

# Fast Matching of Twig Patterns

Jiang Li and Junhu Wang

School of Information and Communication Technology  
Griffith University, Gold Coast, Australia  
Jiang.Li@student.griffith.edu.au, J.Wang@griffith.edu.au

**Abstract.** Twig pattern matching plays a crucial role in XML data processing. Existing twig pattern matching algorithms can be classified into two-phase algorithms and one-phase algorithms. While the two-phase algorithms (e.g., `TwigStack`) suffer from expensive merging cost, the one-phase algorithms (e.g., `TwigList`, `Twig2Stack`, `HolisticTwigStack`) either lack efficient filtering of useless elements, or use over-complicated data structures. In this paper, we present two novel one-phase holistic twig matching algorithms, `TwigMix` and `TwigFast`, which combine the efficient selection of useful elements (introduced in `TwigStack`) with the simple lists for storing final solutions (introduced in `TwigList`). `TwigMix` simply introduces the element selection function of `TwigStack` into `TwigList` to avoid manipulation of useless elements in the stack and lists. `TwigFast` further improves this by introducing some pointers in the lists to completely avoid the use of stacks. Our experiments show `TwigMix` significantly and consistently outperforms `TwigList` and `HolisticTwigStack` (up to several times faster), and `TwigFast` is up to two times faster than `TwigMix`.

## 1 Introduction

The importance of fast processing of XML data is well known. *Twig pattern matching*, which is to find all matchings of a query tree pattern in an XML data tree, lies in the center of all XML processing languages. Therefore, finding efficient algorithms for twig pattern matching is an important research problem.

Over the last few years, many algorithms have been proposed to perform twig pattern matching. Al-Khalifa et al [3] gave an algorithm which breaks a query tree into binary (parent-child and ancestor-descendant) relationships, finds solutions for them, and merges such partial solutions to get the final solutions. One problem of this approach is the large number of partial solutions and hence the high cost in the merging phase. To overcome this problem, Bruno et al [4] proposed a holistic twig join algorithm called `TwigStack`, which breaks the query tree into root-to-leaf paths, finds individual root-to-path solutions, and merges these partial solutions to get the final result. One vivid feature of `TwigStack` is the efficient filtering of useless partial solutions through the use of function `getNext()`. It is shown that when there are only `//`-edges, every root-to-leaf path solution returned by the algorithm will contribute to some final solutions. Later

on several improvements of `TwigStack` were made either to deal with `/`-edges (e.g., `TwigStackList` [9]), or to make use of index structures (e.g., `TSGeneric+` [7], `iTwigJoin` [6]). Chen et al [5] observed that the holistic two-phase algorithms still suffer from high merging costs, and they proposed a one-phase algorithm, `Twig2Stack`, which avoids the merging phase by storing final solutions in hierarchical stacks. It is claimed that `Twig2Stack` outperforms `TwigStack`. Qin et al [10] proposed another one-phase algorithm, `TwigList`, which uses a much simpler data structure, a set of lists, to store the final solutions. Due to the simpler data structure and hence the reduction in random memory access, `TwigList` achieves better performance than `Twig2Stack` [10]. `Twig2Stack` and `TwigList` can avoid the high cost of the merging phase, but they lose an important ability of the holistic approach, which is efficiently locating twig occurrences and discarding useless elements. More recently Jiang et al [8] proposed a one-phase holistic twig matching algorithm called `HolisticTwigStack`, which maintains the overall solutions in linked stacks. However, a considerable amount of time is taken to maintain the linked stacks.

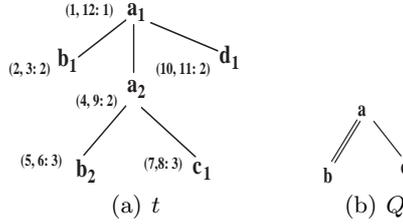
In this paper, we present two novel one-phase holistic twig matching algorithms, `TwigMix` and `TwigFast`, which combine the efficient selection of useful elements introduced in [4] with the simple data structure for storing final solutions introduced in [10]. `TwigMix` simply introduces the `getNext()` function of `TwigStack` into `TwigList` to avoid manipulation of useless elements in the stack and lists. `TwigFast` further improves this by introducing some pointers in the lists to completely avoid the use of stacks, based on the observation that the overhead of maintaining the pointers is generally negligible compared with the pushing/popping-up of elements into/from the stack. We conducted extensive experiments with both real and synthetic data. Our experiments show that (1) `TwigMix` significantly and consistently outperforms `TwigList` and `HolisticTwigStack` (up to several times faster), and `TwigFast` performs even better (up to two times faster) than `TwigMix`; (2) compared with `TwigList`, `TwigMix` saves an average of 75.93% of elements from being pushed into stack and an average of 70.19% of elements from being appended into the result lists. Since the result lists built by our algorithms are far shorter than those built by `TwigList`, our algorithms relieve the problem of memory consumption.

The rest of the paper is organized as follows. Section 2 provides background knowledge and recalls the major features of `TwigStack` and `TwigList`. `TwigMix` is presented in detail in Section 3. In Section 4, we present `TwigFast`. The experiment results are reported in Section 5. Finally, Section 6 concludes this paper.

## 2 Background

### 2.1 Terminology and notations

An XML document is modeled as a node-labeled tree, referred to as the *data tree*. A *twig pattern* is also a node-labeled tree, but it has two types of edges: `/`-edge and `//`-edges, which represent parent-child and ancestor-descendent relationships



**Fig. 1.** example data tree  $t$  and tree pattern  $Q$

respectively. The *twig matching* problem is to find all occurrences of the twig pattern in the data tree. Fig.1 shows a data tree  $t$  (where we use  $a_i$  to denote nodes labeled  $a$ , and so on) and a twig pattern  $Q$ . There is one occurrence of  $Q$  in  $t$ :  $(a_2, b_2, c_1)$ .

For data trees, we adopt the same region-based coding scheme used in **TwigStack**. Each node  $v$  is coded with a tuple of three values:  $(v.start, v.end: v.level)$ . Such a coding scheme has several useful properties: (1) ancestor-descendant and parent-child relationships can be identified in constant time:  $\forall v_1, v_2 \in Nodes(t)$ ,  $v_1$  is an ancestor of  $v_2$  iff  $v_1.start < v_2.start \leq v_2.end < v_1.end$ , and  $v_1$  is the parent of  $v_2$  iff it is the ancestor of  $v_2$ , and  $v_2.level - v_1.level = 1$ . (2)  $v_1, v_2$  do not have ancestor-descendant relationship, and  $v_1$  lies in a path to the left of the path where  $v_2$  lies iff  $v_1.end < v_2.start$  (See Fig. 1 (a)). These properties will be used extensively in our algorithms.

Below, we will use *elements* to refer to nodes in a data tree, and *nodes* to refer to nodes in a twig pattern. We will also use *x-child* (resp. *x-descendant*, *x-element*) to refer to a child (resp. descendant, element) labeled  $x$ . As in **TwigStack**, for each node  $n$ , there is a stream,  $T_n$ , consisting of all elements with the same label as  $n$  arranged in ascending order of their *start* values. Note that an element may appear in several streams if there are nodes with identical labels in  $Q$ . For each stream  $T_n$ , there exists a pointer  $PT_n$  pointing to the current element in  $T_n$ . The function  $Advance(T_n)$  moves the pointer  $PT_n$  to the next element in  $T_n$ . The function  $getElement(T_n)$  retrieves the current element of  $T_n$ . The function  $isEnd(T_n)$  judges whether  $PT_n$  points to the position after the last element in  $T_n$ . In addition, for node  $n$ , the functions  $isRoot(n)$  (resp.  $isLeaf(n)$ ) checks whether node  $n$  is the root (resp. leaf), and  $parent(n)$  (resp.  $children(n)$ ) returns the parent (resp. set of children) of  $n$ .

## 2.2 TwigStack and TwigList

To facilitate our explanation, we briefly recall the major features of **TwigStack** and **TwigList** here.

As mentioned earlier, **TwigStack** uses a function  $getNext(q)$  to efficiently filter useless elements. For self-containment, we copy the function into Algorithm 1. In the function,  $nextL(T_n)$  and  $nextR(T_n)$  return  $getElement(T_n).start$  and  $getElement(T_n).end$  respectively. The function has the following properties: if  $q$

---

**Algorithm 1** getNext( $q$ ) [4]

---

```
1: if (isLeaf( $q$ )) return  $q$ 
2: for  $q_i \in children(q)$  do
3:    $n_i = getNext(q_i)$ 
4:   if ( $n_i \neq q_i$ ) return  $n_i$ 
5:  $n_{min} = minarg_{n_i} nextL(T_{n_i})$ 
6:  $n_{max} = maxarg_{n_i} nextL(T_{n_i})$ 
7: while ( $nextR(T_q) < nextL(T_{n_{max}})$ ) do
8:   Advance( $T_q$ )
9: if ( $nextL(T_q) < nextL(T_{n_{min}})$ ) return  $q$  else return  $n_{min}$ 
```

---

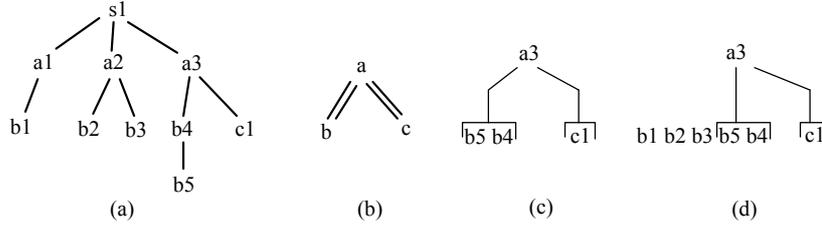
is  $root(Q)$  (the root of  $Q$ ), then  $getNext(q)$  always returns a node  $n$  that has a *minimal descendant extension* [4], i.e., (1) for each child  $n'$  of  $n$ , the current element of  $T_n$  has a descendant which is the current element of  $T_{n'}$ , and each child of  $n$  recursively has this property; (2) the current element of  $n$  has the minimum *start* value among all nodes that have property (1). The function also moves the pointer  $PT(T_{n_i})$  when the current element in  $T_{n_i}$  no longer has descendants in  $T_{n_j}$ , for some of child  $n_j$  of  $n_i$  (lines 7,8).

**TwigList** is based on the following observation [10]: for each  $a$ -element  $v$ , its  $b$ -descendants can be arranged in a minimal interval, such that every  $b$ -descendant of  $v$  falls into this interval, and  $b$ -elements that are not descendants of  $v$  do not fall into the interval. As a consequence, we can use a pair of position values,  $v_{start_b}$  and  $v_{end_b}$ , to specify the interval for all  $b$ -descendants of  $v$ . For example, for the data tree shown in Fig.2 (a), all descendants of the  $a$ -nodes can be arranged in a list  $b_1, b_2, b_3, b_5, b_4$ , and  $a1_{start_b} = a1_{end_b} = 1$ ,  $a2_{start_b} = 2$ ,  $a2_{end_b} = 3$ ,  $a3_{start_b} = 4$  and  $a3_{end_b} = 5$  will tell us the  $b$ -descendants of each  $a$ -element. The data structure used in **TwigList** is thus a set of lists, one list,  $L_n$ , for each node  $n$  in  $Q$ . Each element  $v$  in  $L_n$  has pairs of start and end pointers pointing to the start and end positions of descendant intervals (one interval for each child of  $n$ ). These lists are used to store the final solutions. For instance, for the data tree and query in fig.2 (a),(b), the lists built by **TwigList** are shown in fig.2 (d). In the figure,  $a1$ ,  $a2$  are not put into list  $L_a$  because they do not have  $c$ -descendants. The main algorithm of **TwigList** is a procedure to construct the lists, once this is done, it uses another procedure **TwigList-Enumerate** to efficiently enumerate the final solutions. To construct the lists, **TwigList** uses a stack,  $S$ . Elements are pushed into the stack in pre-order, and  $top(S)$  is popped up when a non-descendant of  $top(S)$  arrives, and it is then checked to see whether it should be appended to the corresponding list.

### 3 TwigMix: introducing efficient element filtering into TwigList

#### 3.1 Overview of TwigMix

We explain the basic ideas used in **TwigMix** using the example in Fig. 2. **TwigMix** uses the same data structure as **TwigList**, but it introduces the  $getNext()$  func-



**Fig. 2.** An example to explain the basic ideas of TwigMix

tion to avoid pushing useless elements into the stack  $S$  and appending useless elements into the lists. In Fig. 2, if we apply the `TwigList` algorithm, all of the elements will be pushed into  $S$ . When the elements are popped up from the stack, the algorithm will determine whether to append them to the result lists. For this example,  $a1$  and  $a2$  are not appended to the result lists because they can not find their  $c$ -descendants. However,  $b1, b2$  and  $b3$  are still appended to the result list although they do not contribute to the final solutions. Fig. 2 (d) shows the structure of the final lists constructed by `TwigList`. For `TwigMix`, due to the introduction of `getNext()`,  $a1$  and  $a2$  can be directly abandoned and will not be pushed into  $S$ . The elements  $b1, b2, b3$  will not be pushed into  $S$  either because they can not find their ancestors in  $S$ . The final result lists are shown in Fig. 2 (c). Therefore, `TwigMix` does not waste time in pushing/popping-up  $b1, b2$ , and  $b3$  into/from stack and appending them to result list  $L_b$ . It also saves memory because  $b1, b2$  and  $b3$  do not need to be stored in the lists. If the data tree is large, the savings of time and space will be quite significant (see Section 5 for examples).

### 3.2 TwigMix

`TwigMix` differs from `TwigList` in its way of constructing the final result lists. Once the lists are constructed, it uses the same procedure `TwigList-Enumerate` in [10] to enumerate all final solutions.

Our new algorithm for building the result lists, `TwigMix-Construct`, is shown in Algorithm 2. Like `TwigList`, we use a stack  $S$  to achieve bottom-up processing of elements. For each node  $n_i \in Nodes(Q)$ , we use a counter  $n_i.counter$  to record the number of elements in stack  $S$  for that query node. In Algorithm 2, after initialization, the function `getNext(q)` is repeatedly called (lines 3,4) to get the query node which has a minimal descendant extension (see Section 2.2). The loop will stop until there are no elements not processed for any of the leaf nodes (see the `end(q)` function). Line 7 is particularly important. If the returned query node  $n_{act}$  is the root, its current element is directly pushed into the stack  $S$ . However, if it is not the root, the counter of  $parent(n_{act})$  is checked to see whether any elements of  $parent(n_{act})$  are in the stack. We push the current element of  $T_{n_{act}}$  into  $S$  only when there are elements of  $parent(n_{act})$  in  $S$  (this is why the elements  $b1, b2$  and  $b3$  in Fig.2 (a) are not pushed into stack). The

---

**Algorithm 2** TwigMix-Construct( $Q$ )

---

```
1: initialize stack  $S$  as empty;
2: initialize the list  $L_{n_j}$  as empty,  $n_j.counter$  as 0, for all nodes  $n_j \in Nodes(Q)$ ;
3: while  $\neg end(Q)$  do
4:    $n_{act} = getNext(root(Q))$ 
5:    $v_{act} = getElement(n_{act})$  //  $region(v)$  denotes the interval  $(v.start, v.end)$ 
6:    $toList(S, region(v_{act}))$ 
7:   if  $isRoot(n_{act})$  OR  $parent(n_{act}).counter > 0$  then
8:     for  $n_k \in children_{n_{act}}$  do
9:        $v_{act}.start_{n_k} = length(L_{n_k}) + 1$ 
10:     $push(S, v_{act})$ 
11:     $n_{act}.counter ++$ 
12:     $Advance(T_{n_{act}})$ 
13:   $toList(S, (\infty, \infty))$ 
14: procedure  $END(q)$ 
15:   return  $\forall n_i \in Nodes(q) : isLeaf(n_i) \Rightarrow isEnd(T_{n_i})$ 
16: procedure  $toList(S, r)$ 
17:   while  $S \neq \emptyset$  AND  $r \notin reg(top(S))$  do
18:      $v_j = pop(S)$ 
19:     let  $v_j$ 's type be  $n_j$  // the type  $n_j$  is memorized when  $v_j$  is pushed into  $S$ 
20:      $n_j.counter --$ 
21:     for  $n_k \in children_{n_{act}}$  do
22:        $v_j.end_{n_k} = length(L_{n_k})$ 
23:     append  $v_j$  into list  $L_{n_j}$ 
```

---

counters are maintained at line 11 and line 20, when an element is pushed into or popped up from  $S$ . When an element is pushed into  $S$ , the start positions of its descendant intervals are set (lines 8,9). In the sub-procedure  $toList(S, r)$ , we check whether the current element in the node returned by  $getNext(root(Q))$  is a descendant of  $top(S)$ , if not, we pop up  $top(S)$ , set the end positions of its descendant intervals, and append it directly to the corresponding list. Note that, unlike the procedure in **TwigList**, we do not need to check whether  $top(S)$  can be appended to list because all elements pushed into the stack are guaranteed to appear in some final solution (provided  $Q$  has no  $/$ -edges). At the end of the algorithm, we apply an infinite interval to  $toList$  in order to pop up all elements from  $S$ .

*Example 1.* Consider the twig pattern and the data tree in Fig. 2. Initially, the current elements of the query nodes are  $(a1, b1, c1)$ . All the first three calls of  $getNext(a)$  return node  $b$ . Because the counter of  $b$ 's parent  $a$  is 0, the elements  $b1, b2, b3$  are not pushed into the stack  $S$ . The fourth call of  $getNext(a)$  returns node  $a$ . Node  $a$  is the root of the query tree, so  $a3$  is directly pushed into  $S$  and the start positions of its descendant intervals are recorded. The counter of node  $a$  increases by 1. The next two calls of  $getNext(a)$  return node  $b$ . Because the counter of node  $a$  is 1, the elements  $b5, b4$  are pushed into the  $S$  stack. Next,  $getNext(a)$  returns node  $c$ . The coming of  $c1$  results in  $b5$  and  $b4$  being popped up and appended to  $L_b$ . Finally, the range  $(\infty, \infty)$  makes  $c1$  and  $a3$  pop up and they are appended to  $L_c$  and  $L_a$  respectively. When  $a3$  is appended to  $L_a$ , the end positions of its descendant intervals are recorded.

### 3.3 Analysis of TwigMix

In this section, we show the correctness of `TwigMix` and analyze its time and space complexity. We prove the following lemma first.

**Lemma 1.** *Suppose  $Q$  has no  $/$ -edges. `TwigMix` pushes an element into stack  $S$  iff the element contributes to some final solutions.*

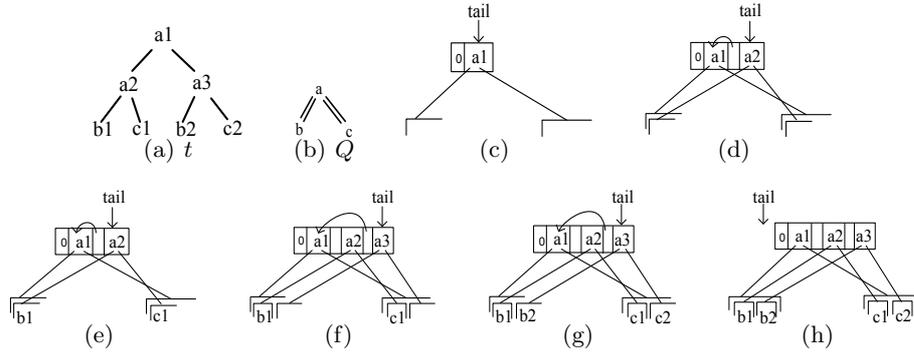
**Proof [sketch]** (only if) If  $getNext(root(Q))$  returns  $q_{act}$ , then  $q_{act}$  has a minimal descendant extension (see Section 2.2). Therefore, the current element  $v_{act}$  of  $T_{q_{act}}$  (line 5) has a descendant in  $T_{n_i}$  for each child  $n_i$  of  $q_{act}$ . Line 7 and line 10 make sure that only if  $q_N$  is  $root(Q)$  or  $S$  contains an element of type  $parent(q_{act})$  do we push  $v_{act}$  into  $S$ . In both cases,  $v_{act}$  participates in at least one final solution, since we assume there are only  $/$ -edges in  $Q$ .

(if) If an element  $v$  of type  $n$  participates in some final solution,  $getNext(root(Q))$  will return  $n$  when the current element of  $T_n$  is  $v$ . If  $n$  is the root,  $v$  will be pushed into  $S$  directly. Otherwise, the stack  $S$  will contain at least one element of type  $parent(n)$  when  $v$  is returned in line 5, because elements are pushed into stack in pre-order, and an element will be popped up from  $S$  after its descendants have been popped-up (line 17-18). Hence  $v$  will also be pushed into  $S$ .  $\perp$

**Theorem 1.** *Given a twig pattern (that has  $/$ -edges only) and an XML data tree, `TwigMix` correctly builds up the final result lists.*

**Proof [sketch]** We only need to show that (1) elements contributing to final solutions will be appended to the lists, and (2) for each element in the list, its descendant intervals are correctly set. (1) is true because of Lemma 1 and the fact that every element pushed in the stack  $S$  is appended in the result list. (2) is true because, for any element  $v_{act}$  satisfying the condition in line 7, it is pushed into  $S$  before any of its descendants are pushed into  $S$ . Therefore, the lists of the children nodes of  $n_{act}$  has no descendants of  $v_{act}$  before  $v_{act}$  is pushed into  $S$  at line 10. However, after  $v_{act}$  is pushed into  $S$ , the next element in  $T_{n_i}$  pushed into  $S$  for any child  $n_i$  of  $n$  must be a descendant of  $v_{act}$ . Therefore, line 9 correctly sets the start positions of the descendant intervals for  $v_{act}$ . Furthermore,  $v_j$  is popped up from  $S$  only when all of its descendants have been popped up and appended to lists. Therefore, line 21-22 correctly sets the end positions of  $v_j$ 's descendant intervals.  $\perp$

**Complexity analysis** Algorithm 2 scans each stream  $T_n$  from start to end once, through the functions  $getNext()$  and  $Advance(T_{n_{act}})$  at line 12. For each element in  $T_n$  it may push it into stack, pop it up from stack, append it to list, and set its start and end positions for its descendants. Suppose  $d$  is the maximum degree of nodes in  $Q$ . For each element appended to result lists, at most  $d$  intervals need to be recorded and recording an interval needs constant time. Pushing/popping-up an element into/from the stack  $S$  can be finished in constant time. Therefore, the worst-case time complexity is  $O(d \cdot N)$  ( $N$  is the sum of the sizes of the input streams), which is linear in  $N$ . The worst-case space



**Fig. 3.** An example to illustrate the basic ideas of **TwigFast**

complexity is linear in the sum of the sizes of the occurrences of the twig pattern (the sum of the sizes of the final lists).

**Considerations of  $\backslash$ -edges**  $getNext(q)$  does not guarantee the returned node can be expanded to a solution when  $\backslash$ -edges exist. Therefore, Algorithm 2 does not guarantee all of the elements moved into the stack  $S$  and result lists will appear in final solutions when  $\backslash$ -edges exist. To make sure the final results enumerated are still correct, we need to modify the enumeration algorithm so that it checks the satisfaction of parent-child relationship, for  $\backslash$ -edges, when outputting final solutions.

To improve the efficiency of enumeration, one can use the strategy of adding sibling links as in [10]. This strategy can not prevent useless elements from being pushed into the stack  $S$  and appended into the result lists. To reduce the manipulation of useless elements, we can incorporate the  $getNext(q)$  function of algorithms that try to reduce the useless intermediate path solutions when  $\backslash$ -edges exist (e.g. **TwigStackList** [9], **iTwigJoin** [6], etc). However, these algorithms may result in the elements of the query nodes returned by  $getNext(q)$  are not in *pre-order*. Therefore, **TwigMix-Construct** needs to be adjusted.

## 4 TwigFast: avoiding manipulation of elements in stacks

### 4.1 Limitations of TwigMix

**TwigMix** integrates the holistic approach into **TwigList**, so only potentially useful elements are pushed into stack  $S$  and result lists. The time taken by pushing/popping-up elements into/from stack will become significant for large data trees. In order to get a glimpse of the number of elements that pass through  $S$ , we implemented **TwigMix** and did some experiments over the DBLP data set. The selected queries are listed in Table 1. As shown in the table, for all three queries, the number of elements pushed into  $S$  is very large. Therefore, if we can directly build up the final lists without using the stack, the performance can be significantly improved.

---

**Algorithm 3** TwigFast(Q)

---

```
1: initialize the list  $L_{n_i}$  as empty, and set  $n_i.tail = 0$ , for all  $n_i \in Nodes(Q)$ ;  
2: while  $\neg end(Q)$  do  
3:    $n_{act} = getNext(root(Q))$   
4:    $v_{act} = getElement(n_{act})$   
5:   if  $\neg isRoot(n_{act})$  then  
6:      $SetEndPointers(parent(n_{act}), v_{act}.start)$   
7:   if  $isRoot(n_{act}) \vee parent(n_{act}).tail \neq 0$  then  
8:     if  $\neg isLeaf(n_{act})$  then  
9:        $SetEndPointers(n_{act}, v_{act}.start)$   
10:      for  $n_k \in children(n_{act})$  do  
11:         $v_{act}.start_{n_k} = length(L_{n_k}) + 1$   
12:         $v_{act}.cancestor = n_{act}.tail$   
13:         $n_{act}.tail = length(L_{n_{act}}) + 1$   
14:      append  $v_{act}$  into list  $L_{n_{act}}$   
15:       $Advance(T_{n_{act}})$   
16:  $SetRestEndPointers(Q, \infty)$   
  
17: procedure  $SETENDPOINTERS(n, actL)$   
18:   while  $n.tail \neq 0$  do  
19:      $v_n = element(n.tail)$   
20:     if  $v_n.end < actL$  then  
21:       for  $n_k \in children(n)$  do  
22:          $v_n.end_{n_k} = length(L_{n_k})$   
23:        $n.tail = v_n.cancestor$   
24:     else  
25:       break  
  
26: procedure  $SETRestEndPointers(n, actL)$   
27:   if  $\neg isLeaf(n)$  then  
28:      $SetEndPointers(n, actL)$   
29:   for  $q_i$  in  $children(n)$  do  
30:      $SetRestEndPointers(n_i, actL)$ 
```

---

Query	Number of elements pushed into S
//dblp//inproceedings[/title//author	915,856
//dblp//article[/author][/title//year	553,062
//dblp//inproceedings[/cite][/title//author	149,015

**Table 1.** Limitation of TwigMix

## 4.2 TwigFast

**TwigFast** uses a data structure that is essentially the same as that of **TwigMix**, but to avoid the use of stack  $S$ , it adds some pointers in the lists. More specifically, each element appended to the result list has a pointer, *cancestor*, that points to its closest ancestor in the same list. With these pointers, the elements on the same path can be linked together. For example, in Fig. 3(f), the element  $a3$  has a pointer pointing to its closest ancestor  $a1$ . For each result list, a *tail* pointer is also maintained to point to the last element that still has potential descendants in the future. Together with the pointers that point to closest ancestors, we can easily maintain a list of elements which still have potential descendants, and these elements must be on the same path. For example, in Fig.3(f), with the pointers, we can easily find  $a3$  and  $a1$  still have potential de-

scendants, but  $a2$  will not contribute to any new solutions in the future, so it is skipped by the pointer.

The purpose of the *cancesor* and *tail* pointers is to make it possible to correctly set descendant intervals for each element. When an element  $e$  is about to be appended to  $L_E$ , the start positions of intervals are determined (line 10 to 11). For each child  $C_i$  of query node  $E$ , the start position is equal to  $length(L_{C_i}) + 1$ . The end positions of an element can be determined when the element will not have any new descendants coming in the future (line 9). For each child  $C_i$  of query node  $E$ , the end position is equal to  $length(L_{C_i})$ . For example in Fig.3(f), the coming of  $a3$  indicates  $a2$  will not have any new descendants in the future, so the end positions of  $a2$  are determined.

*Example 2.* Consider the data tree and twig pattern shown in Fig.3. The first call of  $getNext()$  returns  $a$ , with  $a1$  being the current element ( $v_{act}$ ) of  $T_a$ . Since  $a$  is the root of  $Q$ , and  $a$  is not the leaf, the procedure  $SetEndPointers(a, v_{act}.start)$  is called but it does nothing since  $a.tail = 0$ . Now the start positions of  $a1$ 's descendant intervals are set to 1, and  $a1.cancestor = 0$ ,  $a.tail = 1$ , and  $a1$  is appended to list  $L_a$ , and current element of  $T_a$  is set to  $a2$  (Fig.3 (c)). The second call of  $getNext$  also returns  $a$ , and  $SetEndPointers(a, a2.start)$  is called. Since  $a.tail \neq 0$ , and  $a1.end \geq a2.start$  (i.e.,  $a2$  is a descendant of  $a1$ ), the procedure finishes with nothing done. Now lines 10 to 15 sets the start positions of  $a2$ 's descendant intervals as 1, and  $a2.cancestor = 1$ ,  $a.tail = 2$ , appends  $a2$  to  $L_a$  ((Fig.3 (d))), and advances  $T_a$  to  $a3$ . The next call of  $getNext()$  returns  $b$ , which is a leaf node. The current element of  $T_b$  is  $b1$ . Therefore,  $SetEndPointers(a, b1.start)$  is called. Since  $a.tail \neq 0$ , and  $a2.end > b1.start$ , the procedure returns to line 7. Since  $a.tail > 0$ ,  $b_1$  is appended to  $L_b$  (line 14), and  $PT(T_b)$  points to  $b2$ . Similarly, the next call of  $getNext()$  returns  $c$ , and we append  $c1$  to  $L_c$ , and make  $PT(T_c)$  point to  $c2$  (Fig.3 (e)). The next call of  $getNext()$  returns  $a$  with  $v_{act} = a3$ .  $SetEndPointers(a, a3.start)$  is called. Since  $a2.end < a3.start$ , i.e.,  $a2$  no longer has  $b$ -descendants or  $c$ -descendants, we set the end positions of  $a2$  as 1 and 1 for  $b$  and  $c$ . We then set the start positions of  $a3$  as 2 and 2,  $a3.cancestor = 1$  (pointing to  $a1$ ),  $a.tail = 3$ , append  $a3$  to  $L_a$  and advance  $T_a$  (Fig.3 (f)). The next two calls of  $getNext()$  return  $b$  and  $c$  respectively, so we append  $b_2$  and  $c_2$  to  $L_b$  and  $L_c$  respectively, and advance  $T_b$  and  $T_c$  (Fig.3 (f)). Now we use the infinite value to set the remaining end positions. That is, the end positions of  $a3$  to 2. The final lists are shown in Fig.3 (h).

**Correctness and complexity** Both the correctness of **TwigFast** and the linear time and space complexity of Algorithm 3 can be established, in a way similar to **TwigMix**.

**Considerations of /-edges** For **TwigFast**, the strategy of adding sibling links [10] can also be applied. But one thing should be noted. **TwigFast** directly builds up the final solutions into result lists, so ancestors are always appended

to result lists before their descendants. Therefore, when we set end pointers for an element, if it can not find its children, it should be marked as useless. The enumeration algorithm will skip this element.

## 5 Experiments

In this section, we present the experiment results on the performance of `TwigMix` and `TwigFast` against `TwigList` [10] and `HolisticTwigStack` [8], with both real-world and synthetic data sets. `TwigList` is the most up-to-date one-phase twig pattern matching algorithm that applies the bottom-up approach. It is claimed to significantly outperform `Twig2Stack` [5] which, in turn, is claimed to be faster than `TwigStack`. `HolisticTwigStack` is also a one-phase holistic twig pattern matching algorithm, but the data structure used is complicated and expensive to maintain.

The algorithms are evaluated with the following metrics: (1) number of elements pushed into the S stack and result lists, (2) processing time.

### 5.1 Experiment set-up

The XML document parser we used is `Libxml2` [2]. We implemented a generator in C to generate element encodings (*start, end, level*) for each element in an XML document. A simple XPath parser is also implemented, which generates the twig tree from an XPath expression.

We implemented `TwigMix`, `TwigFast`, `TwigList` and `HolisticTwigStack` in C++. All the experiments were performed on 1.6GHz Intel Centrino Duo processor with 1G RAM. The operating system is Windows XP. We used the following three data sets for evaluation:

**TreeBank:** We obtained TreeBank XML document from the University of Washington XML repository [1]. The data is deep and has many recursive elements with the same label. The maximal depth is 36 and there are more than 240 million elements.

**DBLP:** DBLP XML document is also obtained from the University of Washington XML repository [1]. This data set is wide and shallow. There are more than 330 million elements.

**XMark:** XMark is a synthetic data set, which is generated by the XML Benchmark Project [11]. We set the scaling factor as 2. The generated document is 226M with more than 333 million elements.

### 5.2 Experiment results

We compared the algorithms `TwigMix`, `TwigFast` against `TwigList` and `HolisticTwigStack` with different twig pattern queries over the three data sets above. The queries are listed in Table 2.

Data set	Query	XPath expression
TreeBank	TQ1	//S[//MD]//ADJ
TreeBank	TQ2	//S[//VP//IN]//NP
TreeBank	TQ3	//S//VP//PP[//NP//VBN]//IN
TreeBank	TQ4	//S//VP//PP[//NN][//NP[//CD]//VBN]//IN
TreeBank	TQ5	//S[//VP][//NP]//VP//PP[//IN]//NP//VBN
DBLP	DQ1	//dblp//inproceedings[//title]//author
DBLP	DQ2	//dblp//article[//author][//title]//year
DBLP	DQ3	//dblp//inproceedings[//cite][//title]//author
DBLP	DQ4	//dblp//article[//author][//url]//ee
DBLP	DQ5	//article[//volume][//cite]//journal
XMark	XQ1	//item[//location]//description//keyword
XMark	XQ2	//people//person[//address//zipcode]//profile//education
XMark	XQ3	//item[//location][//mailbox//mail//emph]//description//keyword
XMark	XQ4	//open.auction[//parlist]//bidder
XMark	XQ5	//people//person[//address//zipcode]//profile

**Table 2.** Queries over TreeBank, DBLP and XMark

**Number of elements moved to  $S$  and result lists** We compared the number of elements pushed into stack  $S$  and appended to the result lists during processing. **TwigFast** and **HolisticTwigStack** do not push an element into the stack  $S$ , so we compared **TwigMix** with **TwigList**. The comparison results are presented in Table 3 and 4. Apart from the number of elements, we also calculated the reduction percentage made by **TwigMix**.

As shown in the tables, **TwigMix** reduces a large percentage (up to 99.9%) of elements moved to stack  $S$  and result lists. In some queries, the number of elements reduced is over 1 million. Even though one operation on stack or list is minor, such a large percentage of reduction is enough to significantly reduce the overall time. Additionally, the reduction is significant over all of the three data sets regardless of the structural characteristics of the data, which means the performance improvements brought by **TwigMix** are consistent.

The reduction of elements appended to result lists shows the advantage of **TwigMix** in *memory consumption*. Since the elements appended to result lists will not be released until the results enumeration finishes, they will waste memory space if they do not contribute to the final solutions. Therefore, the useless elements eliminated by **TwigMix** can significantly reduce the usage of memory.

**Processing time** The comparison of processing time is illustrated in Fig.4. As shown, both **TwigMix** and **TwigFast** significantly outperform **TwigList** and **HolisticTwigStack**. **TwigFast** shows better performance than **TwigMix** because it does not need to push elements into stack. This demonstrates that the overhead of maintaining the *ancestor* and *tail* pointers in **TwigFast** is well worthwhile. If we observe the figure together with Table 3 and Table 4, we can see that the processing time is closely related to the number of elements moved to  $S$  and result lists. In other words, the reduction of elements for processing directly brings the improvement of performance. For example, for query TQ4, the percentage of reduction is up to 99.1% such that the gap of processing time is huge. For query DQ1, against **TwigMix**, **TwigFast** saves 915,856 elements from being pushed into the stack, so the processing time nearly decreases by 2 times.

Query	TwigMix Elements	TwigList Elements	Reduction percentage	Useful Elements
TQ1	34	166,940	99.9%	34
TQ2	608,683	883,479	31.1%	608,683
TQ3	40,058	1,047,564	96.1%	40,058
TQ4	11,728	1,283,194	99.1%	11,728
TQ5	64,745	1,637,551	96.0%	64,745
DQ1	915,856	1,257,621	27.2%	915,856
DQ2	553,062	1,485,788	62.8%	553,062
DQ3	149,015	1,428,692	89.6%	149,015
DQ4	126,490	1,270,476	90.0%	126,490
DQ5	52,783	508,499	89.6%	52,783
XQ1	124,066	316,594	60.8%	124,066
XQ2	31,861	140,254	77.3%	31,861
XQ3	63,124	541,558	88.3%	63,124
XQ4	52,941	184,874	71.4%	52,941
XQ5	51,325	127,410	59.7%	51,325

Table 3. Number of elements pushed into S

Query	TwigMix Elements	TwigList Elements	Reduction percentage	Useful Elements
TQ1	34	13,686	99.8%	34
TQ2	608,683	770,052	21.0%	608,683
TQ3	40,058	207,930	80.7%	40,058
TQ4	11,728	414,380	97.2%	11,728
TQ5	64,745	797,917	91.9%	64,745
DQ1	915,856	1,257,384	27.2%	915,856
DQ2	553,062	1,484,711	62.7%	553,062
DQ3	149,015	1,222,789	87.8%	149,015
DQ4	126,490	1,183,417	89.3%	126,490
DQ5	52,783	398,708	86.8%	52,783
XQ1	124,066	255,278	51.4%	124,066
XQ2	31,861	82,829	61.5%	31,861
XQ3	63,124	410,540	84.6%	63,124
XQ4	52,941	167,433	68.4%	52,941
XQ5	51,325	89,241	42.5%	51,325

Table 4. Number of elements appended to result lists

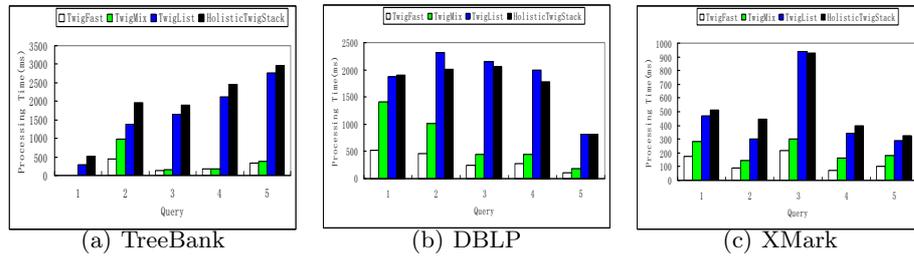


Fig. 4. Processing Time(ms)

## 6 Conclusion

We presented two novel one-phase twig pattern matching algorithms that efficiently find twig pattern occurrences. **TwigMix** introduces holistic ideas into the original bottom-up approach, such that the elements that do not contribute to final solutions are not moved into the stack and result lists. **TwigFast** directly builds up final solutions without pushing/popping-up elements into/from the stack. The better overall performance of our algorithms has been substantiated in our experiments. Since the result lists built by our algorithms are far shorter than those built by **TwigList**, our algorithms relieve the problem of memory consumption.

**Acknowledgement** This work is partially supported by Griffith University New Researcher’s Grant (GUNRG36621).

## References

1. <http://www.cs.washington.edu/research/xmldatasets/>.
2. <http://xmlsoft.org/>.
3. S. Al-Khalifa, H. V. Jagadish, J. M. Patel, Y. Wu, N. Koudas, and D. Srivastava. Structural joins: A primitive for efficient XML query pattern matching. In *ICDE*, pages 141–, 2002.
4. N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: optimal XML pattern matching. In *SIGMOD Conference*, pages 310–321, 2002.
5. S. Chen, H.-G. Li, J. Tatemura, W.-P. Hsiung, D. Agrawal, and K. S. Candan. Twig<sup>2</sup>stack: Bottom-up processing of generalized-tree-pattern queries over XML documents. In *VLDB*, pages 283–294, 2006.
6. T. Chen, J. Lu, and T. W. Ling. On boosting holism in XML twig pattern matching using structural indexing techniques. In *SIGMOD Conference*, pages 455–466, 2005.
7. H. Jiang, W. Wang, H. Lu, and J. X. Yu. Holistic twig joins on indexed XML documents. In *VLDB*, pages 273–284, 2003.
8. Z. Jiang, C. Luo, W.-C. Hou, Q. Zhu, and D. Che. Efficient processing of XML twig pattern: A novel one-phase holistic solution. In *DEXA*, pages 87–97, 2007.
9. J. Lu, T. Chen, and T. W. Ling. Efficient processing of XML twig patterns with parent child edges: a look-ahead approach. In *CIKM*, pages 533–542, 2004.
10. L. Qin, J. X. Yu, and B. Ding. *TwigList* : Make twig pattern matching fast. In *DASFAA*, pages 850–862, 2007.
11. A. Schmidt, F. Waas, M. Kersten, D. Florescu, I. Manolescu, M. Carey, and R. Busse. The XML benchmark project. Technical Report INS-R0103, CWI, April 2001.