Information Sciences 441 (2018) 171–186

Contents lists available at ScienceDirect

Information Sciences

journal homepage: www.elsevier.com/locate/ins

# Opportunistic mining of top-*n* high utility patterns

Junqiang Liu[a,*], Xingxing Zhang[a], Benjamin C.M. Fung[b], Jiuyong Li[c], Farkhund Iqbal[d]

[a] School of Information and Electronic Engineering, Zhejiang Gongshang University, Hangzhou 310018, China
[b] School of Information Studies, McGill University, Montreal, Quebec H3A 1X1, Canada
[c] School of Information Technology & Mathematical Sciences, University of South Australia, Adelaide, South Australia 5001, Australia
[d] College of Technological Innovation, Zayed University, United Arab Emirates

ARTICLE INFO

*Article history:*
Received 18 December 2016
Revised 10 December 2017
Accepted 15 February 2018
Available online 15 February 2018

*Keywords:*
Utility mining
Pattern mining
High utility patterns
Frequent patterns
Top-*n* interesting patterns

ABSTRACT

Mining high utility patterns is an important data mining problem that is formulated as finding patterns whose utilities are no less than a threshold. As the mining results are very sensitive to such a threshold, it is difficult for users to specify an appropriate one. An alternative formulation of the problem is to find the top-*n* high utility patterns. However, the second formulation is more challenging because the corresponding threshold is unknown in advance and the solution search space becomes even larger. When there are very long patterns prior algorithms simply cannot work to mine top-*n* high utility patterns even for very small *n*.

This paper proposes a novel algorithm for mining top-*n* high utility patterns that are long. The proposed algorithm adopts an opportunistic pattern growth approach and proposes five opportunistic strategies for scalably maintaining shortlisted patterns, for efficiently computing utilities, and for estimating tight upper bounds to prune search space. Extensive experiments show that the proposed algorithm is 1 to 3 orders of magnitude more efficient than the state-of-the-art top-*n* high utility pattern mining algorithms, and it is even up to 2 orders of magnitude faster than high utility pattern mining algorithms that are tuned with an optimal threshold.



## 1. Introduction

High utility pattern mining [40] has recently emerged to address the limitation of frequent pattern mining by considering the user's expectation as well as the raw data. Among several high utility pattern mining problems [9,10,18,21,28,30,38,39], mining high utility patterns with the share framework [18,38,39] is the hardest. This problem is difficult to solve since the utility measure is neither anti-monotone nor monotone. Many algorithms [6,13,19,20,27,32,34,38] resort to an interim, anti-monotone measure, called transaction weighted utilization (TWU), and work in two phases. `UP-Growth+` [32,34] is the best two-phase algorithm, but it suffers from the scalability and efficiency bottleneck due to the huge number of high TWU patterns (candidates) generated in Phase I. Recently, one-phase algorithms [24–26,41] were proposed to address the bottleneck, among them is the `HUI-Miner` algorithm [26]. `HUI-Miner` is not scalable due to its vertical data structure and is not efficient when mining large databases due to its join operations on the vertical structure. `EFIM` [41] and `d`$^2$`HUP` [24,25] are the latest one-phase algorithms.

* Corresponding author.
*E-mail address:* jjliu@alumni.sfu.ca (J. Liu).

https://doi.org/10.1016/j.ins.2018.02.035
0020-0255/

On the other hand, it is difficult for users to specify a minimum utility threshold as the mining results are very sensitive to the threshold. An alternative is to reformulate the problem by applying the top-*n* interesting pattern model [16], i.e., to find top-*n* high utility patterns. The general approach to the top-*n* interesting pattern mining problem [1,12,15,35,37] is to dynamically raise a border threshold that will eventually reach the optimal minimum threshold that results in top-*n* interesting patterns while the optimal threshold is unknown in advance. Clearly, the second formulation means a much larger solution search space and a computationally more challenging problem.

There are very few works on the top-*n* high utility pattern mining problem, probably due to the aforementioned challenges. The TKU [33,36] and TKO [33] algorithms are the latest. TKU [33,36] extends UP-Growth+ [32,34]. The bottleneck with UP-Growth+ persists and becomes worse with TKU since the border threshold cannot be raised to the optimal threshold in Phase I. TKO [33] extends HUI-Miner [26] and also inherits the challenges with HUI-Miner [26], i.e., it is not scalable and not efficient when mining large databases.

Moreover, when there are extremely long patterns, TKU and TKO simply cannot work even for very small *n*, for example, for $n = 10$ on the BMS-WebView-1 and Pumsb databases where the maximum pattern lengths are 148 and 22, respectively. These challenges stem from the inherent complexity of the top-*n* high utility pattern mining problem.

To address the challenges, this paper proposes a novel algorithm called TONUP (*TOp-N high Utility Pattern mining*) that directly grows top-*n* high utility patterns by enumerating patterns as prefix extensions, by shortlisting patterns with the first *n* greatest utilities among the enumerated patterns, and by employing the *n*-th greatest utility as a border threshold to prune the search space.

Our contributions are listed as follows, including five opportunistic strategies that are not used in prior algorithms.

First, our TONUP algorithm is the first efficient and scalable algorithm that can find the top-*n* high utility patterns that are extremely long. TONUP extends the $d^2$HUP algorithm [24,25] with the top-*n* interesting pattern model and embodies an opportunistic pattern growth approach. TONUP improves $d^2$HUP by an improved data structure iCAUL and two strategies, which is why TONUP beats $d^2$HUP by a factor up to 8 when both are tuned with an optimal threshold.

- The AutoMaterial strategy is to make a balance between pseudo projection and materialized projection of transaction sets that are maintained in iCAUL, by a self-adjusting threshold.
- The DynaDescend strategy is to help raise the border threshold and prune search space by dynamically resorting items in the descending order of local utility upper bounds, which is different from the static ordering employed in $d^2$HUP, while TKO and TKU employ an improper ordering.

Second, more opportunistic strategies dedicated to top-*n* high utility pattern mining are proposed for taking every opportunity to improve the efficiency and scalability, which facilitates the full-strength TONUP.

- The ExactBorder strategy is to quickly raise the border threshold by the exact utility of each enumerated pattern, which is more effective than four strategies in TKU and one strategy in TKO that raise the border threshold by estimating utility lower bounds.
- The SuffixTree strategy is to efficiently maintain shortlisted patterns by a suffix tree, which is much more efficient than by linear lists, such as arrays. When shortlisting patterns, TONUP considers the ties for the *n*-th greatest utility, which is ignored in TKO and TKU.
- The OppoShift strategy is to opportunistically shift to a two-round approach when the enumerated patterns are extremely long, which improves efficiency in orders of magnitude in such a case.

Third, insights into our approach and strategies are presented based on complexity analysis and extensive experimental comparison with the state-of-the-art algorithms. Concretely, the proposed algorithm TONUP is 1 to 3 orders of magnitude more efficient than the top-*n* high utility pattern mining algorithms TKO [33] and TKU [33,36], and is 1 to 2 orders of magnitude faster than the high utility pattern mining algorithms UP-Growth+ [32,34], HUI-Miner [26], and EFIM [41] tuned with an optimal threshold unknown to TONUP.

The rest of the paper is organized as follows. Section 2 defines the top-*n* high utility pattern mining problem. Section 3 surveys the related works. Section 4 proposes our opportunistic pattern growth approach. Section 5 proposes opportunistic strategies for efficiency and scalability. Section 6 evaluates our algorithm by comparing it with the state-of-the-art algorithms. Section 7 analyzes each strategy that facilitates our approach. Section 8 concludes the paper.

## 2. Top-*n* high utility pattern mining problem

Let $I$ be the universe of items. Let $D$ be a database of transactions $\{t_1, \ldots, t_n\}$, where each transaction $t_i \subseteq I$. Each item in a transaction is assigned a non-zero share or quantity. Each distinct item has a weight or price independent of any transaction, given by an eXternal Utility Table (*XUT*). The research problem is formally defined as follows.

**Definition 1.** The utility of an item $i$ in a transaction $t$, denoted $u(i, t)$, is a function $f$ of the share of $i$ in $t$, $iu(i, t)$, and the weight of $i$ independent of any transaction, $eu(i)$. That is, $u(i, t) = f(iu(i, t), eu(i))$. We also call $iu(i, t)$ the internal utility of $i$ in $t$, and $eu(i)$ the external utility of $i$.

**Table 1**
Transaction database $D$ and external utility table $XUT$.

| TID | (a) Transaction database | | | | | | | (b) External utility table | |
|---|---|---|---|---|---|---|---|---|---|
| | Item | | | | | | | Item | Price ($) |
| | a | b | c | d | e | f | g | | |
| | | | | | | | | a | 1 |
| $t_1$ | 1 | | 1 | | 1 | | | b | 3 |
| $t_2$ | 6 | 2 | 2 | | | 5 | | c | 5 |
| $t_3$ | 1 | 1 | 1 | 2 | 6 | | 5 | d | 2 |
| $t_4$ | 3 | 1 | | 4 | 3 | | | e | 2 |
| $t_5$ | 2 | 1 | | 2 | | 2 | | f | 1 |
| | | | | | | | | g | 1 |

**Definition 2.**

(a) A transaction $t$ *contains* a pattern $X$ if $X \subseteq t$, which means that for every item $i$ in $X$, $iu(i, t) \neq 0$.
(b) The *transaction set* of a pattern $X$, denoted $TS(X)$, is the set of transactions that contain $X$. The number of transactions in $TS(X)$ is the *support* of $X$, denoted $s(X)$.

**Definition 3.**

(a) For a pattern $X$ contained in a transaction $t$, i.e., $X \subseteq t$, the utility of $X$ in $t$, denoted $u(X, t)$, is the sum of the utility of every constituent item of $X$ in $t$, i.e.,

$$u(X, t) = \sum_{i \in X \subseteq t} u(i, t).$$

(b) The utility of $X$, denoted $u(X)$, is the sum of the utility of $X$ in every transaction containing $X$, i.e.,

$$u(X) = \sum_{t \in TS(X)} u(X, t) = \sum_{t \in TS(X)} \sum_{i \in X} u(i, t).$$

**Example 1.** For $D$ and $XUT$ in Table 1, $I = \{a,b,c,d,e,f,g\}$, and $u(i, t) = eu(i) \cdot iu(i, t)$. For transaction $t_1 = \{a,c,e\}$, we have $iu(a, t_1) = 1, iu(c, t_1) = 1,\ iu(e, t_1) = 1,\ eu(a) = 1, eu(c) = 5,$ and $eu(e) = 2$. And $u(a, t_1) = 1,\ u(c, t_1) = 5,$ and $u(e, t_1) = 2$. $TS(\{a,c\}) = \{t_1, t_2, t_3\}, s(\{a,c\}) = 3,\quad u(\{a,c\}) = u(\{a,c\}, t_1) + u(\{a,c\}, t_2) + u(\{a,c\}, t_3) = u(a, t_1) + u(c, t_1) + u(a, t_2) + u(c, t_2) + u(a, t_3) + u(c, t_3) = 28,$ and so on.

**Definition 4.** A pattern $X$ is a *high utility pattern*, abbreviated as HUP, if the utility of $X$ is no less than a user-defined *minimum utility threshold*, $\mu$.

***High utility pattern mining*** is to discover the set, $HUPset(\mu)$, of all high utility patterns from a database $D$ given an external utility table $XUT$ and $\mu$, i.e., $HUPset(\mu) = \{X | X \subseteq I, u(X) \geq \mu\}$.

**Definition 5.** A pattern $X$ is a top-$n$ *high utility pattern*, abbreviated as top-$n$ HUP, if there are fewer than $n$ patterns whose utilities are greater than the utility of $X$. In other words, $X$ is a top-$n$ HUP if $u(X) \geq \mu^*(n)$, where $\mu^*(n)$, called the optimal minimum utility threshold, is the $n$-th greatest pattern utility among the utilities of all patterns.

**Top-** $n$ **high utility pattern mining** is to find the set, $HUPset^*(n)$, of all top-$n$ high utility patterns, given $D$, $XUT$, and $n$, i.e., $HUPset^*(n) = HUPset(\mu^*(n)) = \{X | X \subseteq I, u(X) \geq \mu^*(n)\}$.

**Example 2.** Given $n = 6$, we have $\mu^*(n) = 30$. Thus, $\{a,c\}$ is not a top-$n$ high utility pattern as $u(\{a,c\}) = 28 < \mu^*(n)$, $\{a,b,c\}$ is as $u(\{a,b,c\}) = 31,$ and so on. Consequently, $HUPset^*(6) = HUPset(30) =$ { {a,b,c}, {a,b,d}, {a,d,e}, {a,b,d,e}, {b,d,e}, {d,e}, {a,b,c,d,e,g} }. Note that {d,e} and {a,b,c,d,e,g} are tied for the 6th place with a utility of 30. Finding $HUPset^*(n)$ is harder than finding $HUPset(\mu)$ because $\mu^*(n)$ is unknown in advance.

## 3. Related works

### 3.1. Frequent pattern mining

Conceptually, frequent patterns [4,5,11,17,31] and high utility patterns [18,38–40] can be organized by a tree, and the mining process can be thought of as searching such a tree. Mining algorithms fall into two categories: breadth-first search and depth-first search. Apriori by Agrawal and Srikant [4] and FP-growth by Han et al. [17] are well-known frequent pattern algorithms of the two categories, respectively. Generally, depth-first search algorithms [2,7,8,17,29] achieve better performance than breadth-first search algorithms [3,4]. In particular, FP-growth [17] outperforms Apriori [4].

### 3.2. High utility pattern mining with the share framework

Hilderman et al. and Yao et al. [18,38–40] proposed the utility mining problem with the itemset share framework. Liu et al. [27] proposed the TWU (transaction weighted utilization) property and developed the Two-Phase algorithm by adapting Apriori [4], which generates high TWU patterns (candidates) in Phase I and identifies high utility patterns from candidates in Phase II. Many algorithms adopt the two-phase approach [6,13,19,20,32,34]. Tseng et al. [32,34] proposed the best two-phase algorithms, UP-Growth and UP-Growth+, based on FP-growth. UP-Growth employs four strategies: DGU, DGN, DLU, and DLN. UP-Growth+ uses two more strategies: DNU and DNN.

Recently, one-phase algorithms [24–26,41] were proposed. Liu and Qu [26] proposed HUI-Miner that employs a vertical data structure, called utility-lists, and performs join operations on utility-lists. Liu et al. [24,25] proposed d$^2$HUP with a few contributions. First, two utility upper bounds facilitate the powerful pruning of the search space. Second, a linear data structure enables efficient computation of utilities and upper bounds. Third, a singleton property and a closure property help identify high utility patterns without enumeration. Fourth, a pattern growth approach searches a reverse set enumeration tree and employs a static sorting of items to enhance the likelihood for pruning. Zida et al. [41] proposed EFIM, which shares the same pattern growth approach, pruning techniques, and utility upper bounds as d$^2$HUP [24,25]. The contributions are new database projection, transaction merging, and utility counting techniques suitable for dense databases with short patterns. But, d$^2$HUP outperforms EFIM on dense databases with long patterns and on sparse and mixed databases by up to 2 orders of magnitude in efficiency.

This paper enhances d$^2$HUP [24,25] by an improved data structure, extends it with the top-$n$ interesting pattern model, and proposes five new strategies to improve the scalability and efficiency.

### 3.3. Top-n interesting pattern mining

Fu et al. [16] proposed the top-$n$ interesting pattern mining model. Wang et al. [35] proposed the TFP algorithm for mining top-$n$ frequent closed patterns. Afrati et al. [1] proposed using $n$ patterns to approximate frequent patterns. Xin et al. [37] proposed a method for extracting $n$-representative patterns. Cong et al. [12] discovered top-$n$ covering rule groups for each row of gene expression profiles. Fournier-Viger and Tseng proposed algorithms for mining top-$n$ association rules [15] and top-$n$ sequential rules [14].

Wu et al. [36] and Tseng et al. [33] proposed TKU and TKO, and Zihayat et al. proposed T-HUDS [42], for mining top-$n$ high utility patterns. TKU [33,36] extends UP-Growth+ [32,34] and works in two phases. It incorporates the PE, NU, MD, and MC strategies for estimating lower bounds on utilities of candidates in Phase I and the SE strategy for sorting candidates in Phase II. T-HUDS [42] also works in two phases and has the same performance as TKU. TKO [33] extends HUI-Miner [26] with the top-$n$ interesting pattern model by the RUC strategy and reduces utility estimates by removing zero-elements by the RUZ strategy. TKO processes itemsets in decreasing order of their estimated utility value by the EPB strategy, which is however questionable.

This paper addresses the issues with prior works by proposing a novel algorithm that directly discovers top-$n$ high utility patterns that are very long in a single phase without generating candidates.

## 4. Opportunistic top-$n$ high utility pattern growth and the baseline TONUP

This section proposes an opportunistic top-$n$ high utility pattern growth approach that extends d$^2$HUP with the top-$n$ interesting pattern model and presents the baseline version of our TONUP algorithm.

The proposed approach is to grow top-$n$ high utility patterns by enumerating patterns as prefix extensions, to compute the utility of each enumerated pattern by an improved memory-resident structure, to shortlist the enumerated patterns whose utilities are among the first $n$ greatest utilities, and to use the $n$-th greatest utility as the running border threshold to prune the patterns whose estimated utility upper bounds are below the threshold.

### 4.1. Enumerating patterns by prefix extensions

The basic idea is to enumerate a pattern as a prefix extension of another pattern. In order to avoid repetitive enumeration of patterns, an ordering of items is imposed, with which a pattern can also be represented as an ordered sequence. For brevity, we use the set notation, for example, {a,b,c}, in place of the sequence notation, for example, $<$a,b,c$>$.

**Definition 6.** The *imposed ordering of items*, denoted $\Omega$, is a pre-determined, ordered sequence of all the items in $I$. Accordingly, for items $i$ and $j$, $i \prec j$ denotes that $i$ is listed before $j$, $i \prec X$ denotes that $i \prec j$ for every $j \in X$, and $W \prec X$ denotes that $i \prec X$ for every $i \in W$, in accordance with $\Omega$.

**Definition 7.** Given an ordering $\Omega$, a pattern $Y$ is a *prefix extension* of a pattern $X$, if $X$ is a suffix of $Y$, i.e., if $Y = W \cup X$ for $W \prec X$ in $\Omega$.

**Definition 8.** Given an ordering $\Omega$, a pattern $Y$ is the *full prefix extension* of a pattern $X$ w.r.t. a transaction $t$ containing $X$, denoted $Y = fpe(X, t)$, if $Y$ is a prefix extension of $X$ derived by adding exactly all the items in $t$ that are listed before $X$ in $\Omega$, i.e., if $Y = W \cup X$ with $W = \{i | i \in t \wedge i \prec X \wedge X \subseteq t\}$.

Our pattern growth approach enumerates patterns of length 1 as prefix extensions of the empty pattern {}, patterns of length 2 as prefix extensions of patterns of length 1, and so on.

**Example 3.** Suppose the items are in the lexicographic order, i.e., $\Omega$ = {a,b,c,d,e,f,g}, then a $\prec$ b, a $\prec$ c, a $\prec${b,c}, {a,b} $\prec$ {c,d}, and so on. Therefore, {a} through {g} are enumerated as a prefix extension of {}, {a,b} as that of {b}, {a,c} and {b,c} as that of {c}, {a,b,c} as that of {b,c}, and so forth.

### 4.2. Shortlisting enumerated patterns

Our major strategy is to raise a border threshold $\mu_b$, starting from 0 to the optimal minimum utility threshold $\mu^*(n)$, by using the exact utilities of enumerated patterns. This strategy is straightforward and also more effective than the counterparts [33,36] that employ utility estimates (lower bounds) and pre-computation to raise a border threshold.

**Strategy 1** (ExactBorder - Exact utilities to raise a Border threshold)**.** When enumerating a prefix extension $Y$ of a pattern $X$, updating by $u(Y)$ and $Y$ the set ShortList($n$) of shortlisted patterns, i.e., the patterns whose utilities are no less than the $n$-th greatest among the patterns enumerated by far. The minimum utility in this set, $ShortList(n).\mu$, is the running border threshold $\mu_b$.

*ShortList*($n$) becomes the set of all top-$n$ high utility patterns, *HUPset*$^*$($n$), at the end of the pattern enumeration process. Clearly, the best data structure for *ShortList*($n$) is a minimum heap *minHeap* with a fixed capacity $n$, supplemented by a list *tieList* of patterns whose utility values are tied for the $n$-th greatest. Note that the ties are ignored in [33,36].

The root of the minimum heap keeps the minimum utility. In other words, $minHeap.root.utility = ShortList(n).\mu = \mu_b$. A pattern whose utility is less than $\mu_b$ is not a top-$n$ high utility pattern. Moreover, if an estimated upper bound of the utility of a pattern is less than $\mu_b$, the pattern can be pruned without actually computing its exact utility. In particular, we present two pruning techniques based on the following two upper bounds, respectively.

**Theorem 1** (Upper bound on prefix extensions of a pattern with an item)**.** *For a pattern $X$ and an item $i \prec X$, the utility of any prefix extension $Y$ of $X$ that contains $i$ is no more than the sum of the utility of the full prefix extension of $X$ w.r.t. every transaction in $TS(\{i\}\cup X)$, denoted $uB_{item}(i, X)$, i.e.,*

$$u(Y) \le \sum_{t\in TS(\{i\}\cup X)} u(fpe(X,t),t) = uB_{item}(i,X) \tag{1}$$

**Pruning 1.** If $uB_{item}(i, X) < \mu_b$ for a pattern $X$ and an item $i \prec X$, then $i$ is irrelevant and can be pruned in enumerating prefix extensions of $X$ since every prefix extension of $X$ containing $i$ is not a top-$n$ high utility pattern.

**Example 4.** When going to enumerate the prefix extensions of {d}, the shortlisted patterns with utilities are {({c}, 20), ({e}, 20), ({a,b}, 27), ({a,c}, 28),({b,c}, 24), ({a,b,c}, 31)}, and $\mu_b$ is raised to 20. At the moment, we have $uB_{item}$(a, {d}) = $uB_{item}$(b, {d}) = $u(\{a,b,c,d\}, t_3) + u(\{a,b,d\}, t_4) + u(\{a,b,d\}, t_5) = 36 > \mu_b$, and $uB_{item}$(c, {d}) = $u(\{a,b,c,d\}, t_3) = 13 < \mu_b$. Thus, the items a and b are relevant, and the item c is irrelevant in growing the prefix extensions of {d}.

**Theorem 2** (Upper bound on prefix extensions of a pattern)**.** *For a pattern $X$, the utility of any prefix extension $Y$ of $X$ is no more than the sum of the utility of the full prefix extension of $X$ w.r.t. each transaction in $TS(X)$, denoted $uB_{fpe}(X)$, i.e.,*

$$u(Y) \le \sum_{t\in TS(X)} u(fpe(X,t),t) = uB_{fpe}(X) \tag{2}$$

**Pruning 2.** If $uB_{fpe}(X) < \mu_b$ for a pattern $X$, then $X$ and all its prefix extensions can be pruned since none of them is a top-$n$ high utility pattern.

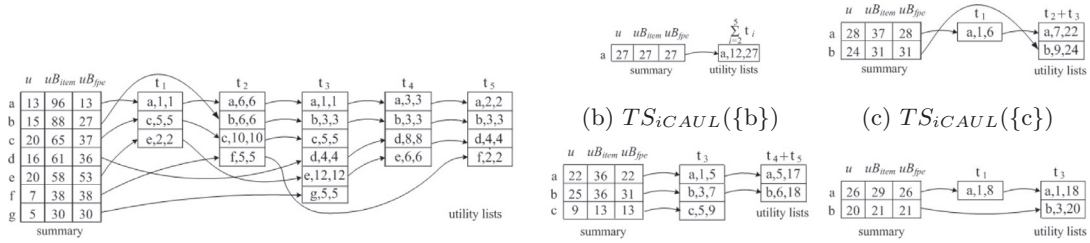
**Example 5.** When enumerating {c,e}, $\mu_b$ is raised to 27, and $uB_{fpe}(\{c,e\}) = u(\{a,c,e\}, t_1) + u(\{a,b,c,e\}, t_3) = 29 > \mu_b$. Thus, {c,e} cannot be pruned. However, {a,c,e} together with all its prefix extensions can be pruned and so can {b,c,e} since $uB_{fpe}(\{a,c,e\}) = 26 < \mu_b$ and $uB_{fpe}(\{b,c,e\}) = 21 < \mu_b$.

### 4.3. Computing utilities of enumerated patterns

For each enumerated pattern $X$, we compute the utility $u(\{i\}\cup X)$ and upper bounds $uB_{item}(i, X)$ and $uB_{fpe}(\{i\}\cup X)$ for every item $i \prec X$, which depends on the set $TS(X)$ of transactions that support $X$. How to represent and maintain $TS(X)$ is a key factor to the scalability and efficiency. We propose an improved version iCAUL of the data structure CAUL [24,25] to address this.

**iCAUL (improved Chain of Accurate Utility Lists)** for a pattern $X$ is a memory-resident structure to represent $TS(X)$. Concretely, $TS(X)$ by iCAUL, denoted as $TS_{iCAUL}(X)$, consists of 2 parts.

**Utility lists**: For each transaction $t \in TS(X)$, there is a utility list holding the utilities of all the items in $t$ relevant in growing prefix extensions of $X$. $\forall i \in fpe(X, t) \wedge i \prec X$, a quadruple ($i$, $u(i, t)$, $u(\{i\}\cup X, t)$, $link(i)$) is stored in the utility list in the imposed ordering $\Omega$. Moreover, utility lists made of an identical set of items are merged into one.




(a) $TS_{iCAUL}(\{\})$ (b) $TS_{iCAUL}(\{b\})$ (c) $TS_{iCAUL}(\{c\})$

(d) $TS_{iCAUL}(\{d\})$ (e) $TS_{iCAUL}(\{c, e\})$

**Fig. 1.** $TS_{iCAUL}$: the iCAUL representing transaction sets.

**A summary table**: For each distinct item $i$ relevant in growing prefix extensions of $X$, an entry $summary[i] = (u(\{i\} \cup X), uB_{item}(i, X), uB_{fpe}(\{i\} \cup X), link(i))$ is maintained in the table. The entries are arranged by the imposed ordering $\Omega$. The same items $i$ in different utility lists are threaded together by $link(i)$ starting from $summary[i]$. Clearly, the process of constructing $TS_{iCAUL}$ is the process of computing the utilities and bounds.

**Example 6.** Fig. 1a shows $TS_{iCAUL}(\{\})$ where the first list represents $t_1$ with its first element storing the item a, $u(a, t_1) = 1$, and $u(\{a\} \cup \{\}, t_1) = 1$, its second element storing the item c, $u(c, t_1) = 5$, and $u(\{c\} \cup \{\}, t_1) = 5$, and so on. The occurrences of the item a in all the five lists are threaded by $link$(a) starting from the summary entry $summary$[a]. The other components of $summary$[a] are $u(\{a\})$, $uB_{item}$(a, {}), and $uB_{fpe}(\{a\})$. Fig. 1b through Fig. 1e show the subsets of $TS_{iCAUL}(\{\})$, the $TS_{iCAUL}$ for {b}, {c}, {d}, and {c, e}, where $TS_{iCAUL}(\{d\})$ and $TS_{iCAUL}(\{c, e\})$ are used in Examples 4 and 5.

### 4.4. The baseline TONUP algorithm

This section proposes the baseline version of our algorithm, called *TOp-N high Utility Pattern mining* (TONUP), which works in three steps, as shown in Algorithm 1.

**Algorithm 1** TONUP(*D, XUT, n*) /* a high level description */ .

**Input:** database $D$, external utility table $XUT$, $n$
**Output:** $HUPset^*(n)$
1: $TS_{iCAUL}(\{\}).create(D, XUT, n)$;
2: $ShortList(n).update(u(\{i\}), \{i\})$ for each item $i$ in $TS_{iCAUL}(\{\})$;
3: enumPrefixExt({}, $TS_{iCAUL}(\{\})$, $ShortList(n)$);
4: **return** $ShortList(n)$;

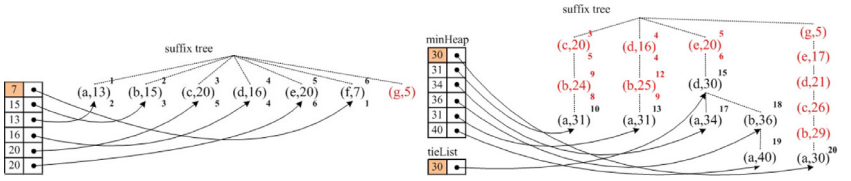
Step 1: TONUP creates $TS_{iCAUL}(\{\})$ by scanning the database $D$ and the external utility table $XUT$ to compute $u(\{i\})$, $uB_{item}(i, \{\})$, and $uB_{fpe}(\{i\})$ for each item $i$.

Step 2: TONUP initializes $ShortList(n)$ by utility $u(\{i\})$ of each item $i$ in $TS_{iCAUL}(\{\})$.

Step 3: TONUP invokes Algorithm 2 to enumerate the prefix extensions of the empty pattern {}, which works as follows. Given a pattern $X$, it enumerates each prefix extension $Y$ that is a concatenation of $X$ with an item $i \prec X$ in an imposed ordering $\Omega$ (Lines 1 to 2). According to Theorem 2, if the utility upper bound on $Y$ is less than $\mu_b$, the prefix extensions of $Y$ will not be enumerated by Pruning 2 (Line 3); otherwise, $TS_{iCAUL}(\{Y\})$ is projected from $TS_{iCAUL}(\{X\})$, $ShortList(n)$ is updated, and the prefix extensions of $Y$ will be recursively enumerated (Lines 4 to 6).

**Algorithm 2** enumPrefixExt($X$, $TS_{iCAUL}(X)$, $ShortList(n)$).

**Input:** $X$, $TS_{iCAUL}(X)$, $ShortList(n)$
**Output:** $ShortList(n)$
1: **for** each item $i$ in the summary table of $TS_{iCAUL}(X)$ by ordering $\Omega$ **do**
2: $Y \leftarrow \{i\} \cup X$;
3: **if** $uB_{fpe}(Y) \geq (\mu_b \leftarrow ShortList(n).\mu)$ **then**
4: $TS_{iCAUL}(Y) \leftarrow TS_{iCAUL}(X).project(Y)$;
5: $ShortList(n).update(u(\{j\} \cup Y), \{j\} \cup Y)$ for item $j$ in $TS_{iCAUL}(Y)$;
6: enumPrefixExt($Y$, $TS_{iCAUL}(Y)$, $ShortList(n)$);
7: **end if**
8: **end for**




(a) $ShortList(6)$: just initialized (b) $ShortList(6)$: the final result

**Fig. 2.** Shortlisted patterns by a suffix tree where the superscript and subscript indicate when a pattern pushed into and when popped from *ShortList*(*n*).

## 5. Opportunistic strategies and the full-strength TONUP

In this section, opportunistic strategies are proposed, for scalably maintaining shortlisted patterns, for efficiently computing utilities, for quickly raising the border threshold, and for dealing with extremely long patterns, to take every opportunity to improve the efficiency and scalability. Then, the implementation of our full-strength algorithm is discussed, and complexity analysis is presented.

### 5.1. Efficient strategy for maintaining the shortlisted patterns

Strategy 1 suggests maintaining the exact utilities in a minimum heap *minHeap* together with a tie list *tieList*. A naive way is to represent each shortlisted pattern as an array directly kept in *minHeap* or *tieList*, which is, however, inefficient. We propose to employ a suffix tree.

**Strategy 2** (SuffixTree - Suffix Tree to maintain patterns)**.** A suffix tree is employed to keep the shortlisted patterns. When a pattern $X$ is newly shortlisted, a new node $Node(X)$ will be created on the suffix tree with the path starting from $Node(X)$ to the null root representing the pattern $X$. The link to $Node(X)$ with the utility $u(X)$ is stored in *minHeap* or *tieList*.

Algorithm 3 will be called when a pattern $Z$ is enumerated. If $u(Z)$ is no less than the running border threshold, then the shortlisted patterns will be updated as follows. First, a new suffix tree node $Node(Z)$ will be created with the path from $Node(Z)$ to the root representing $Z$ (Line 2). Then, the pattern $Z$ will be shortlisted (Lines 3 to 11).

**Algorithm 3** *ShortList*(*n*).*update*($u(Z)$, $Z$)).

**Input:** $u(Z)$, $Z$, *minHeap*, *tieList*, the suffix tree
**Output:** *minHeap*, *tieList*, the suffix tree
1: **if** $u(Z) \geq (Border_{prev} \leftarrow minHeap.root.utility)$ **then**
2: create a suffix tree node $Node(Z)$ representing $Z$;
3: **if** *minHeap.isFull*() **then**
4: copy *minHeap.root* into *tieList*;
5: replace *minHeap.root* by $(u(Z), Node(Z))$ and heapify *minHeap.root*;
6: **if** $Border_{prev} < minHeap.root.utility$) **then**
7: delete each element from *tieList* and purge its suffix tree path;
8: **end if**
9: **else**
10: push $(u(Z), Node(Z))$ into *minHeap* and heapify *minHeap* if it is full;
11: **end if**
12: **end if**

**Example 7.** Given $n = 6$, Algorithm 1 enumerates a total of 27 patterns, 20 of which are shortlisted. Fig. 2a shows the suffix tree together with *minHeap* and *tieList* where $\mu_b = 7$ right after enumerating all distinct items. Fig. 2b shows the final top-*n* high utility patterns when Algorithm 1 ends, where {d, e} and {a, b, c, d, e, g} are tied for the 6th with $\mu_b = 30$.

### 5.2. Opportunistic strategy for projecting transaction sets

For a prefix extension $Y$ of a pattern $X$ with $Y = \{i\} \cup X$, $TS_{iCAUL}(Y)$ is projected from $TS_{iCAUL}(X)$ as in Algorithm 2 (Line 4), and $TS_{iCAUL}(Y)$ can be a pseudo or materialized projection of $TS_{iCAUL}(X)$.

**Pseudo projection**. $TS_{iCAUL}(Y)$ can share the same memory space [23] with $TS_{iCAUL}(X)$, where the utility lists of the pseudo $TS_{iCAUL}(Y)$ are delimited by following the chain threaded by the summary entry $link[i]$ for the item $i$ in $TS_{iCAUL}(X)$, and the summary entry for each item $j \prec i$ of the pseudo $TS_{iCAUL}(Y)$ is computed by scanning each delimited utility list.

**Materialized projection**. The materialized $TS_{iCAUL}(Y)$ is made by copying the pseudo $TS_{iCAUL}(Y)$ to memory space separate from $TS_{iCAUL}(X)$. According to Theorem 1, any item $j$ with $uB_{item}(j, Y) < \mu_b$ is left out from the materialized $TS_{iCAUL}(Y)$ by Pruning 1. Moreover, lists with an identical set of items are merged. Materialization has both benefit and drawback. On one hand, it is beneficial to the efficiency as irrelevant items are filtered out when enumerating the prefix extensions of $Y$. On the other hand, it incurs additional overhead for allocating memory space and copying transactions. In general, there is more benefit than overhead if many prefix extensions of $Y$ will be enumerated, which is more likely when the length of $Y$ is short and less likely when the length is long. Therefore, we propose to decide whether materializing a transaction set for a pattern based on the pattern length and the percentage of relevant items.

**Strategy 3** (AutoMaterial - Automatic Materialization in projecting TS)**.** When the percentage of relevant items in the pseudo projection is below a materialization threshold, $\phi$, a materialized copy will be made. And $\phi$ decreases with the increase of the pattern length $\ell$ as expressed by $\phi = \alpha^{\ell-1}$ with $0 \leq \alpha \leq 1$.

This strategy is incorporated in the procedure $TS_{iCAUL}(X).project(Y)$ at Line 4 in Algorithm 2. It is independent of data characteristics and, hence, is highly expected as it alleviates the burden on users, though the benefit and drawback also depend on data characteristics, i.e., pseudo projection is better on sparse datasets and materialized projection is better on dense datasets.

### 5.3. Dynamic strategy for raising border threshold and for pruning

Our TONUP algorithm enumerates patterns in a depth-first manner. The order of listing items has significant impact on the depth-first search process. Based on the following observations, we propose Strategy 4 that is incorporated into Algorithm 2 (Line 1).

- Roughly speaking, listing items in the descending order of $uB_{item}$ is in accordance with the descending order of utilities. Such an ordering increases the likelihood for early enumeration of patterns with large utilities and, hence, helps quickly raise $\mu_b$.
- If we list items in the proposed order, the enumeration of patterns containing items with low $uB_{item}$ will be delayed, which increases the likelihood for pruning such patterns without enumeration, as an item $i$ with low $uB_{item}$ is less relevant in growing top-$n$ high utility patterns.

**Strategy 4** (DynaDescend)**.** When enumerating each prefix extension $Y$ of a pattern $X$, **Dyna**mically resorting items $i$ in $TS_{iCAUL}(X)$ by the **Descend**ing order of the local $uB_{item}(i, X)$.

Although it incurs additional computational overhead to dynamically reorder items in each $TS_{iCAUL}$ by the local $uB_{item}$, the overhead is offset by the benefit of quickly raising the border threshold and the benefit of improving the pruning. This is different from the static ordering based on the global $uB_{item}$ in [24,25] and also different from the counterparts in [33,36].

### 5.4. Opportunistic strategy for a two-round approach

When there are extremely long patterns to be enumerated, keeping the shortlisted patterns on a suffix tree becomes computationally expensive, although more efficient than using arrays, since many long paths representing shortlisted patterns will first be created and then be purged. Note that the suffix tree keeps roughly $n$ paths dynamically by creating new paths and removing old paths. For such a case, we propose to shift to a two-round approach.

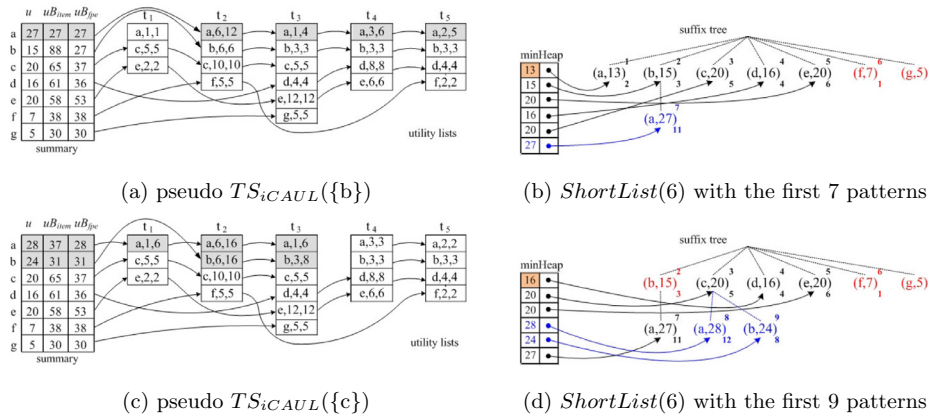
**Strategy 5** (OppoShift - Opportunistic Shift to a two-round approach)**.** When the average length of patterns enumerated by far is over a threshold $\eta$, the mining process will shift to a two-round approach. The first round will keep the utilities without the patterns and thus will only find the optimal threshold, $\mu^*(n)$, which will be fed into a second round to mine the patterns whose utilities are no less than $\mu^*(n)$.

When shifting to the two-round approach, both rounds can employ a closure property [22,24,25]. Concretely, when all the items in $TS_{iCAUL}(X)$ have the same support, we can compute the utility of each prefix extension of $X$ without enumeration, which helps improve the efficiency.

**Example 8.** If the threshold $\eta$ for shifting to a two-round approach is set to 0, i.e., employing the two-round approach from the very beginning, the first round enumerates 13 patterns to finally get $\mu^*(6) = 30$, and the second round enumerates 7 patterns to output all the top-6 high utility patterns.

### 5.5. Implementing the full-strength TONUP

Integrating the opportunistic strategies with Algorithm 1 results in our full-strength TONUP algorithm, which employs a suffix tree to maintain and shortlist enumerated patterns. Consequently, the implementation is mainly about constructing a suffix tree path to represent each enumerated pattern, maintaining a link to the path in *minHeap* or *tieList* if the pattern is shortlisted, and removing the path if the pattern and all of its prefix extensions are not shortlisted. Algorithm 1 is implemented as follows.




(a) pseudo $TS_{iCAUL}(\{b\})$ (b) $ShortList(6)$ with the first 7 patterns

(c) pseudo $TS_{iCAUL}(\{c\})$ (d) $ShortList(6)$ with the first 9 patterns

**Fig. 3.** Explanation of the implementation by the running example.

**Step 1**: TONUP creates the null root of the suffix tree to represent the empty pattern {}, and then creates $TS_{iCAUL}(\{\})$ by scanning the database $D$ twice. The first database scan is performed to compute $uB_{item}$ for each item in the summary table and to sort the items by $uB_{item}$. In the second database scan, the items of each transaction are sorted and maintained by a utility list. For instance, Fig. 1a shows $TS_{iCAUL}(\{\})$ created in Step 1 as detailed in Example 6.

**Step 2**: To initialize $ShortList(n)$, TONUP creates a child node for the suffix tree root to represent each of the $\omega$ items in the summary table and passes each child node to Algorithm 3 to update the shortlist. For instance, Fig. 2a shows the result of Step 2, as detailed in Example 7.

**Step 3**: TONUP calls Algorithm 2, enumPrefixExt, by passing the suffix tree root representing {} and $TS_{iCAUL}(\{\})$. Algorithm 2 is implemented as constructing and searching the subtree rooted at a given node $Node(X)$ representing a pattern $X$ in a depth-first manner. For a pattern $Y = \{i\} \cup X$ represented by a child node $Node(Y)$ of the given node $Node(X)$. If the upper bound on the utilities of prefix extensions of $Y$ is no less than the running threshold, then $Y$ is not pruned and Algorithm 2 does the following.

First, it gets $TS_{iCAUL}(\{Y\})$ from $TS_{iCAUL}(\{X\})$ by pseudo projection, and if necessary it materializes $TS_{iCAUL}(\{Y\})$ by Strategy 3 and sorts the items in materialized $TS_{iCAUL}(\{Y\})$ by Strategy 4 (Line 4). Second, it updates $ShortList(n)$ by first creating a child node $Node(Z)$ of $Node(Y)$ for each item $j$ in $TS_{iCAUL}(\{Y\})$ with $Z = \{j\} \cup Y$ and by passing to Algorithm 3 the utility of $Z$ and the pattern $Z$ represented by the link to $Node(Z)$ (Line 5). Finally, the recursion continues with $Y$ represented by $Node(Y)$ (Line 6).

**Example 9.** Step 3 starts with the null root in Fig. 2a. Nothing happens with the first child node (a, 13) as no item is before the item a by Ω. For the second child node (b, 15), the pseudo $TS_{iCAUL}(\{b\})$ in Fig. 3a is projected from $TS_{iCAUL}(\{\})$, whose materialized version is shown in Fig. 1b. Then, a grandchild node (a,27) is created and $ShortList(n)$ is updated accordingly, which raises $\mu_b$ to 13 in Fig. 3b. Similarly, nothing happens with the grandchild node (a,27). Step 3 continues with the third child node (c, 20) of the null root by projecting $TS_{iCAUL}(\{\})$ to the pseudo $TS_{iCAUL}(\{c\})$ in Fig. 3c, whose materialization is in Fig. 1c, by creating two child nodes (a, 28) and (b, 24) for the node (c,20) and by updating $ShortList(n)$ accordingly, which raises $\mu_b$ to 16 in Fig. 3d.

### 5.6. Complexity analysis

We analyze the time and space complexity of our algorithm, $Time(\texttt{TONUP})$ and $Space(\texttt{TONUP})$. First of all, let $d = |D|$ denote the number of transactions in the database, $\omega$ the number of items in $|I|$ (and in Ω), $\tau_a$ and $\tau_m$ the average and maximum lengths of a transaction, $\rho_a$ and $\rho_m$ the average and maximum lengths of a top-$n$ pattern, and $s_a$ and $s_m$ the average and maximum supports of an item, as shown in Table 2. Furthermore, let $\Psi$, $\ell_a$, and $\ell_m$ denote the number, the average length, and the maximum length of candidates. Obviously, $\rho_a \le \rho_m \le \tau_m$, $\ell_a \le \ell_m \le \tau_m$, $\tau_a \le \tau_m \le \omega$, $s_a \le s_m \le d$, and $\omega \le d \cdot \tau_a$.

**Time complexity**. According to Sections 4.4 and 5.5, the complexity of Algorithm 1 is as follows.

$Time(\text{Step 1}) = 2\ c_1 \cdot d \cdot \tau_a + c_2 \cdot \omega \cdot \log\omega + c_3 \cdot d \cdot \tau_a \cdot \log\tau_a$ where the first component is the time for two database scans with the constant $c_1$ for reading an item and computing its utility, and the second and the third for sorting items in the summary table and in all transactions, respectively, with $c_2$ and $c_3$ depending on the respective sorting procedures. Asymptotically, $Time(\text{Step 1}) = O(d \cdot \tau_a \cdot \log\tau_a + \omega \cdot \log\omega)$.

$Time(\text{Step 2}) = c_4 \cdot \omega \cdot \log n = O(\omega \cdot \log n)$ where $c_4 \cdot \log n$ is the time for one execution of Algorithm 3 with the constant $c_4$ depending the heapify operation, and $\omega$ is the number of times that Algorithm 3 is called.

$Time(\text{Step 3}) = \Psi \cdot (c_5 \cdot s_a \cdot \tau_a + c_4 \cdot \log n + c_6 \cdot \ell_a) = O(\Psi \cdot (s_a \cdot \tau_a + \log n + \ell_a))$ where the first factor, $\Psi$, is the number of times that Algorithm 2 is called, i.e., the total number of candidates (patterns) enumerated by TONUP, while the second

**Table 2**
Experimental datasets.

| Dataset | $\tau_a$ | $\tau_m$ | $\omega = \|I\|$ | top-10K to top-100K | | | $d = \|D\|$ | $s_m$ | $s_a$ | *Type* | *PatLen* | *Size* |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | *Density* | $\rho_a$ | $\rho_m$ | | | | | | |
| Connect | 43.0 | 43 | 129 | 1,839.1 | 15.9 | 21.8 | 67,557 | 67,473 | 22,695 | dense | medium | medium |
| Accidents | 33.8 | 51 | 468 | 1,798.5 | 8.5 | 14.8 | 340,183 | 340,151 | 24,575 | dense | short | medium |
| Pumsb | 74.0 | 74 | 2113 | 1,961.8 | 17.8 | 26.0 | 49,046 | 48,944 | 1,718 | dense | long | small |
| Mushroom | 23.0 | 23 | 119 | 1,202.9 | 9.1 | 15.8 | 8124 | 8,124 | 1,570 | dense | short | small |
| WV1 | 2.5 | 267 | 497 | 346.3 | 144.7 | 148.0 | 59,602 | 3,658 | 301 | mixed | very long | medium |
| BMS-POS | 6.5 | 164 | 1657 | 229.0 | 4.7 | 9.4 | 515,597 | 308,656 | 2,032 | mixed | short | large |
| WV2 | 4.6 | 161 | 3340 | 186.2 | 7.6 | 15.4 | 77,512 | 3,766 | 107 | mixed | short | medium |
| T40 | 39.6 | 77 | 942 | 239.8 | 9.5 | 17.7 | 100,000 | 28,738 | 4,204 | mixed | short | medium |
| Chainstore | 7.2 | 170 | 46,086 | 10.2 | 17.2 | 33.0 | 1,112,949 | 63,818 | 175 | sparse | long | large |
| Retail | 10.3 | 76 | 16,470 | 16.9 | 11.8 | 45.2 | 88,162 | 50,675 | 55 | sparse | medium | medium |
| Foodmart | 4.5 | 29 | 1559 | 36.2 | 21.3 | 27.0 | 34,015 | 143 | 105 | sparse | long | small |
| T20 | 20.0 | 49 | 1000 | 75.8 | 6.5 | 14.0 | 999,287 | 142,667 | 20,221 | sparse | short | large |

factor consists of the time for projecting a transaction set, the time for one execution of Algorithm 3, and the time for deleting a suffix tree path.

Therefore, $Time(\texttt{TONUP}) = Time(\text{Step 1}) + Time(\text{Step 2}) + Time(\text{Step 3}) = O(d \cdot \tau_a \cdot \log \tau_a + \omega \cdot (\log \omega + \log n) + \Psi \cdot s_a \cdot \tau_a + \Psi \cdot (\log n + \ell_a))$, where $\Psi \ll \sum_{l=1}^{\ell_m} \mathcal{C}(\omega, l) \ll 2^{\omega}$, $\ell_a \propto \rho_a$, and $\ell_m \propto \rho_m$, which depends on our pattern growth approach and Strategies 1, 4 and 5. While the constants $c_1$ through $c_4$ are not correlated to a strategy, $c_5$ is correlated to Strategy 3 and $c_6$ to Strategies 2 and 5.

**Space complexity**. The space required by TONUP has three parts. The first part is the space for $TS_{iCAUL}(\{\})$, which is created in Step 1 and persists to the end of TONUP. This part consists of a utility list for each transaction and a summary table of $\omega$ entries, whose space complexity is $O(d \cdot \tau_a + \omega) = O(d \cdot \tau_a)$.

The second is the space for *ShortList*($n$) initialized in Step 2 and kept on updating in Step 3. It consists of a minimum heap of size $n$ and a suffix tree of $n$ shortlisted patterns, whose space complexity is $O(n + n \cdot \ell_a) = O(n \cdot \ell_a)$.

The third part is the space for materializing the transaction sets of enumerated patterns in Step 3. Since there are at most $\ell_m$ transaction sets along a suffix tree path that are depth-first searched in Step 3, with a shrinking rate $\alpha$, the space complexity is at most $O(s_a \cdot \tau_a \cdot \sum_{l=0}^{\ell_m} \alpha^l)$.

Therefore, $Space(\texttt{TONUP}) = O(d \cdot \tau_a + n \cdot \ell_a + s_a \cdot \tau_a \cdot \sum_{l=0}^{\ell_m} \alpha^l) = O(d \cdot \tau_a + n \cdot \ell_a)$ since $\sum_{l=0}^{\ell_m} \alpha^l$ approaches a constant for $0 < \alpha < 1$ and $O(s_a \cdot \tau_a) < O(d \cdot \tau_a)$.

**Observations on complexity analysis**. First, selecting a large $\alpha$ for Strategy 3 makes a large memory footprint, but it results in a relatively small constant, $c_5$, to help improve efficiency. Second, selecting a large $\eta$ for Strategy 5 favours a one-round approach and also makes a large memory footprint, but it results in a relatively smaller $\Psi$. Third, $Time(\texttt{TONUP}) < Time(\texttt{TKO}) < Time(\texttt{TKU})$ since TONUP has much smaller $\Psi$, $\ell_a$, $\ell_m$, and constants due to its powerful pruning and efficient computation enabled by iCAUL. $Time(\texttt{TKU})$ is the highest as it has a component $O(\Psi \cdot \log \Psi)$ for sorting candidates in Phase II, while $\Psi$ by TKU is the largest as it cannot find the optimal threshold $\mu^*(n)$ in Phase I.

## 6. Experimental comparison of TONUP with state-of-the-art works

We evaluate our algorithm TONUP by comparing with the state-of-the-art top-$n$ high utility pattern mining algorithms TKU [33,36] and TKO [33]. We also compare with high utility pattern mining algorithms UP-Growth+ [32,34], HUI-Miner [26], EFIM [41], and d$^2$HUP [24,25] tuned with the optimal threshold $\mu^*(n)$, denoted UP-Growth+$_{op}$, HUI-Miner$_{op}$, EFIM$_{op}$, and d$^2$HUP$_{op}$, respectively. The code of d$^2$HUP, EFIM, and HUI-Miner are provided by the original authors. But, due to unavailability, UP-Growth+, TKU, and TKO are implemented by us. We have greatly improved Phase II [25] in our implementation of UP-Growth+ and TKU. Our implementation of TKO adapts and improves the code of HUI-Miner[1], and only outputs $\mu^*(n)$ without the top-$n$ patterns. To some extent, the running time of UP-Growth+, TKU, and TKO is under-estimated.

Twelve datasets are used in the experiments. They are summarized by Table 2 where the parameters $\tau_a$, $\tau_m$, $\omega$, $\rho_a$, $\rho_m$, $d$, $s_m$, and $s_a$ are defined in Section 5.6, *Density* is the average number of top-$n$ patterns per item, *Type* is a categorization by *Density*, *PatLen* is a categorization by $\rho_a$, and *Size* is a categorization by $d$. And WV1, WV2, T20, and T40 stand for the datasets BMS-WebView-1, BMS-WebView-2, T20I1KD1M, and T40I1KD100K, respectively. Most of the datasets are from the ML Repository[2] or the FIMI Repository[3] without utility information, and we generate the utility information by following

[1] http://philippe-fournier-viger.com/.
[2] http://www.ics.uci.edu/~mlearn/MLRepository.html.
[3] http://fimi.ua.ac.be/data/.

**Table 3**
Highlighting the experimental comparisons shown in Fig. 4.

| Dataset | $n$ | running time (s) by three top-$n$ algs & four optimal algs | | | | | | | $\Psi$ = candidates( $\times 10^3$) | | | memory ftpt (MB) | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | TONUP | TKO | TKU | UP-G+$_o$ | HUI-M$_o$ | EFIM$_o$ | d$^2$HUP$_o$ | TONUP | TKO | TKU | TONUP | TKO | TKU |
| Connect | 50 | 6 | 5960 | – | – | 29 | 3 | 3 | 6 | 773 | – | 89 | 275 | – |
| Connect | 1K | 6.2 | – | – | – | 79 | 3 | 3 | 43 | – | – | 89 | – | – |
| Accidents | 5K | 50 | 5076 | – | 2622 | 72 | 16 | 60 | 22 | 7K | – | 385 | 569 | – |
| Pumsb | 10K | 13 | – | – | – | 3381 | 111 | 24 | 362 | – | – | 117 | – | – |
| Mushroom | 10K | 0.4 | 32 | 183 | 13 | 0.95 | 0.24 | 0.27 | 73 | 1.2K | 4.5K | 7 | 10 | 137 |
| WV1 | 5K | 11.2 | – | – | – | – | 2348 | 1.2 | 11K | – | – | 5 | – | – |
| WV1 | 10K | 19.5 | – | – | – | – | 2951 | 1.5 | 20K | – | – | 7 | – | – |
| BMS-POS | 5K | 15 | 3297 | 99 | 20 | 10 | 18 | 14 | 19 | 91K | 386 | 95 | 83 | 62 |
| WV2 | 10K | 0.46 | – | 8.35 | 1.88 | 2.45 | 3.9 | 0.56 | 46 | – | 1.5K | 11 | – | 51 |
| T40 | 10K | 26 | 876 | 250 | 102 | 67 | 27 | 9 | 283 | 1.4K | 4.1K | 127 | 100 | 136 |
| Chainstore | 10K | 22 | 1928 | 590 | 130 | 709 | 767 | 20 | 60 | 26K | 359 | 237 | 185 | 237 |
| Retail | 10K | 2.6 | 131 | – | 5.1 | 29 | 31 | 2.3 | 55 | 11K | – | 29 | 23 | – |
| Foodmart | 10K | 0.5 | 4.4 | – | 31 | 2.5 | 1.1 | 0.33 | 264 | 160 | – | 6.9 | 4.4 | – |
| T20 | 10K | 126 | 1110 | 383 | 200 | 244 | 216 | 71 | 181 | 211 | 754 | 612 | 467 | 617 |

[27] while Chainstore is from [27] and Foodmart is the sample database shipped with the Microsoft Analysis Service[4]. The datasets and our TONUP code are ready for download[5].

The experiments were performed on a workstation with a 2.40 GHz CPU and 4 GB memory, running CentOS 6.3. The default setting for automatic materialization and for opportunistic shift are $\alpha = 0.85$ and $\eta = 20$, respectively, according to the complexity analysis in Section 5.6 and the sensitivity analyses in Sections 7.3 and 7.5. The mining parameter $n$ changes from 100 to 100K, or from 1 if TKU and TKO cannot run for $n \geq 100$.

### 6.1. Comparison in running time

Fig. 4a shows the running time for mining top-$n$ high utility patterns by our TONUP algorithm, TKO, and TKU with changing $n$, as well as the running time for mining high utility patterns by UP-Growth+$_{op}$, HUI-Miner$_{op}$, EFIM$_{op}$[6], and d$^2$HUP$_{op}$ that are tuned with the optimal threshold $\mu^*(n)$, which is also highlighted by the first nine columns of Table 3.

On dense datasets as in Fig. 4a(i)–(iv) and in the first five rows of Table 3, TONUP is up to 3 orders of magnitude more efficient than TKO and TKU, and even up to 2 orders of magnitude more efficient than UP-Growth+$_{op}$ and HUI-Miner$_{op}$. EFIM$_{op}$ is up to 3 times faster than TONUP and d$^2$HUP$_{op}$ when the patterns are short. On the contrary, TONUP is up to 1 order of magnitude faster than EFIM$_{op}$ when the patterns are long. On mixed datasets, TONUP is up to 2 orders of magnitude more efficient than TKO and TKU, and up to 1 order of magnitude more efficient than UP-Growth+$_{op}$, HUI-Miner$_{op}$, and EFIM$_{op}$. On sparse datasets, TONUP is up to 1 order of magnitude more efficient than TKO and TKU, and up to 1 order of magnitude more efficient than UP-Growth+$_{op}$, HUI-Miner$_{op}$, and EFIM$_{op}$.

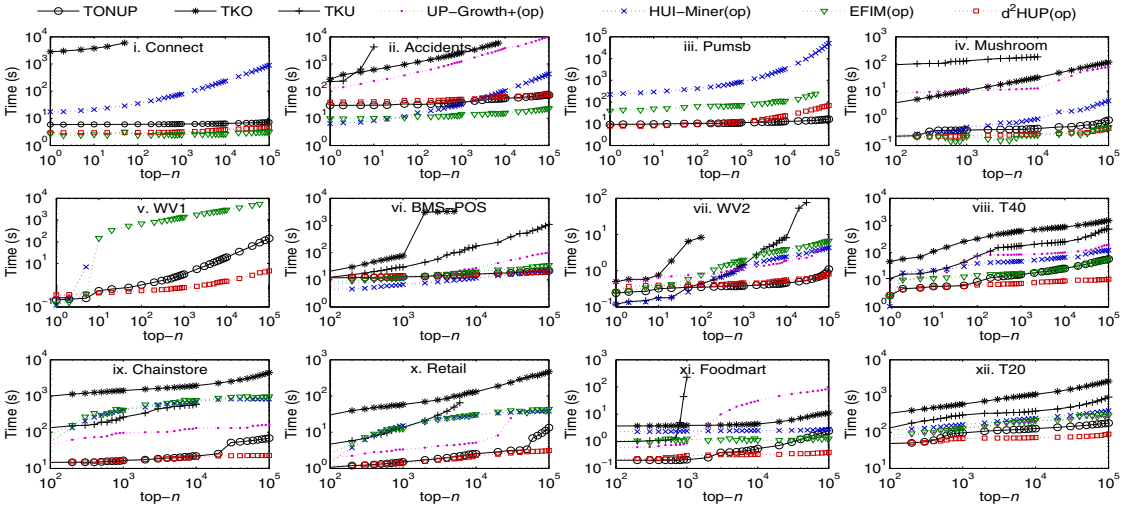
We have three observations in the experimental results. First, TONUP is up to 1 to 3 orders of magnitude more efficient than TKO and TKU. One reason is that using a suffix tree to maintain shortlisted patterns (Strategy 2) and opportunistically shifting to a two-round approach (Strategy 5) are effective for mining top-$n$ high utility patterns. Second, TONUP is 1 to 2 orders of magnitude faster than HUI-Miner$_{op}$, UP-Growth+$_{op}$, and EFIM$_{op}$ when the optimal threshold $\mu^*(n)$ unknown to TONUP is given to the latter algorithms. One exception is that EFIM$_{op}$ beats TONUP and d$^2$HUP$_{op}$ on dense datasets with short patterns. Third, TONUP is just a little slower than d$^2$HUP$_{op}$, usually within a factor of 2 to 3. This confirms that the data structure iCAUL and Strategies 3 (AutoMaterial) and 4 (DynaDescend) improve efficiency.

### 6.2. Comparison in number of candidates

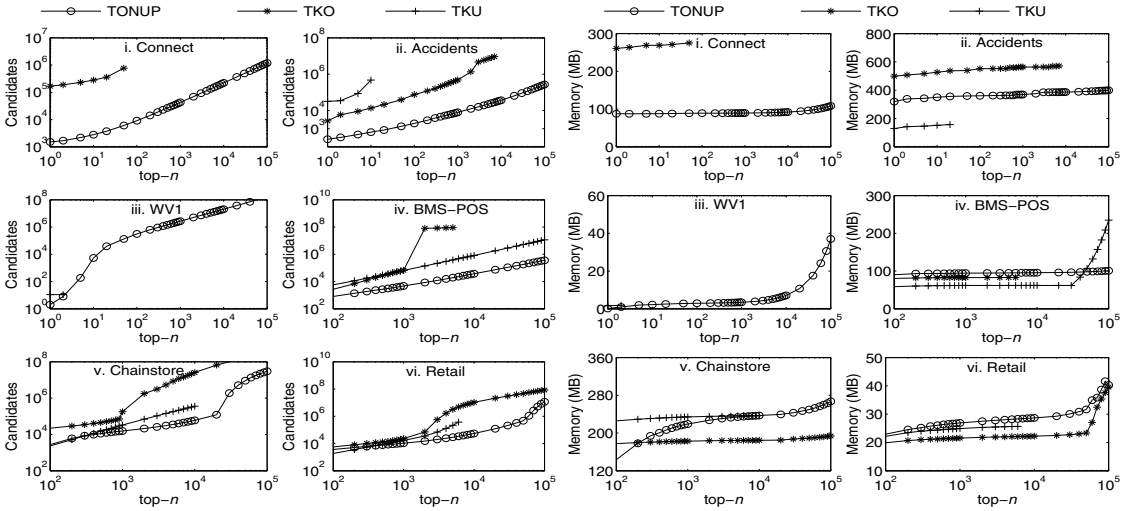
The number $\Psi$ of candidates enumerated by our TONUP algorithm, TKO, and TKU is shown in Fig. 4b and in the tenth through twelfth columns of Table 3. On dense datasets as in Fig. 4b(i)–(ii) and in the first five rows of Table 3, TKU enumerates the most candidates, and TKO and TKU enumerate 1 to 2 orders of magnitude more than TONUP. On mixed datasets, TKO and TKU enumerate 1 to 2 orders of magnitude more candidates than TONUP. On a sparse dataset, TKO enumerates the most candidates, and TKO and TKU enumerate 1 to 2 orders of magnitude more than TONUP.

In short, TONUP enumerates 1 to 2 orders of magnitude fewer candidates than TKO and TKU on all datasets. The reason is that using the exact utilities instead of the estimates to raise the border threshold (Strategy 1) and dynamically sorting items in the descending order of local $uB_{item}$ (Strategy 4) are much more effective than the counterparts in TKO and TKU. The latter algorithms employ either an improper ordering or an irrelevant ordering measure.

[4] http://www.dagira.com/tips/foodmart_download/.
[5] http://kddlab.zjgsu.edu.cn:7200/implementations.html/.
[6] Experiment comparing EFIM and d$^2$HUP in [41] employed an improper implementation of d$^2$HUP without two key techniques.

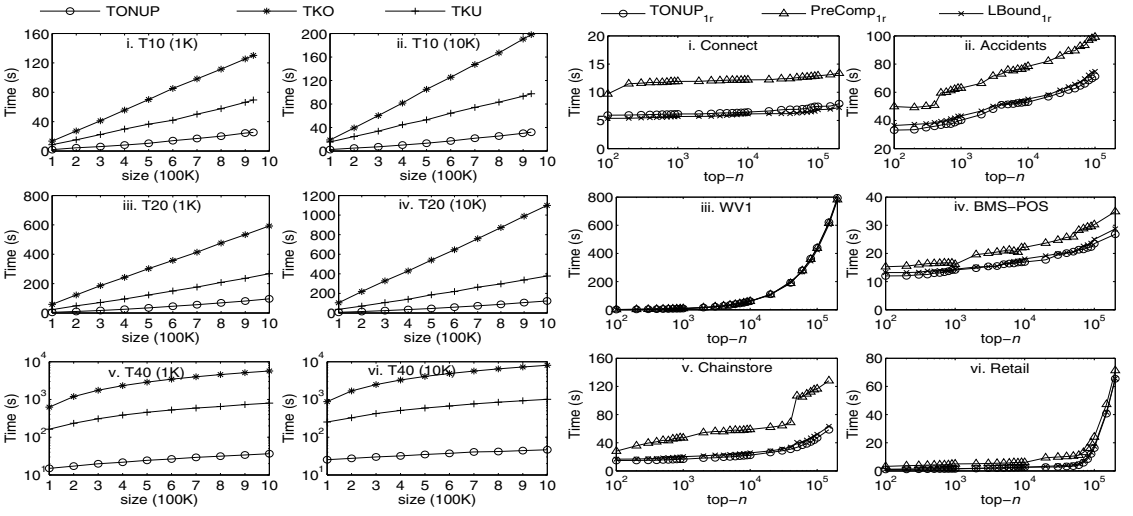



(a) TONUP, TKO, and TKU with changing $n$, and algorithms with $\mu^*(n)$, in running time




(b) TONUP, TKO, TKU in num Ψ of candidates

(c) TONUP, TKO, TKU in memory footprint




(d) Scalability

(e) Strategy 1 ExactBorder

**Fig. 4.** Running time, candidate number, memory footprint, scalability, and Strategy 1.

### 6.3. Comparison in memory footprint

The memory footprints of our TONUP algorithm, TKO, and TKU are shown in Fig. 4c and in the last three columns of Table 3. On dense datasets, TONUP uses the lowest amount of memory, and TKU uses the highest and often runs out of memory. On mixed datasets, TONUP uses the lowest, and TKU uses the highest. On sparse datasets, TKO uses the lowest, and TKU still uses the highest.

In short, TKU uses the highest amount of memory when $n$ is large on all datasets. TONUP uses the lowest on dense and mixed datasets, and TKO uses the lowest on sparse datasets. While TONUP is the most efficient, its memory footprint is reasonably small, due to Strategy 3 for balancing the efficiency and scalability.

### 6.4. Comparison in scalability

We evaluate the scalability of the three algorithms by performing experiments on three datasets T10, T20, and T40 with the size of each dataset varying from 100K to 1000K and for the top-$n$ parameter $n = 1$K and $n = 10$K, respectively. The result is shown in Fig. 4d.

TONUP is the most scalable algorithm, and TKO is less scalable than TKU. Concretely, the running time of TONUP in every setting is the shortest. For example, on the dataset T10 with size 500K, TONUP takes 11s, TKO 70s, and TKU 37s for $n = 1$K; TONUP takes 13s, TKO 105s, and TKU 45s for $n = 10$K. Most importantly, the slope of the curve of TONUP is the smallest in every setting.

## 7. Experimental analysis of the TONUP algorithm

This section further analyzes the five new strategies and an improved data structure that enable our opportunistic pattern growth approach. While TONUP is integrated with all strategies, $\texttt{TONUP}_{1r}$ disables the OppoShift strategy, i.e., sticking to the one-round approach.

### 7.1. Analysis of the ExactBorder strategy

We analyze the effectiveness of the ExactBorder strategy (Strategy 1) by pushing into $\texttt{TONUP}_{1r}$ the strategies for raising the border threshold by pre-computing utilities of item pairs and by estimating utility lower bounds, denoted as $\texttt{PreComp}_{1r}$ and $\texttt{LBound}_{1r}$ in Fig. 4e, respectively. $\texttt{PreComp}_{1r}$ takes 30% to 2 times more time than $\texttt{TONUP}_{1r}$, and $\texttt{LBound}_{1r}$ takes marginally more time than $\texttt{TONUP}_{1r}$ for most cases, as shown in Fig. 4e. For example, on Connect, $\texttt{LBound}_{1r}$ takes 7.3s, $\texttt{TONUP}_{1r}$ 7.9s, and $\texttt{PreComp}_{1r}$ 13.4s for $n = 200$K. On BMS-POS, $\texttt{LBound}_{1r}$ takes 29s, $\texttt{TONUP}_{1r}$ 27s, and $\texttt{PreComp}_{1r}$ 35s for $n = 200$K. On Chainstore, $\texttt{LBound}_{1r}$ takes 52s, $\texttt{TONUP}_{1r}$ 47s, and $\texttt{PreComp}_{1r}$ 117s for $n = 100$K.

It is confirmed that pre-computing utilities of item pairs ($\texttt{PreComp}_{1r}$) or estimating lower bounds ($\texttt{LBound}_{1r}$) does not help much in raising the border threshold. Our basic strategy of raising the border threshold by exact utilities is effective, and the four strategies in TKU and one strategy in TKO that the raise border threshold by estimating utility lower bounds do not work well.

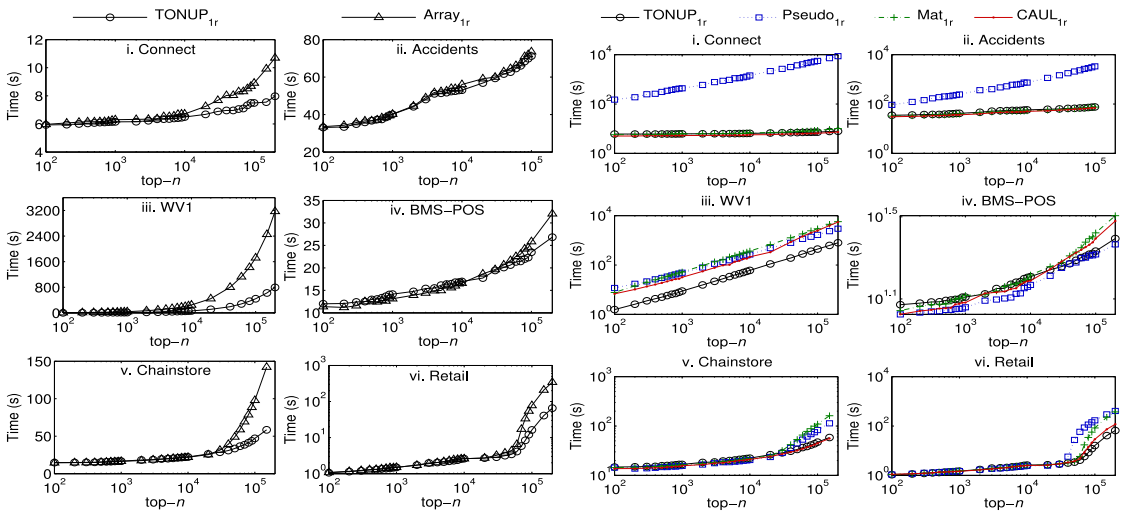
### 7.2. Analysis of the SuffixTree strategy

We analyze the SuffixTree strategy (Strategy 2) by comparing $\texttt{TONUP}_{1r}$ with $\texttt{Array}_{1r}$ that maintains shortlisted patterns in arrays. $\texttt{TONUP}_{1r}$ is up to 20% to 4 times faster than $\texttt{Array}_{1r}$, as shown in Fig. 5a. For example, on Connect, $\texttt{TONUP}_{1r}$ takes 7.9s and $\texttt{Array}_{1r}$ 10.7s for $n = 200$K. On WV1, $\texttt{TONUP}_{1r}$ takes 60s and $\texttt{Array}_{1r}$ 246s for $n = 10$K. On BMS-POS, $\texttt{TONUP}_{1r}$ takes 27s and $\texttt{Array}_{1r}$ 32s for $n = 200$K. On Chainstore, $\texttt{TONUP}_{1r}$ takes 47s and $\texttt{Array}_{1r}$ 97s for $n = 100$K. On Retail, $\texttt{TONUP}_{1r}$ takes 16s and $\texttt{Array}_{1r}$ 78s for $n = 100$K.

The observation is that maintaining shortlisted patterns by a suffix tree instead of arrays improves the efficiency significantly. There is no counterpart in TKO and TKU. Moreover, TONUP considers the ties for the $n$-th greatest utility, while ties are ignored in TKO and TKU.

### 7.3. Analysis of the AutoMaterial strategy

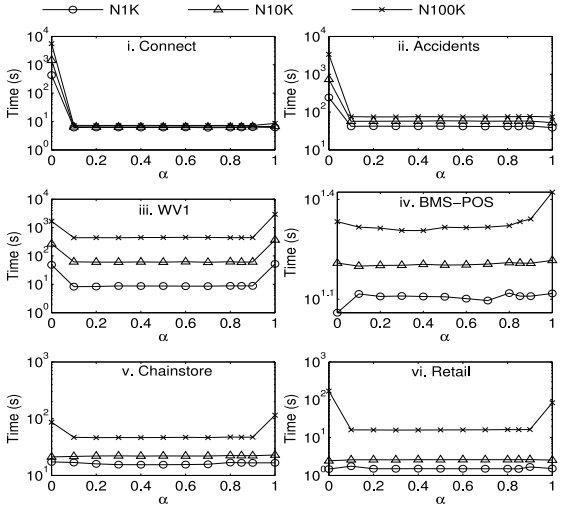
We analyze the AutoMaterial strategy (Strategy 3) by comparing $\texttt{TONUP}_{1r}$ with $\texttt{Pseudo}_{1r}$ that only uses pseudo projections and $\texttt{Mat}_{1r}$ that only uses materialized projections. We also analyze the improved data structure iCAUL by comparing with $\texttt{CAUL}_{1r}$ that uses CAUL [24,25] instead of iCAUL. The result is shown in Fig. 5b. On dense datasets, $\texttt{TONUP}_{1r}$, $\texttt{Mat}_{1r}$, and $\texttt{CAUL}_{1r}$ have the same performance. $\texttt{Pseudo}_{1r}$ is 2 to 3 orders of magnitude slower than the others. For example on Connect, $\texttt{TONUP}_{1r}$ takes 6.4s , $\texttt{Pseudo}_{1r}$ 1,435s, $\texttt{Mat}_{1r}$ 6.8s, and $\texttt{CAUL}_{1r}$ 5.7s for $n = 10$K. On mixed datasets, $\texttt{TONUP}_{1r}$ is up to 6 times faster than others. $\texttt{Pseudo}_{1r}$ is up to 2 times faster than $\texttt{Mat}_{1r}$, and $\texttt{CAUL}_{1r}$ is marginally faster than $\texttt{Mat}_{1r}$. For example on WV1, $\texttt{TONUP}_{1r}$ takes 60s, $\texttt{Pseudo}_{1r}$ 266s, $\texttt{Mat}_{1r}$ 366s, and $\texttt{CAUL}_{1r}$ 214s for $n = 10$K. On sparse datasets, $\texttt{TONUP}_{1r}$ is up to 1 order of magnitude faster than $\texttt{Pseudo}_{1r}$, up to 5 times faster than $\texttt{Mat}_{1r}$, and up to 80% faster than $\texttt{CAUL}_{1r}$. For example on Retail, $\texttt{TONUP}_{1r}$ takes 16s, $\texttt{Pseudo}_{1r}$ 170s, $\texttt{Mat}_{1r}$ 84s, and $\texttt{CAUL}_{1r}$ 29s for $n = 100$K.



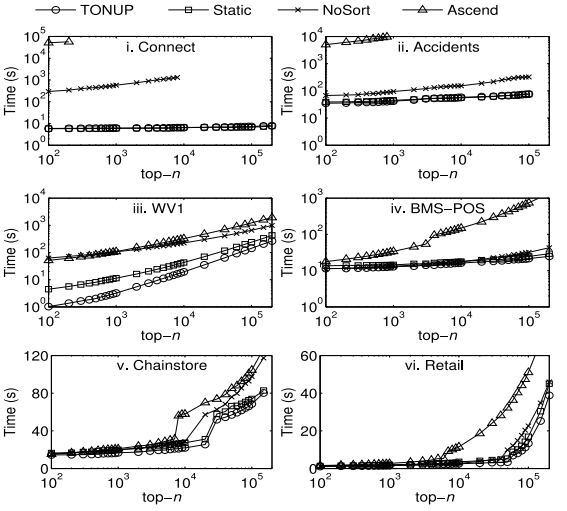

(a) Strategy 2 SuffixTree



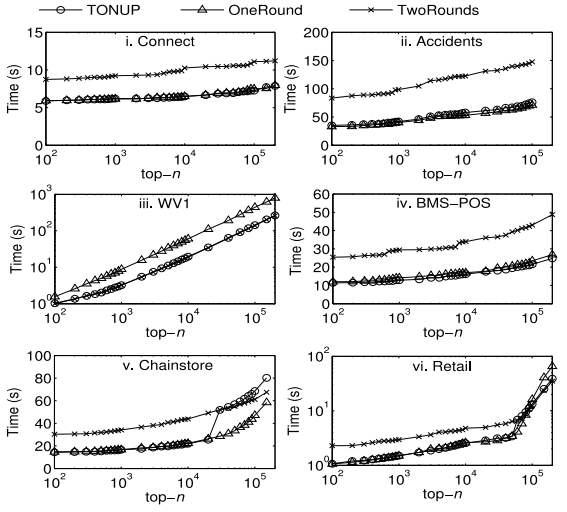

(b) Strategy 3 AutoMaterial



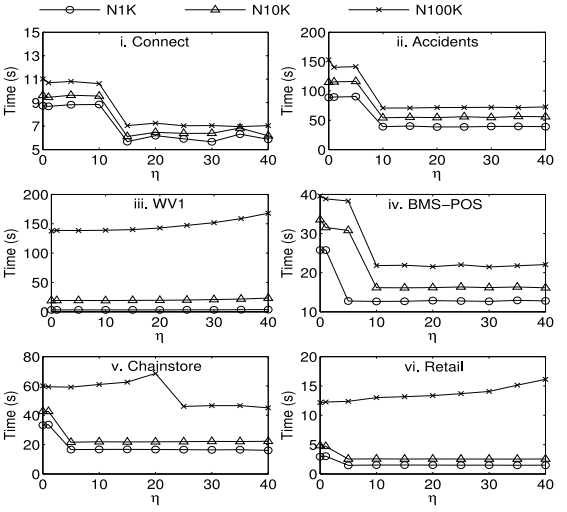

(c) Parameter $\alpha$ for Strategy 3




(d) Strategy 4 DynaDescend




(e) Strategy 5 OppoShift




(f) Parameter $\eta$ for Strategy 5

**Fig. 5.** Evaluating Strategies 2 through 5, and analyzing parameter sensitivity.

There are two observations. First, the AutoMaterial strategy makes a balance between pseudo projection and materialization and thus helps improve the efficiency by several orders of magnitude as neither pseudo projection nor materialized projection consistently outperforms the other. Second, the improved data structure iCAUL also helps improve the efficiency significantly. This also explains why TONUP outperforms $d^2HUP_{op}$ when both are tuned with the optimal threshold.

**Sensitivity and Default Value of Parameter $\alpha$.** We analyze the parameter $\alpha$, which is used to calibrate Strategy 3, by performing experiments evaluating $TONUP_{1r}$ with changing $\alpha$ in the range [0, 1] for $n = 1K$, 10K, and 100K, respectively. Fig. 5c shows the result. In dense datasets, the running time of $TONUP_{1r}$ decreases with the increase of $\alpha$ and becomes insensitive for $\alpha \geq 0.1$. In sparse and mixed datasets, the running time for $\alpha$ in the range [0.1, 0.9] approaches the shortest and changes little. Therefore, a large value in the range [0.1, 0.9], say 0.85, is a good choice for the default value of $\alpha$.

### 7.4. Analysis of the DynaDescend strategy

We analyze the DyanDescend strategy (Strategy 4) by comparing TONUP with Ascend that sorts items in the ascending order of $uB_{item}$, Static in the descending order of the global $uB_{item}$, and NoSort in the lexicographic order. TONUP is up to 4 orders of magnitude faster than Ascend, up to 2 orders of magnitude faster than NoSort, and up to 100% faster than Static, as shown in Fig. 5d. For example, on Connect, Ascend takes 56,979s, NoSort 342s, Static 5.8s, and TONUP 5.9s for $n = 200$. On Accidents, Ascend takes 10,127s, NoSort 89s, Static 45s, and TONUP 41s for $n = 900$. On WV1, Ascend takes 108s, NoSort 106s, Static 11s, and TONUP 3.2s for $n = 1K$. On Chainstore, Ascend takes 105s, NoSort 98s, Static 74s, and TONUP 69s for $n = 100K$.

This set of experiments confirms that dynamically sorting items in the descending order of the local $uB_{item}$ helps raise the border threshold quickly and helps prune the search space. Notice that TKU sorts items in the ascending order of *TWUs* or UP-Tree path utilities, which does not help quickly raise the border threshold, and TKO sorts items in the descending order of a measure similar to our $uB_{fpe}$ rather than $uB_{item}$, which is not effective either.

### 7.5. Analysis of the OppoShift strategy

We compare TONUP that embodies the OppoShift strategy (Strategy 5) with OneRound a.k.a. $TONUP_{1r}$ and TwoRounds that always takes a two-round approach. Fig. 5e shows the result.

When OneRound is faster than TwoRounds, TONUP is as efficient as OneRound and is 70% to 100% more efficient than TwoRounds. For example, on Connect, OneRound takes 7.1s, TONUP 7.5s, and TwoRounds 11.2s for $n = 100K$. On BMS-POS, OneRound takes 23.5s, TONUP 21.5s, and TwoRounds 43s for $n = 100K$.

When TwoRounds is faster than OneRound, TONUP is as efficient as TwoRounds and is up to 3 times faster than OneRound. For example, on WV1, OneRound takes 795s, TONUP 264s, and TwoRounds 259s for $n = 200K$. On Retail, OneRound takes 16s, TONUP 13s, and TwoRounds 12.4s for $n = 100K$.

**Sensitivity and Default Value of Parameter $\eta$.** We analyze the parameter $\eta$, which is used to calibrate Strategy 5, by performing experiments evaluating TONUP with changing $\eta$ for $n = 1K$, 10K, and 100K, respectively. In most cases, as shown in Fig. 5f, the running time of TONUP decreases with the increase of $\eta$ and becomes insensitive for $\eta \geq 10$, i.e., there is little change in the running time for $\eta \geq 10$. Therefore, any value no less than 10, say 20, is a good default value for $\eta$.

## 8. Conclusion and future work

This paper proposes a novel algorithm TONUP for mining top-$n$ high utility patterns that are very long. The baseline TONUP adopts an opportunistic pattern growth approach that grows patterns as prefix extensions, shortlists patterns whose utilities are the first $n$ greatest, and prunes the search space by utility upper bounding. The full-strength TONUP proposes five opportunistic strategies for scalably maintaining shortlisted patterns, for efficiently computing utilities and estimating upper bounds, and for improving pruning. TONUP is up to 1 to 3 orders of magnitude more efficient and is more scalable than the state-of-the-art algorithms TKU and TKO. Surprisingly, TONUP is even 1 to 2 orders of magnitude faster than high utility pattern mining algorithms UP-Growth+, HUI-Miner, and EFIM that are tuned with an optimal threshold unknown to TONUP.

The proposed algorithm, TONUP, heavily relies on memory resident structures and will eventually confront the scalability issue. In this regard, an interesting future work is to propose parallel and distributed algorithms for handling big data. TONUP only works for static data, and the state-of-the-art algorithms for dynamic data consider either insertion or deletion of transactions but not both. Therefore, another future work is to propose an incremental algorithm to mine (top-$n$) high utility patterns for dynamic data with both insertion and deletion. In the future, we will also extend the top-$n$ pattern mining model by introducing more interestingness measures and by applying multiple measures.

## Acknowledgments

This work was supported in part by the National Natural Science Foundation of China [grant number 61272306], the Zhejiang Provincial Natural Science Foundation of China [grant number LY17F020004], and the Australian Research Council [grant number DP140103617].

## References


[1] F.N. Afrati, A. Gionis, H. Mannila, Approximating a collection of frequent sets, in: Proceedings of the Tenth ACM SIGKDD International Conference Knowledge Discovery and Data Mining, Seattle, Washington, USA, 2004, pp. 12–19.
[2] R. Agarwal, C. Aggarwal, V. Prasad, Depth first generation of long patterns, in: Proceedings of the Sixth ACM SIGKDD International Conference Knowledge Discovery and Data Mining, Boston, MA, USA, 2000, pp. 108–118.
[3] R. Agarwal, C. Aggarwal, V. Prasad, A tree projection algorithm for generation of frequent item sets, J. Parallel Distrib. Comput. 61 (3) (2001) 350–371.
[4] R. Agrawal, R. Srikant, Fast algorithms for mining association rules, in: Proceedings of the Twentieth International Conference Very Large Data Bases, Santiago de Chile, Chile, 1994, pp. 487–499.
[5] A.U. Ahmed, C.F. Ahmed, M. Samiullah, N. Adnan, C.K.-S. Leung, Mining interesting patterns from uncertain databases, Inf. Sci. (Ny) 354 (2016) 60–85.
[6] C.F. Ahmed, S.K. Tanbeer, B.S. Jeong, Y.K. Lee, Efficient tree structures for high utility pattern mining in incremental databases, IEEE Trans. Knowl. Data Eng. 21 (12) (2009) 1708–1721.
[7] R. Bayardo, Efficiently mining long patterns from databases, in: Proceedings of the ACM SIGMOD International Conference Management of Data, Seattle, Washington, USA, 1998, pp. 85–93.
[8] D. Burdick, M. Calimlim, J. Gehrke, MAFIA: A maximal frequent item set algorithm for transactional databases, in: Proceedings of the Seventeenth IEEE International Conference Data Engineering, Heidelberg, Germany, 2001, pp. 443–452.
[9] C.H. Cai, A.W.C. Fu, C.H. Cheng, W.W. Kwong, Mining association rules with weighted items, in: Proceedings of the IEEE International Database Engineering and Applications Symposium, 1998, pp. 68–77.
[10] R. Chan, Q. Yang, Y. Shen, Mining high utility itemsets, in: Proceedings of the Third IEEE International Conference Data Mining, Melbourne, Florida, USA, 2003, pp. 19–26.
[11] H. Chen, L. Shu, J. Xia, Q. Deng, Mining frequent patterns in a varying-size sliding window of online transactional data streams, Inf. Sci. (Ny) 215 (2012) 15–36.
[12] G. Cong, K.L. Tan, A.K.H. Tung, X. Xu, Mining top-*k* covering rule groups for gene expression data, in: Proceedings of the ACM SIGMOD International Conference Management of Data, Baltimore, Maryland, USA, 2005, pp. 670–681.
[13] A. Erwin, R.P. Gopalan, N.R. Achuthan, Efficient mining of high utility item sets from large datasets, in: Proceedings of the Twelfth Pacific-Asia Conference Knowledge Discovery and Data Mining, Osaka, Japan, 2008, pp. 554–561.
[14] P. Fournier-Viger, V.S. Tseng, Mining top-*k* sequential rules, in: Proceedings of the Seventh International Conference Advanced Data Mining and Applications, Beijing, China, 2011, pp. II:180–194.
[15] P. Fournier-Viger, V.S. Tseng, Mining top-*k* non-redundant association rules, in: Proceedings of the International Symposium Methodologies for Intelligent Systems, Macau, China, 2012, pp. 31–40.
[16] A.W.-C. Fu, R.W.-W. Kwong, J. Tang, Mining *n*-most interesting itemsets, in: Proceedings Twelfth International Symposium Methodologies for Intelligent Systems, Charlotte, NC, USA, 2000, pp. 59–67.
[17] J. Han, J. Pei, Y. Yin, Mining frequent patterns without candidate generation, in: Proceedings of the ACM SIGMOD International Conference Management of Data, Dallas, Texas, USA, 2000, pp. 1–12.
[18] R.J. Hilderman, C.L. Carter, H.J. Hamilton, N. Cercone, Mining market basket data using share measures and characterized itemsets, in: Proceedings of the Second Pacific-Asia Conference Knowledge Discovery and Data Mining, Melbourne, Australia, 1998, pp. 159–170. April
[19] Y.C. Li, J.S. Yeh, C.C. Chang, Isolated items discarding strategy for discovering high utility itemsets, Data Knowl. Eng. 64 (1) (2008) 198–217.
[20] M.-Y. Lin, T.-F. Tu, S.-C. Hsueh, High utility pattern mining using the maximal itemset property and lexicographic tree structures, Inf. Sci. (Ny) 215 (2012) 1–14.
[21] T.Y. Lin, Y.Y. Yao, E. Louie, Value added association rules, in: Proceedings of the Sixth Pacific-Asia Conference Knowledge Discovery and Data Mining, Taipei, Taiwan, 2002, pp. 328–333.
[22] H. Liu, X. Wang, J. He, J. Han, D. Xin, Z. Shao, Top-down mining of frequent closed patterns from very high dimensional data, Inf. Sci. (Ny) 179 (2009) 899–924.
[23] J. Liu, Y. Pan, K. Wang, J. Han, Mining frequent item sets by opportunistic projection, in: Proceedings of the ACM SIGKDD International Conference Knowledge Discovery and Data Mining, Edmonton, Alberta, Canada, 2002, pp. 229–238.
[24] J. Liu, K. Wang, B.C.M. Fung, Direct discovery of high utility itemsets without candidate generation, in: Proceedings of the Twelfth IEEE International Conference Data Mining, Brussels, Belgium, 2012, pp. 984–989.
[25] J. Liu, K. Wang, B.C.M. Fung, Mining high utility patterns in one phase without generating candidates, IEEE Trans. Knowl. Data Eng. 28 (5) (2016) 1245–1257.
[26] M. Liu, J. Qu, Mining high utility itemsets without candidate generation, in: Proceedings of the Twenty First ACM International Conference Information and Knowledge Management, Maui, HI, USA, 2012, pp. 55–64.
[27] Y. Liu, W. Liao, A. Choudhary, A fast high utility itemsets mining algorithm, in: Proceedings of the ACM SIGKDD International Conference on Utility-Based Data Mining Workshop (UBDM), 2005, pp. 253–262.
[28] S. Lu, H. Hu, F. Li, Mining weighted association rules, Intell. Data Anal. 5 (3) (2001) 211–225.
[29] J. Pei, J. Han, H. Pinto, Q. Chen, U. Dayal, M. Hsu, PrefixSpan: Mining sequential patterns efficiently by prefix-projected pattern growth, in: Proceedings of the Seventeenth IEEE International Conference Data Engineering, Heidelberg, Germany, 2001, pp. 215–224.
[30] Y. Shen, Q. Yang, Z. Zhang, Objective-oriented utility-based association mining, in: Proceedings of the Second IEEE International Conference Data Mining, Maebashi City, Japan, 2002, pp. 426–433.
[31] F.S.C. Tseng, Y.-H. Kuo, Y.-M. Huang, Toward boosting distributed association rule mining by data de-clustering, Inf. Sci. (Ny) 180 (2010) 4263–4289.
[32] V.S. Tseng, B.E. Shie, C.W. Wu, P.S. Yu, Efficient algorithms for mining high utility itemsets from transactional databases, IEEE Trans. Knowl. Data Eng. 25 (8) (2013) 1772–1786.
[33] V.S. Tseng, C.-W. Wu, P. Fournier-Viger, P.S. Yu, Efficient algorithms for mining top-*k* high utility itemsets, IEEE Trans. Knowl. Data Eng. 28 (1) (2016) 54–67.
[34] V.S. Tseng, C.W. Wu, B.E. Shie, P.S. Yu, UP-Growth: an efficient algorithm for high utility itemset mining, in: Proceedings of the Sixteenth ACM SIGKDD International Conference Knowledge Discovery and Data Mining, Washington, DC, USA, 2010, pp. 253–262.
[35] J. Wang, J. Han, Y. Lu, P. Tzvetkov, TFP: an efficient algorithm for mining top-*k* frequent closed itemsets, IEEE Trans. Knowl. Data Eng. 17 (5) (2005) 652–664.
[36] C.W. Wu, B.-E. Shie, P.S. Yu, V.S. Tseng, Mining top-*k* high utility itemsets, in: Proceedings of the Eighteenth ACM SIGKDD International Conference Knowledge Discovery and Data Mining, Beijing, China, 2012, pp. 78–86.
[37] D. Xin, H. Cheng, X. Yan, J. Han, Extracting redundancy-aware top-*k* patterns, in: Proceedings of the Twelfth ACM SIGKDD International Conference Knowledge Discovery and Data Mining, Philadelphia, PA, USA, 2006, pp. 444–453.
[38] H. Yao, H.J. Hamilton, Mining itemset utilities from transaction databases, Data Knowl. Eng. 59 (3) (2006) 603–626.
[39] H. Yao, H.J. Hamilton, C.J. Butz, A foundational approach to mining itemset utilities from databases, in: Proceedings of the Fourth SIAM International Conference Data Mining, Lake Buena Vista, Florida, USA, 2004, pp. 482–486.
[40] H. Yao, H.J. Hamilton, L. Geng, A unified framework for utility-based measures for mining itemsets, in: Proceedings of the ACM SIGKDD International Conference on Utility-Based Data Mining Workshop (UBDM), 2006, pp. 28–37.
[41] S. Zida, P. Fournier-Viger, J.C.-W. Lin, C.-W. Wu, V.S. Tseng, EFIM: a fast and memory efficient algorithm for high-utility itemset mining, Knowl. Inf. Syst. 51 (2) (2017) 595–625.
[42] M. Zihayat, A. An, Mining top-*k* high utility patterns over data streams, Inf. Sci. (Ny) 285 (2014) 138–161.