

Twig Pattern Matching: A Revisit

Jiang Li¹, Junhu Wang¹, and Maolin Huang²

¹ School of Information and Communication Technology
Griffith University, Gold Coast, Australia

Jiang.Li@griffithuni.edu.au, J.Wang@griffith.edu.au

² Faculty of Engineering and Information Technology
The University of Technology, Sydney, Australia
maolin@it.uts.edu.au

Abstract. Twig pattern matching plays a crucial role in XML query processing. In order to reduce the processing time, some existing holistic one-phase twig pattern matching algorithms (e.g., `HolisticTwigStack` [3], `TwigFast` [5], etc) use the core function `getNext` of `TwigStack` [2] to effectively and efficiently filter out the useless elements. However, using `getNext` as a filter may incur other redundant computation. We propose two approaches, namely *re-test checking* and *forward-to-end*, which can avoid the redundant computation and can be easily applied to both holistic one-phase and two-phase algorithms. The experiments show that our approach can significantly improve the efficiency by avoiding the redundant computation.

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. Bruno et al [2] proposed a two-phase 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`. Later on several one-phase holistic algorithms (e.g., [3], [5]) also use `getNext` to filter out useless elements. Using `getNext` as a filter can efficiently discard useless elements. However, `getNext` may incur other redundant computation. Li et al [4] try to resolve the redundant computation and propose `TJEssential`, but their approach involves much overhead and can not avoid some important types of redundant computation.

In this paper, we propose a different approach to avoid redundant computation, and this approach imposes less overheads and can be easily applied to both holistic one-phase and two-phase twig pattern matching algorithms that

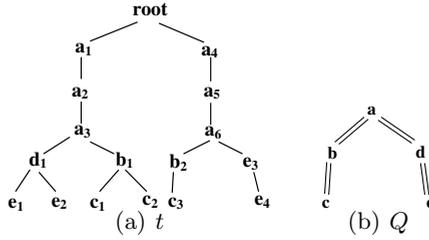


Fig. 1. Example data tree t and tree pattern Q

are based on `TwigStack`. We present the algorithms `TwigFast*` and `TwigStack*` which extend `TwigFast` and `TwigStack` respectively by applying our proposed approach.

The rest of the paper is organized as follows. Section 2 provides background knowledge and recalls the major features of `getNext` and `TwigFast`. Redundant computation in `getNext` is explained in Section 3. Our approach for resolving redundant computation is presented in Section 4. The experiment results are reported in Section 5. Finally, Section 6 concludes this paper.

2 Terminology and Notation

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 respectively. The *twig matching* problem is to find all occurrences of the twig pattern in the data tree. For data trees, we adopt the region coding scheme [2]. Each node v is coded with a tuple of three values: $(v.start, v.end, v.level)$.

Below, we will use *elements* to refer to nodes in a data tree, and *nodes* to refer to nodes in a twig pattern. 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. 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 . $ancestors(n)$ (resp. $descendants(n)$) returns the set of ancestors (resp. set of descendants) of n .

3 Deficiencies in Previous Algorithms

Many previous twig pattern matching algorithms use `getNext` of `TwigStack` to filter out useless elements. However, `getNext` may bring other redundant computation. In this section, we explain where the redundant computation comes

Table 1. Example of redundant calls of *getNext*

Step	getNext(a)	getNext(b)	getNext(c)	getNext(d)	getNext(e)
1	a(a_1)	b	c	d	e
2	a(a_2)	b	c	d	e
3	a(a_3)	b	c	d	e
4	d(d_1)	b	c	d	e
5	e(e_1)	b	c	e	e
6	e(e_2)	b	c	e	e
7	e(e_3)	b	c	e	e
8	e(e_4)	b	c	e	e
9	b(b_1)	b	c	d	e
10	c(c_1)	c	c		
11	c(c_2)	c	c		
12	b(b_2)	b	c	d	e
13	c(c_3)	c	c		

from. For ease of understanding, we use the query and data tree in Fig. 1 to exemplify the redundant computation of *getNext*. We present each step¹ of calling *getNext* over the root of the query tree (i.e., a) in Table 1.

Basically, the redundant computation mainly comes from the following *redundant test* and *late end*.

Redundant test. *redundant test* is making redundant calls of *getNext* over some nodes in the query tree. The current elements of these nodes did not change in the previous step. Consider the data tree t and query Q in Fig. 1. a_1 - a_3 are self-nested nodes. After we found a_1 has a solution extension in step 1, it is unnecessary to call *getNext* over the query trees rooted at the nodes b and d when testing whether a_2 has a solution extension. This is mainly because the current elements of the nodes b , c , d and e do not change during and after step 1. This also happens on a_3 when checking if a_3 has a solution extension. Therefore, the calls of *getNext* over the nodes b , c , d and e in step 2-3 are redundant, and they are grayed in Table 1. For the similar reason, it is unnecessary to call *getNext* over the nodes b , c , d and e in step 4. This also happens in step 9 and 12, and the calls of *getNext* over d and e are redundant. Furthermore, during step 5-7, the calls of *getNext* over the subtree rooted at node b are redundant.

Late end. *late end* is wasting time on the elements that will not contribute to any solutions when some cursors of the streams reach ends. Suppose there are no elements to be processed in the stream of node q , it is possible to skip all the rest of the elements in the streams of nodes *ancestors*(q) and *descendants*(q). This can avoid some calls of *getNext* and the time spent on scanning the elements in some streams. For the example above, when there are no elements left in the stream T_d after step 4, we can directly set PT_a to the end because the remaining elements in stream T_a will not contribute to any solutions. Then, when we found the rest of the elements in the streams of *descendant*(q) will not contribute any solutions, we can set the PT pointers of these streams to the ends. In Table 1, calls of *getNext* in step 8 and 13 are redundant and can be pruned.

¹ In this paper, a step is a call of *getNext* over the root of a query tree including the recursive calls.

Algorithm 1. $getNext^*(q)$

```

1: if  $isLeaf(q)$  then
2:   return  $q$ 
3: for  $q_i \in children(q)$  do
4:   if  $q_i.retest = true$  then
5:      $n_i = getNext^*(q_i)$ 
6:     if  $n_i \neq q_i$  then
7:        $q_i.retest = true$ 
8:       return  $n_i$ 
9:  $n_{min} = \min \arg_{q_i \in children(q)} nextL(T_{q_i})$ 
10:  $n_{max} = \max \arg_{q_i \in children(q)} nextL(T_{q_i})$ 
11: while ( $nextR(T_q) < nextL(T_{n_{max}})$ ) do
12:   Advance( $T_q$ )
13: if  $nextL(T_q) < nextL(T_{n_{min}})$  then
14:    $q.retest = false$ 
15:   return  $q$ 
16: else
17:    $n_{min}.retest = true$ 
18:   return  $n_{min}$ 

```

4 Approach for Avoiding Redundant Computation

4.1 Re-test Checking

Our solution for *redundant test* is called *re-test checking* and is mainly based on the following observation:

Observation. $getNext(n)$ is used for testing whether a solution extension can be found for the current element of node n . Suppose $getNext$ has been called over the node n before. If the current element of any node in the query tree rooted at n changes, it is necessary to call $getNext(n)$ again for re-testing. Otherwise, $getNext(n)$ does not need to be called.

We introduce an extra value *retest* for each query node to record whether $getNext$ need to be called on this node in the next step. The initial value of *retest* is *true*, and this value is dynamic during computation. The new version of $getNext$ is presented in Algorithm 1.

$getNext^*$ has the following properties: (1) Given a query rooted at Q , $getNext^*$ is only called over the nodes whose value of *retest* is *true*, including the nodes that have not been tested by $getNext^*$ before and the nodes have been tested by $getNext^*$ but need to be tested again. (2) Suppose $getNext^*$ has been called over each node at least once. If $getNext^*(Q)$ returns a node n in a step, $getNext^*$ will only be called over the nodes on the path from Q to n in the next step. The number of times $getNext^*$ is called will be bounded by the maximal depth of the query tree in the following steps.

With the properties above, the number of times $getNext^*$ is called can be significantly reduced, particularly when the query tree has many branches.

4.2 Forward-to-End

When the pointer PT_n of the stream T_n reaches the end, the rest of the elements in the streams of n 's ancestors and descendants may become useless. Therefore,

Algorithm 2. Forward-to-end

```

1: procedure FORWARDANSTOEND( $n$ )
2:   for each  $p$  in  $ancestors(n)$  do
3:      $ForwardtoEnd(T_p)$ 

4: procedure FORWARDDESTOEND( $n$ )
5:   for each  $d$  in  $descendants(n)$  do
6:      $ForwardtoEnd(T_d)$ 

```

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:       $v_{act}.ancestor = n_{act}.tail$ 
13:       $n_{act}.tail = length(L_{n_{act}}) + 1$ 
14:      append  $v_{act}$  into list  $L_{n_{act}}$ 
15:   else if  $isEnd(T_{parent(n_{act})}) = true$  then
16:      $ForwardDestoEnd(parent(n_{act}))$ 
17:      $Advance(T_{n_{act}})$ 
18:   if  $isEnd(T_{n_{act}}) = true$  then
19:      $ForwardAnstoEnd(n_{act})$ 
20:  $SetRestEndPointers(Q, \infty)$ 

```

we need to find a solution to skip these useless elements. In our approach *forward-to-end*, we select two time points for skipping. Consider a query tree rooted at Q . Suppose $getNext(Q)$ returns node n in a step, and PT_n reaches the end after calling $Advance(T_n)$. The two time points for skipping the rest of the elements in the streams of n 's ancestors and descendants are as follows:

Time point 1. We immediately skip the rest of the elements in the streams of n 's ancestors after calling $Advance(T_n)$ because we can not find any solution extension for them in the following steps.

Time point 2. We *can not* immediately skip the rest of the elements in the streams of n 's descendants after calling $Advance(T_n)$ because they are still potential elements that may contribute to final solutions. We have to wait until there are no elements in the stack S_n for the two-phase algorithms that use stacks for storing intermediate results and all the end positions in the list L_n have been set for the one-phase algorithms that use lists for storing final results.

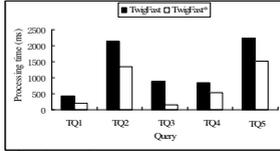
The pseudocode of skipping the rest of the elements in the streams of the node n and n 's ancestors and descendants is shown in Algorithm 2.

Algorithm 4. TwigStack*(Q)

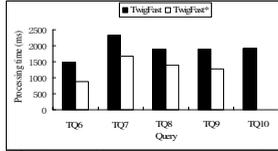
```

1: while  $\neg \text{end}(Q)$  do
2:    $n_{act} = \text{getNext}^*(\text{root}(Q))$ 
3:    $v_{act} = \text{getElement}(n_{act})$ 
4:   if  $\neg \text{isRoot}(n_{act})$  then
5:      $\text{CleanStack}(\text{parent}(n_{act}), v_{act}.start)$ 
6:   if  $\text{isRoot}(n_{act}) \vee \neg \text{empty}(S_{\text{parent}(n_{act})})$  then
7:      $\text{CleanStack}(n_{act}, v_{act}.start)$ 
8:      $\text{MoveStreamtoStack}(T_{q_{act}}, S_{q_{act}}, \text{pointer to top}(S_{\text{parent}(S_{q_{act}})}))$ 
9:     if  $\neg \text{isLeaf}(n_{act})$  then
10:       $\text{ShowSolutionswithBlocking}(S_{n_{act}}, 1)$ 
11:       $\text{Pop}(S_{n_{act}})$ 
12:     else if  $\text{isEnd}(T_{\text{parent}(n_{act})}) = \text{true}$  then
13:        $\text{ForwardDestoEnd}(\text{parent}(n_{act}))$ 
14:        $\text{Advance}(T_{n_{act}})$ 
15:       if  $\text{isEnd}(T_{n_{act}}) = \text{true}$  then
16:          $\text{ForwardAnstoEnd}(n_{act})$ 
17:  $\text{mergeAllPathSolutions}()$ 

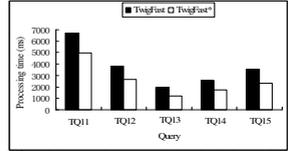
```



(a) low frequency on non-leaf nodes



(b) low frequency on leaf nodes



(c) all nodes with high frequencies

Fig. 2. Processing time of queries with different characteristics**4.3 TwigFast* and TwigStack***

Algorithm 3 and 4 extend the algorithm TwigFast and TwigStack respectively by applying re-test checking and forward-to-end.

5 Experiments

In this section, we present the experiment results on the performance of TwigFast* against TwigFast [5] and TwigStack* against TwigStack [2] and TJEssential [4], with both real-world and synthetic data sets. The algorithms are evaluated with the metrics of processing time. We selected the queries with different characteristics for more accurate evaluation.

We implemented TwigFast*, TwigFast, TwigStack*, TwigStack and TJEssential in C++. All the experiments were performed on 1.7GHz Intel Pentium M processor with 1G RAM. The operating system is Windows 7. We used the following three data sets for evaluation: TreeBank [1], DBLP [1] and XMark [6]. The queries for evaluation are listed in Table 2, which contain ‘//’ and ‘/’ edges.

Performance of answering the queries with different characteristics.In order to make the experiments more objective, we selected the queries with differ-

Table 2. Queries over TreeBank, DBLP and XMark

Data set	Query	XPath expression
TreeBank	TQ1	//V//S
TreeBank	TQ2	//ADV//S//PP//NP
TreeBank	TQ3	//A//S//VP
TreeBank	TQ4	//ADJ//NN//DT
TreeBank	TQ5	//VP//ADV//VP//NP//S
TreeBank	TQ6	//S//NP//CONJ
TreeBank	TQ7	//NP//NP//PP//_NL_
TreeBank	TQ8	//S//VP//NP//_HASH_
TreeBank	TQ9	//S//ADV//PP//NP
TreeBank	TQ10	//VP//NP//PP//FILE
TreeBank	TQ11	//VP//NP//NP//S//PP//VP//NN
TreeBank	TQ12	//NP//S//VP//NP//PP//NP//VBN
TreeBank	TQ13	//S//VP//PP//NP//VBN//IN
TreeBank	TQ14	//S//VP//PP//NP//NP//CD//VBN//IN
TreeBank	TQ15	//S//VP//PP//NP//S//PP//JJ//VBN//PP//NP//_NONE_
TreeBank	TQ16	//S//VP//NP//VP//NP//ADJ//NP//PP//VBN//DT//NN
TreeBank	TQ17	//S//VP//NP//ADV//VBN//VP
TreeBank	TQ18	//S//VP//NP//JJ//NP//PP//ADJ
TreeBank	TQ19	//S//VP//NP//JJ//PP//NN//V
TreeBank	TQ20	//S//VP//NP//PP//A//ADJ
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//title//url//ee/year
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	//people/person//address/zipcode//id//profile//age//education
XMark	XQ5	//open_auction//annotation//parlist//bidder//increase

ent characteristics over the TreeBank dataset. For the queries TQ1-TQ5, there is at least one non-leaf node with low frequency in each query. On the contrary, the nodes with low frequencies appear on leaf nodes in the queries TQ6-TQ10. For the queries TQ11-TQ15, all the nodes have high frequencies. We compare **TwigFast*** with **TwigFast** on these three types of queries. The results are shown in Fig. 2. As shown in this figure, **TwigFast*** achieves better performance than **TwigFast** on all these three types of queries, and is more than 30% faster than **TwigFast** on most queries. The better performance of **TwigFast*** on the queries TQ1-TQ10 suggests that the *forward to end* approach can avoid the redundant computation brought by *late end*. On the other hand, **TwigFast*** achieves better performance than **TwigFast** on the queries TQ11-TQ15 mainly because *re-test checking* approach avoids a large amount of unnecessary calls of *getNext*.

Performance of answering the queries over different datasets. We first compare **TwigFast*** with **TwigFast** over the datasets TreeBank, DBLP and XMark. The queries TQ16-TQ20 over TreeBank dataset mix different characteristics we mentioned above. The results are shown in Fig. 3. As shown in this figure, **TwigFast*** has better efficiency than **TwigFast** on all of the queries over the three datasets. Then we compare **TwigStack*** with **TwigStack** and **TJEssential** over the datasets TreeBank, DBLP and XMark. The results are shown in Fig. 4.

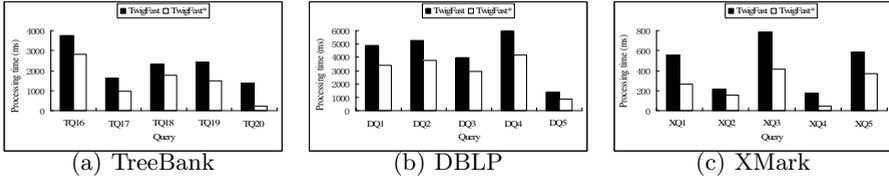


Fig. 3. TwigFast vs TwigFast*

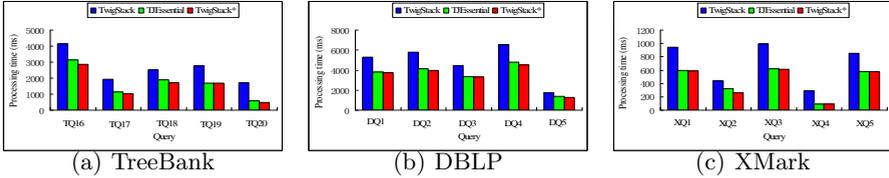


Fig. 4. TwigStack, TJEssential vs TwigStack*

From the results, we can see that both TJEssential and TwigStack* achieves better performance than TwigStack by resolving the redundant computation even though they use different approaches. Additionally, TwigStack* is a littler faster than TJEssential because TwigStack* can avoid some redundant computation that TJEssential can not avoid and TwigStack* imposes less overheads.

6 Conclusion

We presented the approaches *re-test checking* and *forward-to-end*, which can be easily applied to both holistic one-phase and two-phase twig pattern matching algorithms that are based on TwigStack, to resolve the redundant computation in *getNext*. Two algorithms TwigFast* and TwigStack* were presented. The better performance of our algorithms has been verified in our experiments.

References

1. <http://www.cs.washington.edu/research/xmldatasets/>
2. Bruno, N., Koudas, N., Srivastava, D.: Holistic twig joins: optimal XML pattern matching. In: SIGMOD Conference, pp. 310–321 (2002)
3. Jiang, Z., Luo, C., Hou, W.-C., Zhu, Q., Che, D.: Efficient processing of XML twig pattern: A novel one-phase holistic solution. In: Wagner, R., Revell, N., Pernul, G. (eds.) DEXA 2007. LNCS, vol. 4653, pp. 87–97. Springer, Heidelberg (2007)
4. Li, G., Feng, J., Zhang, Y., Zhou, L.: Efficient holistic twig joins in leaf-to-root combining with root-to-leaf way. In: Kotagiri, R., Radha Krishna, P., Mohania, M., Nantajeewarawat, E. (eds.) DASFAA 2007. LNCS, vol. 4443, pp. 834–849. Springer, Heidelberg (2007)
5. Li, J., Wang, J.: Fast matching of twig patterns. In: Bhowmick, S.S., Küng, J., Wagner, R. (eds.) DEXA 2008. LNCS, vol. 5181, pp. 523–536. Springer, Heidelberg (2008)
6. Schmidt, A., Waas, F., Kersten, M., Florescu, D., Manolescu, I., Carey, M., Busse, R.: The XML benchmark project. Technical Report INS-R0103, CWI (April 2001)