ASIPATH: A SIMPLE PATH MINING ALGORITHM

Ayhan Demiriz *
Department of Industrial Engineering,
Sakarya University, 54187 Sakarya, Turkey
Email:demira@rpi.edu

Abstract

A SImple yet very efficient PATH mining algorithm (ASIPATH) is introduced in this paper to analyze the click stream data. As in the case of the preceding sequence mining algorithm (webSPADE), ASIPATH requires only one full scan of the data and several partial scans. In parallel algorithm design, it is very common to parallelize a serial algorithm, but ASIPATH is designed for multi-CPU environment in the first place without a preceding serial version. Due to the choice of the application environment i.e. Windows, ASIPATH is not designed as a distributed algorithm but can easily be modified for a distributed environment. In contrast with many existing path mining algorithms, it is not based on a tree-like data structure and search method. By using very efficient join operations, the parallelization of the algorithm is simplified considerably.

Keywords: Path Mining, Traversal Pattern Mining, Web Logs

1 Introduction

One of the most common application areas of sequence mining is the analysis of click stream data. Many successful applications exist [15, 11, 10, 1] including our webSPADE algorithm [3, 4]. By design, sequence mining algorithms find all possible combinations of frequent item sets under temporal, as well as non-temporal, and minimum support constraints. Thus, a sequence mining algorithm will find a rule related to two pages that are not necessarily linked directly (no hypertext link between these two pages). But, in commercial applications, many reports such as attrition reports might require the next page click analysis (web traversal path analysis). Thus we might have to incorporate site structure to filter the rules found by sequence mining algorithms. It is not always feasible to have a constant web site structure to report on. Most of the commercial web sites are either dynamic or frequently changed in nature.

A simple yet very efficient algorithm, ASIPATH, is introduced in this paper to find the frequent paths that exist in click stream data. Although ASIPATH inherits some design properties from our earlier sequence mining application webSPADE [3, 4], it is a totally new approach to path mining. From the beginning, the algorithm has been designed in a parallel mode.

Borrowing the notation from [1] and adapting it for path mining on click stream data, let $I = \{i_1, i_2, \ldots, i_n\}$ be a set of items (pages). $X \subseteq I$ is called an $itemset$ and $|X|$ is the $size$ of X. A path $p = (p_1, p_2, \ldots, p_m)$ is an ordered list of pages where $p_i \in I,\ i \in \{1, \ldots, m\}$ and the page p_1 is visited just before the page p_2, the page p_2 is visited just before the page p_3 and so on. The ordered list implicitly requires the existence of a hypertext link between consecutive pages, but this link could be broken by the web users if they type a `url` instead of using the hypertext links that exist in the document. In addition, backward movement is allowed in our analysis. The size, m, of a path is the number of pages (items) in the path, i.e. $|p|$.

Note that click stream data is organized by user-sessions. A user can only request (view) a single page at any given time. This results in a fundamental difference between click stream analysis and standard sequence mining. In standard sequence mining, a sequence might be composed of several $itemsets$ that have one or more items in them. Due to this difference, the length, l, of a path is equal to the size of that particular path i.e. $|p|$. The set $\mathcal{F}_l$ represents l-length frequent paths. Particularly, $\mathcal{F}_1$ and $\mathcal{F}_2$ represent the frequent pages and the frequent links (adjacent pages) respectively.

A path $p_a = (a_1, a_2, \ldots, a_n)$ is contained in another path $p_b = (b_1, b_2, \ldots, b_m)$ if there exist integers $1 \leq i_1 < i_2 < \ldots < i_n \leq m$ such that $a_1 = b_{i_1}, a_2 = b_{i_2}, \ldots, a_n = b_{i_n}$. If path p_a is contained in path p_b, then we call p_a a subpath of p_b.

A database, $\mathcal{D}$, is a set of tuples (sid,time,pid), where *sid* is web browser session identifier, *time* is the page request time, and *pid* is the requested page. Tuples in $\mathcal{D}$ are sorted by both *sid* and *tid* and then they are given to the algorithm to be analyzed. The database $\mathcal{D}$ is initially kept fully in the memory. *time* is solely used to sort the data. Instead of *time*, page request index (order) that increments by one is used within the algorithm. The purpose of using the index is to enable the join operation to find the consecutive page requests. Note that the join operation is done at the memory level. It is not in the context of a relational database operator in our implementation.

The main contribution of this paper is to find the frequent paths in a recursive way by joining them with $\mathcal{F}_2$. This is a significant difference from Apriori-like algo-

*This paper was written while the author was an employee of Verizon Communications.

rithms, because they need to scan all the data (full scan) at each step to generate the candidate lists $\mathcal{C}$. Actually, there is no need for generating candidate lists in our approach except to speed up the algorithm. The recursive nature of the algorithm might imply a depth-first or a breadth-first search-like mechanism at the first glance. But our parallel implementation avoids any explicit search method by using a First-In-First-Out (FIFO) queue with multiple servers. The server here is used the same way as in the context of queueing theory. In other words, instead of joining $\mathcal{F}_{k-1}$ with itself, it is easier to parallelize the algorithm by joining $\mathcal{F}_{k-1}$ with already computed $\mathcal{F}_2$. In this case, there is no need to wait for the computation of all $\mathcal{F}_{k-1}$ in order to compute $\mathcal{F}_k$. Therefore, parallelization of the algorithm in a distributed memory and multi-CPU environment is very straightforward.

Another contribution of this paper is the discussion on three different ways of enumerating $\mathcal{F}_2$. First two methods are based on join operations on $\mathcal{F}_1$. The last one requires an additional database scan, however it is not based on join operation.

The rest of the paper is organized as follows. Problem definition and motivation behind ASIPATH are given in the next section. Some of the well-known concepts such as support level are used in a slightly different manner in this paper to address the issues faced in click stream data analysis. Section 2 also emphasizes on giving the rationale behind these modifications. The detail explanation of ASIPATH is given in Section 3. We report the performance of ASIPATH on real click stream data in Section 4 with several ways to enhance the algorithmic performance. It then follows with a conclusion in Section 5.

2 Motivation and Problem Definition

The structure of a web site can be represented by an unweighted graph [8]. A web site can be considered as a set of documents (pages) connected via hypertext links. Each document corresponds to a vertex and each hypertext link corresponds to an arc within the graph representation. Basically, each hypertext link (arc) connects two different documents (vertices). Since there is no tangible cost associated with the movement from one document to another, the underlying graph can be considered unweighted. A consecutive sequence of page views can be considered a traversal pattern, since each pair of consecutive arcs connects two vertices.

Using association mining to analyze the click stream data results in a very large number of associations among the pages. Since association mining does not incorporate web site structure or the page request (view) times, resulting rules will not reflect the user experience. Thus association rules might be misleading and incomplete from an analysis perspective.

The concept of path mining in this paper is used the same meaning as traversal pattern discovery. Thus we define path mining as follows. Given a collection of sessions (transactions) which are paths taken by users and parts of the graph that represents the web site, all subpaths that have sufficient number of frequencies are found. The *sufficient number* corresponds to support level in standard association and sequence mining. In accordance with the terminology of standard association mining, frequent paths are also called as *large* paths.

Sequence mining addresses the same problem, but in a different manner. The difference is that there is no condition on the next page in sequence mining. This means that two consecutive pages in a sequence are not necessarily connected by a hypertext link. There could be a gap within a sequence in terms of hypertext links. Depending on the existence of such a gap, a sequence may not represent a subgraph. In other words, there is no existing arc between two consecutive items (vertices) in a sequence. Analyzing click stream data using sequence mining is addressed in [11] as well as in the preceding work [3, 4] of this paper.

As pointed out earlier, path mining might be a better choice as an analysis tool compared to sequence mining. As long as there is a need for attrition-report-type analysis, path mining would be a preferred tool. However, sequence mining has its own merits such as the ability to catch flow from one application to another. Speaking of `applications`, commercial web sites are composed of several applications e.g. Bill View, Bill Pay and Registration. Usually the first page in any given application gets the highest hits and the last page (e.g., Thank-you and Confirmation pages) gets the least hits. This is so called a funnel type shape in terms of traffic. So, if an analysis is done for the sessions that have the first page in them, the resulting click stream data will be very sparse. However, if the same analysis is done for the last page, the resulting click stream data will be very dense, since the sessions are conditioned to include the successful visits to the last page. In this case, support level will not be able to prune the rule space and number of rules will be very high. For instance, underlying sequence mining algorithm could end up finding 800K sequences for merely 40K hits of click stream data.

2.1 Related Work

Mining traversal patterns was originally discussed in [2]. The problem studied in [2] is to find *maximal forward references*. First, the raw log files are analyzed in [2] to find maximal forward references and then, the frequent traversal patterns (large reference sequences) are found by analyzing maximal forward references. Basically, maximal forward references correspond to the longest paths in a session without visiting a previously visited page (backward reference). To illustrate maximal forward references, assume that following traversal path is used in a session $\{A, B, C, D, C, B, E, G, H, G, W, A, O, U, O, V\}$ - adapted from [2]. Maximal forward references for this session are $\{ABCD, ABEGH, ABEGW, AOU, AOV\}$.

Finding maximal forward references is itself an important step to consider, but the algorithms for finding

the frequent traversal patterns is more related to our work in this paper. Authors propose two different algorithms to analyze maximal forward references in [2] namely `Full Scan` (FS) and `Selective Scan` (SS). The algorithm FS is based on authors' earlier work DHP [9] that utilizes hashing and pruning. Although the algorithm FS trims the transaction database in each level, it is still required to scan the transaction database in each pass. As an alternative to FS, SS is proposed in [2] to avoid unnecessary scans in each level by properly utilizing the candidate lists.

The algorithm FS also utilizes join operation in each level to generate candidate lists. In other words, $\mathcal{C}_k$ can be generated by joining $\mathcal{F}_{k-1}$ with itself, denoted by $\mathcal{F}_{k-1} * \mathcal{F}_{k-1}$. Note that the proposed algorithm in this work, ASIPATH, generates $\mathcal{F}_k$ by the join operation $\mathcal{F}_{k-1} * \mathcal{F}_2$. This is one of the fundamental differences between ASIPATH and FS proposed in [2]. Another difference is that, after generating $\mathcal{C}_k$, the algorithm FS requires a (trimmed) transaction database scan to determine $\mathcal{F}_k$. Join operation in FS is done on sequences that have identical parts after dropping the first item in one sequence and the last item in the other sequence.

A web mining tool, SpeedTracer, is introduced in [13]. SpeedTracer is a stand-alone web mining tool that first sessionizes the raw web log data and uses the algorithms introduced in [2] to analyze the data. In addition to finding the frequent traversal paths, SpeedTracer also provides some standard reports related to user session characteristics, frequent paths and page groups. A page group can essentially be defined as a set of pages that are frequently visited in user sessions. It is a straight implementation of the association mining in the context of web mining.

As explained in [8], transactions may get corrupted (in the context of maximal forward references [2]) by irrelevant page accesses during the user sessions. Such cases can be considered noisy transactions. Thus finding maximal forward references might lead to unexpected outcomes, such as missing the main paths along the web structure. At the end, this results in misleading frequent traversal paths.

To overcome such cases, authors approach the problem of finding traversal paths from a graph theory perspective in [8]. The proposed algorithm in [8] finds the association rules based on the graph structure and takes into account the notion of subpath containment. The resulting algorithm is very similar to the Apriori algorithm with the difference of eliminating the large item candidates that violate the graph structure of the web site. Special data structures are used in [8] to accommodate an efficient algorithm other than the original data structures used in Apriori (hash-tree and hash-table). By using the algorithm proposed in [8], authors also introduce a web prefetching system in [7].

Suffix trees are extensively used for the problem of string search and combinatorial pattern matching. In reality, a string is nothing but an ordered list (sequence) of characters that follow each other. Matching a substring within a large string is primarily used for counting substrings. In this case, a substring is provided to the search algorithm. However, discovering the substrings is a totally different task from the pattern matching task. A naïve sequence mining algorithm would first find all the substrings and then counts them, utilizing a string search algorithm such as suffix trees. A dynamic suffix tree is proposed in [12] in the context of incremental sequence mining when the underlying database is updated (changed.) It was shown that dynamic suffix trees run in time independent of database size for small updates and large databases. Again, the problem in [12] assumes that all the sequences are found in the first place, then a dynamic suffix tree is used to perform an incremental discovery when the database is updated.

Table 1. Sample Problem: Click Stream Data

SID	Click Stream
1	A, D, F
2	A, C, D
3	A, B, D, F, B, D
4	A, C, F ,C
5	A, C, B, D
6	A, D, F

2.2 Revisiting Definitions: Sample Problem

In standard sequence mining, a transaction identifier (*tid*) is used to distinguish the itemsets (X) from each other for a given customer identifier (*cid*). When click stream data is analyzed, we can also incorporate a user identifier. Analyzing repeating customers is an important topic in web mining. This is crucial in the case of marketing oriented web sites such as `amazon.com`. When web sites involve recurring activities such as bill view and bill pay, analyzing the repeating customers will not reveal unknown patterns. Thus analysis at session level is acceptable for our purpose. Nevertheless, ASIPATH can also be used for analyzing the repeating user data with some modifications, but it is out of scope of this paper.

Since the problem domain in this paper is click stream data at session level, the definition of the *support level* is different from the one at transaction level which is used generally in association and sequence mining [15, 10, 1]. Following the same definition as in [3, 4], we define the support of a path, σ, as the number of requests out of all the page requests. Minimum support in this context becomes the number of requests that a page should receive out of the total requests in database $\mathcal{D}$ to be considered as a frequent page. The frequent pages then form the set ($\mathcal{F}_1$). For example, if there are 1M total hits in database $\mathcal{D}$ regardless of the number of sessions, then a page should have at least 1000 requests to be considered as a frequent page at 0.1% support level. Similarly, a path (e.g. $A \rightarrow B \rightarrow C$) should have at least a frequency of 1000 for the same reason to

be considered as a frequent path. If a certain path is taken twice in a session, it will be counted twice as opposed to once in a transaction based consideration.

To illustrate the algorithm and understand the terminology better, a sample problem is given in Table 1. SID stands for the Session IDentifier. Notice that each session is a collection of pages, ordered by their request times. For the simplicity, request times are not included in this sample data. As mentioned above, the order (index) of the click stream data is enough to perform the path mining. Nevertheless, we can also use the index as the time stamp in our sample data.

There are twenty three requests in total from all the six sessions. Assuming that the minimum support is two, then the frequent pages ($\mathcal{F}_1$) are summarized in Table 2. In this case, the support level is equal to $\frac{2}{23} \approx 9\%$. Actually, all the distinct pages in this sample problem end up being as the frequent pages. The next step in our path mining process is to enumerate the frequent two-page paths ($\mathcal{F}_2$). Notice that we can enumerate two-page paths by joining $\mathcal{F}_1$ with itself i.e. $\mathcal{F}_1 * \mathcal{F}_1$. There are two constraints for the join operation that the requests should be

- in the same session
- one-click apart.

To illustrate the join operation, consider joining the sets Page D and Page F (see Table 2) to generate the set D→F. Notice that D and F pairs {(1:2)(1:3),(3:3) (3:4),(6:2)(6:3)} satisfy the above constraints. The resulting set can be found in Table 3. By using an efficient join operation, we can also figure out in the same join operation whether F→D is also a frequent two-page path or not. Obviously, F→D is not a frequent path for this sample problem. Another observation from the sample problem is that the path B→D is taken twice in Session 3. Thus the set B→D has two records from Session 3.

The solution for the enumerating longer frequent paths is a little bit different from enumerating $\mathcal{F}_2$. As mentioned previously, it is sufficient to join $\mathcal{F}_{k-1}$ with $\mathcal{F}_2$ to generate $\mathcal{F}_k$. For example, if we join the sets A→D and D→F, the resulting set will be A→D→F as seen in Table 3.

We briefly illustrate the concept of path mining in this subsection around a sample click-stream data. Implementation of this concept has some challenging issues. We explain the implementation of the algorithm ASIPATH in the next section.

3 Algorithm: Details and Comparisons

Parallel programs are usually developed by modifying their serial versions. In terms of parallelization of the serial programs, the memory type of the computer system is an important factor to consider when designing the algorithm. Since path mining is very similar to sequence mining in nature, it is important to understand the algorithmic challenges of sequence mining in parallel environments.

Sequence mining can be considered as an irregular tree search algorithm [16], with each node corresponding to an equivalence class. According to argument in [16], parallelization of the sequence mining can be achieved either by data or task parallelism. In data parallelism, processors work on distinct partitions of the database but process the global tree structure concurrently. Task parallelism, on the other hand, requires each processor to have a separate copy of the database and run on different branches of the global tree. Experiments in [16] show that task parallelism is more favorable than data parallelism, with the best task parallelism approach using recursive dynamic load balancing [16].

A parallel version of the tree projection algorithm is proposed in [5]. Each node in the projection tree corresponds to k-itemset. The projection tree grows in a breadth-first manner. Data and task parallelism are compared again in [5]. Similar to the results in [16], task parallelism results are more favorable in terms of work load and computation time. An extension to [5] is introduced in [6] by implementing a dynamic load balancing scheme. The dynamic load balancing scheme in [6] performs similar to or better than the static load balancing schemes used in [5].

Algorithm 3.1 (ASIPATH)**.**

Given min_support and database $\mathcal{D}$
$\mathcal{F}_1 = \{$*Frequent items*$\}$
$\mathcal{F}_2 = \{$*Frequent item-pairs (two-item path)*$\}$
$Q = \mathcal{F}_2$
$Enumerate - Freq - Paths(Q)$*;*

A high level pseudo-code of the algorithm ASIPATH is given in Algorithm 3.1. Steps in this pseudo-code are similar to the solution studied in Section 2.2 for the sample problem. Enumerating $\mathcal{F}_2$ is the most CPU intense step in ASIPATH as in the case of webSPADE [4]. ASIPATH runs on a single machine with multiple CPUs utilizing a shared-memory, due to the choice of the application environment. ASIPATH does not require utilization of any search method such as breadth-first and depth-first. It utilizes a job queue in a multi-threaded environment in enumerating the three-page or longer frequent paths. This particular queue resembles a single line First-in-First-Out Multi Server queue from the queueing theory. Thus, it is not necessary that all k-length paths will be found before finding $(k+1)$-length paths. It is very likely that a join operation involving a $(k+1)$-length path might be performed before another join operation involving a k-length path. In other words, parallelization of the algorithm is considerably simplified. The details of the function `Enumerate-Freq-Paths` is given below.

Table 2. Frequent Pages ($\mathcal{F}_1$)

A	
SID	Index
1	1
2	1
3	1
4	1
5	1
6	1

B	
SID	Index
3	2
3	5
5	3

C	
SID	Index
2	2
4	2
4	4
5	2

D	
SID	Index
1	2
2	3
3	3
3	6
5	4
6	2

F	
SID	Index
1	3
3	4
4	3
6	3

Table 3. Frequent Paths ($\mathcal{F}_2$ and $\mathcal{F}_3$)

A→C	
SID	Index
2	2
4	2
5	2

A→D	
SID	Index
1	2
6	2

B→D	
SID	Index
3	3
3	6
5	4

D→F	
SID	Index
1	3
3	4
6	3

A→D→F	
SID	Index
1	3
6	3

Algorithm 3.2 (Enumerate-Freq-Paths)**.**

Given Set S
while S is not empty do:
print the first path (item-list)
Let A_1 be the first item-list in S
for all item-pairs (P_i) in $\mathcal{F}_2$ do:
if last item of A_1 is equal to the first item *of P_i* **then:**
$R = A_1 \bigvee P_i$;
$\mathcal{L}(R) = \mathcal{L}(A_1) \cap \mathcal{L}(P_i)$; ***Join operation***
if $\sigma(R) \geq min\_support$ **then**
S.push_back(R);
endif
endif
S.remove(A_1);
endfor
endwhile

The function `Enumerate-Freq-Paths` could be written in a recursive way to utilize the breadth-first or depth-first search methods, but we intentionally avoid this approach to leave the load balancing to the operating system and to prevent unnecessary housekeeping steps. In this case, a thread is created with the first item in the set (queue) provided that the number of threads is lower than the number of maximum allowable threads i.e. a parameter. The length of the frequent paths is also limited at ten for ASIPATH as in webSPADE [4].

The basic idea in ASIPATH is to join $\mathcal{F}_{k-1}$ with $\mathcal{F}_2$, i.e. $\mathcal{F}_{k-1} * \mathcal{F}_2$, to enumerate $\mathcal{F}_k$ for $k \geq 3$. ASIPATH does not require the utilization of the web site structure in the discovery process. A limited usage of ASIPATH has been in production since May, 2003 to analyze the web logs from Verizon.com. Interested readers can see some screen shots from the application in [4] related to "Page Based Analysis" part. Note that path mining is only mentioned as a tool in [4] without any algorithmic details. The limitation on the length of the frequent paths comes handy when using relational tables efficiently in displaying results in our application.

In general there are two types of join operation in ASIPATH. The first one is for enumerating $\mathcal{F}_2$ and the other is for longer paths. Although enumerating $\mathcal{F}_2$ is done in a parallel mode by using two-way joins in a single scan, this step is the most time consuming due to the exhaustive search requirement. Notice that the merit of the using join operation in path mining as well as in sequence mining is to minimize the memory requirements as observed in [1]. In the next section, we will introduce two different schemes to avoid the exhaustive search in this step.

4 Alternatives to Enumerate $\mathcal{F}_2$: Experimental Section

In this section, we introduce two different ways to enhance the performance of the algorithm ASIPATH. The baseline algorithm ASIPATH, as introduced in previous sections, is called "Full Join" to differentiate from the enhancements studied in this section. The first enhancement is called "Smart Join" and is based on a scheme to do selective joins.

In this way of enumerating $\mathcal{F}_2$, the algorithm tracks the page couplings during the database access. There is no counting process involved, the algorithm only checks during the database scan whether a page comes after another page or not. After completing the database access and find-

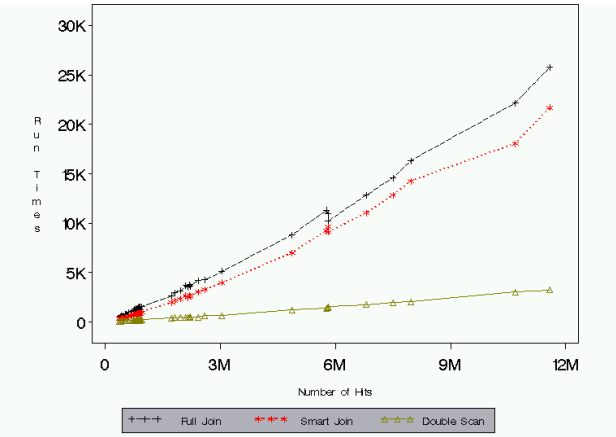

Figure 1. Run Times vs. Number of Hits

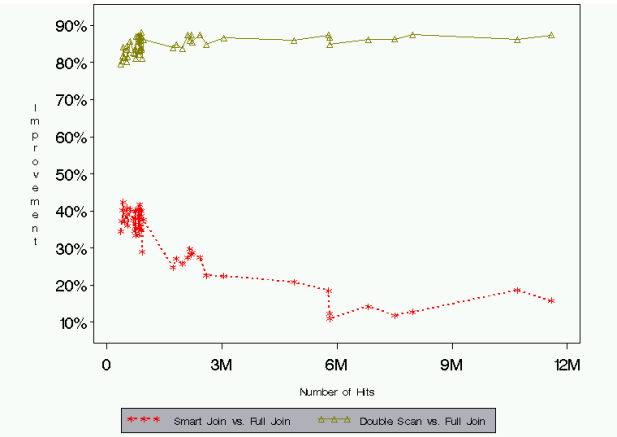

Figure 2. Runtime Improvements

ing $\mathcal{F}_1$, depending on the resulting page couplings join operation is conducted on $\mathcal{F}_1$ with itself. Notice that some of the page couplings might be two-way (e.g. A→B and B→A) because of backward movement and some of them only one-way (e.g. only A→B, but not B→A). Although we devise two different join operations for these cases, our initial experiments show that there is no significant performance changes when we use one-way join operation for the one-way coupling instead of using two-way join that enumerates for an unnecessary path. This is because both join operations are based on an efficient one time scan.

The second way of enhancing ASIPATH is called "Double Scan", as the name suggests, it is based on scanning the database an additional time to completely avoid the join operation for enumerating $\mathcal{F}_2$. The crucial step in "Double Scan" is to use special data structures to count the page couplings efficiently. As in the case of Apriori-like algorithms, double scan first generates a candidate list in the first database scan and then it counts the two-item paths in the second database scan. Finally, the frequent two-item paths are determined based on the support level.

We compare the baseline algorithm and other two enhancements on click stream data collected from Verizon.com during month of August, 2002. To show the scalability of the algorithm, the click stream data have been used in different time periods. First, experiments are conducted on daily click stream data in other words 31 different data chunks. We then use 3-day long data chunks i.e. 10 different measurements. We use weekly, 10-day, and bi-weekly data as well. At the end, we are able to observe the algorithm on 50 different data chunks. Experiments are conducted on a server that has 8 CPUs running at 700 MHz clock speed with 4GB of total memory. A fixed (0.1%) support level is used for all the experiments. Figure 1 depicts the runtimes from August 2002 data by changing the data size. Times are reported in seconds. Number of hits are reported in millions.

Using the smart join approach improves ASIPATH a little bit but not significantly. To investigate the improvement levels at different data sizes, Figure 2 is drawn. It seems smart join shows 40% of improvement for the small size data (i.e. daily). When the data size is increased, its performance diminishes. A logical explanation for this phenomenon is that for the large click stream data it is high likely that there will be more page couplings. Then algorithm behaves in a way that it gets closer to the exhaustive search method for enumerating $\mathcal{F}_2$.

On the other hand, double scan shows well over 80% of improvement constantly for all the data sizes. It also scales very well for the large data. We also observe that using additional scan to enumerate $\mathcal{F}_2$ is very crucial to improve the scalability. Furthermore the size of the data gets smaller by increasing length of paths, the join operation can safely be used for enumerating frequent paths that are longer than two.

5 Conclusion

A path mining algorithm, ASIPATH, based on join operations is introduced in this paper. The original algorithm requires only one database scan but several intermediate partial scans. We also discussed two enhanced versions of this algorithm in this paper. Based on the experiments using click stream data from Verizon.com, we noticed a significant improvement in terms of run time of the algorithm when the second database scan is used for enumerating $\mathcal{F}_2$.

The content of this paper is limited to the introduction of the idea of using join operation in path mining. Although the existing algorithms are designed for slightly different problems, ASIPATH should be compared with them in terms of algorithmic performance. Moreover, we need to study the similarities and the differences between constrained sequence mining [14] and path mining. Path mining can be considered as a special case of sequence mining under constraints of minimum and maximum gaps equal to 1 (see [14]).

Experiments in this paper showed that ASIPATH is very scalable. By developing a user friendly front-end, ASIPATH can easily be utilized in analysis of the click stream data to answer various business questions stemming from the issues related to a commercial web site.

Acknowledgments

I would like to thank to Dr. Mohammed J. Zaki for his comments on an early draft of this paper.

References

[1] J. Ayres, J. Gehrke, T. Yiu, and J. Flannick. Sequential pattern mining using a bitmap representation. In David Hand, Daniel Keim, and Raymond Ng, editors, *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM Press, 2002. Edmonton, CANADA.

[2] M.-S. Chen, J.-S.Park, and P. S. Yu. Efficient data mining for path traversal patterns. *IEEE Transactions on Knowledge and Data Engineering*, 10(2):209–221, 1998.

[3] A. Demiriz. webspade: A parallel sequence mining algorithm to analyze web log data. In *Proceedings of The Second IEEE International Conference on Data Mining (ICDM 2002)*, pages 755–758. IEEE Computer Society, 2002. Maebashi City, Japan.

[4] A. Demiriz. On analyzing web log data: A parallel sequence mining algorithm. In Mehmed Kantardzic and Jozef Zurada, editors, *New Generation of Data Mining Applications (To be published)*. IEEE-Wiley, 2003. Available at http://www.rpi.edu/∼demira/webspadebkchap.pdf.

[5] V. Guralnik, N. Garg, and G. Karypis. Parallel tree projection algorithm for sequence mining. In R. Sakellariou, J. Keane, J. Gurd, and L. Freeman, editors, *Proceedings of Seventh European Conference on Parallel Computing (Euro-Par)*, pages 310–320. Springer, 2001.

[6] V. Guralnik and G. Karypis. Dynamic load balancing algorithms for sequence mining. Technical Report 00-056, Department of Computer Science, University of Minnesota, 2001.

[7] A. Nanopoulos, D. Katsaros, and Y. Manolopoulos. A data mining algorithm for generalized web prefetching. *IEEE Transactions on Knowledge and Data Engineering,*, 15(5), September/October 2003.

[8] A. Nanopoulos and Y. Manolopoulos. Finding generalized path patterns for web log data mining. In J. Stuller, J. Pokorny, B. Thalheim, and Y. Masunaga, editors, *Proceedings of Fourth East-European Conference on Advanced Databases and Information Systems*, pages 215–228. Springer, 2000. Prague, Czech Republic.

[9] J.-S. Park, M.-S. Chen, and P. S. Yu. An effective hash based algorithm for mining association rules. In Michael J. Carey and Donovan A. Schneider, editors, *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, pages 175–186, San Jose, California, 1995.

[10] J. Pei, J. Han, H. Pinto, Q. Chen, U. Dayal, and M.-C. Hsu. Prefixspan: Mining sequential patterns efficiently by prefix-projected pattern growth. In *Proceedings of International Conference on Data Engineering (ICDE'01)*, 2001. Germany.

[11] J. R. Punin, M. S. Krishnamoorthy, and M. J. Zaki. Logml: Log markup language for web usage mining. In Ron Kohavi, Brij M. Masand, Myra Spiliopoulou, and Jaideep Srivastava, editors, *WEBKDD 2001-Mining Web Log Data Accross All Customers Touch Points, San Francisco, CA, USA*, pages 88–112. Springer, 2002.

[12] K. Wang. Discovering patterns from large and dynamic sequential data. *Journal of Intelligent Information Systems*, 9(1):33–56, 1997.

[13] K.-L. Wu, P. S. Yu, and A. Ballman. Speedtracer: A web usage mining and analysis tool. *IBM Systems Journal*, 37(1):89–105, 1998.

[14] M. J. Zaki. Sequence mining in categorical domains: Incorporating constraints. In *Proceedings of the 9th International Conference on Information and Knowledge Management*, pages 422–429, Washington, DC, 2000.

[15] M. J. Zaki. SPADE: An efficient algorithm for mining frequent sequences. *Machine Learning Journal*, 42(1/2):31–60, Jan/Feb 2001. Special issue on Unsupervised Learning (D. Fisher, editor.).

[16] M. J. Zaki. Parallel sequence mining on shared-memory machines. *Journal of Parallel and Distributed Computing,*, 61(3):401–426, March 2001. Special issue on High Performance Data Mining (V. Kumar, S. Ranka and V. Singh, editors.).