

A Valid Candidate Approach to Mining Bi-Directional Traversal Patterns on the WWW *

Jiun-Rung Chen and Ye-In Chang

Dept. of Computer Science and Engineering, National Sun Yat-Sen University

Kaohsiung, Taiwan, R.O.C

E-mail: changyi@cse.nsysu.edu.tw Tel: 886-7-5252000 (ext. 4334) Fax: 886-7-5254301

Presenter: Jiun-Rung Chen Submitted to: ICOMP'06

Abstract

Mining traversal patterns is one of important topics in Web mining. It focuses on how to find the Web page sequences which are frequently browsed by users. In this paper, we propose two algorithms for mining traversal patterns. For the first algorithm, SpeedTracer-I, it is a revised version of the SpeedTracer algorithm. It directly generates and counts all candidate patterns from user sessions. Moreover, it improves the checking step when candidate patterns are generated. Next, based on the SpeedTracer*-I algorithm, we propose the SpeedTracer*-II algorithm, which improves the performance of the SpeedTracer*-I algorithm by decreasing the times to scan the database. From the simulation results, we show that the SpeedTracer*-I algorithm needs less processing time than the SpeedTracer algorithm. Moreover, the SpeedTracer*-II algorithm needs less processing time than SpeedTracer*-I and Apriori-like algorithms (e.g., FS and FDLP algorithms).*

(keywords: association rules, data mining, traversal patterns, Web mining, WWW)

1. Introduction

Mining traversal patterns is one of important topics in Web mining [3, 4]. Finding the traversal patterns will not only help to improve the system design (e.g., providing efficient access between highly correlated objects), but also be able to lead to better marketing decisions (e.g., putting advertisements in proper places) [2].

In the mining traversal patterns problem, the database is a set of user sessions, where a user session is a sequence of Web pages made by a user

during a specified period. The mining traversal patterns problem is to find all the *frequent traversal patterns* in the database. A frequent traversal pattern is a traversal pattern of Web pages that its support is larger than the user-specified minimum support.

There have been several algorithms proposed for mining traversal patterns. Basically, those algorithms can be classified into two types: type 1 considers *simple traversal patterns*, where each page only appears once in a pattern, for example, MF/FS/SS [2] and SpeedTracer [6] algorithms; type 2 considers *non-simple traversal patterns*, where one page may appear more than one time in a pattern, for example, the FDLP algorithm [7].

Most of the previous algorithms [2, 7] based on the Apriori-like algorithms [1, 5] will generate too many invalid candidate patterns (i.e., a candidate pattern with count equal to 0), since they generate candidate patterns from previous frequent patterns. In the SpeedTracer algorithm [6], they utilize the property of Web transactions, i.e., the *continuous* property, to reduce the number of candidate patterns. The continuous property is that the pages of a traversal pattern will occur continuously in any user session. For example, in Figure 1-(a), pattern $\{ACD\}$ is an invalid candidate pattern, because it does not occur continuously in the first user session $\{\underline{A}BC\underline{B}D\}$ or other user sessions. Figure 1-(b) shows a comparison of the related total number of candidate 2-patterns for Figure 1-(a), where the number of candidate patterns in the SpeedTracer algorithm is much smaller than that in Apriori-like algorithms, and there is no invalid candidate pattern generated in the SpeedTracer algorithm.

Although the SpeedTracer algorithm avoids generating invalid candidate patterns, it does not check the subsets of a candidate pattern efficiently. In this paper, we propose two algorithms for min-

*This research was supported in part by the National Science Council of Republic of China under Grant No. NSC-94-2213-E-110-003.

UserID	User Sessions
100	ABCB
200	BCBA
300	ABD
400	ABCBA

(a)

Algorithm	Apriori-like	SpeedTracer
C_2	AB, <u>AC</u> , <u>AD</u> , BA, BC, BD, <u>CA</u> , CB, <u>CD</u> , <u>DA</u> , <u>DB</u> , <u>DC</u>	AB, BA, BC, BD, CB
The number of C_2	12	5

(b)

Figure 1: An example: (a) the database; (b) a comparison of the number of candidate patterns with $L_1 = \{\{A\}, \{B\}, \{C\}, \{D\}\}$.

ing non-simple traversal patterns. For the first algorithm, SpeedTracer*-I, we reduce the number of checks for each candidate pattern, as compared to the SpeedTracer algorithm. Next, to reduce the times to scan the database, we propose the SpeedTracer*-II algorithm. From our simulation results, we show that the SpeedTracer*-I algorithm outperforms the SpeedTracer algorithm in terms of the processing time, and the SpeedTracer*-II algorithm outperforms the SpeedTracer*-I algorithm. Moreover, the SpeedTracer*-II algorithm outperforms Apriori-like algorithms (*e.g.*, FS and FDLP algorithms).

The rest of the paper is organized as follows. Section 2 describes algorithms for mining traversal patterns. Section 3 presents the proposed algorithms. Section 4 gives the performance. Finally, Section 5 gives the conclusions.

2. The SpeedTracer Algorithm

The SpeedTracer algorithm is used to mine simple traversal patterns [6]. It uses an algorithm, which is similar to the MF algorithm [2], to find all the maximum forward references from user sessions. Then, SpeedTracer uses a mining algorithm to find the frequent patterns among these maximum forward references. (Note that although SpeedTracer is used to mine simple traversal patterns, this mining algorithm could also be used to mine non-simple traversal patterns.) Take the maximum forward reference $\{ABCDE\}$ as an example. Different from the Apriori algorithm, SpeedTracer directly generates C_3 from the maximum forward reference $\{ABCDE\}$, *i.e.*, $\{ABC\}$, $\{BCD\}$, and $\{CDE\}$, where each page of each candidate pattern is continuous in the max-

imum forward reference. Candidate patterns like $\{ABE\}$ and $\{ACE\}$, which may be generated in Apriori-like algorithms, will not be generated in the SpeedTracer algorithm. For the first pattern $\{ABC\}$, SpeedTracer checks the supports of its two subsets, $\{AB\}$ and $\{BC\}$. (Note that SpeedTracer does not check $\{AC\}$ since $\{A\}$ and $\{C\}$ are not continuous in $\{ABC\}$.) If one of the supports of $\{AB\}$ and $\{BC\}$ is less than the minimum support, $\{ABC\}$ is pruned. Otherwise, the count of $\{ABC\}$ in C_3 is added by 1. Similarly, for the second pattern $\{BCD\}$, the supports of its two subsets, $\{BC\}$ and $\{CD\}$, are checked. After scanning all user sessions, L_3 could be determined from C_3 . This process will be continued until all the frequent patterns are found.

3. SpeedTracer*-I and SpeedTracer*-II Algorithms

In this Section, first, we present the SpeedTracer*-I algorithm to improve the processing time of SpeedTracer. Next, we present the SpeedTracer*-II algorithm to further improve the performance of the SpeedTracer*-I algorithm.

3.1. The SpeedTracer*-I Algorithm

Similar to the SpeedTracer algorithm, the SpeedTracer*-I algorithm generates the candidate patterns directly from user sessions, instead of being generated from the previous frequent patterns in most of Apriori-like algorithms. If a continuous k -pattern in a user session passes the checking step, which is used to prune unqualified candidate patterns, this pattern is inserted into candidate k -patterns C_k . The main contribution of our SpeedTracer*-I algorithm is the reduction of the number of checks in the checking step, as compared to the SpeedTracer algorithm.

Table 1 shows the variables used in the SpeedTracer*-I algorithm and Figure 2 shows the SpeedTracer*-I algorithm. The input parameter n is used in the SpeedTracer*-II algorithm which will be presented later.

First, the SpeedTracer*-I algorithm scans the database once to count the candidate 1-patterns, C_1 , and determines the frequent 1-patterns, L_1 . Next, it finds frequent k -patterns as follows, where $k > 1$. For each user session, the SpeedTracer*-I algorithm considers all continuous k -patterns in this user session by scanning the database. The variable k is the round number for scanning the database, which also indicates the length of the current generated candidate pattern. A k -pattern could be a candidate pattern if all its continuous

Table 1: Variables used in the proposed algorithms

Variable	Description
L_k	Frequent k -patterns
D	The database of user sessions
t	A user session
i	A variable which records the starting position of the checked pattern
$ t $	The length of t
s	The checked pattern
C_k	Candidate k -patterns
$minsup$	The user-specified minimum support
n	A parameter which is used to call the SpeedTracer*-I procedure to generate L_n
UC	The union of C_k for $k > n$
$UResult$	The union of L_k for $k > n$

subsets with length $(k - 1)$ are in L_{k-1} . Since a k -pattern $\{a_1a_2\dots a_k\}$ is continuous in the user session, we only have to check its two subsets with length $(k - 1)$, *i.e.*, $\{a_1a_2\dots a_{k-1}\}$ (noted as $subset_1$) and $\{a_2a_3\dots a_k\}$ (noted as $subset_2$). In the SpeedTracer algorithm, it will check whether both subsets are in L_{k-1} for each k -pattern. However, in our SpeedTracer*-I algorithm, most of the time, we do not check whether $subset_1$ is in L_{k-1} for k -patterns. (Note that the checking step of the subset could be skipped if this subset is already checked before.) The process of the generation of candidate k -patterns of a user session is described in lines 10 to 37 of procedure *SpeedTracer*-I* shown in Figure 2.

Take Figure 3 as an example, where Figure 3-(a) shows a user session and Figure 3-(b) shows the frequent 1-patterns. Figure 4 shows the process of the SpeedTracer*-I algorithm in this example. The Boolean flag *skip* is used to indicate whether the checking step of the subset of the pattern could be skipped or not. Procedure *contSubset*(t, j, l, r) shown in Figure 5 is used to generate a continuous subset of t , where the subset starts from the j th page of t and its length is l . Since we have $\{AB\}$ as the first continuous 2-pattern of the user session, we have $\{A\}$ as the first subset recorded in $subset_1$ generated by procedure *contSubset*.

Next, we process the checking step; that is, we check whether $subset_1 \in L_{k-1}$. In this example, we check whether $\{A\} \in L_1$. As mentioned above, we only perform the checking step for $subset_1$ when any of the following two conditions occur: (1) the checked k -pattern is the first continuous pattern of a user session (*e.g.*, Figure 4-(a)); (2) the previous checked k -pattern does not pass the checking step (*e.g.*, Figure 4-(e)). While

```

01 /* this procedure will find all  $L_k$  for  $k \leq n$  */
02 procedure SpeedTracer*-I( $n$ , var  $L_n$ );
03 begin
04    $k := 1$ ;
05    $L_1 := \{\text{all frequent 1-patterns in } D\}$ ;
06   while  $((L_k \neq \emptyset) \text{ and } (k < n))$  do
07     begin
08        $k := k + 1$ ;
09       for each  $t \in D$  do
10         begin
11            $i := 1$ ;
12            $skip := \text{FALSE}$ ;
13           while  $(i \leq (|t| - k + 1))$  do
14             begin
15               if  $(skip = \text{FALSE})$  then do
16                 begin
17                   contSubset( $t, i, k - 1, subset_1$ );
18                   if  $(subset_1 \notin L_{k-1})$  then
19                     begin
20                        $i := i + 1$ ;
21                       continue the while loop;
22                     end;
23                   end;
24                   contSubset( $t, i + 1, k - 1, subset_2$ );
25                   if  $(subset_2 \notin L_{k-1})$  then
26                     begin
27                        $i := i + 2$ ;
28                        $skip := \text{FALSE}$ ;
29                       continue the while loop;
30                     end;
31                   contSubset( $t, i, k, s$ );
32                   addCount( $s, C_k$ );
33                    $i := i + 1$ ;
34                    $subset_1 := subset_2$ ;
35                    $skip := \text{TRUE}$ ;
36                 end;
37               end;
38                $L_k := \{cp \in C_k \mid cp.count \geq minsup\}$ ;
39             end;
40           end;

```

Figure 2: The *SpeedTracer*-I* algorithm

in the SpeedTracer algorithm, they always process the checking step for both subsets of the checked k -pattern. That is why our proposed algorithm could provide better performance than the SpeedTracer algorithm. Since $\{A\} \in L_1$, we skip the **if** statements (lines 18 to 22), which are used to skip the current checked pattern (we will explain it in details later by an example).

Then, we generate the second subset of the checked k -pattern, $subset_2$ (line 24). If $subset_2 \in L_{k-1}$, we skip the following **if** statements (lines 25 to 30) which are also used to skip the current checked pattern. In this example, we have $\{B\} \in L_1$, so we skip the **if** statements.

Up to this point, the two subsets of the checked k -pattern have passed the checking step. That is, it is a valid candidate pattern. Therefore, we gen-

UserID	User Session
100	ABCBDEA

L_1
A
B
C
E

(a) (b)

Figure 3: An example: (a) a user session; (b) the frequent 1-patterns L_1 .

<u>A</u> <u>B</u> C B D E A	<u>A</u> <u>B</u> <u>C</u> B D E A
(a)	(b)
A B C <u>B</u> D E A	A B C <u>B</u> D E A
(c)	(d)
A B C B D <u>E</u> <u>A</u>	
(e)	

Figure 4: The checking process of the SpeedTracer*-I algorithm for the example in Figure 3: (a) $\{AB\}$; (b) $\{BC\}$; (c) $\{CB\}$; (d) $\{BD\}$; (e) $\{EA\}$.

erate this candidate k -pattern, s (line 31), and call procedure *addCount* shown in Figure 6 to increase its count. The checking point is then moved to the next page (line 33), since we have finished the checking step of the current candidate k -pattern. We also assign $subset_2$ to $subset_1$, since the current $subset_2$ will be the new $subset_1$ for the next continuous k -pattern. In this example, the next 2-pattern is $\{BC\}$ and we have $subset_1$ of $\{BC\}$ equal to $\{B\}$. In addition, we set the Boolean flag $skip = \mathbf{TRUE}$ (line 35), because we already know that $subset_1$ of the new k -pattern is in L_{k-1} .

In our example, the second 2-pattern is $\{BC\}$ and we only have to check the second subset, *i.e.*, $\{C\}$. The checking step of $subset_1$, *i.e.*, $\{B\}$, is skipped, since we have $skip = \mathbf{TRUE}$. Following the similar steps, we have $\{CB\}$ as the next continuous 2-pattern, which also passes the checking step and is a valid candidate pattern recorded in C_2 . While we process $\{BD\}$ as the continuous 2-pattern, we have $subset_1 = \{B\}$

```

procedure contSubset( $t, j, l, \mathbf{var} r$ );
begin
   $r := \emptyset$ ;
  for  $i = 0$  to  $l - 1$  do
     $r := result \cup$  the  $(j + i)$ th page of  $t$ ;
end;

```

Figure 5: Procedure *contSubset*

```

procedure addCount( $s, \mathbf{var} C_k$ );
begin
  if  $s \in C_k$  then
     $s.count := s.count + 1$ 
  else
    begin
       $C_k := C_k \cup s$ ;
       $s.count := 1$ ;
    end;
  end;
end;

```

Figure 6: Procedure *addCount*

C_2	count	step
AB	1	(a)
BC	1	(b)
CB	1	(c)
EA	1	(e)

Figure 7: The result of the candidate 2-patterns for the example in Figure 3

and $subset_2 = \{D\}$. $subset_1$ passes the checking step immediately. However, for $subset_2$, we have $\{D\} \notin L_1$. Therefore, we skip current pattern $\{BD\}$, reset the Boolean flag $skip$ to **FALSE**, and move the checking point to the next two pages. In this example, the checking point is moved to page E and the current pattern is $\{EA\}$. The Boolean flag $skip$ is **FALSE** now and we have to check both $subset_1$ and $subset_2$. Since both $\{E\}$ and $\{A\}$ are in L_1 , $\{EA\}$ is a valid candidate pattern. The checking point is now moved to the last page of the user session, *i.e.*, page A . Since we have $i > (|t| - k + 1)$, the whole **while** loop stops (line 13).

Up to this point, we finish the processing of the user session in this example. The result of candidate 2-patterns is shown in Figure 7. We have to repeat the processing of the rest of user sessions in the database for generating C_2 and accumulate the related counts. (Note that the checking step is performed in the main memory.) When the last user session is scanned, L_2 can be determined (line 38). We continue to scan the whole database to generate new C_k (lines 9 to 37) and determine L_k (line 38) until $L_k = \emptyset$, where $k > 2$. (Note that we will assign the input variable n with a very large value such that the condition $(k < n)$ is always true in the SpeedTracer*-I algorithm. The value of n will be assigned to a suitable value in the SpeedTracer*-II algorithm.) When $L_k = \emptyset$, procedure *SpeedTracer*-I* stops and we obtain all frequent patterns. The total times to scan the

database is k .

As compared to the SpeedTracer algorithm, the main contribution of the SpeedTracer*-I algorithm is that the SpeedTracer*-I algorithm reduces the number of checks of the checking step. For example, in Figure 4, we only need 7 checks to generate C_2 for this user session. However, in the original SpeedTracer algorithm, it will need two checks for each continuous 2-pattern in the user session. Therefore, it needs 12 checks to generate C_2 for this user session, as compared to 7 checks in our SpeedTracer*-I algorithm.

3.2. The SpeedTracer*-II Algorithm

In the previous section, the SpeedTracer*-I algorithm uses the continuous property to generate the candidate patterns efficiently. However, it needs to scan the database for many times as most of other algorithms need. For example, if the maximal candidate pattern is C_m , the SpeedTracer*-I algorithm needs to scan the database for m times to generate and count the candidate patterns. Therefore, we present the SpeedTracer*-II algorithm to improve the performance of the SpeedTracer*-I algorithm by decreasing the times to scan the database. Basically, in the SpeedTracer*-II algorithm, given a parameter n , we apply the SpeedTracer*-I algorithm to find L_n first, and directly generate and count C_k from user sessions based on L_n , where $k > n$.

Table 1 shows the variables used in the SpeedTracer*-II algorithm. Figure 8 shows the SpeedTracer*-II algorithm. The basic idea of the SpeedTracer*-II algorithm is similar to the SpeedTracer*-I algorithm. That is, the SpeedTracer*-II algorithm makes use of the continuous property in generating all candidate patterns. First, we call procedure *SpeedTracer*-I* to generate L_n . This step needs to scan the database for n times. Next, we scan the database for one more time to generate and count all continuous candidate k -patterns, where $k > n$. For each user session, a continuous k -pattern of this user session could be a candidate pattern if all its continuous subsets with length n are in L_n . The generation of candidate k -patterns, $k > n$, is described from lines 5 to 25 of procedure *SpeedTracer*-II* shown in Figure 8.

Inside the **while** loop (line 7), we focus on each of continuous k -patterns with the checking point starting at position i , where $k = n+1, n+2, \dots, |t| - i + 1$. Then, for each such a continuous pattern, we check whether its continuous subsets with length n are in L_n . If the subset of a continuous pattern does not pass the checking step, this continuous

```

01 procedure SpeedTracer*-II( $n$ );
02 begin
03   SpeedTracer*-I( $n, L_n$ );
04   for each  $t \in D$  do
05     begin
06        $i := 1$ ;
07       while ( $i \leq (|t| - n)$ ) do
08         begin
09           contSubset( $t, i, n, subset_1$ );
10           if ( $subset_1 \notin L_n$ ) then
11             begin
12                $i := i + 1$ ;
13               continue the while loop;
14             end;
15           for  $j := 1$  to ( $|t| - n + 1 - i$ ) do
16             begin
17               contSubset( $t, i + j, n, subset_2$ );
18               if ( $subset_2 \notin L_n$ ) then
19                 break the for loop;
20               contSubset( $t, i, n + j, s$ );
21               addCount( $s, UC$ );
22             end;
23              $i := i + 1$ ;
24           end;
25         end;
26       UResult := { $c \in UC | c.count \geq minsup$ };
27 end;
```

Figure 8: The *SpeedTracer*-II* algorithm

pattern could not be a candidate pattern and the checking point is moved to the next position.

Take Figure 9 as an example, where Figure 9-(a) shows a user session and Figure 9-(b) shows the frequent 2-patterns. The parameter n is 2 now. Figure 10 shows the process of the SpeedTracer*-II algorithm. Pattern $\{ABC\}$ is checked first as shown in Figure 10-(a), where $\{ABC\}$ is the first continuous 3-pattern of the user session. The first subset of this pattern, $subset_1$, is set to $\{AB\}$ by calling procedure *contSubset* (shown in Figure 5). Because $\{AB\}$ is in L_2 , the **if** statement (from lines 10 to 14) is not executed. Then, the following **for** loop generates all candidate patterns whose first subset is $subset_1$. That is, the checking point of those continuous k -patterns focuses on the same position 1, where $3 \leq k \leq 7$. For the pattern $\{ABC\}$, the other subset, $subset_2$, is set to $\{BC\}$ (line 17). Because the pattern $\{BC\}$ is also in L_2 , the checked pattern $\{ABC\}$ is generated (line 20) and is inserted into UC by calling procedure *addCount* (shown in Figure 6). Up to this point, the pattern $\{ABC\}$ has passed the checking step. The next checked pattern is $\{ABCB\}$ as shown in Figure 10-(b), which has the same checking point as the pattern $\{ABC\}$ except the length is increased by one. Since the subsets with length equal to 2, *i.e.*, $\{AB\}$ and $\{BC\}$, have

UserID	The User Session
100	ABCBDEA

L_2
AB
BC
CB
BD

(a) (b)

Figure 9: An example: (a) a user session; (b) the frequent 2-patterns L_2 .

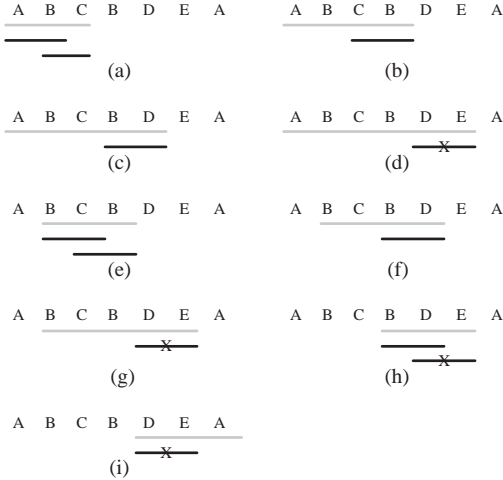


Figure 10: The checking process of the SpeedTracer*-II algorithm for the example shown in Figure 9: (a) $\{ABC\}$; (b) $\{ACB\}$; (c) $\{BCBD\}$; (d) $\{BCBDE\}$; (e) $\{BCB\}$; (f) $\{BCBD\}$; (g) $\{BCBDE\}$; (h) $\{CBD\}$; (i) $\{CBDE\}$; (j) $\{BDE\}$; (k) $\{DEA\}$.

been checked in the previous steps, we only have to check the subset $\{CB\}$ at this time. Since the subset $\{CB\}$ is in L_2 , pattern $\{ACB\}$ is inserted into UC .

Following the similar way, pattern $\{ABCBD\}$ passed the checking step. However, for the next pattern, $\{BCBDE\}$, because subset $\{DE\}$ is not in L_2 , the **for** loop is forced to be finished in line 19. Then, we move the checking point to the next position of the user session by increasing the value of i , which means the first subset of the checked pattern becomes $\{BC\}$ as shown in Figure 10-(e). Following the similar idea, we process the rest of this user session, and the generated candidate patterns are shown in Figure 11. After scanning all user sessions, the SpeedTracer*-III algorithm generates all candidate patterns and obtains their counts. The frequent patterns are determined according to the counts of the candi-

UC	count	step
ABC	1	(a)
ACB	1	(b)
ABCBD	1	(c)
BCB	1	(e)
BCBD	1	(f)
CBD	1	(h)

Figure 11: The result of the candidate patterns of the example shown in Figure 9

Table 2: Parameters

Parameter	Description
H	The average height of the traversal tree
$ D $	The number of user sessions (the size of the database)
P	The average length of user sessions

date patterns.

The parameter n determines the number of the times to scan the database. The times to scan the database will be $(n + 1)$.

4. Performance

In this Section, we study the performance of the proposed algorithms and make a comparison with the FS algorithm [2], the SpeedTracer algorithm [6], and the FDLP algorithm [7]. (Note that although the FS algorithm is an algorithm of type 1, it could also be used to mine non-simple traversal patterns if we do not apply the MF algorithm to convert user sessions first.)

4.1. Generation of Synthetic Data

In order to evaluate the performance of the proposed algorithms, we generated synthetic Web transactions based on [2]. These Web transactions were used to simulate the browsing scenarios of users on the WWW. There are two steps in this method: (1) construct a traversal tree which mimics the WWW structure; (2) generate user sessions among the traversal tree. The details of this method are described in [2]. The parameters used in this method are shown in Table 2.

4.2. Simulation Results

In the section, we discuss the simulation results of different synthetic data sets. Table 3 shows the names and parameter settings for each data set.

Table 3: Parameter values for synthetic data sets

Case	Name	H	P	$ D $	Size
1	H5_P10_D50K	5	10	50K	1.76MB
2	H8_P15_D2K	8	15	2K	126KB
3	H15_P20_D8K	15	20	8K	775KB

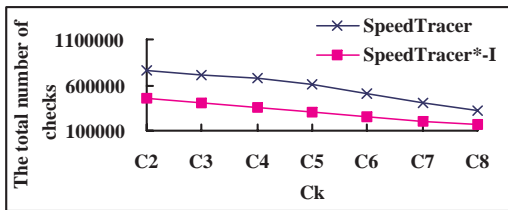


Figure 12: A comparison of the total number of checks with $minsup = 0.5\%$ (Case 1)

4.2.1. SpeedTracer vs. SpeedTracer*-I

When we choose the synthetic data set as H5_P10_D50K (Case 1), a comparison of the total number of checks in SpeedTracer and SpeedTracer*-I algorithms is shown in Figure 12. We could see that the SpeedTracer*-I algorithm will need smaller number of checks than the SpeedTracer algorithm. Therefore, the execution time of the SpeedTracer*-I algorithm is shorter than that of the SpeedTracer algorithm. A comparison of the execution time based on different values of the minimum support is shown in Figure 13.

Figure 14 shows how SpeedTracer and SpeedTracer*-I algorithms scale up as the number of user sessions increases from 2K to 8K, where the minimum support = 0.25%. We use the data set of H8.P15.D2000 (Case 2) as the basis, and extend it to the double size, the triple size, and the quadruple size. From Figure 14, we show that the execution time of the SpeedTracer*-I algorithm is always shorter than that of the SpeedTracer algorithm. We also show that our SpeedTracer*-I algorithm outperforms FS [2] and FDLP [7] algorithms in this case. The reason is that the FDLP algorithm generates too many candidate patterns. Although the FS algorithm utilizes the hash technique to reduce the number of candidate patterns, it needs to construct a very large hash table first. Our SpeedTracer*-I algorithm does not have these disadvantages and thus could provide better performance than FS and FDLP algorithms.

We use the synthetic data set H15_P20_D8K (Case 3) to compare the performance among SpeedTracer, SpeedTracer*-I, and FS algorithms again. Figure 15 shows this comparison based on different values of the minimum support. In this case, we do not consider the performance of the FDLP algorithm, because it needs a two-dimension array with size equal to 5G, which makes it impossible to be applied. From Figure 15, we show that the SpeedTracer*-I algorithm could provide better performance than SpeedTracer and

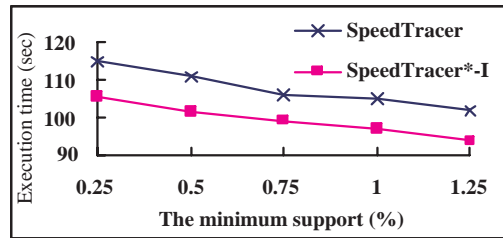


Figure 13: A comparison of the execution time (Case 1)

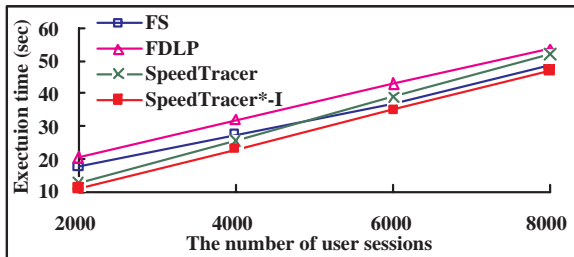


Figure 14: A comparison of the execution time (Case 2)

FS algorithms. The reason is the same as the one mentioned above.

4.2.2. The SpeedTracer*-II Algorithm

We use the synthetic data set of H8_P15_D2K (Case 2) to compare the performance of the SpeedTracer*-II algorithm with the SpeedTracer*-I algorithms. Figure 16 shows the comparison of the execution time based on different values of the minimum support. Figure 17 shows the comparison of the total number of candidate patterns. From Figures 16 and 17, we observe that the SpeedTracer*-I algorithm generates the smaller number of candidate patterns than the SpeedTracer*-II algorithm, while the latter needs the shorter execution time than the former. This is because the SpeedTracer*-II algorithm needs the less times to scan the database than the SpeedTracer*-I algorithms.

4.3. The Performance Result for the Real Web Log

In this section, we study the performance by using the real Web log. The Web log that we use here was generated by the Web server of "http://www.nsysu.edu.tw". There are 141799 user sessions and 2912 different Web pages contained in the Web log.

Figure 18 shows a comparison of execution time among FS, SpeedTracer, SpeedTracer*-I, and

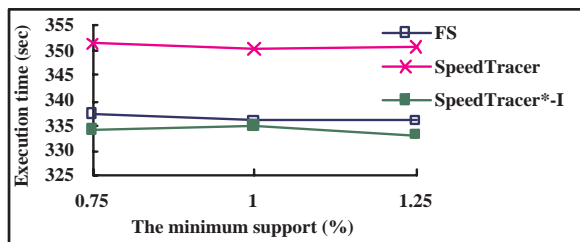


Figure 15: A comparison of the execution time (Case 3)

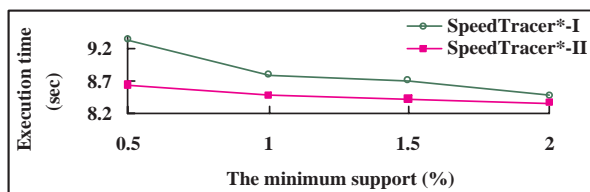


Figure 16: A comparison of the execution time (Case 2)

SpeedTracer*-II algorithms for the Web log. Since the FDLP algorithm needs the structure of the Web site which is not recorded in a real Web log, we do not consider the FDLP algorithm here. (Note that the structure of the Web site could be simulated in synthetic data.) From Figure 18, we show that the SpeedTracer*-I algorithm outperforms the SpeedTracer algorithm no matter what value the minimum support is. We also observe that the FS algorithm outperforms our SpeedTracer*-I algorithm. This is because the FS algorithm will trim the database after each database scanning. When the size of the database is large, this technique is very useful to reduce the I/O time for scanning the database. From Figure 18, we also show that our SpeedTracer*-II algorithm outperforms the FS algorithm. This is because the SpeedTracer*-II algorithm needs less times to scan the database than the FS algorithm.

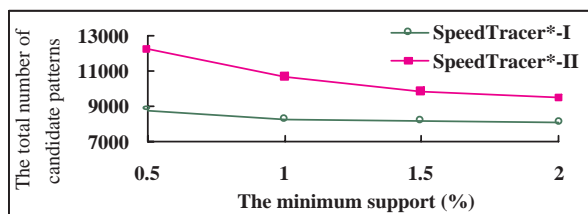


Figure 17: A comparison of the total number of candidate patterns (Case 2)

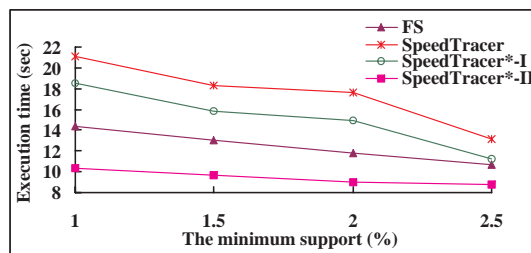


Figure 18: A comparison of the execution time for the real data

5. Conclusion

In this paper, we have proposed two algorithms for mining traversal patterns, SpeedTracer*-I and SpeedTracer*-II. For the SpeedTracer*-I algorithm, it improves the performance of the SpeedTracer algorithm by reducing the number of checks while generating candidate patterns. For the SpeedTracer*-II algorithm, it decreases the times to scan the database. From our simulation results and the experiments on the real Web log, we have shown that the SpeedTracer*-I algorithm could provide better performance than the SpeedTracer algorithm in terms of the processing time. Moreover, the SpeedTracer*-II algorithm outperforms SpeedTracer*-I and Apriori-like algorithms.

References

- [1] R. Agrawal and R. Srikant, "Fast Algorithms for Mining Association Rules," *Proc. of the 20th Int. Conf. on Very Large Data Bases*, pp. 487–499, 1994.
- [2] M. S. Chen, J. S. Park, and P. S. Yu, "Efficient Data Mining for Path Traversal Patterns," *IEEE Trans. on Knowledge and Data Eng.*, Vol. 10, No. 2, pp. 209–221, March/April 1998.
- [3] M. Eirinaki and M. Vazirgiannis, "Web Mining for Web Personalization," *ACM Trans. on Internet Technologies*, Vol. 3, No. 1, pp. 1–27, Feb. 2003.
- [4] Y. S. Lee, M. C. Hsieh, and S. J. Yen, "Efficient Approach for Interactively Mining Web Traversal Patterns," *Int. Conf. on Computational Science and its Applications*, pp. 1055–1065, 2005.
- [5] J. S. Park, M. S. Chen, and P. S. Yu, "Using A Hash-Based Method with Transaction Trimming for Mining Association Rules," *IEEE Trans. on Knowledge and Data Eng.*, Vol. 9, No. 5, pp. 813–825, Sept./Oct. 1997.
- [6] K. L. Wu, P. S. Yu, and A. Ballman, "SpeedTracer: A Web Usage Mining and Analysis Tool," *IBM Systems Journal*, Vol. 37, No. 1, pp. 89–105, Jan. 1998.
- [7] S. J. Yen, Y. S. Lee, and C. H. Hsu, "Mining Frequent Traversal Patterns in a Web Training Environment," *Proc. of National Computer Symp.*, pp. 105–116, 2001.