




# Enhanced Sliding Window-Based Periodic Pattern Mining from Dynamic Streams

Evan W. Madill<sup>1</sup>, Carson K. Leung<sup>1</sup> , and Justin M. Gouge<sup>1,2</sup>

<sup>1</sup> University of Manitoba, Winnipeg, MB, Canada

Carson.Leung@UManitoba.ca

<sup>2</sup> York University, Toronto, ON, Canada

**Abstract.** Discovering frequent patterns has been an important problem for knowledge discovery. The efficient discovery of interesting patterns—such as weighted periodic patterns—from big data has been crucial to the development in new domains. Due to their high velocity of data generation and collection, these big data can form dynamic streams, which can be unbounded. Traditional approaches to this problem consist of the reconstruction of the underlying structure, while recent advances have shown new methods for dynamically updating the underlying structure for each new window. In this paper, we present an enhanced sliding window-based algorithm for mining weighted periodic patterns from dynamic streams. Evaluation results show the effectiveness of this algorithm.

**Keywords:** Data mining · Periodic pattern · Stream mining · Sliding window

## 1 Introduction

In the current era, big data [1] are everywhere. With advances in technology, high volumes of a wide variety of data are generated and collected at a high velocity for numerous real-life applications and services (e.g., healthcare informatics [2], transportation analytics [3–5], business analytics [6, 7], social network analysis [8–12]). Embedded in these big data is implicit, previously unknown and potentially useful information and knowledge. This calls for big data management [13–15], as well as big data analytics and knowledge discovery [16, 17].

As an important big data analytics and knowledge discovery task, *frequent pattern mining* [18–21] aims to discover frequently occurring sets of items (e.g., merchandise items, events) from big data. Due to the continuous and unbounded nature of dynamic data streams, their contents are usually captured in an underlying structure such as a suffix tree [22], from which frequent patterns can be mined. Groups of data (e.g., sequence of characters or a string) are usually discretized and represented by a single symbol (e.g. a character) in the tree. Traditional approaches for data stream mining with sliding windows reconstruct a suffix tree for every sliding of the windows, which can be costly. To deal with this problem, a *dynamic tree based solution to handle sliding window in time (DTSW)* [23] was proposed to dynamically update and maintain the structure of the tree for each modified window, keeping it suitable for pattern mining. Although the DTSW

algorithm avoids reconstruction of suffix trees whenever the window slides, it introduces another problem. When the window slides, the deletion module of the algorithm removes the old batch from an explicit form of the tree, and insertion module inserts the new batch to an implicit form of the tree. Hence, these insertions and deletions requires frequent transformation of the tree between its implicit and explicit forms.

In this paper, we present a new algorithm to address this problem of tree transformations between its implicit and explicit forms. The algorithm eliminates the need to transform the suffix tree leaving the tree in its implicit form at all times when the window slides. Evaluation results show that our algorithm achieves a large performance increase across all window sizes tested, with no significant increase in memory.

*Key contributions* of this paper include design of our enhanced sliding window-based algorithm for mining periodic patterns, which are sequences that periodically occur at least a certain amount of times. With our suffixList structure, our algorithm only needs to maintain the *implicit* form of the suffix tree when capturing important information from dynamic streams (rather than converting back-and-forth between the implicit and explicit forms of the suffix tree as in the related works).

The remainder of this paper is organized as follows. The next section provides background and related works. Section 3 describes our enhanced sliding window-based algorithm for mining periodic patterns from streams. Evaluation results are shown in Sect. 4. Finally, conclusions are drawn in Sect. 5.

## 2 Background and Related Works

A **suffix tree** [22] is a trie containing all the suffixes of a given sequence of characters, or string. Figure 1 shows suffix trees—(a) in its implicit form and (b) in its explicit form—for a string “abcabababc”. A suffix tree is in its explicit form when all suffixes can be found by traversal from the root to a leaf node. An implicit suffix tree may contain suffixes (which are implicit to an edge) that is they do not end in a leaf node, but rather end within an edge. We can force a suffix tree to be in an explicit form by inserting a unique character, usually a “\$” or “#” to the end of the string.

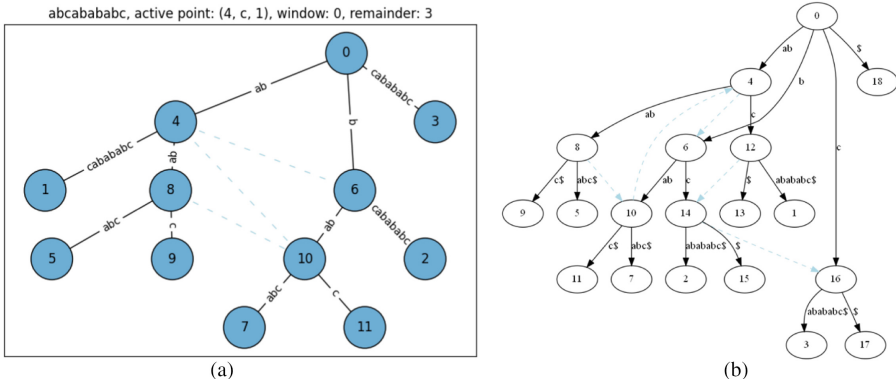
**Ukkonen’s algorithm** [24] is a linear-time algorithm for the construction of an implicit suffix tree. One can add a unique symbol on the end during construction to create an explicit tree. See Fig. 1.

A **sliding window** only stores data relevant to a certain time frame. Since we use the process of discretization to obtain a sequence of characters, we can form a “window” around the characters we want to look at. These characters then make up the underlying suffix tree. Consider an example with the string “abcabababc” and a window size of 3:

$$\boxed{abc} abababc \rightarrow a \boxed{bca} bababc \rightarrow ab \boxed{cab} ababc$$

where the characters preceding the boxed characters (i.e., characters currently in the window) may not have been received yet. In existing approaches, these suffix trees were reconstructed at each window slide.

The **Dynamic Tree Based Solution to Handle Sliding Window in Time Series Data (DTSW)** [23] handles updating of the suffix tree (rather than reconstruction) by



**Fig. 1.** A suffix tree in its (a) implicit form (as plotted by Matplotlib) and (b) explicit form (as plotted by Pydot).

the use of the insertion and deletion modules. The deletion module deletes characters (the first character) from the beginning of the window. However, before deleting any suffix from the suffix tree, the tree must be in its explicit form. To insert a new character (representing an event) onto the end of the sequence, the insertion module needs to revert back the tree from its explicit form to an implicit form for removing the unique symbol and erasing all the effects created due to it. In other words, to insert into (or delete events from) the suffix tree, the insertion and deletion modules need to convert the suffix tree back-and-forth between its explicit and implicit form.

### 3 Our Sliding Window-Based Weighted Periodic Pattern Mining Algorithm

The conversion between implicit and explicit form described by the DTSW is an unnecessary step. Hence, we *keep the tree in its implicit form* by proposing the idea of a *suffixList*, an ordered list of all possible suffixes from the sequence of characters. The initial tree is built with Ukkonen’s algorithm and the *suffixList* is created during construction. We must also be able to maintain the active point in order to use Ukkonen’s algorithm for insertion.

While the initial tree is being constructed using Ukkonen’s algorithm, we create a **suffixList**—which is a list of all possible suffixes from the input string. We keep the list in a sorted order so that it can be easily viewed and manipulated. When deleting a character from *S*, we update our *suffixList* by removing the longest suffix. Then, we do a simple traversal of the suffix tree, and remove the nodes associated with longest suffix. In other words, we can use the *suffixList* to check what is supposed to be there and what is not. The process is the same for both insertion and deletion. Essentially, we use our updated *suffixList*, and check to make sure the necessary nodes are in the tree. Since this algorithm is meant to keep the tree in implicit form only, there is no conversion step from implicit to explicit for both insertion and deletion.

More specifically, the **deletion module** first deletes the longest suffix. Then, it traverses the tree (by traversing only the portion that starts with the character being

removed). On the suffixList, it checks each suffix that starts with the character to be deleted. It goes down this tree path and updates the labels used by Ukkonen’s algorithm if edge compression was used. It uses the existing structure of the tree and updates the labels as necessary, and then deletes any excess nodes that would have in the end.

A special case for deletion would be when the deletion of the first character causes two branches of the suffix tree to merge. For example, in string “abcabc”, if it deletes ‘a’ from the input string, then the branch of the tree with ‘a’ will be removed and subtree under ‘a’ will be merged appropriately with ‘b’ branch in the tree. Otherwise, it simply traverses down the appropriate branch and updates the labels of nodes to match the suffixes as needed. Then, anything past the end will be deleted from the suffix tree.

The deletion module removes the longest suffix from the tree. Since the tree is in its implicit form, this raises the problem of searching edges that may contain smaller implicit suffixes within them as simply deletion of the longest paths leaf is not sufficient. To check how much of the edge to delete, the removal of the longest suffix from the suffixList is required. Then, a traversal checks all suffixes starting with the character being deleted. Upon a node deletion, a cleanup must be performed. Otherwise, we find the index of the longest and update the edge label to reflect the next longest matching suffix along that path.

Moreover, it also has to update the active point if necessary. When deleting the longest suffix, if the active node and edge is present when traversing to the longest suffix, then it simply deletes to the active length. Otherwise, it removes the leaf safely. If it does delete on an active point, then it decrements the remainder and finds the new active point from the remainder.

Since we maintain the active point and remainder in the deletion module, the **insertion module** simply runs Ukkonen’s algorithm on the new character.

## 4 Evaluation

To evaluate our algorithm, we compared with reconstruction and the existing DTSSW algorithm (which all implemented in Python). See Table 1 for differences among the three algorithms. In the evaluation, we used 40 window slides in order to observe the effect of how we may receive data in a stream. All tests were run on a Ryzen 7 3800X 8-Core (3.9 GHz) and 64 GB RAM (3600 MHz). We used several real-world test datasets from the UCI Machine Learning Repository<sup>1</sup>. As results were consistent, we reported the results—which were an average of 50 executions—for the *individual household electric power consumption dataset*, which captures 2,075,259 events discretized into 13 types.

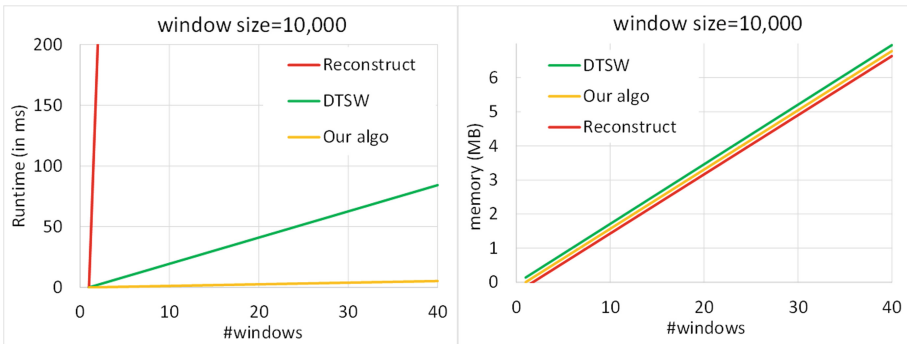
---

<sup>1</sup> <https://archive.ics.uci.edu/ml/index.php>.

**Table 1.** Summary of algorithms.

Algorithm	Delete form	Insert form
Reconstruct	–	–
DTSW	Explicit	Implicit
<b>Our algorithm</b>	<b>Implicit</b>	Implicit

When window size grew from 10 to 1,000 and 10,000, the amount of *runtimes* required by our algorithm dropped from 42% to 22% and 15% of the runtimes required by DTSW. In contrast, for the baseline reconstruction approach (denoted as Reconstruct), runtimes grew from 4.38 ms to 793 ms and 7,950 ms when window size grew from 10 to 1,000 and 10,000, respectively. In terms of memory, as the window size grows larger, the number of characters remains the same. The *space efficiency* of the algorithms seem to, on average, converge. For instance, with 40 windows, our algorithm consumed 97% and 99% of those required by DTSW when window size = 1,000 and 10,000, respectively. See Fig. 2. To summarize, our algorithm consumes almost the same amount of memory space as the existing DTSW (baseline), but our algorithm runs much faster than DTSW.



**Fig. 2.** (a) Runtime (in ms) and (b) memory (in MB) for sliding window performed over 40 windows with a window size of 10,000.

## 5 Conclusions

In this paper, we presented an enhanced sliding window-based periodic pattern stream mining algorithm. It uses suffix tree to capture important contents of the dynamic streams, from which periodic patterns are mined. It makes good use of implicit forms of the suffix tree during deletion and insertion of tree branches due to the sliding of the window capturing batches of the data streams. As such, it achieves shorter runtime and less memory space consumption when compared with the existing dynamic tree based solution to handle sliding window in time (DTSW) algorithm. Evaluation results show that our

algorithm outperformed related works. As *ongoing and future work*, we explore ways (e.g., use the sliding suffix tree [25]) to further improve our algorithm.

**Acknowledgement.** This work is partially supported by NSERC (Canada) & U. Manitoba.

## References

1. Bemarkisika, P., Totohasina, A.: ERAPN, an algorithm for extraction positive and negative association rules in big data. In: Ordonez, C., Bellatreche, L. (eds.) DaWaK 2018. LNCS, vol. 11031, pp. 329–344. Springer, Cham (2018). [https://doi.org/10.1007/978-3-319-98539-8\\_25](https://doi.org/10.1007/978-3-319-98539-8_25)
2. Leung, C.K., Fung, D.L.X., Hoi, C.S.H.: Health analytics on COVID-19 data with few-shot learning. In: Golfarelli, M., Wrembel, R., Kotsis, G., Tjoa, A.M., Khalil, I. (eds.) DaWaK 2021. LNCS, vol. 12925, pp. 67–80. Springer, Cham (2021). [https://doi.org/10.1007/978-3-030-86534-4\\_6](https://doi.org/10.1007/978-3-030-86534-4_6)
3. Audu, A.-R.A., Cuzzocrea, A., Leung, C.K., MacLeod, K.A., Ohin, N.I., Pulgar-Vidal, N.C.: An intelligent predictive analytics system for transportation analytics on open data towards the development of a smart city. In: Barolli, L., Hussain, F.K., Ikeda, M. (eds.) CISIS 2019. AISC, vol. 993, pp. 224–236. Springer, Cham (2020). [https://doi.org/10.1007/978-3-030-22354-0\\_21](https://doi.org/10.1007/978-3-030-22354-0_21)
4. Leung, C.K., Braun, P., Hoi, C.S.H., Souza, J., Cuzzocrea, A.: Urban analytics of big transportation data for supporting smart cities. In: Ordonez, C., Song, I.-Y., Anderst-Kotsis, G., Tjoa, A.M., Khalil, I. (eds.) DaWaK 2019. LNCS, vol. 11708, pp. 24–33. Springer, Cham (2019). [https://doi.org/10.1007/978-3-030-27520-4\\_3](https://doi.org/10.1007/978-3-030-27520-4_3)
5. Leung, C.K., Braun, P., Pazdor, A.G.M.: Effective classification of ground transportation modes for urban data mining in smart cities. In: Ordonez, C., Bellatreche, L. (eds.) DaWaK 2018. LNCS, vol. 11031, pp. 83–97. Springer, Cham (2018). [https://doi.org/10.1007/978-3-319-98539-8\\_7](https://doi.org/10.1007/978-3-319-98539-8_7)
6. Ahn, S., et al.: A fuzzy logic based machine learning tool for supporting big data business analytics in complex artificial intelligence environments. In: FUZZ-IEEE 2019, pp. 1259–1264
7. Morris, K.J., et al.: Token-based adaptive time-series prediction by ensembling linear and non-linear estimators: a machine learning approach for predictive analytics on big stock data. In: IEEE ICMLA 2018, pp. 1486–1491
8. Braun, P., Cuzzocrea, A., Jiang, F., Leung, C.K.-S., Pazdor, A.G.M.: MapReduce-based complex big data analytics over uncertain and imprecise social networks. In: Bellatreche, L., Chakravarthy, S. (eds.) DaWaK 2017. LNCS, vol. 10440, pp. 130–145. Springer, Cham (2017). [https://doi.org/10.1007/978-3-319-64283-3\\_10](https://doi.org/10.1007/978-3-319-64283-3_10)
9. Jiang, F., Leung, C.K.-S.: Mining interesting “following” patterns from social networks. In: Bellatreche, L., Mohania, M.K. (eds.) DaWaK 2014. LNCS, vol. 8646, pp. 308–319. Springer, Cham (2014). [https://doi.org/10.1007/978-3-319-10160-6\\_28](https://doi.org/10.1007/978-3-319-10160-6_28)
10. Leung, C.K.: Mathematical model for propagation of influence in a social network. In: Alhadjj, R., Rokne, J. (eds.) Encyclopedia of Social Network Analysis and Mining, 2nd edn., pp. 1261–1269. Springer, New York, NY (2018). [https://doi.org/10.1007/978-1-4939-7131-2\\_110201](https://doi.org/10.1007/978-1-4939-7131-2_110201)
11. Leung, C.K., et al.: Parallel social network mining for interesting “following” patterns. Concurrency Comput. Pract. Experience **28**(15), 3994–4012 (2016)
12. Leung, C.K.-S., Carmichael, C.L., Teh, E.W.: Visual analytics of social networks: mining and visualizing co-authorship networks. In: Schmorow, D.D., Fidopiastis, C.M. (eds.) FAC 2011. LNCS (LNAI), vol. 6780, pp. 335–345. Springer, Heidelberg (2011). [https://doi.org/10.1007/978-3-642-21852-1\\_40](https://doi.org/10.1007/978-3-642-21852-1_40)

13. Arora, N.R., Lee, W., Leung, C.K.-S., Kim, J., Kumar, H.: Efficient fuzzy ranking for keyword search on graphs. In: Liddle, S.W., Schewe, K.-D., Tjoa, A.M., Zhou, X. (eds.) DEXA 2012, Part I. LNCS, vol. 7446, pp. 502–510. Springer, Heidelberg (2012). [https://doi.org/10.1007/978-3-642-32600-4\\_38](https://doi.org/10.1007/978-3-642-32600-4_38)
14. Eom, C.S., et al.: Effective privacy preserving data publishing by vectorization. *Inf. Sci.* **527**, 311–328 (2020)
15. Olawoyin, A.M., Leung, C.K., Choudhury, R.: Privacy-preserving spatio-temporal patient data publishing. In: Hartmann, S., Küng, J., Kotsis, G., Tjoa, A.M., Khalil, I. (eds.) DEXA 2020, Part II. LNCS, vol. 12392, pp. 407–416. Springer, Cham (2020). [https://doi.org/10.1007/978-3-030-59051-2\\_28](https://doi.org/10.1007/978-3-030-59051-2_28)
16. Leung, C.K.-S., Jiang, F.: Big data analytics of social networks for the discovery of “following” patterns. In: Madria, S., Hara, T. (eds.) DaWaK 2015. LNCS, vol. 9263, pp. 123–135. Springer, Cham (2015). [https://doi.org/10.1007/978-3-319-22729-0\\_10](https://doi.org/10.1007/978-3-319-22729-0_10)
17. Souza, J., Leung, C.K., Cuzzocrea, A.: An innovative big data predictive analytics framework over hybrid big data sources with an application for disease analytics. In: Barolli, L., Amato, F., Moscato, F., Enokido, T., Takizawa, M. (eds.) AINA 2020. AISC, vol. 1151, pp. 669–680. Springer, Cham (2020). [https://doi.org/10.1007/978-3-030-44041-1\\_59](https://doi.org/10.1007/978-3-030-44041-1_59)
18. Jiang, F., Leung, C.K.-S.: Stream mining of frequent patterns from delayed batches of uncertain data. In: Bellatreche, L., Mohania, M.K. (eds.) DaWaK 2013. LNCS, vol. 8057, pp. 209–221. Springer, Heidelberg (2013). [https://doi.org/10.1007/978-3-642-40131-2\\_18](https://doi.org/10.1007/978-3-642-40131-2_18)
19. Leung, C.K.-S., MacKinnon, R.K.: Balancing tree size and accuracy in fast mining of uncertain frequent patterns. In: Madria, S., Hara, T. (eds.) DaWaK 2015. LNCS, vol. 9263, pp. 57–69. Springer, Cham (2015). [https://doi.org/10.1007/978-3-319-22729-0\\_5](https://doi.org/10.1007/978-3-319-22729-0_5)
20. Leung, C.-S., MacKinnon, R.K.: BLIMP: a compact tree structure for uncertain frequent pattern mining. In: Bellatreche, L., Mohania, M.K. (eds.) DaWaK 2014. LNCS, vol. 8646, pp. 115–123. Springer, Cham (2014). [https://doi.org/10.1007/978-3-319-10160-6\\_11](https://doi.org/10.1007/978-3-319-10160-6_11)
21. Leung, C.K.-S., Tanbeer, S.K.: Mining popular patterns from transactional databases. In: Cuzzocrea, A., Dayal, U. (eds.) DaWaK 2012. LNCS, vol. 7448, pp. 291–302. Springer, Heidelberg (2012). [https://doi.org/10.1007/978-3-642-32584-7\\_24](https://doi.org/10.1007/978-3-642-32584-7_24)
22. Rasheed, F., Alshalalfa, M., Alhadj, R.: Efficient periodicity mining in time series databases using suffix trees. *IEEE TKDE* **23**(1), 79–94 (2011)
23. Rizvee, R.A., et al.: Sliding window based weighted periodic pattern mining over time series data. In: *ICDM 2019*, pp. 118–132
24. Ukkonen, E.: On-line construction of suffix trees. *Algorithmica* **14**(3), 249–260 (1995)
25. Brodnik, A., Jekovec, M.: Sliding suffix tree. *Algorithms* **11**, 118:1–118:11 (2018)