free_matrix(right,size-middle);
     return list;
}
/*
     This function takes the following parameters:
     char** nfa: simply the automata
     int afa_count: # of lines in the automata

     It returns the same automata but inverted, that is all the transitions inverted plus final states being
     starts and the starts beign final
*/

char**
inv_afa(char ** nfa, int afa_count){
     int source, sink, line_pos;
     char symbol;
     char** inv_result = init_matrix(afa_count,50);
     char temp[50];
     for(int i=0; i<afa_count; i++){
          line_pos = 0;
          source = get_source_state(nfa[i]);
          sink = get_sink_state(nfa[i]);
          symbol = get_symbol(nfa[i]);
          sprintf(temp,"%d",sink);
          append(inv_result[i],temp,&line_pos);
          append(inv_result[i]," ",&line_pos);
          sprintf(temp,"%c",symbol);
          append(inv_result[i],temp,&line_pos);
          append(inv_result[i]," ",&line_pos);
          sprintf(temp,"%d",source);
          append(inv_result[i],temp,&line_pos);
     }
    char** result = merge_sort(inv_result,afa_count);
    for(int a=0; a<afa_count; a++)
         free(inv_result[a]);
    free(inv_result);
     return result;
}
int
get_view_pos(char symbol){
     int view_pos;
     if (symbol>90) view_pos = symbol - 'A' - 7;
     else view_pos = symbol - 'A';
     return view_pos;
}
/*
     This function takes the following parameters
     char** afa array: in rough condition (only the list of transtiions)
     int afa_count   :number of lines for such transtions

     this function outputs an afa table in vertical order in the following way
     for each line: first cell is the input symbol
                         next cells will be the sink states followed by the -99 integer
*/
int **
get_afa_table(char ** nfa, int afa_count)
{
     int states_num = get_state_count(nfa,afa_count);
     int count = states_num*MAX_QUERIES;
     int** answer = init_table(count,2*MAX_QUERIES);
     char **afa = inv_afa(nfa, afa_count);
     int pos;// traverse the afa column to column
     int matched; // This boolean checks that there is a transtion from a state with a symbol
     int cur_state; // making sure all the states are being treated
     int view_pos;// product from converting a letter into a view
     int line_state; // state present as source on each line
     int cur_line=0;// line of afa being treated
     for(int k=0; k<MAX_QUERIES; k++){
          matched = 0;
          cur_state = -1;
```

```
          for(int cur_line=0; cur_line<afa_count; cur_line++){
             view_pos = get_view_pos(get_symbol(afa[cur_line]));
              if(view_pos==k){
                    matched = 1;
                    line_state = get_source_state(afa[cur_line]);
                    // they are the same, Just append the new sink state
                    if(line_state==cur_state){
                         int sink_state = get_sink_state(afa[cur_line]);
                         // add only if such sink is not already there
                         if(!is_there(sink_state,answer[k*states_num + cur_state],pos)){
                              answer[k*states_num + cur_state][pos] = sink_state;
                              pos++;
                              answer[k*states_num + cur_state][pos] = -99;
                         }
                    }
                    else{
                         // there are states in between not being treated
                         if(line_state>cur_state+1){
                              for(int h=cur_state ; h<line_state-1; h++){
                                   cur_state++;
                                   pos = 0;
                                   answer[k*states_num + cur_state][pos]= cur_state;
                                   pos++;
                                   answer[k*states_num + cur_state][pos] = -1;
                                   pos++;
                                   answer[k*states_num + cur_state][pos] = -99;
                              }
                         }
                         // Adding a new state
                         cur_state++;
                         pos=0;
                         answer[k*states_num + cur_state][pos] = cur_state;
                         pos++;
                         answer[k*states_num + cur_state][pos] = get_sink_state(afa[cur_line]);
                         pos++;
                         answer[k*states_num + cur_state][pos] = -99;
                    }// end of else
               }// end of matching
          }//end of for each line
          while(cur_state<states_num-1){
               cur_state++;
               pos = 0;
               answer[k*states_num + cur_state][pos]= cur_state;
               pos++;
               answer[k*states_num + cur_state][pos] = -1;
               pos++;
               answer[k*states_num + cur_state][pos] = -99;
          }
          // Checking if the letter is even present in the afa
          if(!matched){
               for(int p = 0; p < states_num ; p++){
                    pos = 0;
                    answer[k*states_num + p][pos] = p;
                    pos++;
                    answer[k*states_num + p][pos] = -1;
                    pos++;
                    answer[k*states_num + p][pos] = -99;
               }// end of for
          }//end of not matching
     }//end of for each view
     for(int a=0; a<afa_count; a++)
         free(afa[a]);
     free(afa);
     return answer;
}
/*
     this function takes the following parameters:
        int value: bit vector in decimal form
        int total_bits

     This function simply prints th bit_vector
     It is used mainly for debuging purposes
*/
void
print_bit_vector(int value, int total_bits)
{
     int * bit_vector = (int *)malloc(total_bits * sizeof(int));
     for(int j=0;j<total_bits;j++){
     bit_vector[j] = value%2;
          value = value/2;
     }
     for(int i=total_bits-1; i>=0; i--){
          cout<< bit_vector[i];
     }
     free(bit_vector);
}
void
print_bit_vector(unsigned long value, int total_bits)
{
     unsigned long mask;
     for(int i=total_bits-1; i>=0; i--){
```

```
          mask = 1L<<i;
          if((value&mask)>0)
               cout<< 1;
          else
               cout<<0;
     }
}
/*
     This function takes the following parameters:
     int prev_vector       :     prev_vector to use
     char input symbol : the one that proceeds to the transition
     char **  afa_table:  table to use for the vector generator

     It returns the result from evaluating prev_vector on the afa_table
*/
int get_afa_vector(int prev_vector, char input, int** afa_table, int afa_states)
{
     int count = MAX_QUERIES*afa_states;
     int i =0;
     int residue; // for finding the right bit
     // this piece of code checks the original view postion taken from the letter
     int view_pos = 0;
     if (input>90) view_pos = input - 'A' - 7;
     else view_pos = input - 'A';
     int value = prev_vector;
     int * bit_vector = (int *)malloc(afa_states * sizeof(int));
     for(int j=0;j<afa_states;j++){
     bit_vector[j] = value%2;
          value = value/2;
     }
     int new_vector = 0;
     //find the corresponding vector
     for(int m=0; m< afa_states; m++){
          int afa_pos =1;
          // it is possible that more than one possible state transition is present
          while(afa_table[view_pos*afa_states+m][afa_pos]!=-99){
               //also it takes care if there is no transition at all
               int state_value = afa_table[view_pos*afa_states+m][afa_pos];
               // for that postion the bit is on
               if((state_value>-1)&&(bit_vector[state_value]))
                    new_vector+=pow((double)2,m);
               afa_pos++;
          }
     }
     free(bit_vector);
     return new_vector;
}

/*
     This function takes the following paramaters:
     - u: it represents the prev vector
     - input: Symbol that proceeds to the new vector
     - table:  it contains the alpha values for every pair(state, symbol) on the AFA
     - masks: A selection of bit arrays that has one bit turned on each array.
*/
unsigned long
get_afa_vector(unsigned long u, char input, unsigned long* table, int afa_states, unsigned long* masks)
{
     // this piece of code checks the original view postion taken from the letter
     int view_pos = 0;
     if (input>90) view_pos = input - 'A' - 7;
     else view_pos = input - 'A';
     unsigned long alpha;
     unsigned long answer = 0L;
     for(int m=0; m<afa_states; m++){
          alpha = table[view_pos*afa_states + m];
          if((alpha&u)>0)
               answer = answer|masks[m];
     }
     return answer;
}
/*
     This function returns the fundamental vector
     The f-vector is a bit vector that contains all the
     final states buts on and the rest off.
*/
unsigned long
get_f_vector(int r_final_state)
{
     unsigned long result = 0;
     result = 1L<<r_final_state;
     return result;
}

char*
combine_string(unsigned long num1, int num2){
     char* result = (char*)malloc(90*sizeof(char));
     char temp[50];
     int line_pos = 0;
     sprintf(temp,"%lu",num1);
     append(result,temp,&line_pos);
```

```
     append(result," vector ",&line_pos);
     sprintf(temp,"%lu",num2);
     append(result,temp,&line_pos);

     return result;
}


char*
combine_string(unsigned long num1, char letter, int num2){
     char* result = (char*)malloc(90*sizeof(char));
     char temp[50];
     int line_pos = 0;
     sprintf(temp,"%lu",num1);
     append(result,temp,&line_pos);
     append(result," ",&line_pos);
     sprintf(temp,"%c",letter);
     append(result,temp,&line_pos);
     append(result," ",&line_pos);
     sprintf(temp,"%lu",num2);
     append(result,temp,&line_pos);
     return result;
}
/*
     This function retuns the alpha table as specified in bit vector implementation
     input: afa_table: contains the trasition table of the AFA
            state_count: number of states on the AFA table
            mask: list of bit arrays with one bit activated on each array.
*/
unsigned long*
get_alpha_table(int** afa_table, int state_count, unsigned long* mask)
{
     int count = state_count * MAX_QUERIES;
     unsigned long* alpha = (unsigned long*)malloc(count*sizeof(unsigned long));
     int mask_index;
     for(int afa_line=0; afa_line<count; afa_line++){
          int pos = 1;
          alpha[afa_line] = 0;
          while(afa_table[afa_line][pos]!=-99){
               if(afa_table[afa_line][pos]>-1){
                    mask_index = afa_table[afa_line][pos];
                    alpha[afa_line] = alpha[afa_line]|mask[mask_index];
               }
               pos++;
          }
     }
     return alpha;
}
/*
     This function prints the alpha table as specified in bit vector implementation
     input: table : contains the representation of the afa table
            state_count: number of distict states being present in the table
     output: the alpha table of bit vectors
*/

void
print_alpha(unsigned long* table, int state_count)
{
     int count = state_count*MAX_QUERIES;
     for(int i =0; i < count; i++)
     {
          print_bit_vector(table[i],state_count);
          cout << "\n";
     }
}

/*
This function takes the following parameters:
     char** VG :         database array
     int    VG_count:      # of lines on the database
     char **  afa :      afa array, this afa needs to be inverted to get the r-afa effect
     int afa_count :          # of lines on the view
     int* final states:     contains the final states for the view
     int  final count:     #of final states on the view
     int start_state:    The start state for the afa
     int * reach_count:
*/
long
find_reach(char **VG, int VG_count,  char **afa, int afa_count, int * final_states,
            int final_count, int start_state,  int order)
{
     char temp[50];
     char* out_string = (char*)malloc(90*sizeof(char));
     int line_pos;
     //-------------------------------------------------------------------------------------------
     int** VG_pairs = get_pairs(VG, VG_count);
     char* VG_symbols = get_symbols(VG, VG_count);
     long reach_count = 0;
     //Getting the elements for r-afa
     int r_afa_state_count = get_state_count(afa, afa_count);
     unsigned long* masks = (unsigned long*)malloc(r_afa_state_count*sizeof(unsigned long));
```

```
     for(int q=0; q< r_afa_state_count; q++ ){
          masks[q] = 1L<<q;
     }
     int** r_afa_table = get_afa_table(afa, afa_count);
     unsigned long* alpha_table = get_alpha_table(r_afa_table,r_afa_state_count, masks);
     int* r_start_states = final_states;
     int r_start_count = final_count;
     int r_final_state = start_state;
     unsigned long f_vector = get_f_vector(r_final_state);
     //-------------------------------------------------------------------------------------------
     queue<char*> reach_queue;
     map<string, int> reach_hash;
     map<string, int>::iterator it;
     //-------------------------------------------------------------------------------------------
     int VG_source_state;
     int VG_sink_state;
     int afa_source_state;
     int afa_sink_state;
     char symbol;
     fstream  f_argout;
     char file_name[50];
     sprintf(file_name,"afa_viewgraph/%i",order);
     f_argout.open(file_name, ios::out);
     //-------------------- ALGORITHM STARTS HERE --------------------------------
     int VG_state_count = get_state_count(VG,VG_count);
     char *temp_string;
     int empty_bool;
     unsigned long new_vector, prev_vector;
     for(int p=0; p<VG_state_count; p++){
          // setting up the root the tree
          empty_bool = 0;
          VG_source_state = p;
          prev_vector = f_vector;
          temp_string = combine_string(VG_source_state, prev_vector);
          string my_string2(temp_string);
          reach_hash[my_string2] = 0;
          while(!empty_bool){
               for(int i =0; i<VG_count;i++){
                    if(VG_pairs[i][0] == VG_source_state){
                         symbol = VG_symbols[i];
                         VG_sink_state = VG_pairs[i][1];
                         new_vector = get_afa_vector(prev_vector, symbol, alpha_table, r_afa_state_count, masks);
                         temp_string = combine_string(VG_sink_state, new_vector);
                         string my_string(temp_string);
                         if(!is_there(my_string,reach_hash)){
                              reach_hash[my_string] = 0;
                              temp_string = combine_string(new_vector, symbol, VG_sink_state);
                              reach_queue.push(temp_string);
                         }
                    }
               }
               if(reach_queue.empty()){
                    empty_bool = 1;
               }
               else{
                    temp_string = reach_queue.front();
                    reach_queue.pop();
                    VG_source_state = get_sink_state(temp_string);
                    prev_vector = get_source_state(temp_string);
               }
          }
     }
     // this is to add to the final output the states that reached final together with its originator
     for( it = reach_hash.begin(); it != reach_hash.end(); it++ ){
          //cout<< (*it).first<< "\n";
          f_argout << (*it).first<< "\n";
          (reach_count)++;
     }
     f_argout <<"total of : " <<(reach_count)<< " states\n";

     //freeing memory
     for (int c1=0; c1<VG_count; c1++)
        free(VG_pairs[c1]);
     free(VG_pairs);
     free(VG_symbols);
     free(out_string);
     free(r_start_states);
     return (reach_count);
}

int
main(int argc, char** argv)
{
     char**  VG;
     char** afa;
     time_t start, finish;
     char seconds[100];

     //keeps count on the number of lines
     int afa_line_count;
```

```
  int VG_line_count;

  //This part reads/writes the files and puts them into their respective arrays
  fstream  f_argout2;

  //more variables
  char * temp;
  int * final_states;
  int  start_state;
  int final_states_count;
  char file_name[100];
  //keep track of the tree size
  int reach_count = 0;
  //Algorithm for iterations

  // Initilizing the array for the VIewGraph
  VG = (char **)malloc(200000 * sizeof(char *));
  for(int k =  0; k < 200000; k++)
       VG[k] = (char *)malloc(50 * sizeof(char));

  // Initializing the array for afa
  afa = (char**)malloc(60000 * sizeof(char *));
  for(int b =  0; b< 60000; b++)
    afa[b] = (char *)malloc(50 * sizeof(char));

  int g = atoi(argv[1]);
  afa_line_count = 0;
  VG_line_count = 0;

  //populate the ViewGrah
  fstream     f_VGin;
  sprintf(file_name, "viewgraphs_10000/%i",g);
  f_VGin.open(file_name, ios::in);
  while(!f_VGin.eof()){
       f_VGin.getline(VG[VG_line_count], 30);
       VG_line_count++;
  }
  f_VGin.close();
  //-------------------------------------------------------------------------------------
  //populate afa array. Notice here that there is a specific format for populating the afa

  temp = (char*)malloc(90*sizeof(char));
  final_states;
  final_states_count=0;
  fstream     f_afain;
  sprintf(file_name, "rewritings/%i",g);
  f_afain.open(file_name, ios::in);
  f_afain.getline(temp, 30);
  start_state = get_source_state(temp);
  f_afain.getline(temp, 30);

  while(!is_final_info(temp)){
       string_copy(afa[afa_line_count],temp);
       afa_line_count++;
       f_afain.getline(temp, 30);
  }

  //counting the number of final states.  This is almost leading to the end of the file
  final_states = (int*)malloc(afa_line_count*sizeof(int));
  final_states[final_states_count] = get_source_state(temp);
  final_states_count++;
  while(!f_afain.eof()){
       f_afain.getline(temp, 30);
       final_states[final_states_count] = get_source_state(temp);
       final_states_count++;
}
free(temp);
f_afain.close();
//-----------------------------------------------------------------------------------------------------
//Proceed to find the reach set from the viewgraph and the afa
time(&start);
reach_count = find_reach(VG, VG_line_count, afa, afa_line_count,final_states,final_states_count,start_state, g);
time(&finish);

sprintf(seconds,"%d", (finish-start));
cout<<g<<"\t"<<reach_count<<"\t"<<get_state_count(afa,afa_line_count);
cout<<"\t"<<get_state_count(VG,VG_line_count)<<"\t"<<seconds<<"\n";

f_argout2.open("results_afa_viewgraph", ios::app | ios::ate | ios::out);
f_argout2<<g<<"\t"<<reach_count<<"\t"<<get_state_count(afa,afa_line_count)<<"\t";
f_argout2<<get_state_count(VG,VG_line_count)<<"\t"<<seconds<<"\n";
f_argout2.close();

free(final_states);
//Free memory from the afa
for (int v=0; v<60000; v++)
   free(afa[v]);
free(afa);
//Free memory from the ViewGraph
for (int c=0; c<200000; c++)
   free(VG[c]);
```

```
   free(VG);
   return 1;
}
```

## 8.10 DFA vs ViewGraph

```
#include    <strstream.h>
#include    <fstream.h>
#include    <iostream.h>
#include    <stdlib.h>
#include    <map>
/*******************************************************************************
DFA vs ViewGraph evaluation

- This Program takes one parameter.  This parameter is the name of the file that contains
  the Viewgraph, and the DFA.  These are two separate files, but they should have the same
  name.  e.g.
            % dfa_vs_viewgraph.out 0
- 0 in this case is the name of the viewgraph file and also the name of the corresponding
  dfa file.  Both are in different directories.

- compilation:
  %gcc dfa_vs_viewgraph.C -o dfa_vs_viewgraph.out    OR
  %CC dfa_vs_viewgraph.C -o dfa_vs_viewgraph.out
*******************************************************************************/
using namespace std;
int  MAX_QUERIES = 40;

int
get_state_length(char *str, int pos){
   int i = 0;
   while((str[pos]>='0')&&(str[pos]<='9')){
        pos++;
        i++;
   }
   return i;
}
int
get_symbol_length(char *str, int pos){
   int i = 0;
   while((str[pos]>='a')&&(str[pos]<='z')){
        pos++;
        i++;
   }
   return i;
}

//returns the state number appearing in some position
char *
get_state(char* str, int index){
     int number_digits = get_state_length(str,index);
     char str_state[20];

     for(int j = 0; j < number_digits; j++){
          str_state[j] = str[index];
          index++;
     }
     str_state[number_digits] = 0;
     return str_state;

}

//append an string to another string on a specified position
void
append(char* derived,char* source, int* count){
     int i= 0;
     while(source[i]!='\0'){
          derived[(*count)] = source[i];
          i++;
          (*count)++;
     }
     derived[*count] = '\0';

}
void
string_copy(char* derived , char* source){
     int i= 0;
     while(source[i]!='\0'){
          derived[i] = source[i];
          i++;
     }
     derived[i] = '\0';
}
int
get_source_state(char *pool)
{
     char* c_state = get_state(pool,0);
     int state = atoi(c_state);
     return state;
```

```
}

//this function finds the sink state on a object-edge-object table
// this state is located on the right hand side from the same line
int
get_sink_state(char *pool)
{
     int found = 0;
     int i=0;
     while(found<2){
          if(pool[i]== ' ') found++;
          i++;
     }
     char* c_state = get_state(pool,i);
     int state = atoi(c_state);
     return state;
}
int
is_final_info(char* temp)
{
     int i =0;
     while(temp[i]!=0){
        if(temp[i]=='F')
           return 1;
        i++;
     }
     return 0;
}
int**
get_pairs(char **matrix, int count)
{
     int** answer = (int**)malloc(count*sizeof(int*));
     for(int i=0; i<count; i++){
          answer[i] = (int*)malloc(2*sizeof(int));
     }

     for(int j=0; j<count; j++){
          answer[j][0] = get_source_state(matrix[j]);
          answer[j][1] = get_sink_state(matrix[j]);
     }
     return answer;
}

// This two functions are to fill the matrix.
char*
get_symbols(char **matrix, int count)
{
     char* answer = (char*)malloc(count*sizeof(char));
     int h;
     for(int i =0; i<count;i++){
          h=0;
          while(matrix[i][h]!=' ')
               h++;
          answer[i]=matrix[i][h+1];
     }
     return answer;
}

int
is_there(int key, map<int,int> m)
{
     map<int,int>::iterator it;
     it = m.find(key);
     if(it != m.end())
          return 1;
     return 0;
}

int
is_final(int num, int*nums, int nums_count)
{
     for(int i=0; i<nums_count;i++){
          if(nums[i]==num)
               return 1;
     }
     return 0;
}

/*
     this function returns the number of states present
     input: - a matrix of transition
            - number of transition in such matrix
     output: max State.
*/
int
get_state_count(char** fma, int count)
{
     int max = 0;
     for(int i=0; i<count; i++){
          if(get_source_state(fma[i])>max){
               max = get_source_state(fma[i]);
```

```
          }
     }
     for(int g=0; g<count; g++){
          if(get_sink_state(fma[g])>max){
               max = get_sink_state(fma[g]);
          }
     }
     return max+1;
}
/*this function takes the following parameters:
     char** DB :  database array
     int      DB_count :# of lines on the database
     char **  view : view array
     int view_count :  # of lines on the view
     int* final states: contains the final states for the view
     int  final count: #of final states on the view
     int VG_count: number of lines on the viewGraph (this number
                   will be updated by the algorithm)
     int view_index:numeric index for the view being used.
*/
void
gen_ViewGraph(char **VG, int VG_count,  char **dfa, int dfa_count, int * final_states,
                int final_count, int *reach_count, int dfa_index)
{
     char file_name[100];
     //-------------------------------------------------------------------------------------------
     int** VG_pairs = get_pairs(VG, VG_count);
     int** dfa_pairs = get_pairs(dfa, dfa_count);
     char * VG_symbols = get_symbols(VG, VG_count);
     char * dfa_symbols = get_symbols(dfa, dfa_count);
     //-------------------------------------------------------------------------------------------
     map<int, int> queue;
     map<int, int>reach;
     map<int, int> out;
     map<int, int>::iterator it;
     //-------------------------------------------------------------------------------------------
     int VG_source_state;
     int VG_sink_state;
     int dfa_source_state;
     int dfa_sink_state;
     char symbol;
     fstream       f_argout;
     //-------------------- ALGORITHM STARTS HERE --------------------------------
     int VG_state_count = get_state_count(VG,VG_count);
     for(int p=0; p<VG_state_count; p++){
          int first_time=1;
          // just clearing the information
          queue.clear();
          reach.clear();
          queue[p]=dfa_pairs[0][0];
          while(!queue.empty()){
               VG_source_state = (*queue.begin()).first;
               dfa_source_state = (*queue.begin()).second;
               reach[VG_source_state]=dfa_source_state;
               if((is_final(dfa_source_state,final_states, final_count))&&(!first_time)){
                    out[p] = VG_source_state ;
               }
               first_time=0;
               queue.erase(queue.begin());
               //I already have the state and object to find their transitions
               // Assuming the database is already sorted
               int i =0;
               while((i<VG_count)&&(VG_pairs[i][0]!=p)) i++;
               while((i<VG_count)&&(VG_pairs[i][0]==p)){
                    symbol = VG_symbols[i];
                    VG_sink_state = VG_pairs[i][1];
                    for(int m=0;m<dfa_count;m++){
                         if((dfa_pairs[m][0]==dfa_source_state)&&(dfa_symbols[m]==symbol)){
                              dfa_sink_state=dfa_pairs[m][1];
                              //There is a match, make sure is not already there
                              if((!is_there(VG_sink_state,queue))&&
                                 (!is_there(VG_sink_state,reach))){
                                   queue[VG_sink_state] = dfa_sink_state;
                              }
                         }
                    }
                    i++;
               }
          }
    }
    sprintf(file_name,"rewritings/%i",dfa_index);
    f_argout.open(file_name, ios::out);
    // this is to add to the final output the states that reached final
    //together with its originator
    for( it = out.begin(); it != out.end(); it++ ){
          f_argout << (*it).first<< " " << (*it).second <<"\n";
          (*reach_count)++;
    }
    f_argout <<"total of : " <<(*reach_count)<< " states\n";
    f_argout.close();
    //Clean the mess
```

```
    for (int c1=0; c1<VG_count; c1++)
       free(VG_pairs[c1]);
    free(VG_pairs);
    for (int c2=0; c2<dfa_count; c2++)
       free(dfa_pairs[c2]);
    free(dfa_pairs);
    free(VG_symbols);
    free(dfa_symbols);
}
int
main(int argc, char** argv)
{
     char** reach;
     char** dfa;
     char** ViewGraph;
     //keeps count on the number of lines
     int reach_line_count = 0;
     int dfa_line_count = 0;
     int VG_line_count = 0;
     time_t start, finish;
     char seconds[100];
     // Initializing the array of values...
        ViewGraph = (char **)malloc(200000 * sizeof(char *));
     for(int k =  0; k < 200000; k++)
          ViewGraph[k] = (char *)malloc(50 * sizeof(char));
     //This part reads/writes the files and puts them into their respective arrays
     fstream           f_dfain;
     fstream          f_VGin;
     fstream          f_argout2;

     //more variables
     char * temp;
     int * final_states;
     int final_states_count;
     char file_name[100];
     //populate database array
     int g = atoi(argv[1]);
     sprintf(file_name, "viewgraphs_10000/%i",g);
     f_VGin.open(file_name, ios::in);
     while(!f_VGin.eof()){
          f_VGin.getline(ViewGraph[VG_line_count], 90);
          VG_line_count++;
     }
     f_VGin.close();
     reach_line_count=0;
     dfa_line_count = 0;
     // Initializing the array of values...
     dfa = (char **)malloc(60000 * sizeof(char *));
     for(int i =  0; i < 60000; i++)
          dfa[i] = (char *)malloc(50 * sizeof(char));
     temp = (char*)malloc(90*sizeof(char));
     final_states = (int*)malloc(1000*sizeof(int));
     final_states_count=0;
     //populate dfa array. Notice here that there is a specific
     //format for populating the view
     sprintf(file_name, "second_complement/%i", g);
     f_dfain.open(file_name, ios::in);
     f_dfain.getline(temp, 90);
     f_dfain.getline(temp, 90);
     while(!is_final_info(temp)){
          string_copy(dfa[dfa_line_count],temp);
          dfa_line_count++;
          f_dfain.getline(temp, 90);
     }

     final_states = (int*)malloc(dfa_line_count*sizeof(int));
     final_states[final_states_count] = get_source_state(temp);
     final_states_count++;
     while(!f_dfain.eof()){
          f_dfain.getline(temp, 90);
          final_states[final_states_count] = get_source_state(temp);
          final_states_count++;
     }
     f_dfain.close();
     time(&start);
     gen_ViewGraph(ViewGraph, VG_line_count, dfa, dfa_line_count,
           final_states,final_states_count,&reach_line_count,g);
     time(&finish);

    sprintf(seconds,"%d", (finish-start));
    cout<<g<<"\t"<<reach_line_count<<"\t"<<get_state_count(dfa,dfa_line_count)<<"\t";
    cout<<get_state_count(ViewGraph,VG_line_count)<<"\t"<<seconds<<"\n";

    f_argout2.open("results_afa_viewgraph", ios::app | ios::ate | ios::out);
    f_argout2<<g<<"\t"<<reach_line_count<<"\t"<<get_state_count(dfa,dfa_line_count)<<"\t";
    f_argout2<<get_state_count(ViewGraph,VG_line_count)<<"\t"<<seconds<<"\n";
    f_argout2.close();
    //Free memory from the dfa
    for (int b=0; b<60000; b++)
       free(dfa[b]);
    free(dfa);
```

```
    //Free memory from the ViewGraph
    for (int a=0; a<200000; a++)
        free(ViewGraph[a]);
    free(ViewGraph);
    return 1;
}
```