PrefixFPM: A Parallel Framework for General-Purpose Frequent Pattern Mining

Da Yan[*1.1], Wenwen Qu[+*1.2], Guimu Guo[*3], Xiaoling Wang[+†4]

Note: Da Yan and Wenwen Qu are parallel first authors. This work was done during Wenwen Qu's internship at UAB.

[*]*University of Alabama at Birmingham (UAB),* {[1.1]yanda, [1.2]wenwenqu, [3]guimuguo}@uab.edu
[+]*East China Normal University (ECNU),* [1.2]wenwenqu@sei.ecnu.edu.cn, [4]xlwang@cs.ecnu.edu.cn
[†]*Shanghai Institute of Intelligent Science and Technology, Tongji University*

***Abstract*—Frequent pattern mining (FPM) has been a focused theme in data mining research for decades, but there lacks a general programming framework that can be easily customized to mine different kinds of frequent patterns, and existing solutions to FPM over big transaction databases are IO-bound rendering CPU cores underutilized even though FPM is NP-hard.**

This paper presents, *PrefixFPM*, a general-purpose framework for FPM that is able to fully utilize the CPU cores in a multicore machine. PrefixFPM follows the idea of prefix projection to partition the workloads of PFM into independent tasks by divide and conquer. PrefixFPM exposes a unified programming interface to users who can customize it to mine their desired patterns, and the parallel execution engine is transparent to end-users and can be reused for mining all kinds of patterns. We have adapted the state-of-the-art serial algorithms for mining frequent patterns including subsequences, subtrees, and subgraphs on top of PrefixFPM, and extensive experiments demonstrate an excellent speedup ratio of PrefixFPM with the number of cores.

A demo is available at https://youtu.be/PfioC0GDpsw; the code is available at https://github.com/yanlab19870714/PrefixFPM.

***Index Terms*—frequent pattern mining, prefix projection, compute-intensive, CPU-bound, sequence, subgraph, tree.**

I. Introduction

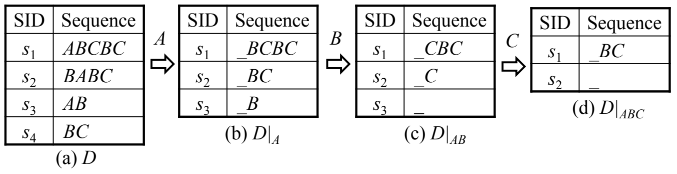
Frequent patterns are substructures that appear in a dataset with frequency no less than a user-specified threshold. Frequent pattern mining (FPM) has been at the core of data mining research [3], where numerous serial algorithms have been proposed for mining various types of substructures, and they are widely used in real applications [5], [6].

With the popularity of Big Data and Hadoop, and the need of FPM over Big Data, many distributed solutions to FPM emerge such as those based on MapReduce [7], [9], [4] and other dedicated ones [10], [11]. However, all these works adopt an Apriori approach where frequent patterns of size $(i+1)$ are generated from all frequent patterns of size i, leading to an iterative algorithm where Iteration i mines all frequent patterns of size i. This straightforward approach has a catastrophic performance impact when implemented in a distributed environment. For example, data instances that contain size-i patterns need to be transmitted across the network to various machines for growing size-$(i+1)$ patterns, and to get their frequency, another round of communication is needed for pattern frequency aggregation. This basically associates a data movement with each computing operation, but the former is the performance bottleneck rendering CPU cores underutilized. The performance is further exacerbated by the fact that the frequent patterns found in each iteration often need to be dumped to Hadoop Distributed File System (which replicates data) and then loaded back in Iteration $(i+1)$.

Researchers have begun to realize the issue that IO-bound Big Data frameworks can be ill-suited for compute-heavy problems such as our FPM. In [1], McSherry indicates that "*the current excitement about distributed computation (e.g. Hadoop, Spark) produced implementations ... whose performance never quite gets to where you would be with a simple single-threaded implementation on a laptop*". There is no exception in the context of FPM. For example, in SOSP 2015, Arabesque [10] is proposed as a distributed system that can handle frequent subgraph mining, but later in OSDI 2018, RStream [11] follows Arabesque's programming model but utilizing relational joins to run out-of-core on a single machine, and is found to beat Arabesque in performance.

So far, there lacks a parallel FPM solution that scales with the number of CPU cores. We propose such a system called *PrefixFPM* for running in a multicore machine. Instead of checking patterns in breadth-first order using Apriori algorithms, PrefixFPM adopts the prefix projection approach pioneered by PrefixSpan [8] and followed by numerous later works. As we shall explain in Section II, prefix projection partitions the workloads of pattern checking by divide and conquer, which naturally fits in a task-based parallel execution model; also, depth-first pattern-growth order allows a small memory footprint as one do not have to keep all size-i patterns for pattern growth, and it allows a pattern α that is grown from β to only examine the subset of data that contain β (called a *projected database* by β). PrefixFPM also features a user-friendly programming model where users can customize it to mine different kinds of patterns by adapting existing serial algorithms. The parallel execution details are handled by the framework itself and are transparent to users. As far as we know, PrefixFPM is the first parallel framework that unifies the mining of different types of frequent patterns. We adapt state-of-the-art serial mining algorithms for frequent subsequences, subgraphs and subtrees to run on top of PrefixFPM, which show an excellent speedup with CPU cores in experiments.

In the sequel, Sec. II overviews the idea of prefix projection and how PrefixFPM utilizes it for task parallelism. Sec. III introduces our programming model for unifying FPM problems. Sec. IV concludes by summarizing our experimental results.


A
B
C
(a) D
(b) $D|_A$
(c) $D|_{AB}$
(d) $D|_{ABC}$

SID	Sequence
s_1	*ABCBC*
s_2	*BABC*
s_3	*AB*
s_4	*BC*

SID	Sequence
s_1	*_BCBC*
s_2	*_BC*
s_3	*_B*

SID	Sequence
s_1	*_CBC*
s_2	*_C*
s_3	*_*

SID	Sequence
s_1	*_BC*
s_2	*_*

Figure 1. Illustration of PrefixSpan

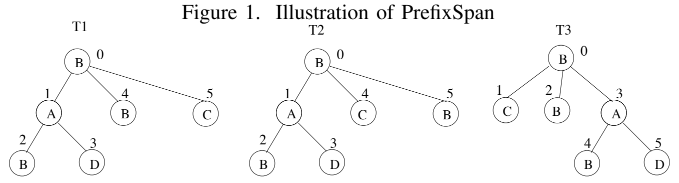


Figure 2. Tree Patterns from Figure 4 of [13]

II. PrefixFPM Solution Overview

PrefixSpan Review. To understand the idea of prefix projection, let us first briefly review the PrefixSpan [8] algorithm for mining frequent sequential patterns from a sequence database.

We denote $\alpha\beta$ to be the sequence resulted from concatenating sequence α with sequence β. We also use $\alpha \sqsubseteq s$ to denote that sequence α occurs as a subsequence of data sequence s.

Given a sequential pattern α and a data sequence s, the α-projected sequence $s|_\alpha$ is defined to be the suffix γ of s such that $s = \beta\gamma$ with β being the minimal prefix of s satisfying $\alpha \sqsubseteq \beta$. To highlight the fact that γ is a suffix, we write it as $\_\gamma$. To illustrate, when $\alpha = BC$ and $s = ABCBC$, we have $\beta = ABC$ and $s|_\alpha = \_\gamma = \_BC$. Given a sequential pattern α and a sequence database D, the α-projected database $D|_\alpha$ is defined to be the set $\{s|_\alpha \mid s \in D \wedge \alpha \sqsubseteq s\}$. Consider the sequence database D shown in Figure 1(a). The projected databases $D|_A$, $D|_{AB}$ and $D|_{ABC}$ are shown in Figure 1(b), (c) and (d).

Let us define the support of a pattern α as the number of sequences in D that contain α as a subsequence, then the support of α is simply the size of $D|_\alpha$. PrefixSpan finds the frequent patterns (with support at least τ_{sup}) by recursively checking the frequentness of patterns with growing lengths. In each recursion, if the current pattern α is checked to be frequent, it will recurse on all the possible patterns α' constructed by appending α with one more element. PrefixSpan checks whether a pattern α is frequent using the projected database $D|_\alpha$, which is constructed from the projected database of the previous iteration. Figure 1 presents one recursion path when $\tau_{sup} = 2$, where, for example, $s_1|_{ABC}$ in $D|_{ABC}$ is obtained by removing the element C from $s_1|_{AB}$ in $D|_{AB}$.

Prefix Projection. The above algorithm actually generalizes to other patterns such as subgraphs and subtrees though some additional processing is necessary. The key idea is that we can establish a one-to-one correspondence between each subgraph/subtree pattern and its sequence encoding, so that we can examine the pattern encodings by a PrefixSpan-style algorithm. The tricky issue is that different patterns that are isomorphic to each other have different encodings, but they actually refer to the same pattern and growing larger patterns from them leads to a lot of redundant computation.

For example, consider the 3 subtrees shown in Figure 2. The Sleuth algorithm [13] encodes a tree T by adding vertex labels to the encoding in a depth-first preorder traversal of T, and by adding a unique symbol "\$" whenever we backtrack from a child to its parent. For example, the encoding of T_1 in Figure 2 is BAB\$D\$\$B\$C\$, while the encoding of T_2 is BAB\$D\$\$C\$B\$. Even though the two encodings are different, if children order does not matter, T_1, T_2 and T_3 are the same.

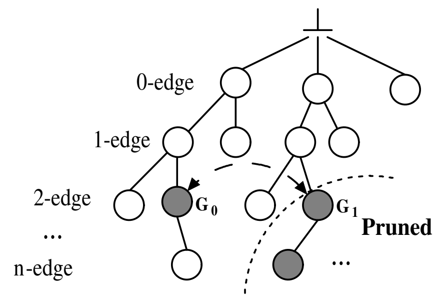


Figure 3. Pattern Search Space from Figure 1 of [12]

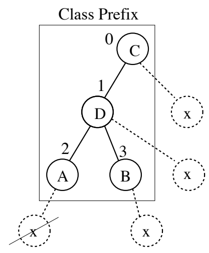


Figure 4. Pattern Extension from Figure 5 of [13]

To avoid processing redundant patterns, existing serial algorithms define the *canonical* encoding of a pattern α, denoted by min(α), as the minimum encoding of all automorphisms of α. A pattern α is examined if its encoding equals min(α).

Figure 3 illustrates the depth-first pattern search tree of the gSpan algorithm [12] for subgraph patterns that are grown by adding adjacent edges, where each node represents a subgraph pattern α to examine, and the subtree under the node contains those patterns grown from α. Assume that G_0 and G_1 are isomorphic and since only G_0's encoding equals its canonical encoding, only G_0 is checked for frequentness and for further pattern growth, while G_1 (and its potential subtree) is pruned.

A pattern α is extended by one more element to generate a child pattern β, and in PrefixSpan we simply append α with all possible labels. However, for a subgraph or subtree pattern α, we cannot extend it with any adjacent edge since some extensions have been considered by a prior node in the depth-first search space tree. For example, in Figure 4, we can only extend the subtree pattern in the box using an adjacent edge on its rightmost path CDB, since the extension from vertex A has a smaller encoding than the subtree pattern itself (CDAx$\cdots$ <CDA\$$\cdots$) and is considered before in DFS order.

Task-Based Model of PrefixFPM. PrefixFPM associates each pattern α (which corresponds to a node in the search tree of Figure 3) with a task t_α which checks the frequentness of α using its projected database $D|_\alpha$, and which grows the pattern by one more element to generate the children patterns $\{\beta\}$ and their projected databases $\{D|_\beta\}$ (computed incrementally from $D|_\alpha$ rather from the entire D). These children patterns give rise to new tasks $\{t_\beta\}$ which are added to a shared task queue to be fetched by computing threads for further processing.

PrefixFPM runs a number of computing threads that fetch pattern-tasks from a shared task queue Q_{task} for concurrent processing. Since each task t_α needs to maintain $D|_\alpha$ to compute the projected databases of the child-patterns grown from α, a depth-first task fetching priority in the pattern search tree tends to minimize the memory footprint of patterns in processing. This is because we tend to grow those patterns that have been grown deeper, which are larger (and thus with smaller projected databases) and are closer to finishing their growth (due to the support becoming less than τ_{sup}).

We thus implement the task queue Q_{task} as a stack where newly-pushed tasks are popped sooner for processing. Note that PrefixFPM processes the pattern search tree in a near-depth-first order but not strictly depth-first: when the leftmost pattern-node α is being processed as a task by some computing thread, its next sibling pattern-node will be popped for processing by another available computing thread (rather than the children of pattern-node α). This allows idle computing threads to fetch patterns for processing ASAP to keep CPU cores busy, but may require more memory than a serial depth-first solution. In fact, the memory cost is the same as a serial algorithm if there is only one computing thread (as tasks are fetched in DFS order), but it is expected to increase with the number of threads. Queue Q_{task} is protected by a lock (mutex) so that only one thread can push or pop a task at a time.

Since fetching tasks from Q_{task} and adding tasks to Q_{task} incur locking overheads, this is only worthwhile if each task contains sufficient computing workloads such that the locking cost of fetching it and inserting child-pattern tasks is negligible. Therefore, when processing a task t_α, we only add child-pattern tasks to Q_{task} if the number of projected data instances in $D|_\alpha$ is above a size threshold τ_{split}, so that the workloads can be divided by other computing threads; otherwise, t_α is not expensive and the current computing thread simply processes its entire search space subtree in depth-first order directly.

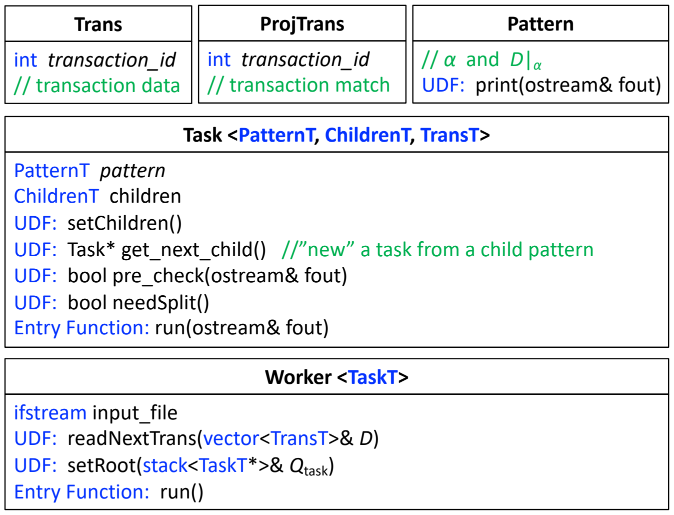
III. PrefixFPM Programming Model

PrefixFPM is written as a set of C++ header files defining some base classes and their virtual functions for users to inherit in their subclasses and to specify the application logic. We call these virtual functions as user-defined functions (UDFs). The base classes also contain C++ template arguments for users to specify with the proper pattern and data structures.

We now introduce these base classes as shown in Figure 5.

Trans. The *Trans* class implements a transaction (i.e., a data instance in the input database) with a predefined transaction ID field. Users implement their transaction subclass by inheriting *Trans* and including additional fields to store the target data instance such as a sequence or a graph. Initially, the input dataset is read into an in-memory transaction database D which is simply an array of objects whose type is the *Trans* subclass.

ProjTrans. The *ProjTrans* class implements a projected transaction $s|_\alpha$ in a projected database $D|_\alpha$. A *ProjTrans* object also has a transaction ID field indicating which transaction $s \in D$ this projected transaction corresponds to. The user-defined *ProjTrans* subclass should also indicate how $s|_\alpha$ is currently matched on s, so that the matching status can be incrementally updated as the pattern α grows.



```
Trans
int transaction_id
// transaction data

ProjTrans
int transaction_id
// transaction match

Pattern
// α and D|α
UDF: print(ostream& fout)

Task <PatternT, ChildrenT, TransT>
PatternT pattern
ChildrenT children
UDF: setChildren()
UDF: Task* get_next_child()  //"new" a task from a child pattern
UDF: bool pre_check(ostream& fout)
UDF: bool needSplit()
Entry Function: run(ostream& fout)

Worker <TaskT>
ifstream input_file
UDF: readNextTrans(vector<TransT>& D)
UDF: setRoot(stack<TaskT*>& Q_task)
Entry Function: run()
```

Figure 5. PrefixFPM Programming Interface

Pattern. The *Pattern* class specifies the data structure of a pattern α, and contains a (pure) virtual function *print*(*fout*) specifying how to output the object of a *Pattern* subclass into an output file stream *fout*. Recall that PrefixFPM runs multiple task computing threads, and each thread actually appends the frequent patterns found by it to a file of its own (with file handle *fout*). When a job finishes, frequent patterns are simply recorded by the files written by all task computing threads.

A *Pattern* subclass usually also includes the projected database $D|_\alpha$ as a field. In *print*(*fout*), users may choose to output $D|_\alpha$ along with α, to capture the matched transactions.

Task. This class is to specify the algorithmic logic. Recall that a task t_α checks the frequentness of pattern α using $D|_\alpha$, and grows α by one more element to generate children patterns $\{\beta\}$ and their projected databases $\{D|_\beta\}$ for further mining. Here, an object of base class *Task<PatternT, ChildrenT, TransT>* implements a task t_α with 3 template arguments:

- *PatternT*: the user-defined *Pattern* subclass (with $D|_\alpha$);
- *ChildrenT*: the type a table *children* that keeps $\{D|_\beta\}$;
- *TransT*: the user-defined *Trans* subclass.

A *Task* object t_α maintains 2 fields: a pattern α of type *PatternT* (often containing $D|_\alpha$), and the children table *children* that keeps $\{D|_\beta\}$, which is typically implemented as std::map with $children[e] = D|_\beta$ if β is grown from α with element e.

TransT is needed since the *Task* class provides a function to access the global static transaction database D for users to call in their *Task* UDFs, which is useful since a projected transaction $s_i|_\alpha$ usually only keeps a compact matching status towards $s_i \in D$, and to extend it with one more element e in s_i to generate $s_i|_\beta$, we need to access s_i as $D[i]$ where i is the transaction ID field of $s_i|_\alpha$ whose type is a *ProjTrans* subclass.

For example, the projected transaction of PrefixSpan only keeps the position of the last match, i.e., '_' in Figure 1, to minimize the memory consumed by projected databases.

```
1   void run(ostream& fout){
2          if(!pre_check(fout)) return;
3          //generate new patterns
4          setChildren(children);
5          //run new child tasks
6          while(Task* t= get_next_child()){
7                 if(needSplit()){
8                        q_mtx.lock();
9                        queue().push(t);
10                       q_mtx.unlock();
11                }
12                else{
13                       t->run(fout);
14                       delete t;
15                }
16         }
17  }
```

Figure 6. The *run*(*fout*) function of base class *Task*

Task has an internal function *run*(*fout*) which executes the processing logic of the task t_α. The behavior of *run*(.) is specified by *Task* UDFs which are called in *run*(.).

Figure 6 shows *run*(.). In Line 2, t_α first runs UDF *pre_check*(*fout*) to see if α is frequent (and its encoding is canonical if applicable), and if so, to output α to *fout*. If α is not pruned, Line 4 then runs UDF *setChildren*(*children*) to scan $D|_\alpha$ and compute $\{D|_\beta\}$ into the table field *children*. In this step, every infrequent child pattern β should be removed from the table *children* as a post processing after $\{D|_\beta\}$ are constructed. Line 6 then wraps each child pattern β in table *children* as a task t_β, and calls the UDF *needSplit*() to check if t_β is time consuming (e.g., $D|_\beta$ is big). If so, we add t_β to the task queue Q_{task} (Lines 8–10) to be fetched by available task computing threads (recall that Q_{task} is a global last-in-first-out task stack protected by a mutex), which divides the computing workloads by multithreading. Otherwise, we recursively call t_β's *run*(*fout*) to process the entire checking and extension of β by the current thread, which avoids contention on Q_{task}.

Worker. A PrefixFPM program is executed by subclassing the *Worker<TaskT>* base class, and call its *run*() function. Here, *TaskT* refers to the user-defined *Task* subclass, from which *Worker* derives the other necessary types such as *TransT*.

The *run*() function (1) keeps calling UDF *getNextTrans*() to read transactions and appends them to the transaction database D, (2) calls the UDF *setRoot*() to generate root tasks (where α contains only one element) into Q_{task}, and (3) creates k task computing threads to concurrently process the tasks in Q_{task}.

Implementing *Worker*::*setRoot*(.) should be similar to implementing *Task*::*setChildren*(.): instead of constructing $\{D|_\beta\}$ form $D|_\alpha$, we construct $\{D|_e\}$ form D. Each seed task $t_e = \langle e, D|_e \rangle$ is added to Q_{task} to initiate parallel computation.

At the beginning of *Worker*::*setRoot*(.), we also need to get the element frequency statistics and eliminate infrequent elements (i.e., they are not considered when growing patterns).

During parallel task computation, each computing thread keeps fetching a task t_α from Q_{task} to call its *run*(*fout*) function, and gets hanged to release CPU core when it cannot find a task in Q_{task}. Note that while Q_{task} is currently empty, another thread may be processing a task and could add more subtasks back to Q_{task}. The job terminates only if all k task computing threads are hanged and no task is found in Q_{task}.

Thread #	Time (sec)	Speedup
1	5750.37	1.00
2	2177.47	2.64
4	953.23	6.03
8	456.48	12.60
16	279.02	20.61
32	231.09	24.88

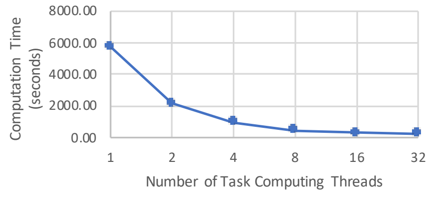


Figure 7. Scale-up Results on *NCI*

IV. Experiments and Conclusion

We have used PrefixFPM to parallelize 3 state-of-the-art prefix-projection algorithms for mining 3 different kinds of frequent patterns, i.e., sequences, subgraphs and subtrees. Due to space limit, we refer interested readers to https://github.com/yanlab19870714/PrefixFPM for their implementation. We plan to describe them in detail in a complete journal paper.

We tested those 3 programs (parallel versions of PrefixSpan, gSpan and Sleuth) on large real datasets and found that in all experiments, an ideal speedup is obtained for at least 16 cores, and performance continues to improve till all 32 cores in our machine are used. As an illustration, Figure 7 shows the scalability results of gSpan's parallel implementation with PrefixSpan on the *NCI* [2] molecular structure dataset with 265,242 subgraphs where we set $\tau_{sup} = 60,000$.

Acknowledgments: The research of Da Yan and Guimu Guo is supported by NSF OAC 1755464 and NSF DGE 1723250. The research of Wenwen Qu and Xiaoling Wang is supported by NSFC grant (No. 61532021) and Zhejiang Lab (No. 2019KB0AB04).

References

[1] COST in the Land of Databases. https://github.com/frankmcsherry/blog/blob/master/posts/2017-09-23.md.

[2] NCI Dataset. https://cactus.nci.nih.gov/download/nci/.

[3] C. C. Aggarwal and J. Han, editors. *Frequent Pattern Mining*. Springer, 2014.

[4] M. Bhuiyan and M. A. Hasan. An iterative mapreduce based frequent subgraph mining algorithm. *IEEE Trans. Knowl. Data Eng.*, 27(3):608–620, 2015.

[5] J. Cheng, Y. Ke, W. Ng, and A. Lu. Fg-index: towards verification-free query processing on graph databases. In *SIGMOD*, pages 857–872, 2007.

[6] T. Kudo, E. Maeda, and Y. Matsumoto. An application of boosting to graph classification. In *NIPS*, pages 729–736, 2004.

[7] W. Lin, X. Xiao, and G. Ghinita. Large-scale frequent subgraph mining in mapreduce. In *ICDE*, pages 844–855, 2014.

[8] J. Pei, J. Han, B. Mortazavi-Asl, H. Pinto, Q. Chen, U. Dayal, and M. Hsu. Prefixspan: Mining sequential patterns by prefix-projected growth. In *Proceedings of the 17th International Conference on Data Engineering, April 2-6, 2001, Heidelberg, Germany*, pages 215–224, 2001.

[9] Z. Peng, T. Wang, W. Lu, H. Huang, X. Du, F. Zhao, and A. K. H. Tung. Mining frequent subgraphs from tremendous amount of small graphs using mapreduce. *Knowl. Inf. Syst.*, 56(3):663–690, 2018.

[10] C. H. C. Teixeira, A. J. Fonseca, M. Serafini, G. Siganos, M. J. Zaki, and A. Aboulnaga. Arabesque: a system for distributed graph mining. In *SOSP*, pages 425–440, 2015.

[11] K. Wang, Z. Zuo, J. Thorpe, T. Q. Nguyen, and G. H. Xu. Rstream: Marrying relational algebra with streaming for efficient graph mining on A single machine. In *OSDI*, pages 763–782, 2018.

[12] X. Yan and J. Han. gspan: Graph-based substructure pattern mining. In *ICDM*, pages 721–724, 2002.

[13] M. J. Zaki. Efficiently mining frequent embedded unordered trees. *Fundam. Inform.*, 66(1-2):33–52, 2005.