Optimizing Regular Expression Clustering for Massive Pattern Search

Nikolay Laptev 1, Hamid Mousavi 2, Alexander Shkapsky 3, Carlo Zaniolo 4

Computer Science Department, University of California at Los Angeles
Los Angeles, CA, USA
1nlaptev@cs.ucla.edu
2hmousavi@cs.ucla.edu
3shkapsky@cs.ucla.edu
4zaniolo@cs.ucla.edu

Abstract—Optimizing the search for a large sets of patterns, due to their prevalence, is becoming critical in a variety of applications including financial services, healthcare monitoring, RFID-based inventory management, publish-subscribe and network intrusion detection systems (NIDS). For example in NIDS, to identify particular packets in a packet stream, modern network devices need to perform deep packet inspection at high rates given a limited memory and a large number of signatures. It is often proposed to group together regular expressions (REs) denoting similar patterns into a DFA or an NFA to decrease the memory usage and increase the average throughput. In this paper, we propose a framework comprised of three novel distance metrics used to cluster REs and an execution technique that uses the information aggregated by the distance metrics to optimize the number of memory reads during execution, thereby increasing the overall throughput. Our distance metrics are unique in that each metric is specialized for a particular type of patterns. The first two metrics optimize for compactness of REs and specialize in regular expressions that have long, contiguous similarity segments. Our third metric optimizes for redundancy and specializes in REs that have random segments of similar. We show how compression, prefetching and memory access optimizations are made possible when the notion of redundancy is leveraged. We validate our approach by implementing it in a modern NIDS Snort, which matches potentially thousands of patterns against an incoming network stream. We observe a five-fold speed improvement over the unmodified version of Snort.

I. INTRODUCTION

In domains such as financial services, healthcare monitoring, RFID-based inventory management, publish-subscribe and network intrusion detection, systems are used to monitor data streams for desired patterns in the incoming data. These systems often utilize regular expressions (REs) and complex pattern queries to perform searches in real-time so end-users can be notified immediately when a match is detected. Because of the real-time response requirement and due to thousands of queries to be monitored, these systems must utilize computing resources efficiently.

To find matches when dealing with a single RE, it is common to translate the RE into a finite state Automaton (FSA), specifically a Non-Deterministic or a Deterministic Finite Automaton (NFA or DFA respectively). Thus, to find matches over a given input, one can simply follow the transition(s) of the FSA for each input symbol, and check if the final state is reached. Using an NFA, the pattern representation will be space efficient, but due to multiple active transitions being allowed, processing a single input using an NFA takes $O(N)$ running time where $N$ is the number of states. Alternatively, the DFA has a $O(1)$ processing time per input symbol, however the number of states in a DFA may be exponential in the length of the RE. The DFA also suffers from state explosion when kleene closures are used in REs.

When multiple REs are searched over the input, a similar to the above approach can be used. That is we can combine all the REs into a single DFA or an NFA using disjunction, and find the matches over that FSA. However, this approach is not scalable due to space and time issues. Combining multiple REs into a composite DFA may result in a state explosion. For a combined NFA the matching time may become unacceptable. Therefore, exclusively using a DFA or an NFA is ill-suited for online applications where both light memory usage and high speed are important. Thus several hybrid approaches have been proposed [1]. In this paper we propose novel distance metrics that can be used to combine REs into clusters such that memory and performance are optimized. A motivating example is presented next.

Example 1: Consider DNA patterns in which one wants to find matches for nine patterns $(A|C|G)^* AAAAA$, $(A|C|G)^* AAAC$, $(A|C|G)^* AACA$, $(A|C|G)^* AACC$, ..., $(A|C|G)^* ACCC$, and $(A|C)^* AAA$ in a given DNA sequence. It is not practical to consider these REs separately (or equivalently consider single NFA for their combination) because of the time constraint in streaming systems [2]. A more efficient solution would be to combine all nine of these REs into a single DFA and run any of the existing matching algorithms, such as that in [2] to find matching instances. Although this approach is fast, it requires storing 33 states. A more space-friendly alternative is to group the first eight REs into one cluster and consider the remaining REs as another cluster. This would generate two DFAs, a DFA with 16 states and a DFA with four states. Thus we eliminated 13 states and several transitions, however the matching algorithm must now examine two DFAs, which may decrease the overall throughput.
We propose a general framework to cluster a range of pattern query types based on three novel distance metrics. The distance metrics, which are the main focus of this paper, are mainly designed for the purpose of improving throughput and memory utilization. Each of the proposed metrics is designed for a particular family of patterns. Furthermore, the techniques we propose can be applied to both patterns and XML query patterns because both of these types of patterns can be represented as automata. In other words, general patterns with predicates and nested XML queries can be translated to an NFA or a DFA using techniques presented in [3] and [4] respectively. Note that, we refer to both patterns, pattern queries and REs interchangeably, because both are an RE with additional predicates per symbol, and can be expressed as an automaton.

Distance metrics in clustering approaches are usually based on the concept of similarity between objects. This is why distance metrics are sometimes referred to as similarity metrics. As for REs, this type of metric intuitively should combine REs that share large pattern segments in common, such that the combined automaton represents a language that is as compact as possible. As an alternative to a similarity metric, distance metrics based on redundancy aim to cluster REs based on how many redundant transitions there are between combined states. Our insight is that seemingly dissimilar REs, when converted to a merged DFA, may have states that have a lot of transitions in common. We refer to such transitions as redundant and to the metric that maximizes the number of such transitions as a redundancy metric. It is possible to compress many of the redundant transitions, resulting in an automaton that uses less memory than the automaton derived by utilizing similarity techniques alone. Thus redundancy metrics allow us to reduce the overall memory usage through compression techniques and increase throughput by decreasing the number of memory reads with the help of prefetching. The example in Figure 1 gives a motivation for our Redundancy Metric:

**Example 2:** Figure 1, demonstrates how to reduce the memory footprint by storing only the required transitions and states of the DFA. In Figure 1 (a), observe that state A and state B have the same transitions. Here, we can use a bitmap to only store a single set of transitions for state A and refer to it in state B. This will result in a DFA in Figure 1 (b). By reducing the number of transitions we are able to increase throughput and decrease memory usage. We are able to store entire states and the corresponding transitions in cache thus also reducing the number of memory reads.

Based on redundancy and similarity, we present three new clustering distance metrics used to cluster REs. Two of the newly proposed metrics are based on similarity. The third metric takes redundancy into account, which provides a very different view of measuring distance between two patterns. We also identify types of patterns which are appropriate for each of these metrics, and we consider ways of combining these distance metrics to produce more compact clusters. Next we briefly introduce these metrics.

The first metric extends the idea of Minimum Description Length (MDL) [5]. The main goal of MDL is to group REs in order to reduce the overall bits required to encode a character using the language described by the chosen REs. MDL, however, is oblivious to where the similarity between patterns occurs, and is instead concerned with the overall similarity. For example, using MDL RE \( re_1 = a(ba)^*b \) is as similar to \( re_2 = a(cd)^*(ba)^*b \) as it is similar to \( re_3 = a(ba)^*(cd)^*b \). This means that using MDL, \( re_1 \) and \( re_2 \) will be placed in the same cluster, however it is better to group \( re_1 \) with \( re_3 \) because the former pair shares a majority of the prefix in common, which has intuitively has a higher probability of being accessed.

Our second similarity metric is based on a Fixed Point Computation (FPC). FPC first converts each RE into its minimized DFA form. Then, considering the initial states and the final states in each DFA, FPC constructs a propagation graph. Similarity values for each possible pair of states are then initialized. Using the propagation graph similarity values are propagated from nodes to their neighbors until the similarity value converges.

The third metric is based on the idea of redundancy. A high redundancy between two patterns is interpreted as a repetition of a part of the language which is obtained when two patterns are combined. Calculating redundancy for a cluster is done via propagation of a redundancy measure using a depth first traversal. Because the clusters can be quite large, we propose algorithms that can quickly approximate the underlying redundancy of the cluster via pruning.

Table I categorizes the types of patterns that are appropriate for the two types of similarity approaches and the redundancy measure we introduced.

Given the above mentioned approaches and the types of patterns that are appropriate for each approach, we show how combining the two similarity techniques with a redundancy metric results in a distance measure that can greatly reduce the size of the resulting clusters, thereby increasing throughput. To unify these approaches, we present a framework consisting of clustering and execution components. The clustering component, utilized during application startup and prior to any matching, clusters REs based on similarity and/or redundancy metrics to provide compact, memory efficient DFAs that
deliver high throughput in a resource constrained environment. Next, during application execution, the execution component utilizes the redundancy metric for prefetching purposes to minimize the number of memory reads and thus to increase throughput.

The redundancy measure postulates the probability distribution of the transitions to be taken, thus the redundancy of each state can be used for effective prefetching by greedily prefetching states with highest redundancy measure. Given a cache limit of size $k$, authors in [6] show that any optimal character by character compressor can be converted into a prefetcher that has an optimal fault rate. Thus, we utilize the redundancy measure to determine which $k$ pages to prefetch.

The main contributions of this paper are:

- We present three novel distance metrics used for clustering REs and efficient algorithms for computing these metrics.
- We categorize each distance metric based on the type of pattern queries which are best suited for it and present a novel hybrid clustering approach that unifies the presented distance metrics to cluster REs in an environment with limited memory and a strict requirement on a high throughput.
- We show how prefetching and compression can be applied to resulting clusters generated via our new redundancy metric.

The remainder of this paper is organized as follows: in section 2, we discuss previous work. Section 3, describes our framework, outlining each component. In section 4, we discuss our experiments and conclusions are presented in Section 5.

II. RELATED WORK

In this section, we categorize related works based on pattern (REs) similarity measurements and graph compression. In addition, we also mention pattern SQL proposals and systems that can benefit from our approach.

A. Pattern Similarity Measurements

As a measurement of structural similarity over XML fragments, Tree Edit Distance (TED) [7] has been widely used. TED of two trees $A$ and $B$ is defined as the minimum number of operations, such as insertion, deletion, and replacement, over nodes which are needed to transform tree $A$ into tree $B$. However, TED is costly to compute, and it is therefore difficult to apply it over data streams. There exist other recent works on tree-pattern similarity estimation which are mostly useful for XML-based systems such as [8][9].

Wang et al. proposed a new clustering method called pCluster in which the main focus is to capture subspace similarities of patterns and not the similarity of the patterns as a whole [10]. Given patterns of different dimensions, the proposed approach picks a subset of dimensions that will maximize the similarity between a given set of patterns.

A lot of attention has been given to computing the similarity of regular expressions [11][12][13][8][9]. These works take a general notion of similarity used in clustering and apply it to a specific application, namely clustering of REs. As one of the early works on this, Powell used a simple dynamic programming algorithm to find the similarity of two REs [11]. This approach is efficient, but one of its limitations is the lack of support for REs with kleene stars.

Melnik et al. proposed a general graph similarity measure approach based on fixpoint computation [12]. The authors convert each model (graph) to a directed labeled graph. Each pair of the resulting graphs are then used to construct the pairwise connectivity graph and consequently a propagation graph. Using the propagation graph, the similarity value of each pair is iteratively propagated to its neighbors until a convergence is reached and the best matching nodes are reported. However, the approach cannot be used directly on DFAs, because of three main reasons: (1) The schema of backward edges is set up in the propagation graph, (2) No consideration is given to the initial and the final states of DFAs, and (3) Edge weights are based on labels’ similarities.

In [13], Sokolsky et al. adds the concept of initial node into labeled directed graph and computes the similarity of graphs using a simulation-based approach. This approach, similarly to [12], suffers from the fact that the authors have not considered feedback edges in the simulation graph. Thus the states with self-loops in a DFA would receive majority of the weight.

When dealing with a DFA state explosion problem (for example when converting a single pattern ".A...BC" into a DFA) previous approaches utilize techniques such as pattern rewriting [2] to get rid of non-determinism. These types of approaches help reduce memory usage, but unfortunately at the expense of introducing false matches.

B. Graph Compression

An important applications of graph compression is the Web graph. Boldi and Vigna, in [14], proposed an efficient way to compress the Web graph, by exploiting gap compression. The authors use the locality feature of the Web to find pairs of nodes with similar out-going links. Thus, it is enough to store the list of links for one of the nodes and to construct the differential list for the other node. Despite being very efficient on the Web graph, the approach can not be directly applied to DFAs because of structural differences between the Web graph and the DFA.
In [15], authors present a $\delta$FA which is based on the observation that many states in a DFA share a lot of transitions in common. Thus compression techniques can be used to considerably reduce the memory usage of a DFA. The authors in [15] however only provide a brief overview of the compression techniques and do not deal with clustering of patterns.

C. SQL Proposals

There exist many languages and systems that can benefit from our approach such as SQL-TS, MATCH-RECOGNIZE and SASE+.

MATCH-RECOGNIZE [16] comes from a 2007 ANSI standard proposal to add the new SQL functionality for finding patterns defined as regular expressions over sequence of rows via a MATCH_RECOGNIZE clause. The new syntax allows for pattern definition via a PATTERN component which is used to specify the regular expression. Some of the operations supported in the PATTERN component are: * (0 or more rows), + (1 or more rows), ? (0 or 1 row). In addition, aggregate operators can be used on group variables like AVG(A) where A is a group variable. In addition the columns of the matched rows are also accessible via D.price > A.price.

SQL-TS [17] (Simple Query Language for Time Series) is another main proposal for SQL language extension to support pattern queries. SQL-TS, besides proposing a set of new language constructs also discusses the optimizations of pattern matching. An extension of the well-known Knuth, Morris, and Pratt (KMP) algorithm is introduced in order to optimize pattern queries.

SASE+[18] proposes a compact language that can be used to define a wide variety of patterns containing Kleene Closures. The authors rigorously studied language semantics and analyzed the expressive power of SASE+, SQL-TS and Cayuga [19], which is a stateful publish/subscribe system based on an NFA. Cayuga is an example of a system that can easily benefit from our approach because it operates on a large number of queries, many of which contain Kleene Closures.

III. OUR APPROACH

In this section, we present our framework, that consists of clustering and execution modules. The clustering module utilizes the three distance metrics, discussed in detail in section III-C, to cluster REs into groups such that memory is minimized and potential throughput is maximized. Compression techniques are used to further minimize the memory used by compressing any redundant transitions. The clustering module is also responsible for building a light-weight index on the resulting clusters. Note that the clustering module does not depend on any particular clustering technique. Our main contribution is the design of various distance metrics applicable to patterns exhibiting certain characteristics. In the experimental section we used a simple clustering technique where the distance between all patterns is compared and most similar patterns are merged. Thus any clustering algorithm that uses distance as a criterion for merging objects can be used with our technique. The execution module is responsible for using the information gathered by the distance metrics to increase throughput via prefetching. We first present preliminary information and then discuss the details of our framework.

A. Preliminaries

Our framework is flexible in terms of the type of automata used for representing patterns. Thus for simplicity we use a DFA as an underlying pattern representation, however our distance metrics are applicable to an NFA and other types of automata that supports intersection and union operations. As stated our framework is independent of the clustering technique used, nevertheless for completeness we describe the frequently used clustering techniques next.

In general, the goal of clustering is to group a set of points in a feature space based on proximity, in order to form a set of clusters. To cluster objects such as points in n dimensional space, unified near-linear time complexity techniques are used such as k-means and Scatter/Gather [20]. These techniques are all partition based. Partition based approaches simply separate a flat collection of items into a single set of “bins”. A hierarchy is built by recursively applying a partition algorithm. The partition algorithms each have complexity of $O(N)$ with respect to the number of objects, $N$, so the overall hierarchy is generated in $O(N\log N)$ time when a balanced hierarchy is assumed.

Partition based algorithms have three stages: seed selection, center adjustment, and cluster refinement. Seed selection is the process of choosing $k$ candidate points in the feature space to serve as centers for the partitions. During center adjustment, objects are repeatedly assigned to the nearest center and the center is recalculated based on the average location of all objects assigned to it, thereby moving it through the feature space. This process may be repeated multiple times. Afterwards, all objects are removed from the centers, and reassigned to the new closest center. Thus, it is important that the centers be distributed effectively enough that they each attract sufficient nearby, topically related objects. Cluster refinement is an optional final step for improving the new partitions [21].

Next, we define the data structures that are frequently used for pattern matching.

Definitions: Traditional pattern matching approaches generally construct one finite state machine (a deterministic or a non-deterministic finite automata (DFA or NFA respectively)) for all patterns to be matched. There are standard techniques to convert a pattern (RE) into an NFA or a DFA [22] so we do not describe them here. Input used for traditional pattern queries (trajectory queries, double-bottom queries etc) seldom matches even the first few predicates of the query, thus traditional approaches involving a single composite DFA are wasteful. Due to scalability issues (state explosion) when a single DFA is constructed for thousands of patterns and because the tail of the pattern is rarely visited, constructing $m$ smaller DFAs for $k$ first symbols and a slower (NFA) for the rest of the pattern is a better choice.
A DFA is a quintuple \( M(S, s_0, \sigma, \Sigma, F) \) where \( S \) is a finite set of states, \( s_0 \) is an initial state, \( \sigma : S \times \Sigma \rightarrow S \) is a transition function, \( \Sigma \) is the alphabet and \( F \subseteq S \) is a set of accepting states. An NFA is defined similarly except that the transition function \( \sigma \) may return multiple states, thus multiple states may be active for input \( i \).

In the next subsection we present the architecture of our framework.

### B. Framework

Figure 2(a) depicts our architecture. Its components include: Index, Compression, Prefetching and Clustering Engine. We implement our framework as a standalone application (with a naive automaton string matcher) and on top of Snort (with Snort’s sophisticated automaton string matcher) for experimental purposes. Snort handles query matching as well as input and output operations. For this reason, the explanation of our framework is provided while referring to Snort and its underlying methods. First, Snort rules are read and clustered according to the clustering metric described next, which, depending on the type of patterns to be clustered, can either include a hybrid approach that utilizes both redundancy and similarity measures or one of the standalone clustering measures presented. In the hybrid approach the weights between similarity and redundancy metrics are determined experimentally. Because our approach is independent of the clustering technique used, we use a naive approach where we merge most similar patterns based on a given distance metric until only \( k \) patterns remain where \( k \) is dependant on the available system resources. Section III-E1 provides a brief intuition of finding optimal number of clusters, but this is not the focus of the paper and the details are left as future work.

After the clustering step is complete, a light-weight index is constructed which only utilizes the initial states of each cluster. This index quickly returns a set of clusters \( C \) with initial states that match the input. Clusters contain redundancy information \( R_n \) for each node \( n \). \( R_n \) is utilized when the prefetch component of our architecture is invoked during the execution phase. The prefetch component prefetches states with probability \( p \) which is proportional to state \( i \)'s redundancy \( R_i \).

The preprocessing step of our framework generates the clusters from a set of REs. During cluster generation, the following set of functions is invoked (where \( x \) can be substituted by an NFA or a DFA):

1. \( x\text{New}(P, C_n) \) – Create a new state machine given a set of patterns \( P \) and a desired number of clusters to be generated \( C_n \).
2. \( x\text{AddPattern}(p) \) – Add a pattern \( p \) to the cluster that minimizes the appropriate distance metric currently used.
3. \( x\text{Compile()} \) – Compile all clusters.

Given a set of clusters, an index on the initial state of each cluster is constructed. In the execution step, for each input \( i \), first, a set of possibly matching clusters is retrieved from the index. On the subsequent input, \( x\text{Search()} \) is invoked. \( x\text{Search()} \) first checks the cache to get the next transition. If there is a cache-miss, the disk is accessed to get the next transition. \( x\text{Search()} \) uses prefetching to minimize the number of memory reads.

When implementing our framework in Snort, the execution engine is modified such that:

- We store the patterns separately in arrays corresponding to particular clusters.
- The matching lists of patterns for each state and for each cluster are stored separately as well.
- We modify Snort’s \( \_\text{list_get_next_state} \) which is the method responsible for retrieving an automata’s next state given a transition. These modifications are necessary when using compressed automata.

Because each state of our DFA may have sparse transitions (especially the possibly compressed DFAs) we have to store each state in a sparse list. Sparse transition arrays are searched using linear and binary search strategies depending on the number of entries to search in each state.

### C. Clustering Phase

We suggest that by aggregating the results of different distance metrics we can significantly improve the robustness and the quality of the final clustering. The idea is that different algorithms make different “mistakes” that can be “canceled out” in the final aggregation. Furthermore, for objects that are outliers or noise, it is most likely that there will be no consensus on how they should be clustered, and thus they will be singled out by our hybrid algorithm. The intuition is similar to performing rank aggregation for improving the results of web searches. Our experiments indicate that using a hybrid distance metric can significantly improve the results of individual algorithms.

Given a set of REs, we group them into clusters using a combination of the three distance metrics presented below.

**Minimum Description Length (MDL):** We employ MDL [5] as one of the similarity measures. MDL is an approach that provides a way to select the most compact model efficiently. The MDL principle favors a model that can represent the data in the simplest way. A popular alternative to MDL is the rate of growth measure, which states that language \( L_1 \) is larger than language \( L_2 \) if \( L_1 \) grows faster than \( L_2 \). However, the rate of growth fails to identify the larger language in some cases [23].

The main theory behind MDL is based on what is called the “Occam’s Razor”, which states that the best scientific theory is the one that is simplest and explains all facts. Thus, applying it to the automata theory, MDL prescribes that, that FSA should be chosen which can encode the input most succinctly. Therefore, we apply a similar theory to a general automata, and say that given two automata \( M_i \) and \( M_j \) and if language of \( M_i \) denoted as \( L(M_i) \) is larger than the language of \( M_j \) \( (L(M_j)) \) then the per symbol cost to encode a string using \( M_i \) will be higher than the per symbol cost to encode a string using \( M_j \) [24].

The per symbol cost of a string \( w \in M \) is simply the MDL measure of \( w \) using \( M \) divided by the length of \( w \) denoted as...
|w|. Clearly the larger we choose |w| to be the more accurate our MDL measure will be. A good |w| measure according to [23] is slightly greater than |M| which ensures that some accepting state(s) along with some transitions are visited. In cases where the maximum length of the input is known, |w| should be set to that length.

We use MDL to compute similarity as a measure of compactness between two patterns i and j. First i and j are merged into a single automaton. Then to compute the similarity, the MDL notion is used which is defined as follows:

**Definition 3.1: The MDL Principle**

Let \( D^{(1,2)}, D^{(1,3)}, \ldots \) be a list of candidate models (e.g., \( D^{(i,j)} \) is the union of automaton \( i \) with automaton \( j \)). The best model \( D \in \{ D^{(1,2)}, D^{(1,3)}, \ldots \} \) to explain strings \( S \in L(D) \) is the one which minimizes the sum \( L(D) + L(S|D) \), where

- \( L(D) \) is the length of the model in bits (i.e., finite state automata)
- \( L(S|D) \) is the length, in bits, of the description of the data when encoded with the help of \( D \).

Thus the best model is the model that explains \( S \) most succinctly.

We specify the size of language \( L \) as

\[
|L(M)| = \frac{MDL(M, w)}{|w|} \quad (1)
\]

where \( MDL(M, w) \) denotes the cost of MDL-based encoding of a string \( w \in L(M) \) which is defined as: \( MDL(M, w) = \sum_{i=1}^{n} \log(\nu_i) \) where \( \nu_i \) denotes the number of out-going transitions from state \( s_i \) which is visited by our string \( w \) and \( \log(\nu_i) \) denotes the number of bits required to encode the outgoing transition from state \( s_i \). It is important to note that \( M \) should be a minimal DFA without any non-essential transitions because an MDL measurement very much depends on the number of transitions. The following example demonstrates how MDL is used:

**Example 3.2:** Consider two automata \( M_1 \) and \( M_2 \) in Figure 2(b). Consider also two strings \( w_1 = abcaaaaaab \in L(M_1) \) and \( w_2 = acbabbabab \in L(M_2) \). The per symbol cost of encoding a string using \( M_1 \) and \( M_2 \) respectively is:

\[
|L(M_1)| = (9 \times \log_2(2) + 0)/10 = 0.9
\]

\[
|L(M_2)| = ((2 \times 0) + (8 \times \log_2(3)))/10 = 1.26
\]

The MDL metric correctly indicates that \( L(M_2) \) is larger than \( L(M_1) \). Thus, when clustering a set of REs, \( M_1 \) would be preferred as a cluster as opposed to \( M_2 \).

**FixPoint Comparison Similarity:** The previous subsection showed how MDL can be used to compute the similarity of two REs. Although the goal of MDL is to reduce the number of states generated by combining all the REs of each cluster into a single DFA, as mentioned earlier, for some cases this might not be a good choice. For instance it is better to put the REs which have a longer prefix in common into the same cluster to decrease the total matching time due to the prefix being accessed more frequently. Furthermore, combining all REs into a single DFA is inefficient due to a possible space explosion.

Thus we have designed another similarity measure approach termed FixPoint Comparison (FPC). FPC is heavily inspired by Melnik et al., and its main goal is to compute the similarity of REs (or equivalently DFAs) based on the size of their longest common prefix.

Informally, to compute the similarity of two REs, FPC first converts them to the minimized DFA form. Next, considering the initial states and final states in the pair of DFAs, it constructs a propagation graph. Finally, FPC initializes the similarity values for each possible pair of states and uses the propagation graph to repeatedly propagate the similarity values of each pair over their neighbors until the convergence is reached.

More specifically, FPC takes the following steps to compute the similarity of two REs, say \( re_1 \) and \( re_2 \):

- Convert REs \( re_1 \) and \( re_2 \) to minimized DFAs \( df_{a1} \) and \( df_{a2} \). These DFAs should be complete, meaning that from each node and for each possible input, the DFA must have exactly one transition. To achieve this, each state in the DFA must have a transition to a dummy state for disallowed input.

- Combine two DFAs into a single labeled directed graph called a mixed graph. For each state \( s_1 \) in \( df_{a1} \) and \( s_2 \) in \( df_{a2} \), we add a node \( <s_1, s_2> \) into the mixed graph.
Then, we connect node \(<s_1, s_2>\) to \(<t_1, t_2>\) through an edge labeled \(l\) in the mixed graph if there is an edge labeled \(l\) from both \(s_1\) to \(t_1\) and from \(s_2\) to \(t_2\) in \(dfa_1\) and \(dfa_2\) respectively.

- Create a propagation graph \(p\), which is a weighted directed graph. This graph has the exact same nodes as the mixed graph. For the edges, we add every node \(<s_1, s_2>\) to node \(<t_1, t_2>\) with weight \(w\), where \(w\) is the number of edges from \(<s_1, s_2>\) to \(<t_1, t_2>\) in the mixed graph divided by alphabet size of our REs. At this point, we make sure that the similarity values keep propagating through the propagation graph. To this end, we connect edge \(<st_1, st_2>\) to \(<z_1, z_2>\) with weight 1, where \(st_i\) and \(z_i\) are respectively the start and the dummy states in \(dfa_i\). It is worth mentioning that the purpose of the propagation graph is to specify the probability that a pair of nodes is similar based on the similarity of all the other pairs of nodes (one from each DFA).

- Initialize a similarity matrix \(\sigma_0\). This matrix indicates how similar each pair of states is. FPC initializes \(\sigma_0[st_1][st_2]\) to 1 and all others to 0. Note that unlike similar approaches, it may make a difference if another initializing scheme is used.

- Compute the fixpoint similarities. In each iteration, \(k\), we update the similarity using equation 2 and then normalize the values. This would be repeated until the similarities converge, meaning the difference between the results of two consecutive iterations is less than a pre-specified threshold:

\[
\sigma_{k+1}[i][j] = \sigma_k[i][j] + \sum_{n,m} \{p[n,m][i,j] \times \sigma_k[n][m]\}
\] (2)

- Compute the final aggregate similarity value of two REs. This can be done in several ways:
  1) Consider the similarity of the two initial states \((\sigma_0[st_1][st_2])\).
  2) For each state in \(dfa_1\) find the best matching non-dummy state in \(dfa_2\) and add the similarity of these pairs together. In this paper, we use this approach to compute the final similarity values, since it provides the similarities based on the biggest common prefix of REs.
  3) Add up the similarity of pairs of final states (those pairs containing final states from each DFA.) This is a good choice when it is desirable to know what portion of the corresponding languages of the REs intersect.

Note that the number of iterations needed for the system to converge is usually less than 30. The only problem is that the propagation graph size is quadratic, and for big REs this approach may require too much memory and may also run slowly. To overcome this problem, FPC uses a sparse matrix implementation for all the graphs. For large DFAs, it also shrinks the DFAs by considering only those nodes that are within a particular distance from the start node. FPC is performed during the preprocessing phase prior to matching, so these encumbrances can be considered tolerable.

**Redundancy Metric:** Traditional clustering techniques look at structural and/or semantic similarity, whereas maximizing redundant information in merged REs (or clusters) has been overlooked. By maximizing redundant information, we can make better use of various compression techniques in order to efficiently represent the transition state table. Therefore, by merging two patterns that seem dissimilar and with the help of compression techniques, we can potentially maximize redundant information in the merged pattern, thereby producing a potentially best possible cluster. Compression techniques such as those described by the authors in [15] and [14] present novel ways of compressing a finite automaton to both improve memory utilization and reduce the number of memory block accesses.

Recall that in Figure 1 we showed the intuition behind clustering based on redundancy. However, we only considered a neighboring state when considering redundant states, but note that for instance state \(E\) and state \(A\) have overlapping transitions, thus further optimization is possible if non-neighboring states are also considered. Thus when using a redundancy distance metric, we consider not only those REs that result in fewest number of states thus minimizing memory usage, but also those REs that result in greatest average in-degree per state. This approach will let us take advantage of redundancy and thus allow us to use prefetching and compression techniques.

We must point out that in our implementation, the number of memory accesses and the processing cost always work concurrently. Nevertheless, for any operation the memory access usually dominates any processing time, therefore the overall performance almost exclusively depends on the number of memory accesses.

Our compression technique requires a supplementary data structure that stores the difference between a current state \(i\) and another closely related state (or states) \(j\). Such a data-structure must be succinct and be able to fit in-memory. However, this requires the introduction of a supplementary structure that locally stores the transition set of the current state. The main idea is to let the local transition set evolve as a new state is reached: if there is no difference between \(i\) and \(j\) for a given character, then the corresponding transition defined in the local memory is taken. Otherwise, the transition stored in the state is chosen. In all the cases, as a new state is read, the local transition set is updated with all the stored transitions of the state.

**D. Mathematical Model of Redundancy Metric**

In this section, we give the formal definition of the redundancy metric and give a simple example to illustrate the process of redundancy metric score calculation.

**Definition 3.3:** Let \(\text{path}(n_i, n_s)\) denote a sequence of nodes \(n_i, n_{i+1}, \ldots, n_s\) such that \((n_i, n_{i+1}) \in C\) where \(C\) is
a cluster. This is called a path from \( n_i \) to \( n_s \).

**Definition 3.4:** Let \( \text{length}(p) \) denote the length of path \( p \) and is defined as \( \text{length}(p) = |p| \) where \(|p|\) is defined as the number of nodes in \( p \).

**Definition 3.5:** Let \( \text{PATH}(v, u) \) denote the set of all possible paths from node \( v \) to node \( u \).

**Definition 3.6:** Let \( RS(u, v) \) denote the redundancy score of node \( u \) that propagated to node \( v \) through \( \text{PATH}(u, v) \). Thus we define:

\[
RS(u, v) = \sum_{p \in \text{PATH}(u, v)} \frac{w}{u \neq v} 
\]  
(3)

where \( w \) is defined as the weight of a particular in-transition which is based on the node’s depth level in a DFT algorithm and is defined as

\[
w = \frac{d}{\text{length}(p)}
\]  
(4)

where \( d \) is the decay factor and \( \text{length}(p) \) is the level of the node from which in-transition was taken and \( n \) represents the total number of nodes in a cluster. The overall redundancy, \( R(C) \), of a given cluster \( C \) is computed by

\[
R(C) = \sum_{i=0}^{n} RS(n_0, i)
\]

where \( n_0 \) is the initial state. Note that \( C \) represents a set of merged REs. For this reason \( R(C) = R(re_1, re_2) \) where \( re_1 \) and \( re_2 \) are two patterns to be compared.

**The Redundancy Computation Algorithm:** For a better grasp of the redundancy score computation, we start with an example. In order to compute a redundancy score for cluster \( C \), we compute \( RS(n_0, j) \) where \( n_0 \) denotes the starting node and \( 0 \leq j < N \) where \( N \) is the total number of nodes in the cluster automaton. The computation of the redundancy score is implemented similarly to the depth first traversal, the difference being that in our case \( \text{PATH}(i, s) \) can contain a node which has been previously visited.

**Example:** In Figure 3, set \( d = 1 \). \( RS(n_0, n_0) = 1 \), \( RS(n_0, n_1) = (1/2)/4 \), \( RS(n_0, n_2) = ((1/3) + (1/2))/4 \), \( RS(n_0, n_3) = ((1/3) + (1/4))/4 \).

Finally, the overall redundancy of the graph in Figure 3, can be computed as \( R(C) = RS(n_0, n_0) + RS(n_0, n_1) + RS(n_0, n_2) + RS(n_0, n_3) = 1 + (1/8) + (5/24) + (7/48) = 1.48 \).

As mentioned previously one can use a hybrid approach to cluster, \( h(p_1, p_2) \), where the relationship between two pattern \( p_1 \) and \( p_2 \) is evaluated as:

\[
h(p_1, p_2) = \beta R(p_1, p_2) + (1 - \beta) S(p_1, p_2)
\]

where \( S(p_1, p_2) \) is the similarity between patterns \( p_1 \) and \( p_2 \) and \( R(p_1, p_2) \) is the redundancy measure between these two patterns. The similarity approach used for \( S \) is chosen based on the type of patterns to be clustered, and is varied between MDL or FPC. We do not combine MDL and FPC due to performance reasons. \( \beta \) and \((1 - \beta)\) are the weights assigned which are determined experimentally given a target task. Intuitively, the assignment of a higher weight to \( S \) produces a better pruning ability for the resulting cluster and thus is more applicable as an index. Alternatively, increasing the weight for \( R \) provides better memory and prefetch opportunities which can lead to better execution speeds, thus more applicable for secondary storage.

**E. Compression**

Given DFAs that contain a high degree of redundancy, the next step is to cluster them into a single DFA, say \( A \), and then compress the transition table of \( A \). Let \( T_A[v, i, u] \) be the transition table (matrix) indicating that by seeing input \( i \) in state \( v \), \( A \) would transition to state \( u \).

Our first attempt is to compress \( M \) by considering a global list of transitions. For each label and destination \((i, u)\) respectively, the global list stores only one row of data. Thus, if multiple states have a transition with the same label to the same state in the DFA, only one transition would be stored in the transition table.

The above mentioned compression technique is effective enough for the cases where redundant transitions predominantly appear only between two states. However, this is not always the case, and when merging a large number of patterns, the majority of redundant transitions may appear between \( n \) states. For these cases, further compression can be achieved by using a bitmap based approach which is mostly inspired by the work in web graphs [14].

When the average in-degree of states is high, there is a good chance that some of the states share several transitions. State \( i \) and state \( j \) share the same transition if on input \( a \) both \( i \) and \( j \) transit to the same state. Thus, instead of separately referring to the same set of transitions, we can construct the transition list of one state based on the list of transitions of another state.

More specifically, consider state \( s \) that shares some transitions with states \( s_1, s_2, ..., s_m \). From this list, we pick one state that shares the greatest number of transitions with \( s \) and then the second state that shares the greatest number of transitions with the remaining transitions of \( s \). This last step would be repeated until the number of selected items is greater than a predefined threshold \( k \).

Let states \( r_1, r_2, ..., r_k \) be the selected states, and \( t_1, t_2, ..., t_j \) be those transitions in \( s \) which are not covered by these selected states. Now to present the transition list of \( s \), we use a
bitmap of transitions used from each states in \(r_1, r_2, ..., r_k\) and a link to the remaining transitions \(t_1, t_2, ..., t_j\). The bitmap for state \(r_i\) simply indicates which transition of \(r_i\) also appeared in \(s\).

Note that in order for the above approach to reduce the space usage, \(s\) should have at least 2 transitions in common with each of state \(r_i\). This is because for each bitmap construct, for example for state \(r_i\) we need one word for the link to the state \(r_i\) and one word for the bitmap itself. However, the experiments show that \(k\) should be kept as small as possible to make the process of remaking transition lists faster.

1) Optimal Number of Clusters: The optimal number of clusters depends on the average distance \(f\) between existing pattern clusters. Equation 5 computes the average distance between all clusters using distance metric \(D\). Given a maximum similarity threshold \(T\) and available memory \(M\) we merge the clusters until either \(f(D, K) < T\) or until the memory utilized by the clusters is greater than \(M\).

The focus of this paper however is specifically on distance metrics and how these distance metrics can assist in clustering patterns of various properties. Nevertheless the optimization of the search for optimal number of pattern clusters is critical and it is an active interest of our future work.

\[
f(D, K) = \frac{1}{K^2} \sum_{i=1}^{K} \sum_{j=1}^{K} D(i, j)
\]  

Equation 5

F. Execution

Prefetching: We define prefetching as loading a state’s transition table before it is required. The success of our prefetchers is directly related to the average in-degree of the states. Clustering using a distance metric based on redundancy aims at grouping REs such that the opportunities for prefetching are maximized thereby increasing the potential throughput. In this paper, we have investigated the possibility of combining clustering and prefetching together in order to improve the performance further. Note that by using simple LRU technique for prefetching, we get poor results as compared to prefetching states with a high in-degree.

Increasing the cache size (used for prefetching) by a significant factor does not heavily improve performance of our prefetcher however the performance of LRU is greatly improved. As we will see, the performance of our prefetcher with a small cache is equivalent to LRU with a large cache.

Uniform prefetching: As a baseline, we simulate uniform prefetching which works by prefetching \(d\) pages at each prefetching state. This works much like the LRU prefetcher and is used as a baseline for comparison.

Prefetching using Redundancy Metric: We utilize the redundancy to pick which nodes to prefetch. Specifically, at each state the amount of prefetching to be done is directly proportional to the average redundancy of surrounding nodes.

The prefetching based on the redundancy metric is a probabilistic metric, and the probability of a node being prefetched \(P(n)\) is directly proportion to \(n\)'s average in-degree. In our experiments we define \(\epsilon\) which corresponds to the average in-degree threshold value node \(n\) has to have in order to be prefetched. Note that we prefetch a maximum of \(m\) nodes.

G. Complexity of Algorithms

The running times of proposed distance metric algorithms are described in this section.

As with MDL and FPC, the Redundancy metric must be computed efficiently to avoid drastically impacting the overall system performance. Due to a potentially large number of nodes in a cluster, we look at a pruning technique which is adopted by our approach and analyze the time and the space complexity of the resulting algorithm.

To compute redundancy, the length \(l\) of path \(p \in PATH(n_0, n_i)\) is computed for every node \(i\) from a starting node \(n_0\) which has complexity of \(O(n^2)\) where \(n\) is the number of nodes. Computing redundancy for node \(i\) takes constant time given length of \(p\). Thus, computing redundancy for all nodes in a cluster has complexity of \(O(n^2)\). However as we see next, all nodes do not have to be considered to have a good approximation for redundancy.

Observe that as the length of \(p \in PATH(n_0, n_i)\) increases the redundancy drops substantially. Therefore distant nodes have little impact on the overall redundancy score. Thus, the efficiency of redundancy computation can be substantially increased by a pruning technique by limiting the depth \(h\) of our DFT algorithm. Although we have a tradeoff between accuracy and efficiency, the amount of efficiency gained by utilizing the pruning technique is much more than the amount of accuracy achieved. The time complexity therefore is reduced to \(O(m^2)\) where \(m\) is the number of nodes up to level \(h\). Given \(t\) as the average degree nodes, \(m = t^{h+1} - 1\).

In the case of FPC, the propagation graph iteratively computes the similarity values for all possible pairs of nodes. In other words, FPC updates the similarity of \(N^2\) pairs a constant number of times. We can improve the algorithm running time to \(O(m^2)\) by using the pruning technique presented for Redundancy computation.

MDL runs in linear time with respect to the size of the cluster if a constant number of input samples are used. Recall that MDL relies on a sample string \(s \in L(D)\) where \(L(D)\) is the language expressed by a given automaton \(D\). String \(s\) is encoded on \(D\) and the number of bits used for encoding is recorded. For a greater accuracy, the length of \(s\) and the number of strings encoded can be increased.

IV. Experiments

In this section, we evaluate our proposed distance metrics used for clustering patterns.

Comparisons are made by measuring the memory usage and the overall throughput of the underlying automaton representing a cluster of patterns. For the redundancy metric, we evaluate what impact prefetching has on memory reads, and consequently, on throughput. Furthermore, to gauge the efficacy of our approach we compare its performance against the unmodified version of Snort [25]. We modify Snort to encompass the pattern clustering techniques discussed in this
paper, and evaluate the effectiveness of our clustering approach as it pertains to reducing memory usage, increasing throughput and increasing the potential for compression opportunities. When compared to the unmodified version of Snort, our framework provides a five-fold speed increase for traffic dumps obtained from MIT Lincoln Labs [26].

A case is made that shows what types of patterns are best suited for the three distance metrics. We found that the redundancy distance metric is best suited for a pattern set containing random segments of similarity where as FPC and MDL, respectively, are best suited for pattern sets where similarity segments between patterns are located towards the prefix and distributed randomly.

**Overhead of our approach:** We do not measure the clustering performance of our approach as it depends on a particular clustering algorithm used, which is not the focus of the paper. Instead we measure how much benefit can be derived from the distance metrics used. Furthermore the overhead of the larger data structure is offset by the memory savings as shown in Figure 5.

### A. Testbed and Methodology

We use real data sets employed in networking applications, specifically we select REs from the Snort system [25] which contains 1555 regular expressions used for intrusion detection. To simulate network traffic we use a recorded network dump from the MIT Lincoln Labs [26]. The MIT network dump used contains packets that are very long, the average packet payload length being 507.386 bytes.

To convert the pattern into an NFA or a DFA we use a Java package provided by Aarhus University. We do not implement a string matching algorithm, as it is already provided by the underlying Snort system. All the experimental results were obtained on a PC with 2.53 Ghz CPU and 4GB of memory.

Because clustering is an unsupervised technique, the goodness of the result is usually measured by the cost function. The cost function is usually expressed in terms of a distance measure between clusters and between objects within a single cluster. The first two distance metrics, MDL and FPC, group REs generally based on large segments of similarity. The third metric, based on redundancy does not group together necessarily similar objects, but instead it groups together those objects that may be largely dissimilar but produce the most compact representation when compression is used. This is because random portions of patterns that are similar or the presence of *kleene closures* produce states with large in-degree that can be effectively compressed.

**Theoretical Intuition:** The experimental results observed closely follow our theoretical model. For theoretical interpretation we evaluate our clusters via two categories: Execution speed and Memory usage.

For the execution speed we define a Potential Prefetching and Potential Memory Reads for each cluster. Potential prefetching for cluster \( C \) is defined as:

\[
P(C) = \frac{1}{n} \sum_{i} \max(o_i)
\]

Where \( n \) is the number of automatons states in cluster \( C \), and \( o_i \) represents the out-degree of state \( i \) of cluster \( C \). Thus \( P(C) \) can be interpreted as the potential prefetching opportunities given that we can only prefetch one state.

Potential memory reads is defined as follows [0,1]:

\[
M(C) = \frac{1}{n} \sum_{i} t_i
\]

Which can be interpreted as the amount of transitions we have to load per each state, where \( t_i \) is defined as the number of transitions for state \( i \).

For memory usage, we define **Potential Compression** as follows:

\[
K(C) = \frac{1}{n} \sum_{i} y_i
\]

Which can be interpreted as simply the average amount of in-transitions with the same label, where \( y_i \) is the number of incoming transitions for state \( i \) with the same label. Note that the condition that the incoming transitions must have the same label may seem restrictive to achieve high compression rates nevertheless techniques such as [14] allow a more flexible approach to compressing automata but with a higher processing cost. The balance between the amount of compression and processing cost is part of our future work.

### B. Effects of Redundancy on Potential Compression

In this section, we describe how clustering based on the redundancy distance metric can potentially influence the compression rate. To run the experiments we take 1000 random Snort patterns and create 10 clusters. The redundancy and compression results presented are respective averages of the 10 clusters. Intuitively, if there is a lot of redundant information in the transition table, then we can effectively use compression techniques such as [14], [27] to compress the redundant information further. In this experiment we vary the explore depth \( h \) of our DFT algorithm and observe how it affects the redundancy metric. Given the redundancy metric, we also observe how much compression we can achieve. We utilized a simple compression scheme that maps \( n \) identical transitions into a single transition. Compression is computed as \( O_{\text{states/transition}}/N_{\text{states/transition}} \) where \( O_{\text{states/transition}} \) is the number of states and transitions before compression and \( N_{\text{states/transition}} \) is the number of states and transitions after compression. Redundancy is computed as described previously.

As can be observed, the redundancy quickly levels off as the DFT depth increases. This was the main insight behind the approximate computation of the redundancy metric presented in the previous section. Experimentally we can see that setting the DFT depth for our data-set (Snort REs) beyond 5 does not
produce much benefit. Note that the potential compression corresponds to only those nodes located at maximum depth $h$. Observe also that there is a direct correlation between redundancy and compression. It is important to note that both compression and redundancy can spike at higher DFT depth $h$, because seldom there are states at high $h$ that contain kleene closures and therefore a high unweighted redundancy. In our case such states are rarely visited therefore as $h$ increases such states have less and less impact on the redundancy measure.

C. Similarity, Redundancy and Original Snort

In this section we examine how similarity and redundancy approaches complement each other. We have used 1000 Snort patterns. A total of 10 clusters was created using our approach. As a baseline we use Snort (no clustering) and as input we use a sample network traffic. Note that in Figure 5 Snort’s original memory usage and throughput are approximately constant. We show that by only utilizing redundancy, or by only utilizing similarity approaches, we achieve a worse throughput and memory utilization compared to when the hybrid approach is used. Recall that we assign weights $\alpha$ and $\beta$ to similarity and redundancy votes respectively. The weights depend on the application and the dataset. Figure 5 shows that given a mixed pattern dataset (i.e. where similarity between patterns occurs at random segments of the pattern) the hybrid approach clearly achieves a better result. Table II shows the corresponding empirically obtained optimal weights for $\alpha$ and $\beta$ given the Snort pattern data-set. Note that the distance metric used for $\alpha$ can be varied between MDL and FPC depending on the similarity being more prevalent in the prefix (FPC) or randomly distributed (MDL) in a set of patterns. For these experiments we used MDL. Also note that if only redundancy is used, the throughput is lower than when the hybrid approach is employed. This is because if the redundancy metric is used on patterns without kleene closures and with long segments of similarity, the resulting clusters do not contain enough redundant transitions to take advantage of prefetching and compression.

D. Redundancy and Optimization

In this section, we examine how prefetching can be used to speed up the matching time. Specifically, we explore the potential prefetching approach and potential number of memory reads. In this experiment we measure cache fault rates, which correspond to the number of memory reads, when the number of nodes prefetched is varied. The graph below examines the uniform prefetcher and the prefetcher utilizing the node’s redundancy metric based models. When using the uniform prefetcher, each node is prefetched with probability $1/Out(N)$ where $Out(N)$ is the number of outgoing transitions of node $N$. We can see that the uniform prefetcher performs poorly compared to the prefetcher utilizing the redundancy metric information. The uniform prefetcher essentially simulates the LRU cache when maximum nodes prefetched is 1. Prefetching a larger number of nodes clearly narrows the gap between the redundancy-aware and a uniform prefetcher methods. Also note that prefetching more than five nodes produces little results for both approaches.

E. Effects of Number of Clusters on Memory

In this section, we observe how the number of clusters affects memory usage when using only redundancy or only similarity distance metrics. As observed from Figure 7, when using redundancy the overall memory usage is much lower.
than when using similarity. We can also observe that memory does not grow linearly with the number of clusters, which is due to the fact that as the number of clusters varies, the cluster compression does also. Seldom is the case that more clusters in fact require less amount memory. This is mainly because with fewer clusters, dissimilar patterns might have been merged requiring relatively large amounts of memory.

Observe that obtaining the optimal number of clusters is a difficult problem. In our approach we merge patterns until we run out of pre-allocated space. We can however utilize the local change in memory and in the number of states to derive the approximation for when to terminate our merging algorithm. This derivation of approximate number of pattern clusters is however part of our future work.

### V. Conclusion

In this paper, we presented a framework for regular expression clustering and pattern matching over streaming data. We presented two similarity distance metrics, as well as a redundancy distance metric which relies on transition elimination between states in a cluster of REs. We outlined how a hybrid similarity and redundancy metric would function as well as the role of prefetching and compression techniques in our framework. Additionally, we identified the types of patterns that are best suited for each of the three distance metrics presented. Lastly, by experimenting with real world data, we displayed the efficacy of our methods and showed that a redundancy metric can be an effective way to perform clustering. Our initial investigation with the new redundancy metric and with the similarity metrics indicates promising results and are worth further investigation by those interested in clustering approaches.

### REFERENCES


