Frequent Subtree Mining on the Automata Processor: Challenges and Opportunities

Elaheh Sadredini
Dept. of Computer Science
University of Virginia
Charlottesville, VA, 22903 USA
elaheh@virginia.edu

Reza Rahimi
Dept. of Electrical and Computer Eng.
University of Virginia
Charlottesville, VA, 22903 USA
rahimi@virginia.edu

Ke Wang and Kevin Skadron
Dept. of Computer Science
University of Virginia
Charlottesville, VA, 22903 USA
⟨kewang,skadron⟩@virginia.edu

ABSTRACT

Frequency counting of complex patterns such as subtrees is more challenging than for simple itemsets and sequences, as the number of possible candidate patterns in a tree is much higher than one-dimensional data structures, with dramatically higher processing times. In this paper, we propose a new and scalable solution for frequent subtree mining (FTM) on the Automata Processor (AP), a new and highly parallel accelerator architecture. We present a multi-stage pruning framework on the AP, called AP-FTM, to reduce the search space of FTM candidates. This achieves up to 353× speedup at the cost of a small reduction in accuracy, on four real-world and synthetic datasets, when compared with PatternMatcher, a practical and exact CPU solution. To provide a fully accurate and still scalable solution, we propose a hybrid method to combine AP-FTM with a CPU exact-matching approach, and achieve up to 262× speedup over PatternMatcher on a challenging database. We also develop a GPU algorithm for FTM, but show that the AP also outperforms this. The results on a synthetic database show the AP advantage grows further with larger datasets.

CCS CONCEPTS

•Information systems →Frequent subtrees; •Computer systems organization → *Multiple instruction, single data;* **•Hardware** →Emerging architectures;

KEYWORDS

Frequent subtree mining, finite automata, the automata processor, GPU, heterogeneous architecture

ACM Reference format:

Elaheh Sadredini, Reza Rahimi, and Ke Wang and Kevin Skadron. 2017. Frequent Subtree Mining on the Automata Processor: Challenges and Opportunities. In *Proceedings of ICS '17, Chicago, IL, USA, June 14-16, 2017,* 11 pages.

DOI: http://dx.doi.org/10.1145/3079079.3079084

ICS '17, Chicago, IL, USA

DOI: http://dx.doi.org/10.1145/3079079.3079084

1 INTRODUCTION

Frequent subtree mining refers to finding all the patterns in a given forest (database of trees) whose support is more than a given threshold value, called the minimum support. A subtree pattern is called frequent if the number of trees in the dataset that have at least one subtree isomorphic to the given pattern is more than the minimum support. Frequent subtrees have proven to be extremely important and informative in many real world applications such as XML data, parse-trees in natural language processing, bioinformatics, and patient treatment awareness. For instance, in natural language processing (NLP), frequent subtrees mined from the parse tree databases can be used to increase the accuracy of NLP tasks, such as sentiment analysis and question classification problems [1]. However, finding all frequent subtrees becomes infeasible for a large and dense tree database, due to the combinatorial explosion of the subtree candidates.

The main challenges in FTM are efficiently traversing the search space and performing subtree isomorphism. A number of research studies have attempted to improve the performance of the task by proposing different data structures and counting strategies. These are either based on breadth-first search (BFS) or depth-first search (DFS). BFS is a level-wise iterative search method and usually uses a horizontal tree representation. BFS suffers from a long processing time because it requires passing through the entire dataset in each iteration. However, DFS usually projects the database into a vertical tree representation for fast support counting, but encounters memory capacity challenges and costly I/O processing because the set of candidates and their embedding list tend to overflow memory.

Researchers are increasingly exploiting accelerators as performance growth in conventional processors is slowing down. The Micron Automata Processor (AP) is a non-von Neumann, native-hardware implementation of non-deterministic finite automata (NFA). The high bit-level parallelism of the memory-based architecture makes it capable of performing high-speed search and analysis on complex data structures. Recent studies on frequent itemset mining [12], sequential pattern mining [14], disjunctive rule mining [13], random forests [9], and entity resolution [2] have proved that the AP is a promising target accelerator in data-mining, machine learning, and data-matching applications, and these studies have achieved orders of magnitude speed-up over conventional processors. However, the main difficulty in exploiting the AP for FTM is that the AP was intended to support regular languages, whereas tree structures will typically need to be represented by a context-free grammar. By relaxing some of the tree structure constraints, the AP can be effectively utilized to prune the search space of FTM.

In this paper, we first study difficulties of directly implementing the FTM problem on the AP platform. Then, we propose a multi-stage pruning framework to greatly reduce the search space of embedded FTM on the AP. This provides a scalable solution in terms of both memory and execution time on large databases and lower support thresholds. Frequent subtree candidates can be the potential features in classification tasks, and the surviving candidates with lower frequency are especially beneficial to boost classification accuracy of rare classes, because these frequent subtree patterns can represent unique and discriminative features of classes with fewer members. In order to maintain both ancestor-descendant relationship and sibling properties on the tree structure, and at the same time provide a feasible computation to exploit the AP, four complementary string representations of the tree structure and their mapping to the automaton representation are proposed. Pruning kernels presented in this work result in a set of potentially frequent candidates (close to the final set of frequent patterns) which may contain a small number of false positives (however, recall is 100%). For the applications that demand an exact solution, we adapt TreeMinerD [16], a quick DFS approach that detects the distinct occurrences of a pattern, to prune the AP results and provide an exact solution. Finally, we develop a BFS-based solution for embedded FTM on GPU in order to compare the AP solution with an additional accelerator architecture. Overall, this paper makes following principle contributions:

- We propose AP-FTM, a multi-stage pruning framework on a heterogeneous architecture, to reduce the search space of FTM for embedded subtrees. This eliminates a large portion of infrequent candidates in a short amount of time at the expense of a small number of false positives, providing a scalable solution for large databases and lower support thresholds.
- We propose APHybrid-FTM, which combines AP-FTM with TreeMinerD in order to provide an exact and efficient hybrid solution for embedded FTM. The general strategy of combining an inaccurate and fast AP-based solution with an efficient and accurate CPU-based approach can be applied to other complex pattern mining tasks.
- We develop a BFS-based algorithm for embedded FTM on GPUs platform and study the performance of a SIMT platform for the FTM problem.

2 FREQUENT SUBTREE MINING

2.1 Problem Statement

Tree Mining Problem: We define D to be a dataset of trees and a transitive subtree relation $S \preceq T$ for some of the trees (T) in D. Define $t_1, t_2, ..., t_n$ to be the nodes in T and $s_1, s_2, ..., s_m$ be the nodes in S. Then, S is a subtree of T if there are matching labels of the nodes $t_{i_1}, t_{i_2}, ..., t_{i_m}$ such that (1) $label(s_k) = label(t_{i_k})$ for all $k = 1, 2, ..., m$; and (2) for every branch (s_j, s_k) in S, t_{i_j} should be an ancestor of t_{i_k} in T. The latter condition preserves the structure of S in T. This definition of subtree refers to an *embedded subtree*. By restricting the ancestor-descendant relationship to parent-child relationships in T for the second condition, a new kind of subtree, called *induced subtree*, can be defined. Fig. 1 shows an example on different types of subtrees on T_0. There are several other subtree types such as maximal subtree, closed subtree, and partial subtree, which put restrictions on the induced and embedded subtrees and are not considered in this work.

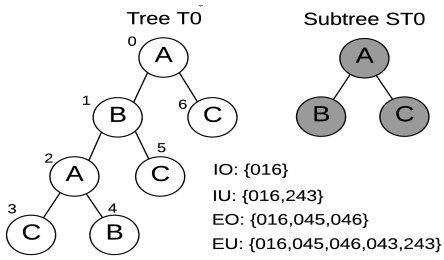


Figure 1: An example of subtrees (I = Induced, E = Embedded, O = Ordered, U = Unordered)

The relative minimum support number (*Rminsup*), defined as the ratio of minimum support number to the total number of transactions (input trees), is used in this paper. We define the size of a tree as the number of nodes in it. Moreover, we represent a candidate of size k with *k-candidate* and a frequent candidate of size k with *k-frequent-candidate* throughout the paper. Many applications are only interested in counting the number of database trees that contain at least one match of a subtree, which is called counting distinct occurrences. On the other hand, weighted counting refers to enumeration of all possible occurrences over all possible trees in the database. In this work, we focus on mining distinct occurrences of embedded subtrees from rooted, ordered, and labeled trees as those types of datasets dominate in data mining applications [16]. Embedded subtree mining has a larger search space and higher mining complexity than induced subtrees [3] [16], and CPU solutions have difficulties dealing with them. The proposed pruning method is not limited to binary trees and can be adopted by unordered embedded and ordered/unordered induced subtree mining with minimal changes.

2.2 Candidate Generation

Our candidate-generation step is based on an equivalence-class, right-most extension approach adopted from [15]. In this approach, the *(k+1)-candidates* are generated from the known *k-frequent-candidates* within an equivalence-class (having the same string prefix). Two frequent patterns can be merged based on the position of the last extended node. In this approach, all the candidates are generated once (avoiding redundancy) and all are the valid candidates. We do not describe the candidate-generation in detail, as the main focus of the paper is accelerating the candidate enumeration step, which is the bottleneck of the algorithm. Details of the candidate-generation and proof of correctness can be found in [15].

3 RELATED WORK

A considerable amount of research has been devoted to frequent subtree mining, due to its significance in different domains such as bioinformatics, web mining, and natural language processing. Frequent subtree mining techniques can be classified based on the subtree and the ordering types to induced ordered, induced unordered, embedded ordered, and embedded unordered subtrees. The way the candidate patterns are generated, the data structure representation in the memory, and the candidate subtree enumeration approach can significantly affect the efficiency of the algorithm.

Several algorithms have been proposed to mine labeled, embedded, and ordered subtrees. TreeMiner [15] is the first algorithm for mining embedded ordered subtrees, suggested by Zaki, which

is based on DFS search, and it introduces the concept of vertical tree representation, a space-efficient string encoding of the tree. New candidates are generated by adding one node to the right-most path of the tree (right-most extension approach). TreeMiner uses an *extension and join* approach for the candidate-generation and enumeration and stores the matches in a vertical representation, which can be very large and consume a lot of memory, especially when the number of overlapping matches is high. The same author proposes TreeMinerD [16], an algorithm which enumerates only distinct occurrences of a pattern, and can be beneficial for some applications. However, when the average number of embeddings of a subtree in a tree is low, TreeMiner and TreeMinerD have almost the same performance. XSpanner [11] is another solution for mining embedded, ordered subtrees and adopts the *FP-Growth* concept and its enumeration model generates valid candidates, and counts the distinct trees. An expensive pseudo-projection phase results in poor cache performance in XSpanner. The idea of pseudo-projection techniques is that, instead of physically constructing a copy of the subtree, they reuse the trees in the original tree database. MB3-Miner [6], yet another solution, applies a *Tree Guided Model* to efficiently generate the candidates. However, Tatikonda in [7] shows that the MB3-Miner solution suffers from high memory usage.

The TRIPS and TIDES [8] solutions are proposed to mine embedded and induced subtrees that can be ordered or unordered. They are based on sequential encoding strategies that provide fast generation of a complete and non-redundant set of candidate subtree patterns. They use an embedded list for the candidate-generation and a hash table for the support counting step. Even though their approaches are cache-conscious due to the simple array-based data structure, they still suffer from high memory consumption with larger datasets and with smaller support thresholds. The same authors proposed an architecture-aware FTM algorithm [7] targeting multi-core systems. Several optimizations are adopted to decrease memory access latency and bandwidth pressure, and a parallel *pattern-growth* approach in the context of the TRIPS [8] algorithm on multicore systems is proposed. They show a nice scale-up with the number of cores, however, their solution crashes far sooner than a single-core implementation. This work is the only parallel solution for FTM, to the best of our knowledge.

PATTERN-MATCHER [15] instead employs a breadth-first iterative search to find frequent subtrees. It employs an equivalence-class notion for candidate-generation and counting, and a prefix-tree data structure for indexing the candidates. It prunes the candidates of size $(k + 1)$ using the frequent candidates of size k. We consider this algorithm as a baseline to compare with TreeMiner. Furthermore, a level-wise push-down automata-based approach for the candidate enumeration step of FSM is proposed in [5]. A deterministic finite automata is generated for each candidate and the experiments are run on a Pentium 4 CPU. However, the authors do not compare the performance of their method with the *FP-growth* algorithms, which are known to be faster than the level-wise solutions.

Chopper [11] proposes a two-stage solution to find frequent subtrees. In the first stage, the database is converted to a preorder traversal label sequence representation and then, frequent sequences are found using sequential pattern mining, which acts as the pruning stage. Then, frequent subtrees are found by removing the false positives. XSpanner outperforms Chopper [11] and Tree Miner [15] outperforms Xspanner, so we do not consider them in the performance comparison. Furthermore, there is no parallel implementation of frequent subtree mining problem on GPU or FPGA architectures.

4 AUTOMATA PROCESSOR

Micron's Automata Processor (AP) [4] is an in-memory, non-von Neumann processor architecture that computes non-deterministic finite state automata (NFAs) natively in hardware. The AP allows a programmer to create NFAs and also provides a stream of input symbols to be computed on the NFAs in parallel. This is a fundamental departure from the sequential instruction/data addressing of von Neumann architectures. A benchmark repository for automata-based applications is presented in [10].

Specifically, the AP is a reconfigurable fabric of State Transition Elements (STEs), counters, and boolean gates. Each STE is capable of matching a set of any 8-bit symbols and activating a set of following STEs on a match. Counter and boolean elements are designed to extend computational capability beyond NFAs. The counter element counts the occurrence of a pattern described by the NFA connected to it and activates other elements or reports when a given threshold is reached. The counters in particular are useful in frequent itemset mining (FIM) [12] and sequential pattern mining (SPM) [14], for counting occurrences against the support threshold. The matching and counting stage of FIM and SPM map to the AP architecture naturally. We transform the matching and counting stage of FTM to several simpler kernels equivalent to the FIM and SPM methods in order to prune the huge search space of the FTM problem and provide a scalable solution for large databases.

Micron's current-generation AP-D480 boards use AP chips built on 50nm DRAM technology, running at an input symbol (8-bit) rate of 133 MHz. A D480 chip has 192 blocks. Each block has 256 STEs, 4 counters and 12 Boolean elements [4]. We assume an AP board with 32 AP chips, so that all AP chips process input data stream in parallel. Each AP D480 chip is projected to have a worst case power consumption of 4W [4].

4.1 Input and Output

The AP takes input streams of 8-bit symbols. The double-buffer strategy for both input and output of the AP chip enables an implicit data transfer/processing overlap. Any type of element on the AP chip can be configured as a reporting element (i.e., accepting state); one reporting element generates a one-bit signal when the element matches the input symbol. If any reporting element reports on a particular cycle, the chip will generate an output vector for all the reporting elements If too many output vectors are generated, the output buffer can fill up and stall the chip. Thus, minimizing output vectors, and hence the frequency at which reporting events can happen, is an important consideration for performance optimization. To address this, we will use the structures that wait until a special end-of-input symbol is seen to generate all of its reports in the same clock cycle.

4.2 Programming and Configuration

Automata Network Markup Language (ANML), an XML-like language for describing automata networks, is the most basic way to program AP chip. ANML describes the properties of each element and how they connect to each other. The Micron's AP SDK also provides C, Java and Python interfaces to describe automata networks, create input streams, parse output and manage computational tasks

on the AP board. A "macro" is a container of automata for encapsulating a given functionality, similar to a function or subroutine in common programming languages. Macros can be templatized (with the structure fixed but the matching rules for STEs and the counter thresholds to be filled in later).

Placing automata onto the AP fabric involves three stages: placement and routing compilation (PRC), routing configuration and STE symbol-set configuration. In the PRC stage, the AP compiler deduces the best element layout and generates a binary of the automata network. Depending on the complexity and the scale of the automata design, PRC takes several seconds to tens minutes. Macros or templates can be precompiled and composed later. This shortens PRC time because only a small macro needs to be processed for PRC, and then the board can be tiled with as many of these macros as fit.

Routing configuration/reconfiguration programs the connections, and needs about 5 ms for a whole AP board. The symbol set configuration/reconfiguration writes the matching rules and initial active states for the STEs and takes 45 ms for a whole board. A precompiled automata only needs the last two steps. If only STE states change, only the last step is required. This feature of fast partial reconfiguration plays a key role in a successful AP implementation of FTM: the fast symbol replacement helps to deal with the case that the total set of candidate patterns exceeds the board capacity; the quick routing reconfiguration enables a fast switch from k to $k+1$.

5 FTM ON THE AP: CHALLENGES

Subtree inclusion checking cannot be accomplished using deterministic finite state machines. The tree structure is more complex than a sequence and cannot be described with regular languages [5]. It implies that instead of a finite state machine, a pushdown automaton (PDA) is needed in order to count the length of a possible branch when searching for a subtree in the input tree. A PDA is a finite automaton with access to a potentially unlimited amount of stack, which is more capable than finite state machines.

We concluded that the AP is an excellent accelerator to *prune* the search space of the candidates in FTM, when relaxing some of the tree constraints in order to make the simpler representations of a tree. In the following section, we propose a set of pruning kernels implemented on the AP to shrink the subtree candidate-set size, which provides a scalable solution to the larger databases and lower support thresholds.

6 FTM ON THE AP: OPPORTUNITIES

In this section, we propose four kernels, *(1) subset pruning, (2) intersection pruning, (3) downward pruning, and (4) connectivity pruning*. The first two kernels are independent from the input transaction, while the last two create a new presentation of the trees in the database and use them as the input stream to match against the candidates. The proposed kernels are complementary to each other to avoid overlapping pruning and applied to the candidates in sequence to accommodate more candidates in the early stage.

6.1 Preliminaries

Frequent Itemset Mining (FIM): We define $I = i_1, i_2, ..., i_m$ as a set of interesting items. Let $T = t_1, t_2, ..., t_n$ be a database of transactions, each transaction t_j is a subset of I. Define $x_i = \{i_{s1}, i_{s2}, ..., i_{sl}\}$ be a set of items in I, called an itemset. The itemset with k items is called k-itemset. A transaction t_p is said to cover the itemset x_q *iff* $x_q \subseteq t_p$. The support of x_q, $Sup(x_q)$, is the number of transactions that cover it. An itemset is frequent *iff* its support is greater than a given threshold value called minimum support, *minsup*. The goal of FIM is to find all itemsets with support greater than *minsup*. Wang et al. [12] proposed a novel automaton template (this is valuable because routing reconfiguration is slower than symbol replacement) for matching and counting stage of FIM on the AP that can handle variable-size itemsets (ME-NFA-VSI) and avoid routing reconfiguration. The whole design makes full usage of the massive parallelism of the AP. By using this template structure, one AP board can match and count 18,432 itemsets in parallel with sizes from 2 to 40 for 8-bit encoding and 2 to 24 for 16-bit encoding (for symbol alphabets > 256). Note that the processing rate is 133 MB/s regardless of encoding.

Sequential Pattern Mining (SPM): We define $I = i_1, i_2, ..., i_m$ as a set of items, where i_k is usually represented by an integer, call item ID. Let $s =< t_1 t_2 ... t_n >$ denotes a sequential pattern, where t_k is a transaction and also can be called as an itemset. We define an element of a sequence by $t_j = \{x_1, x_2, ..., x_m\}$ where $x_k \in I$. We assume that the order within a transaction (itemset) does not matter, so the items within one transaction can be *lexicographically ordered* in preprocessing stage. A sequence with a size k is called a k-sequence. Sequence $s_1 =< t_1 t_2 ... t_m >$ called to be a subsequence of $s_2 =< r_1 r_2 ... r_j >$, if there are integers $1 \preceq k_1 \prec k_2 \prec .. \prec k_{m-1} \prec k_m \preceq j$ such that $t_1 \subseteq r_{k1}, t_2 \subseteq r_{k2}, ..., t_m \subseteq r_{km}$. Such a sequence s_j is called a sequential pattern. A sequence is known as frequent *iff* its support is greater than a given threshold value called minimum support, *minsup*. The goal of SPM is to find out all the sequential patterns with support greater than *minsup*. In [14] Wang et al. derive a compact automaton design for matching and counting of SPM on the AP. A key insight that enables the use of automata for SPM is that hierarchical patterns of sequences can be flattened into strings by using delimiters and place-holders. Again, a template is proposed to accommodate variable-structured sequences. This allows a single, compact template to match any candidate sequence of a given length, so this template can be replicated to make full use of the capacity and massive parallelism of the AP. One AP board can match and count 6,144 sequence patterns in parallel with sizes from 2 to 20 for 8-bit and 16-bit encoding.

6.2 Pruning Kernels

We propose four pruning kernels in this section. Each kernel maps to FIM or SPM definition, and we use the automata structures proposed for FIM [12] and SPM [14] problems, from our previous works, to implement the kernels on the AP and calculate the AP board utilization.

6.2.1 Subset Pruning. According to the downward-closure principle, all sub-patterns of a frequent pattern must themselves be frequent. This means that, when generating a *(k+1)-candidate*, all of its *k-candidates* should be frequent as well. BFS-based FTM approaches can greatly benefit from this property in order to reduce the search space, whereas DFS implementations do not have all the *k-frequent-candidates* when looking at a *(k+1)-candidate*. The subset pruning kernel checks the downward closure property for all the candidates of size three and more. This property can be directly mapped to FIM, where each generated *(k+1)-candidate* represents a candidate itemset and the items in the itemset are the set

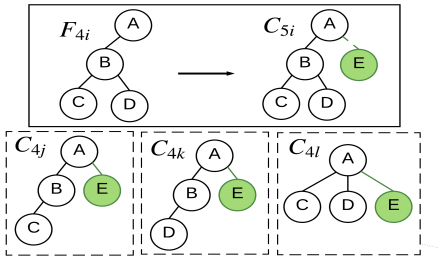


Figure 2: An example of subset pruning

of *k-candidates*. In Fig. 2, C_{5i} (a *5-candidate*) is generated from F_{4i}, which is a *4-frequent-candidate*, by extending the edge *AE*. In subset pruning, we should check C_{4j}, C_{4k}, and C_{4l}, the other subsets of C_{5i}, to verify they are frequent as well. The itemset candidate corresponding to C_{5i} is $C_{5i} = \{C_{4j}, C_{4k}, C_{4l}\}$ and the input dataset has only one transaction, which consists of all the frequent candidates of size 4, e.g., $\{F_{40}, F_{41}, ..., F_{4m}\}$, where m is the number of *4-frequent-candidates*. Therefore, the set of all C_5 creates the candidate itemsets for FIM. A subtree candidate will survive at this stage if it occurs in the input transaction (*Rminsup* is 100% here).

The CPU implementation adds each individual frequent subtree into a hash table. Thus, each subtree check takes O(1) time, and since there can be k subtrees of length *k-1* and n candidates, it takes O(nk) time to perform the pruning check for the patterns in each iteration. In the AP implementation, many candidate itemsets are configured on the AP and checked against the input transaction in parallel. The time complexity of the AP solution is O(m), where m is the number of frequent candidates of the previous level. Because the support threshold here is 100%, we can remove the counter element of the FIM AP design [12], which is the main constraint of the AP board utilization. When the number of generated candidates is relatively small, the CPU solution beats the AP, because of the AP configuration overhead. However, when the number of candidates starts to grow, the AP implementation provides a faster solution. This kernel is very light-weight and does not require a pass of input trees (the input for this kernel is the set of *frequent-candidates* of the previous iteration), however, it prunes a large number of candidates in the early stage.

6.2.2 Intersection Pruning. In order to pass this pruning stage, (1) all the subsets of a *(k+1)-candidate*, which are the members of a *k-frequent-candidates*, should happen in the same input tree, and (2) the number of joint occurrences must be more than the minimum support threshold. Let's assume C_{5i} from Fig. 2 has passed the subset pruning stage and all its subset has been frequent. Also, assume there is a database of four trees $\{T_1, T_2, T3, T4\}$, where F_{4i} occurs in $\{T_1, T_2, T_4\}$, F_{4j} occurs in $\{T_1, T_4\}$, F_{4k} occurs in $\{T_1, T_2, T_3\}$, and F_{4l} occurs in $\{T_1, T_4\}$. As we see, the set of $\{F_{4i}, F_{4j}, F_{4k}, F_{4l}\}$ (which are the subset of C_{5i}) jointly happens in only T_1. As a result, if the *Rminsup* is less than 25%, F_{5i} will pass the second stage, otherwise, it will be pruned.

Intersection pruning can directly map to FIM, where itemsets are the set of *(k+1)-candidates* and items in the itemsets are the set of *k-frequent-candidates* for each candidate. The number of input transactions is equal to the number of trees in the database and the size of each transaction is the number of frequent candidates contained in the transaction, which creates the AP input stream. If all the frequent candidates fit into the AP boards, one pass of the input stream checks the frequency of intersection pruning for all the candidates at the same time; otherwise, the automaton macros will be loaded with a new set of candidates, which requires another pass of the input stream. The CPU implementation uses a 1D array for each frequent candidate to list the set of trees in which it occurs. The size of the array is equal to the number of trees in the database.

6.2.3 Downward Pruning. To further prune the search space, the downward pruning kernel simplifies tree representation to a sequence of root-to-child paths in order to check the ancestor-descendant relationships of a subtree in an input tree. Clearly, the original tree cannot be constructed using these paths, but it has some unique properties which help to identify a set of frequent subtrees and reduces computational complexity.

Downward string representation (DSR): It starts from the root of the tree and traverses all the paths from the root to the terminal children. The delimiter ',' separates different paths and the delimiter '#' represents the end of the downward representation string of an input tree. When mining ordered subtrees, it is important to traverse from the left-most path to the right-most path in order to preserve the order. For example in Fig. 3, the vertical representation of subtree ST_2 is *AC*, *AB*#. When delimiter '#', encoded at the end of subtree downward representation, matches to the input stream, the associated counter counts up by one and then, matching for the next tree starts from the root of the subtree.

Downward Pruning on the AP: For all the surviving *(k+1)-candidates* from the previous stage, the DSR will be created. These candidates can be interpreted as the candidate sequences in SPM, where the nodes in a path represent an itemset and paths create the itemsets. The DSR for the input tree is considered as the input stream for this kernel. We adopt the SPM-AP implementation in [14] for this stage.

DSR creates a sequential pattern of the tree structure, which preserves ancestor-descendant relationships and ignores the sibling information. Fig. 3 shows the DSR of an input tree and three example subtrees. According to the SPM definition, both *DSR-ST*$_1$ and *DSR-ST*$_3$ are the subsequences of *DSR-T*$_0$, so they survive the downward pruning and will be checked further at the next pruning kernel. ST_3 is not a true subtree of T_0 and connectivity kernel, a complementary pruning strategy, will prune it in the next stage. *DSR-ST*$_2$ is not a subsequence of *DSR-T*$_0$ and ST_2 can be safely pruned from the set of candidate subtrees.

Downward pruning ensures that, for all the subtrees candidates with degree no more than one (we call them line-shaped candidates), the final decision regarding their frequency will be made at this stage and no false positive candidate will survive from this kernel. This is particularly true because line-shaped candidates are equivalent to an itemset in SPM, where no branching information is required. The quality of downward pruning directly depends on the topology of the input trees. Deeper trees will benefit more from the downward pruning.

6.2.4 Connectivity Pruning. Connectivity pruning addresses situations when the downward string representation generates two itemsets out of one node, which allows some false positives to survive downward pruning. Connectivity pruning finds a mapping of the subtree *root-path* to the input tree and then looks for the child sequences of the last node in the root-path from left to right in the tree. The root-path of a subtree is the path from the root to the first

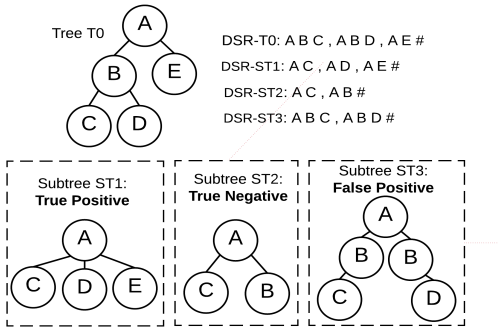


Figure 3: An example of downward pruning

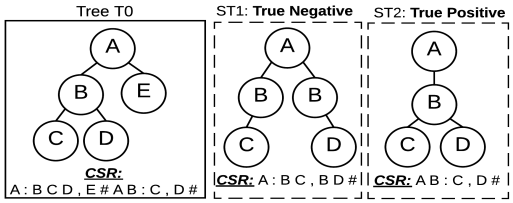


Figure 4: An example of connectivity pruning

node with degree more than one. For example, the root-path of ST_2 in Fig. 4 is AB.

Connectivity string representation (CSR): CSR of a subtree consists of the root-path followed by the delimiter ':', and then, the pre-order representation of the children from the left-most path to the right-most path separated by the delimiter ','. For example in Fig. 4, the CSR of ST_1 is $A : BC, BD\#$, where the root-path is A and the pre-order representations of its children are BC and BD, which are separated by the ','. In order to detect the subtree in an input tree, the CSR of the input tree should be extended by all the paths from the root to all the node with degree more that two. Take the input tree T_0 as an instance, where first, A is considered as the root-path and is followed by the left-side children (BCD) and the right-side child (E), and second, AB is considered as the root path followed by the B's children. Delimiter '#' separates root path sets in the trees and subtree inclusion checking starts from the subtree root after '#' appears in the input stream.

Connectivity Pruning on the AP: This kernel can directly map to the SPM, where the root path and children are the itemsets and the nodes are the items. In SPM, the order between the itemsets matters while the order between the items does not matter. However, having a pre-defined order of the items helps simplify the automata structure [14]. In sequences generated by CSR, both items and itemsets have a pre-defined ordering, which means that the CSR can be easily map to the SPM automata structures. Connectivity pruning does not cause any false negatives, because it just relaxes necessary tree structure properties in order to check subtree inclusion.

6.3 Pruning Corollaries

3-candidates can only have two different topologies; (1) a root and two children connect to the root (triangle-shaped), and (2) a root with one child and one grandchild (line-shaped). As discussed before, lined-shape patterns will be properly pruned in the downward stage. Connectivity pruning also perfectly trims triangle-shape ones. This is because the root path has just one node, which is the root itself and the left and right child are the only node and do not have hierarchical structure, and they only need to appear (in order) in two different branches of the equivalent tree node to the subtree root. These make all the surviving *3-candidates* in the final set to be true *3-frequent-candidates*. This property is very useful because more precise pruning in early iterations will greatly reduce the chance of getting false positives in the later stages.

6.4 Program Infrastructure

Fig. 5 shows the complete workflow of the AP-accelerated FTM proposed in this paper. The input database is in horizontal, string-encoded format (the horizontal format is the pre-order traversal of trees, including backtracking information). The following describes the data pre-processing steps:

Computing 1-frequent-candidates (F_1) and 2-frequent-candidates (F_2): To compute F_1, for each item (node label) in the tree, its count in a 1D array will be incremented, where the total time for each tree with size n is $O(n)$. Other database statistics, such as the maximum number of labels and number of trees, is calculated as well. F_2 counting is done using a 2D integer array of size $F_1 \times F_1$ and the total time is $O(n^2)$ per tree.

Recoding the input trees: Depending on the number of frequent items, the item can be encoded by 8-bit or 16-bit symbols. Different encoding schemes lead to different automaton designs and capacity of patterns for each pruning stage.

Making input streams: We create downward and connectivity string representation of the input trees according to the recoded items and keep them in the memory. After pre-processing and generating tree candidates, the corresponding AP input stream will be generated for each pruning stage. Then, the appropriate pre-compiled template macro of automaton structure for FIM or SPM pattern is selected according to k (size of itemset or sequence candidate) and is configured on the AP board. The candidates are generated on the CPU and are filled into the selected automaton template macro. The input data formulated in pre-processing is then streamed into the AP board for counting. While there are *k-candidates* left to be processed on the AP, the AP computation (symbol replacement and matching) and the AP input generation of the next-level pruning kernel can be done in parallel. Fortunately, the latency of symbol replacement could be significantly reduced in future generations of the AP (because symbol replacement is simply a series of DRAM writes), which would improve the overall performance greatly. At the end of connectivity pruning stage, either k has reached the maximum size or *k-frequent-candidates* set is empty, we have the approximate solution, which is a set of potentially frequent candidates. Depending on the final application, the approximate results can either be directly used with no further final pruning or can be considered as the ground candidate set for an exact FTM solution. We will later show how TreeMinerD [16] can be used to provide an exact solution over the AP results.

7 FTM GPU IMPLEMENTATION

Despite the DFS strategies, where memory becomes a limiting factor for performance and scalability, especially in large datasets and lower support thresholds, BFS solutions do not require a large memory footprint, as they do not project the dataset into memory.

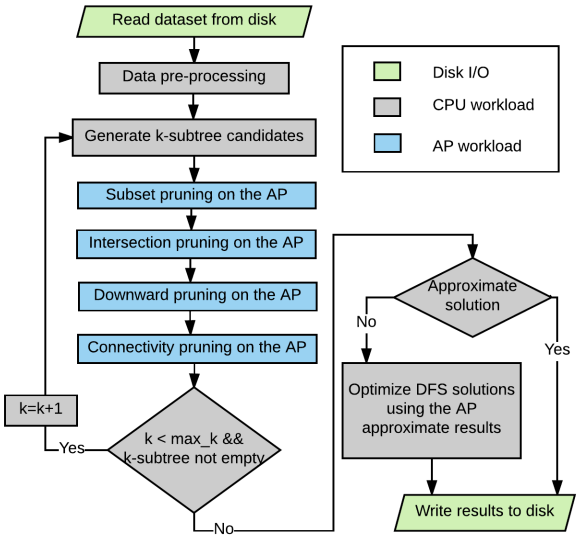


Figure 5: The workflow of the AP-accelerated FTM

However, they require a pass of the dataset in each iteration. Thus, to implement FTM on the GPU platform, we chose to adopt the BFS-based candidate-generation and enumeration strategy because (1) the solution will not be bound by the finite GPU global memory, and (2) it exposes many ready-to-process candidate subtrees in order to fully utilize the GPU cores and ultimately, reduces the overhead of database scanning.

FTM-GPU: After recoding labels, the whole dataset is transferred to the GPU global memory. Then, the algorithm iterates over three steps: (1) generating *(k + 1)-subtree* candidates from the frequent *k-subtrees* on the CPU, (2) pruning the candidates using subset pruning on the CPU, and (3) identifying the frequent *(k + 1)-subtrees* on the GPU. For the GPU computation, we convert both input trees and subtree candidates to one-dimensional arrays. In the CUDA kernel function, each thread is responsible for matching one tree in the input dataset to a candidate. We explore two memory region targets for the subtree candidates. In the first approach, we transfer all the candidates to constant memory (in iterations, if candidate array is larger than constant memory size) and all the threads start matching one candidate to their bound trees. Alternatively, we transfer the candidates to global memory and at each iteration, take just one to shared memory for matching. The constant memory implementation provides a faster solution when the trees in the dataset are similar in terms of size and node labels, otherwise, the shared memory approach shows a better result.

To improve the performance of FTM-GPU, we sort the trees in the database according to the tree size. This sorting tries to provide each warp with a batch of trees of roughly the same size. This greatly helps reduce branch divergence and lessen synchronization time. Once the matching and counting phase is done for all the *(k+1)-candidates*, the results are transferred back to the CPU for the next-level candidate-generation. FTM-GPU is capable of counting both distinct occurrences of subtrees and weighted support. *To the best of our knowledge, this is the first implementation of frequent subtree mining on the GPU platform.*

8 EXPERIMENTAL RESULTS

The number of patterns that can be placed into the board, and the number of candidates that must be checked in each stage, assuming a 32-chip Micron D480 AP board, determines how many passes through the input are required for each pruning kernel, and the input processing rate is fixed at 133 MB/s, which allows a simple calculation to determine the total time on the AP (see hardware parameters in [4]). All the automata designs are selected from the 16-bit encoding for simplicity, so there is no need for reconfiguration when the number of labels is more than 256. In each step of pruning, an appropriate FIM or SPM corresponding to the candidate size will be selected and configured on the AP.

8.1 Comparison with Other Implementations

We compare the performance and accuracy of the proposed AP multistage pruning for FTM (FTM-AP) versus (1) a BFS-based GPU implementation of FTM (FTM-GPU), (2) a multi-core implementation using pthread (TRIPS-12C) [7], (3) a single-core DFS-based implementation capable of weighted counting (TRIPS) [8], (4) a single-core DFS-based implementation which counts distinct occurrences of a pattern (TreeMinerD) [16], and a single-core level-wise BFS approach (PatternMatcher) [15].

We consider PatternMatcher in our comparison because it is the only method that does not fail on challenging datasets in lower support thresholds. Despite being very slow, it gives us a baseline for calculating both performance and accuracy for the proposed AP solution. Similarly to TreeMinerD, the FTM-AP solution is designed to only enumerate distinct occurrences of a pattern, thus providing a very fast solution in comparison with TreeMiner [16]. Therefore, we do not compare FTM-AP with TreeMiner. Moreover, TRIPS and TIDES [8] claim that they are orders of magnitude better than TreeMiner. In the same paper, they show that XSpanner is worse than TreeMiner, so we do not compare with XSpanner either.

FIM and SPM implementations on the AP are much faster than their GPU solutions, especially for large datasets [12] [14]. In FTM-AP, all the kernels are mapped to either FIM or SPM, and we can conclude that FTM-AP will outperform the GPU implementation of pruning kernels. Moreover, GPU implementations of subsequence inclusion checking in a sequence and subtree inclusion checking in a tree have almost similar complexity and synchronization overhead. Thus, we predict that exact FTM solution on the GPU (FTM-GPU) will outperform the GPU implementation of pruning kernels (inexact-FTM-GPU), because inexact-FTM-GPU requires at least twice as many subsequence inclusion checking operation as FTM-GPU requires subtree inclusion checking.

8.2 Platform and Parameters

All of the above implementations are tested using the following hardware:

- CPU: Intel CPU i7-5820K (6 cores, 3.30GHz), memory: 32GB, 2.133 GHz
- GPU: Nvidia Kepler K80C, 560 MHz clock, 4992 CUDA cores, 24GB global memory
- AP: D480 board, 133 MHz, 32 AP chips (simulation)

Furthermore, we test CPU solutions in a large-memory machine with 512GB of memory later in Section 8.7. For each benchmark, we compare the performance of the above implementations over a range of relative minimum support (*Rminsup*) values. To observe

the behavior of different implementations and finish all our experiments in a reasonable amount of time, we select *Rminsup* numbers that produce computation times up to one day.

8.3 Datasets

We evaluate the proposed algorithm on four different datasets, two real-world (CSLOGS [1] and TREEBANK [2]), and two synthetically generated by the tree generation program provided by Zaki[1] (T1M and T2M). Table 1 shows the details of the datasets. CSLOGS consists of user browsing subtrees of the CS department web site at the Rensselaer Polytechnic Institute. TREEBANK is widely used in computational linguistics and consists of XML documents. It provides a syntactic structure of the English text and uses part-of-speech tags to represent the hierarchical structure of the sentences. T1M and T2M are generated based on a mother tree with the maximal depth and fan-out of 10. The total number of nodes in T1M and T2M are 1,000,000 and 100,000, respectively. The datasets are then generated by creating subtrees of the mother tree. The synthetic tree generator provides a preorder-like representation, while TRIPS and TRIPS-12C work with the Prüfer sequence and postorder tree representation. Thus, we convert the datasets to their compatible input format offline and do not consider it in the time calculation.

8.4 AP-FTM Breakdown and Speedup Analysis

We choose TREEBANK dataset to study the pruning efficiency of each kernel and compare the performance of the CPU implementation and the AP-FTM for each of them. We also compare the scalability and efficiency of the kernel methods with the counting stage of PatternMatcher. TREEBANK is the most challenging dataset, because it consists of very wide and large trees (the largest tree has 684 nodes) and it has a large number of items and relatively high standard deviation of tree size, which makes it difficult for the CPU solutions to process. Excluding PatternMatcher, other solutions either crash or quickly run out of memory when decreasing the support threshold.

We have implemented all four pruning kernels on the CPU (in C++) in order to isolate the performance difference of the AP vs. CPU for the same work and highlight the AP architectural contribution. Fig. 6 shows that subset, intersection, downward, and connectivity kernels achieve up to 163×, 19×, 3144×, and 2635× speedups over their counterpart CPU implementations, while they prune at least 80%, 0.5%, 3.5%, and 4.8% of the total generated candidates in TREEBANK, when ranging *Rminsup* from 0.9 to 0.3 (Fig. 7). Due to the AP configuration time overhead, *subset_pruning_CPU* is faster than *subset_pruning_AP* at higher support thresholds. However, when *Rminsup* decreases and more candidates are generated, searching the larger dictionary of a frequent subtree on the CPU

[1]http://www.cs.rpi.edu/~zaki/software/
[2]http://www.cs.washington.edu/research/xmldatasets/

Table 1: Datasets

Name	#Trees	Ave_Node	SD_Node	#Items	Size(MB)
T1M	1,000,000	5.5	6.2	500	49.3
T2M	2,000,000	2.95	3.3	100	60.1
CSLOGS	59691	12.94	22.47	13361	6.3
TREEBANK	52581	68.03	32.46	1387266	27.3

Ave_Node = Average number of nodes per tree
SD_Node = Standard deviation of number of nodes per tree
#Items = Label set size

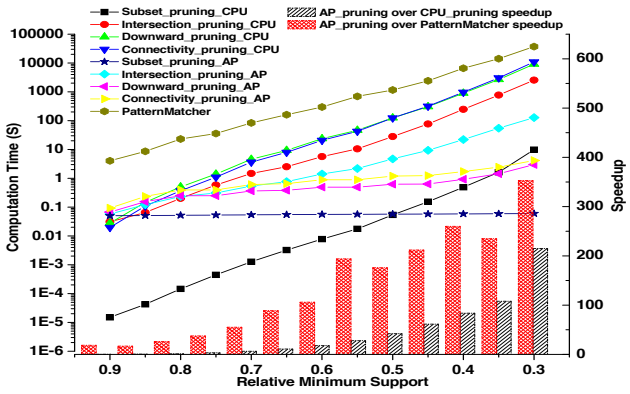


Figure 6: Kernels' execution time and speedup

takes longer, while the AP solution is almost constant. However, the subset kernel has a very small effect in total execution time, yet at the same time, it has a major contribution in pruning the candidates. Although intersection pruning has the lowest pruning contribution among others, it has the highest computation time on the AP, because the necessary input stream for this kernel is very large (due to the repetition of frequent candidates in different input trees). In total, the AP kernels show up to 215× speedup over the CPU pruning kernels, which implies the AP architectural contribution (black bars in Fig. 6). The subset kernel is more effective in lower *Rminsup*, where larger candidates are introduced and survive. This is because larger candidates have more frequent subsets, which increases the probability of pruning false positives. On the other hand, downward and connectivity kernels are more efficient on the smaller candidates because the effect of relaxing tree constraints is less destructive on them. This observation can be clearly seen in Fig. 7, where by decreasing the *Rminsup* (which means the population of the larger candidates relative to the smaller ones grows), the contribution of subset pruning increases, whereas the contribution of the other three kernels decreases.

We also compare the AP-FTM solution with PatternMatcher in order to show the trade-off between accuracy and execution time, and also study the algorithmic/heuristic contribution of our pruning kernels (the ratio of red to black bars in Fig. 6). In total, at least 86% of the generated candidates are pruned using the pruning kernel within less than 105 seconds for the lowest support threshold, where the PatternMatcher takes more than 10 hours to find the exact frequent candidates (about 353× speedup considering pre-processing time). Note that the pruning portions and timing are calculated just for the candidates of size three and more, and we do not consider the number of candidates for *1-frequent-candidates* and *2-frequent-candidates*, as they will be easily detected either on the AP or on the CPU. In order to further study the behavior of the pruning kernels, we test the effects of taking the intersection kernel out of the pruning framework, because as we have already observed, it has the least pruning efficiency and largest computational time. The results show that the AP achieves up to 1530× and 2190× speedup over the CPU-based pruning kernels and PatternMatcher, respectively. However, the AP worst case accuracy decreases from 86% to 83%. Therefore, having more sophisticated kernels can improve the accuracy and depend on the target constraints, and the user can make the trade-off between kernel selection and desirable speedup.

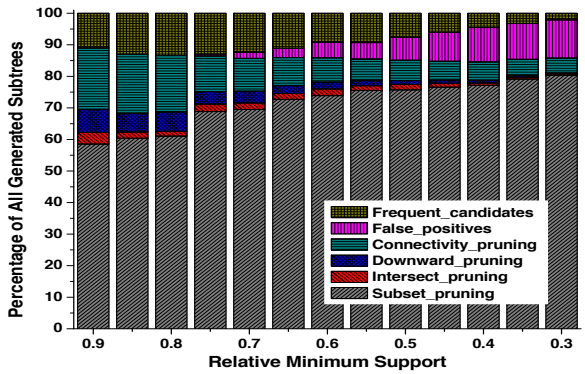


Figure 7: Generated subtrees' breakdown

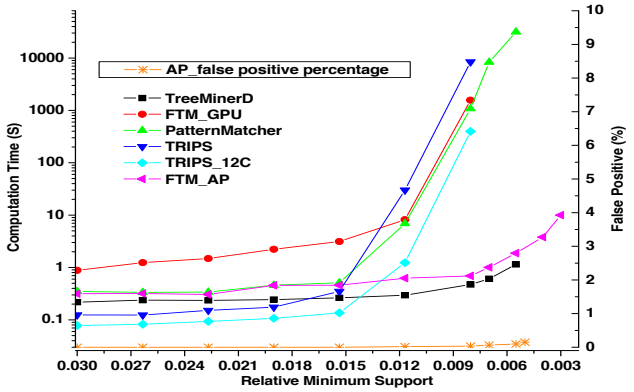


Figure 8: Performance comparison on CSLOGS

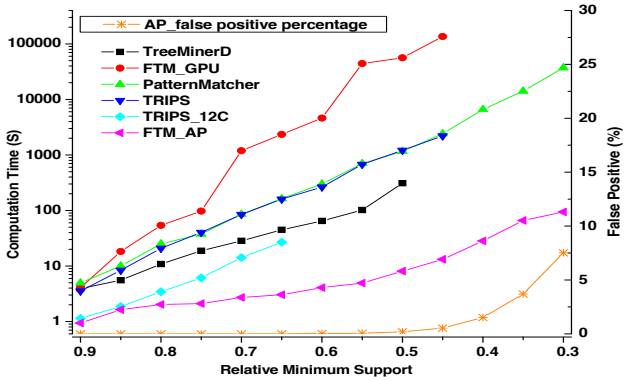


Figure 9: Performance comparison on TREEBANK

8.5 AP-FTM vs. Other FTM Algorithms

Fig 8 - 11 represent performance comparison (left vertical axis) among single-core CPU implementations (TreeMinerD, PatternMatcher, TRIPS), a multi-core (TRIPS-12C) approach, a GPU solution (FTM-GPU), and our proposed method (FTM-AP) for mining distinct occurrences of embedded subtrees in four databases of ordered labeled trees. All methods are end-to-end solutions and apart from FTM-AP, have an accuracy of 100% for the final frequent set. The right vertical axes in the graphs represent the percentage of the false positives in the output of FTM-AP. The main goal of these graphs is to compare the trade-off between the speed and accuracy of the AP solution versus the existing FTM implementations.

Most existing state-of-the-art tools have difficulty with larger inputs and smaller support thresholds and fail due to scalability limits. Common limits include long execution time, insufficient system memory, limitations in internal data-structures, or crashing/reporting an error due to their design not anticipating a large input. TRIPS and TRIPS-12C either crash or get struck for unknown reasons when decreasing the *Rminsup*. For example, in TREEBANK, TRIPS-12C breaks at $Rminsup = 0.65$ and TRIPS stucks at $Rminsup = 0.5$. TRIPS-12C shows an unstable behavior in T1M and T2M, and from our experience, changing the number of running threads and turning hyper-threading off do not make a difference. TreeMinerD is the fastest accurate solution in real-world datasets for smaller thresholds. In CSLOGS (Fig. 8), TreeMinerD is even faster than the FTM-AP solution, however, it runs out of memory when $Rminsup<0.005$ (we will discuss the memory usage of TreeMinerD in the next subsection). PatternMatcher requires the least amount of memory among other solutions, but its execution time takes more than 10 hours at $Rminsup<0.006$ and $Rminsup<0.35$ in CSLOGS and TREEBANK, respectively.

Database statistics such as average and standard deviation of the number of nodes per tree, the number of items, and the number of trees directly affect the performance of the FTM-GPU. In Fig. 8 and 9, FTM-GPU shows almost the worst performance results among others. This is because *SD_Node* and #*Items* in TREEBANK and CSLOGS are relatively high. Higher *SD_Node* implies uneven distribution of trees with different sizes in the database, and causes the synchronization time between the thread in a warp to increase. Higher #*Items* increases the chance of thread divergence in a warp, because the possibility of checking the subtree node against different labels in the input trees of the same warp increases. In CSLOGS at $Rminsup \leq 0.008$ and TREEBANK at $Rminsup \leq 0.45$, the FTM-GPU takes more than 10 hours to run. T1M and T2M in Fig. 10 and 11 show that FTM-GPU has much better relative performance among accurate solutions as the *SD_Node* and #*Items* are lower (Table 1). Overall, the FTM-GPU results show that the GPU platform does not provide a reliable and scalable solution for the FTM.

In Fig. 8, for $Rminsup \geq 0.008$, the execution time of TreeMinerD and FTM-AP are almost the same, which is less than a second. In the range of $0.008 < Rminsup \leq 0.006$, PatternMatcher is the only running accurate solution, which takes 31,456 sec when $Rminsup = 0.006$. Eventually, for $Rminsup \leq 0.006$, FTM-AP continues to be the only reliable running solution, and we are not able to calculate the accuracy of the AP-FTM, as there is no exact solution running in this range. The maximum accuracy reduction for FTM-AP in CSLOGS is 0.09%. For T1M and T2M, AP-FTM beats TRIPS, which is the fastest solution, by factors of 22× and 9.2×, while losing at most 6.5% and 0.1% accuracy, respectively. Overall, FTM-AP, with at most 7.5% false positives, beats PatternMatcher, the feasible and exact CPU solution, by a factor of 394×. The low memory requirement and huge speedup achieved by the AP-FTM makes it a scalable and reliable solution with a final application tolerance of a few percentage points for false positives. Another advantage of the AP is that it gives consistently good performance, while the performance of other techniques varies based on the database characteristics.

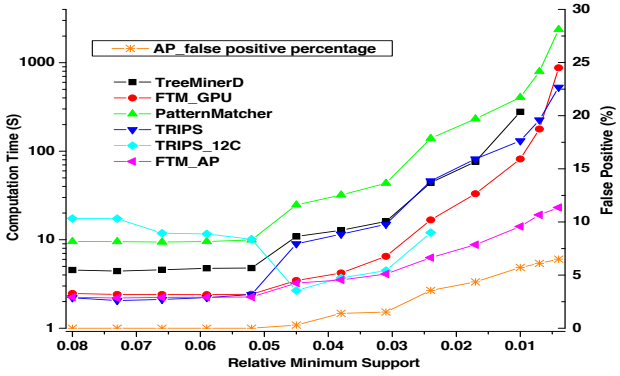


Figure 10: Performance comparison on T1M

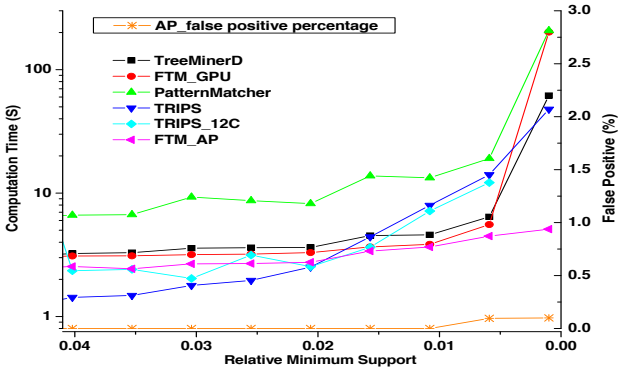


Figure 11: Performance comparison on T2M

8.6 Performance Scaling with Data Size and Support Threshold

In this subsection, we study the scaling of performance as a function of input data sizes and minimum support threshold. In order to generate synthetic datasets, we adopt the parameters used to generate T1M (Table 1) and increase the number of trees from 10^6 (49MB) to 2×10^6 (99MB), 4×10^6 (196MB), 8×10^6 (386MB), and 16×10^6 (770MB), while preserving the other parameters. Fig. 12 represents the performance of TreeMinerD, FTM-GPU, PatternMatcher, TRIPS, and FTM-AP for these five synthetically generated dataset using two relative minimum supports. FTM-AP beats FTM-GPU by a factor of 2.6× at *Rminsup* = 0.04 and 3.9× at *Rminsup* = 0.02, and PatternMatcher by a factor of 3.3× at *Rminsup* = 0.04 and 45.1× at *Rminsup* = 0.02, while losing at most 1% accuracy. TreeMinerD and TRIPS do not provide a scalable solution, because they both run out of memory when increasing the input size and when decreasing the minimum support threshold. The results show that the FTM-AP always has the lowest execution time and its performance advantage grows when *Rminsup* decreases and input size increases. We are evaluating FTM performance for one node. Sufficiently large CPU/GPU clusters can handle larger FTM problems and run them faster than a single AP, but a cluster of APs would be even faster.

8.7 Exact Solution on the FTM-AP

The output of the AP pruning kernels is a set of potentially frequent candidates that preserve a subset of tree topological and label attributes. Depending on the target application, one can directly

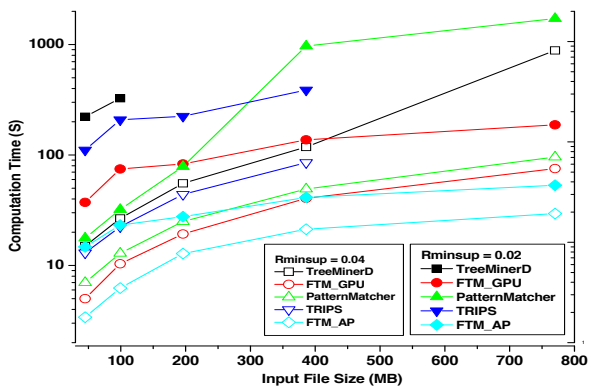


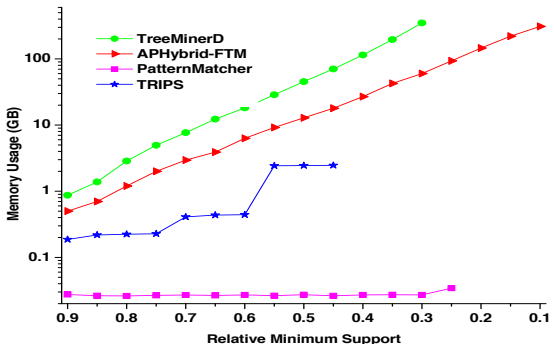
Figure 12: Performance scaling with input data-size and Rminsup

use the AP output as the set of frequent subtrees, especially in classification tasks, where the applications are able to recover from the introduced false positives. Frequent subtrees are used in many natural language processing (NLP) tasks, because tree structures can capture and represent the complex relations and dependencies of a natural language. Agarwal et al. in [1] demonstrated how combining subtree features with sequential patterns and bag-of-words can increase sentiment analysis accuracy. Since the AP results are already examined for sequential properties in downward pruning, and all the false positives in the final candidate set can be translated as sequential pattern features (albeit with low support), it suggests that the AP final results can be directly used for those tasks without potentially affecting the final accuracy, while achieving a huge speedup.

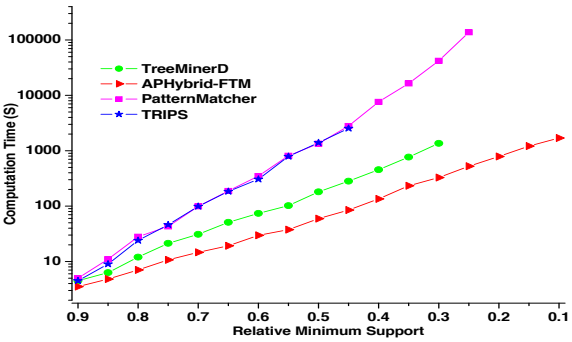
On the other hand, having an exact set of frequent subtrees is a must for some applications. In order to prune the false positives from the AP output, we propose APHybrid-FTM, to combine the AP with TreeMinerD, where the AP solution can help to reduce the memory requirement and increase the speed of the TreeMinerD approach while maintaining its 100% accuracy. As mentioned before, TreeMinerD implements a DFS-based algorithm, and a *(k+1)-candidate* is generated by combining two *k-candidates* on the same equivalent-class, under some circumstances [16]. We store the set of potential frequent subtrees in a dictionary ($\mathcal{D}$) and check whether the *(k+1)-candidates*, generated by the TreeMinerD candidate-generation step, exist in $\mathcal{D}$. If it is a hit, TreeMinerD continues the matching and counting stage (as discussed in Section 6.3, the set of *3-candidates* and line-shaped candidates are 100% accurate and do not need to be checked for frequency), otherwise, the candidate is infrequent, which avoids the unnecessary enumeration step, and the algorithm continues to generate the next candidate. The AP-FTM framework greatly helps TreeMinerD to (1) reduce its execution time, and (2) alleviate its memory footprint because many infrequent candidates in TreeMinerD will be pruned early in the candidate-generation step and their occurrences (embedding information in the database) do not need to be stored in memory.

In order to analyze APHybrid-FTM performance and study the effect of memory usage on the FTM scalability, we run TreeMinerD, APHybrid-FTM, PatternMatcher, and TRIPS on a node[3] with 512GB memory for TREEBANK. Fig. 13a represents the required maximum memory size and Fig. 13b shows the execution time of these

[3]Intel(R) Xeon(R) CPU E5-2670 (24 cores, 2.30GHz), memory: 512GB, 2.133 GHz



(a) Exact solutions memory usage



(b) Exact solution execution time

Figure 13: Execution time vs memory usage on TREEBANK

methods on a log scale. The speed and memory trade-off among TreeMinerD, PatternMatcher, and TRIPS on the CPU can be easily observed by looking at these two graphs, where TreeMinerD is the fastest tool among these three and demands the largest memory footprint, whereas PatternMatcher, with the lowest performance, requires the smallest memory capacity (less than 100MB). However, APHybrid-FTM is the best. It alleviates memory usage of TreeMinerD up to 5.8× and reduces its execution time up to 4.14 at $Rminsup = 0.3$. For $Rminsup \leq 0.3$, TreeMinerD runs out of memory, while APHybrid-FTM continues. Furthermore, APHybrid-FTM performs 262× better than PatternMatcher at $Rminsup = 0.25$ (it takes more than a day for $Rminsup < 0.25$) and 30× better than TRIPS at $Rminsup = 0.45$ (lowest working threshold for TRIPS).

In summary, APHybrid-FTM provides the fastest exact solution (Fig. 13b), which in turn extends the scalability of TreeMinerD and its advantages increase at lower support thresholds and larger databases. Overall, the proposed pruning approach can be adopted as a general strategy to accelerate complex and/or memory-intensive pattern-mining problems.

9 CONCLUSIONS AND FUTURE WORK

We develop FTM-AP, a multi-stage pruning strategy on the AP, to reduce the candidate search space of the frequent subtree mining problem in a very short amount of time, providing a fast and scalable solution at the cost of a small reduction in accuracy. FTM-AP achieves up to 394× speedup with at most 7.5% false positives over PatternMatcher, a feasible and exact CPU solution. For problems requiring an exact solution, we use the output of FTM-AP as the candidate screen for TreeMinerD, a DFS-based exact solution, in order to remove the false-positive candidates, limit the memory requirements, and achieve up to 262× speedup. The benefits the AP provides for FTM increase with larger datasets and lower support thresholds. The pruning framework on the heterogeneous architecture (CPU and the AP) can also potentially be adopted to extend the scalability of the other subtree types and graph mining problems, an interesting direction for future work.

Additional performance improvements could be achieved with hardware support to minimize symbol replacement latency and maximize capacity of resources on the AP, as well as better support for push-down automata capabilities. The proposed pruning kernels are capable of running in a pipeline system, assuming four AP boards are available, and this also allows scaling of larger problems to cluster- or datacenter-scale resources, another interesting direction for future work.

10 ACKNOWLEDGEMENTS

This work was supported in part by the Virginia CIT CRCF program under grant no. MF16-032-IT; by C-FAR, one of the six SRC STARnet Centers, sponsored by MARCO, and DARPA; NSF XPS grant CCF-1629450; and a grant from Micron Technology.

REFERENCES

[1] A. Agarwal, B. Xie, I. Vovsha, O. Rambow, and R. Passonneau. 2011. Sentiment analysis of twitter data. In *Proceedings of the Workshop on Languages in Social Media*. Association for Computational Linguistics.
[2] C. Bo, K. Wang, J. Fox, and K. Skadron. 2016. Entity Resolution Acceleration using the Automata Processor. In *Proceedings of the IEEE International Conference on Big Data*. IEEE.
[3] Y. Chi and J. Kok. 2001. Frequent subtree mining–an overview. *Fundamenta Informaticae* 21.
[4] P. Dlugosch, D. Brown, P. Glendenning, M. Leventhal, and H. Noyes. 2014. An efficient and scalable semiconductor architecture for parallel automata processing. *IEEE Transactions on Parallel and Distributed Systems (TPDS)* 25, 12.
[5] R. Iváncsy and I. Vajk. 2007. Automata Theory Approach for Solving Frequent Pattern Discovery Problems. *International Journal of Computer, Electrical, Automation, Control and Information Engineering, World Academy of Science, Engineering and Technology* 1, 8.
[6] H. Tan, F. Hadzic, T. S Dillon, E. Chang, and L. Feng. 2008. Tree model guided candidate generation for mining frequent subtrees from XML documents. *ACM Transactions on Knowledge Discovery from Data (TKDD)*.
[7] S. Tatikonda and S. Parthasarathy. 2009. Mining tree-structured data on multicore systems. *Very Large Data Base (VLDB), ACM*.
[8] S. Tatikonda, S. Parthasarathy, and T. Kurc. 2006. TRIPS and TIDES: new algorithms for tree mining. *The Conference on Information and Knowledge Management (CIKM), ACM*.
[9] T. Tracy II, Y. Fu, I. Roy, E. Jonas, and P. Glendenning. 2016. Towards machine learning on the Automata Processor. In *International Conference on High Performance Computing*. Springer.
[10] J. Wadden, V. Dang, N. Brunelle, T. Tracy II, D. Guo, E. Sadredini, K. Wang, C. Bo, G. Robins, M. Stan, and K. Skadron. 2016. ANMLzoo: a benchmark suite for exploring bottlenecks in Automata Processing engines and architectures. In *in Workload Characterization (IISWC)*. IEEE.
[11] C. Wang, M. Hong, J. Pei, H. Zhou, W. Wang, and B. Shi. 2004. Efficient pattern-growth methods for frequent tree pattern mining. In *Proc. of the Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD)*. Springer.
[12] K. Wang, Y. Qi, J. J Fox, M. R Stan, and K. Skadron. 2015. Association rule mining with the Micron Automata Processor. In *IEEE International Parallel Distributed Processing Symposium (IPDPS)*.
[13] K. Wang, E. Sadredini, and K. Skadron. Hierarchical Pattern Mining with the Micron Automata Processor. In *International Journal of Parallel Programming (IJPP)*. 2017.
[14] K. Wang, E. Sadredini, and K. Skadron. 2016. Sequential Pattern Mining with the Micron Automata Processor. In *ACM International Conference on Computing Frontiers (CF)*.
[15] M. J. Zaki. 2002. Efficiently mining frequent trees in a forest. In *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*.
[16] M. J. Zaki. 2005. Efficiently mining frequent trees in a forest: Algorithms and applications. *IEEE Transactions on Knowledge and Data Engineering (TKDE)* 17, 8.