LCIM: Mining Low Cost High Utility Itemsets

M. Saqib Nawaz[1], Philippe Fournier-Viger[1], Naji Alhusaini[2], Yulin He[1], Youxi Wu[3], and Debdatta Bhattacharya[4]

[1] Shenzhen University, Shenzhen, China
[2] Chuzhou University, Anhui, China
[3] Hebei University of Technology, Tianjin, China
[4] Koneru Lakshmaiah Education Foundation, Vaddeswaram, India
{msaqibnawaz, philfv}@szu.edu, naji@chzu.edu.cn, yulinhe@szu.edu.cn, wuc567@163.com,debdatta122001@gmail.com

Abstract. High utility itemset mining is a data mining task that consists of analyzing data to find the sets of items (values) that bring benefits as measured by a utility function. A representative application is to identify the sets of items bought by customers that yield a high profit. However, these studies focus on the utility (benefits) provided by patterns while ignoring information about their cost (e.g. time, effort, money or other resources that are consumed). This paper addresses this issue by studying a novel problem called low cost high utility itemset mining. The aim is to find patterns that have a high average utility, but also a low average-cost. An example application is to find patterns indicating learners' studying patterns in an e-learning platform that results in obtaining high grades (utility) for a relatively small effort (cost). An efficient algorithm named LCIM (Low Cost Itemset Miner) is proposed to solve this problem. It integrates a novel lower bound on the average cost called Average Cost Bound (ACB) to reduce the search space. Experiments show that LCIM is efficient and can reveal interesting patterns.

Keywords: Pattern Mining · Itemset · Cost Function · Utility Function.

1 Introduction

More and more data is being stored in databases. To allow users to gain insights from data, a popular subfield of data mining is *pattern mining* [1]. The aim is to apply algorithms to find interesting patterns in data that meet some user-defined constraints. Initial studies in this field have focused on identifying frequent itemsets using the support (frequency) function. A frequent itemset is a set of values that appears more than *minsup* times in a database, where *minsup* is a user-defined threshold [1, 7]. Frequent itemset mining (FIM) has many applications such as to analyze university course selection data to find the sets of courses that are frequently selected together by students, and identifying the sets of products frequently purchased together in a store. However, a drawback of FIM is that frequency is not the only criterion that matters in many situations.

Hence, FIM was generalized as the problem of high utility itemset mining (HUIM) [2, 6]. The goal is to find high utility itemsets, that is sets of values that bring high benefits, as measured by a utility function. For instance, in the context of shopping, utility may represent the profit obtained by selling sets of products. The search space of patterns can be very large in HUIM, and efficient algorithms have been designed [8, 10–12]. HUIM can reveal interesting patterns in many domains [6]. Nonetheless, the focus of HUIM is on the utility of patterns and the cost associated with these patterns is ignored.

For domains such as e-learning, utility and cost are two important dimensions that should be jointly considered. For example, the cost may be the study time during learning activities and the utility may be the grades subsequently obtained. Another example is medical data, where the cost may be the time or money spent by a patient for each treatment and the utility may be the result of finally being cured or not. Hence, the *utility* of a pattern can be viewed as the benefits obtained such as the grades or profit that is yielded by the pattern, while the *cost* can represent the drawbacks of the pattern such as the effort, time or resources consumed to apply this pattern.

Jointly considering utility and cost in pattern mining is desirable but not simple and could be done in many ways. One way is to subtract the cost from the utility and then apply HUIM algorithms. However, this approach is inadequate for applications such as e-learning as utility and cost may be expressed in different units (e.g. hours, grades). Besides, for such applications, it is more meaningful to consider the average utility and the average cost of patterns rather than their sums. For example, it is useless to know that some students took a total of 350 minutes to do activities and received a total of 400 points at the final exam, but it is meaningful to know that on average they took 1.2 hours each with an average grade of 80 points. Hence, this motivates us to separately model the utility and cost, and to consider their average. So far, only one study [4] has integrated the cost in pattern mining. But algorithms from that study are for analyzing event sequences, while this paper focuses on mining itemsets in transactions, which is different and more common in real-life. For example, such data are sets of courses selected by students or treatments taken by hospital patients. Due to different data types, prior algorithms cannot be reused.

The key contributions of this paper are fourfold. (1) A novel problem called low cost high utility itemset mining is formalized to introduce the concept of cost in itemset mining. The aim is to find itemsets that have a high average utility, and a low average cost. (2) An algorithm named LCIM (Low Cost Itemset Miner) is designed to solve this problem efficiently. (3) To reduce the search space, LCIM applies a lower bound on the average cost called Average Cost Bound (ACB). (4) Experiments were carried out to evaluate LCIM. Results show that LCIM is efficient and that interesting patterns are found in e-learning data.

The rest of this paper is organized as follows. Section 2 describes related work. Section 3 defines the problem. Section 4 presents the LCIM algorithm. Section 5 reports experimental results. Finally, Section 6 draws the conclusion.

2 Related work

The input of FIM [1] is a *transaction database*, where each transaction (record) is a set of items (symbols). The goal is to find frequent itemsets, where the support of an itemset is its occurrence frequency. All efficient FIM algorithms such as Apriori, FP-Growth and Eclat [7] find the frequent itemsets without exploring all possibilities by relying on the *anti-monotonicity property* of the support [1] (the support of an itemset cannot be more than that of its subsets).

HUIM is a generalization of FIM, initially designed to find profitable sets of items in shopping data [2, 6]. The input is a *quantitative transaction database* containing additional utility information. Each item in a transaction has a utility value (positive number) indicating its importance in that transaction (e.g. profit). The aim of HUIM is to find the itemsets that have a high utility as measured by a utility function [2, 6]. The two most popular functions are the total utility [2] (e.g. total profit) and the length-average utility[5] [9] (e.g. the profit of an itemset divided by its number of items). HUIM is harder than FIM because there is no anti-monotonicity property for most utility functions. Hence, FIM algorithms cannot be directly applied for HUIM. Efficient HUIM algorithms such as MLHUI-Miner [11], UD-Growth [14], ULB-Miner [3], and REX [12]) utilize upper bounds on the utility that are anti-monotonic to reduce the search space. Besides, various strategies are used to find patterns such as a breadth-first or depth-first search [6], or a horizontal [8] or vertical data format [13]. Until now, no HUIM studies has integrated the concept of cost.

Recently, the concept of cost was studied to find sequential patterns (subsequences) in multiple sequences of events [4]. That study has shown that the cost is an interesting function for some applications. But the aim of analyzing sequences is very different from the focus of this paper on analyzing transactions to find itemsets. Thus, algorithms from that study cannot be reused.

There are major differences between this paper and prior work on HUIM. In the current paper, two functions (utility and cost) are combined to identify itemsets having a low cost and a high utility. This combination is motivated by applications such as e-learning where these two dimensions are important. The data representation is different from HUIM. Rather than associating a utility value to each item, a single utility value is associated to each transaction, such that the utility represents its outcome (e.g. the final grade at an exam) after using items (e.g. studying some lessons). Then, a novel utility function is used to evaluate itemsets called average utility (e.g. average time spent on a lesson), which is different from the length-average utility function from HUIM (e.g. average time spent on lessons divided by the number of lessons) and more meaningful for applications such as e-learning. Besides, a cost value is assigned to each item to indicate its cost (e.g. time spent on each lesson). Due this different definition of utility and the addition of cost, prior HUIM cannot be applied to solve the new problem. A key challenge of this paper is also that there is an aim to maxi-

[5] The original name is *average utility* but it is renamed to be more precise.

mize utility while minimizing cost. Thus, not only an upper bound on the utility must be used but also a lower bound on the cost to mine patterns efficiently.

3 Problem definition

The proposed problem aims to discover itemsets in a *transaction database with cost values*. The data format is defined as follows. Let $I = \{i_1, i_2, \ldots, i_n\}$ be a finite set of items (symbols). A **transaction database** is a set of transactions $D = \{T_1, T_2, \ldots T_m\}$ where each **transaction** $T_j \in D$ is a finite set of items, i.e. $T_j \subseteq I$. Each transaction $T_j \in D$ has a unique identifier j. For each item i appearing in a transaction T, i.e. $i \in T$, a positive number $c(i, T)$ is assigned representing the **cost of the item** i in T. Moreover, a positive number is assigned to each transaction T, denoted as $u(T)$, which is called the **utility of transaction** T.

For instance, Table 1 shows an example database containing seven distinct items $I = \{a, b, c, d, e, f, g\}$ and five transactions $T_1, T_2, \ldots T_5$. The first transaction T_1 contains the items a, b, c, d, e, and f, with the cost values $c(a, T_1) = 5$, $c(b, T_1) = 10$, $c(c, T_1) = 1$, $c(d, T_1) = 6$, $c(e, T_1) = 3$, and $c(f, T_1) = 5$. The utility values of the transactions are $u(T_1) = 40$, $u(T_2) = 20$, $u(T_3) = 18$, $u(T_4) = 37$, and $u(T_5) = 21$. Each transaction can represent the lessons (items) taken by some students (transactions) where cost is time and utility is final grades.

Table 1. A small transaction database with cost and utility values

Transaction	Items $(item, cost)$	Utility
T_1	$(a, 5), (b, 10), (c, 1), (d, 6), (e, 3), (f, 5)$	40
T_2	$(b, 8), (c, 3), (d, 6), (e, 3)$	20
T_3	$(a, 5), (c, 1), (d, 2)$	18
T_4	$(a, 10), (c, 6), (e, 6), (g, 5)$	37
T_5	$(b, 4), (c, 2), (e, 3), (g, 2)$	21

An **itemset** X is a set such that $X \subseteq I$. An itemset X having k items is called a k-itemset. Let $g(X) = \{T | X \subseteq T \in D\}$ be the set of transactions where X appears. The **support of an itemset** X in a database D is defined and denoted as $s(X) = |g(X)|$.

For example, the itemset $\{b, c\}$ is a 2-itemset. It appears in the transactions $g(\{b, c\}) = \{T_1, T_2, T_5\}$. Thus, $s(\{b, c\}) = 3$.

The **average cost of an itemset** X in a transaction T such that $X \subseteq T$ is defined and denoted as $c(X, T) = \sum_{i \in X} c(i, T)$. The **cost of an itemset** X in a database D is defined and denoted as $c(X) = \sum_{T \in g(X)} c(X, T)$. The **average cost of an itemset** X in a database D is defined and denoted as $ac(X) = c(X) \div s(X)$.

For instance, the cost of itemset $\{b, c\}$ in transaction T_1 is $c(\{b, c\}, T_1) = c(b, T_1) + c(c, T_1) = 10 + 1 = 11$. The average cost of itemset $\{b, c\}$ is $[c(\{b, c\}, T_1) + c(\{b, c\}, T_2) + c(\{b, c\}, T_5)] \div s(\{b, c\}) = [11 + 11 + 6] \div 3 = 28/3$, which can represent the average time to study lesson b and c in an e-learning context.

The **utility of an itemset** X in a database D is defined and denoted as $u(X) = \sum_{T \in g(X)} u(T)$ and represents how positive the outcome of transaction is.

The **average utility of an itemset** X in a database D is defined and denoted as $ac(X) = u(X) \div s(X)$.

For example, the utility of itemset $\{b, c\}$ is $u(\{b, c\}) = 40 + 20 + 21 = 81$. The average utility of itemset $\{b, c\}$ is $u(\{b, c\}) \div s(\{b, c\}) = 81 \div 3 = 27$. In e-learning, this can indicate that the average grade is 27 for students doing lessons b and c. The proposed problem is defined as follows.

Definition 1 (problem definition). *Let there be some user-specified thresholds $minsup > 0$, $minutil > 0$, and $maxcost > 0$. An itemset X is a **low cost itemset** if and only if $au(X) \geq minutil$, $ac(X) \leq maxcost$ and $s(X) \geq minsup$.*

This definition is meaningful for applications such as e-learning to find sets of lessons that on average lead to high scores with a small study time.

4 The LCIM algorithm

This section presents the proposed LCIM algorithm. Subsection 4.1 first describes the search space pruning properties of LCIM. Subsection 4.2 presents its novel cost-list data structure. Finally, Subsection 4.3 describes the algorithm.

4.1 Search space exploration and pruning properties

The proposed algorithm explores the search space of itemsets starting from itemsets containing single items, and then recursively searches for larger itemsets. To avoid considering the same itemset multiple times, larger itemsets are explored by applying a process called *extension*, defined as follows.

Definition 2 (Extension). *Without loss of generality, it is assumed that items in each itemset are sorted according to a total order $\succ$, called the **processing order**. Two itemsets X and Y can be joined together to obtain a new itemset $Z = X \cup Y$ if all items in X and Y are the same except the last one according to $\succ$. The itemset Z is then said to be an **extension** of X and Y.*

For instance, assume that the processing order is the lexicographical order $(f \succ e \succ d \succ c \succ b \succ a)$. Then, the two itemsets $X = \{a\}$ and $X = \{b\}$ can be joined to obtain an itemset $Z = \{a, b\}$. Similarly, the itemsets $\{a, b\}$ and $\{a, c\}$ can be joined to obtain an itemset $\{a, b, c\}$. However, the itemsets $\{a, b, c\}$ and $\{b, e\}$ cannot be joined because they are not the same except the last item.

By recursively performing extensions, it can be shown that the whole search space of itemsets can be explored. However, to have an efficient algorithm, it is necessary to be able to reduce the search space. To reduce the search space, the Apriori pruning property based on the support is used [1].

Property 1 (Support pruning). For any two itemsets $X \subseteq Y$, $s(X) \geq s(Y)$.

Besides, to reduce the search space using the cost, a lower bound on the cost is introduced (inspired by the ASC lower bound used in sequence mining [4]):

Definition 3 (Lower bound on the cost). *Let there be an itemset X, which appears in the transactions $g(X) = \{T'_1, T'_2, \ldots, T'_N\}$. The* ***sequence of cost values of*** *X is the unique sequence $A(X) = (a_i)_{i=1}^{N}$ where $a_i = c(X, T'_i)$. Let $sort(A(X))$ be the unique non decreasing sequence $B = (b_i)_{i=1}^{N}$ satisfying $\forall i \leq K \leq N$ $|\{i \in \mathbb{N} | a_i = a_K\}| = |\{i \in \mathbb{N} | b_i = a_K\}|$. The K* ***largest cost values*** *of X is the sequence $A(X)^{(K)} = (c_i)_{i=1}^{K}$ of real numbers satisfying $\forall 1 \leq i \leq K, c_i = b_{N-K+i}$. The* ***average cost bound (ACB)*** *of X is defined as $acb(X) = \frac{\sum_{c_i \in A(X)^{(minsup)}} c_i}{s(X)}$.*

For instance, let $minsup = 1$ and $X = \{b, c\}$. The sequence of cost values of X is $A(X) = \langle 11, 11, 6 \rangle$. Then, $sort(A(X)) = \langle 6, 11, 11 \rangle$, and $A(X)^{(1)} = \langle 6 \rangle$. The average cost bound of X is $acb(X) = \frac{6}{3} = 2$. If $minsup = 2$, $A(X)^{(2)} = \langle 6, 11 \rangle$, and $acb(X) = \frac{6+11}{3} = \frac{17}{2}$.

Property 2 (lower bound of the ACB on the average cost). For any itemset X, the average cost bound of X is a lower bound on the average cost of X. In other words, $acb(X) \geq ac(X)$.

Proof. By definition, $acb(X) = \frac{\sum_{c_i \in A(X)^{(minsup)}} c_i}{s(X)}$, while the average cost can be expressed as $ac(X) = \frac{\sum_{c_i \in sort(A(X))} c_i}{s(X)}$. It is clear that $A(X)^{(minsup)}$ is a subsequence of $sort(A(X))$. Hence, the numerator of $acb(X)$ is no greater than that of $ac(X)$. Thus, $acb(X) \leq ac(X)$. □

Property 3 (anti-monotonicity of the ACB). For any two itemsets $X \subseteq Y$, then $acb(X) \leq acb(Y)$.

Proof. We have $acb(X) = \frac{\sum_{c_i \in A(X)^{(minsup)}} c_i}{s(X)}$ and $acb(Y) = \frac{\sum_{c_i \in A(Y)^{(minsup)}} c_i}{s(Y)}$. Since $X \subseteq Y$, it follows that $g(Y) \subseteq g(X)$ and that $s(Y) \leq s(X)$. Hence, the denominator of $acb(Y)$ is greater or equal to that of $acb(X)$. For the numerators, we know that $A(X)^{(minsup)}$ and $A(Y)^{(minsup)}$ both contain $minsup$ cost values but cost values in the latter must be greater than those in the former since Y is a larger itemset. Hence, the numerator of $acb(Y)$ must be larger than that of $acb(X)$. Thus, $acb(X) \leq acb(Y)$. □

Property 4 (Search space pruning using the ACB). For an itemset X, if $acb(X) > maxcost$, X and its supersets are not low cost itemsets.

Proof. This directly follows from the two previous properties.

4.2 The cost-list data structure

Another key consideration for the design of an efficient algorithm is how to efficiently calculate the utility and cost values of itemsets and also the ACB lower bound for reducing the search space. For this purpose, the designed LCIM algorithm relies on a novel data structure called the *cost-list*.

Definition 4 (cost-list). *The **cost-list** of an itemset X is a tuple $L(X) = (utility, cost, tids, costs)$ that stores information in four fields. The field utility contains $u(X)$. The field cost stores $c(X)$. The field tids stores $g(X)$, while costs stores $A(X)$. In the following, the notation $L(X).field$ refers to the value of the field $field$ in $L(X)$.*

For instance, Table 4.2 shows the cost-lists of itemset $\{a\}$, $\{b\}$ and $\{a, b\}$.

Table 2. The cost-lists of $\{a\}$, $\{b\}$ and $\{a, b\}$

$L(\{a\}$	
utility	95
cost	20
tids	$\{T_1, T_3, T_4\}$
costs	$\langle 5, 5, 10\rangle$

$L(\{b\}$	
utility	81
cost	22
tids	$\{T_1, T_2, T_5\}$
costs	$\langle 10, 8, 4\rangle$

$L(\{a, b\}$	
utility	40
cost	15
tids	$\{T_1\}$
costs	$\langle 15\rangle$

The cost-lists of an itemset X is useful as it contains all the required information about it. The cost-list of X allows to directly obtain its support as $s(X) = |L(X).tids|$), its average utility as $au(X) = L(X).utility/s(X)$, and its average cost as $ac(X) = L(X).utility/s(X)$. Moreover, the ACB lower bound can be calculated by finding the *minsup* smallest values in $L(X).costs$.

The proposed algorithm builds the cost-list of each 1-itemset containing by scanning the database. Then, the cost-list of any larger itemset is obtained without scanning the database, by applying a join operation on cost-lists of some of its subsets, as follows.

Let there be two itemsets X and Y that are joined to obtain an extension $Z = X \cup Y$. The cost-list $L(Z)$ is derived directly from the cost-lists $L(X)$ and $L(Y)$ as follows. The field $L(Z).costs$ is obtained by merging the cost values corresponding to the same transactions in $L(X).cost$ and $L(Y).costs$. The field $L(Z).tids = L(X).tids \cap L(Y).tids$. The field $L(Z).cost$ is the sum of values in $L(Z).costs$. The field $L(Z).utility$ is calculated as the sum of utility values for transactions in $L(Z).tids$.

4.3 The algorithm

LCIM (Algorithm 1) takes as input a transaction database with cost/utility and the *minsup*, *minutil* and *maxcost* thresholds. LCIM first scans the database to calculate the support of each item. This allows determining the set I^* of all frequent items (having a support no less than *minsup*). Then, LCIM utilizes this information to establish a processing order $\succ$ on items that is the ascending order of support. Thereafter, LCIM scans the database again to create the cost-lists of all items in I^*. This provides the information for calculating the ACB lower bound of each item $i \in I^*$. Each item having a lower bound value that is no greater than *maxcost* is put in a set I^{**}. Then, the recursive procedure *Search* is called with I^{**} to search for low cost itemsets. Other items can be ignored as they cannot be part of a low cost itemset based on Property 4.

The *Search* procedure is shown in Algorithm 2. It takes a input a set of itemsets P and their cost-lists, as well as the *minsup*, *minutil* and *maxcost* thresholds. The procedure outputs the set of low cost itemsets in P or itemsets that are extensions of itemsets in P. For each itemset $X \in P$, the procedure first calculates its average utility and average cost from its cost-list $L(X)$. Then, if the average utility of X is no less than *minutil* and its average cost is no greater than *maxcost*, the itemset X is output as a low cost itemset. Then, the procedure initializes a variable $ExtensionsOfX$ to store extensions of P that contain one more item than X and may be low cost itemsets. Thereafter, for each itemset Y in P that can be joined with X, the extension $Z = X \cup Y$ is created, as well as its cost-list $L(Z)$ by calling the *Construct* procedure. The cost-list $L(Z)$ allows to directly obtain the support $s(Z)$ and the ACB lower bound $acb(Z)$ without scanning the database. Then, if $s(Z)$ is no less than *minsup* and the $acb(Z)$ is no greater than *maxcost*, the itemset Z is added to $ExtensionsOfX$. This is because Z and its recursive extensions may be low cost itemsets. Afterwards, the search procedure is called with the set $ExtensionsOfX$ to explore extensions of X recursively. When the loop of all itemsets in P ends, all low cost itemsets that are in P or are extensions of itemsets in P have been output.

Algorithm 1: The LCIM algorithm

input : D: a transaction database,
$minsup, minutil, maxcost$: the user-specified thresholds
output : all the set of low cost itemsets

Scan D to calculate the support $s(\{i\})$ of each item i;
$I^* \leftarrow$ each item i such that $s(\{i\}) \geq minsup$;
Let $\succ$ be the total order of support ascending values on I^*;
Scan D to build the cost-list of each item $i \in I^*$;
$I^{**} \leftarrow$ each item $i \in I^*$ such that $acb(\{i\}) \leq maxcost$ according to $L(\{i\})$;
`Search` (I^{**}, $minsup$, $minutil$, $maxcost$);

Algorithm 2: The *Search* procedure

input : P: a set of itemsets and their cost-lists,
$minsup, minutil, maxcost$: the user-specified thresholds
output : a set of low cost itemsets

foreach *itemset* $X \in$ P **do**
  if $au(X) \geq minutil \wedge ac(X) \leq maxcost$ **then** Output X;
  $ExtensionsOfX \leftarrow \emptyset$;
  foreach *itemset* $Y \in$ P *that can be joined with* X **do**
    $Z \leftarrow X \cup Y$;
    $L(Z) \leftarrow$ `Construct` $(L(X), L(Y))$;
    if $s(Z) \geq minsup \wedge acb(Z) \leq maxcost$ **then**
      $ExtensionsOfX \leftarrow ExtensionsOfX \cup \{Z\}$;
    end
  end
  `Search` ($ExtensionsOfX$, $minsup, minutil, maxcost$);
end

The *Construct* procedure is shown in Algorithm 3. The input is the cost-lists $L(X)$ and $L(Y)$ of two itemsets X and Y that can be joined together to form an extension $Z = X \cup Y$. The output is the cost-list $L(Z)$. This procedure first initializes the cost-list of $L(Z)$ such that $L(Z).utility = 0$, $L(Z).tids = \emptyset$ and $L(Z).tids = \emptyset$. Then a loop is performed to check each transaction T_w that is in

$L(X).tids$ to see if it also appears in $L(Y).tids$. For each such transaction T_w, it is added to $L(Z).tids$, and then the utility $u(T_w)$ is added to $L(Z).utility$. Then, $L(Z).costs$ is updated by adding the cost of Z in T_w. This cost is calculated by a procedure called Merge, which does the sum of the cost of X in T_w and the cost of Y in T_w (this information is obtained from $L(X)$ and $L(Y)$). Finally, after the loop is completed, $L(Z).cost$ is calculated as the sum of all values in $L(Z).costs$, and $L(Z)$ is returned.

Algorithm 3: The Construct procedure

input : $L(X), L(Y)$: the cost utility list of two itemsets X and Y
output : the cost-list $L(Z)$ of $Z = X \cup Y$

```
1  Initialize a cost-list L(Z) such that L(Z).utility = 0, L(Z).tids = ∅, and L(Z).costs = ∅;
2  foreach transaction T_w ∈ L(X).tids do
3      if ∃T_w ∈ L(Y).tids then
4          L(Z).tids ← L(Z).tids ∪ {T_w};
5          L(Z).utility ← L(Z).utility + u(T_w);
6          L(Z).costs ← Merge(L(X).costs, L(Y).costs);
7      end
8  end
9  L(Z).cost = ∑ L(Z).costs;
10 return L(Z);
```

It can be observed that the whole search space of itemsets can be explored by recursively performing extensions, and that only itemsets that are not low cost itemsets are pruned by the pruning properties, which are proven in Section 4.1. Hence, LCIM can find all low cost itemsets. The complexity of LCIM is a function of the number of itemsets that LCIM visits in the search space, which depends on how *minsup*, *minutil* and *maxcost* are set. In the worst case, there are $2^{|I|} - 1$ itemsets. For each visited itemset, a cost-list is created in linear time, which has a size bounded by the database size.

Two optimizations. We also include two performance optimization: (1) *Matrix Support Pruning* (MSP) consists of precalculating the support of all pairs of items in the initial database scan. Then, two itemsets X and Y are not joined as $X \cup Y$ if their last items have a joint support below *minsup*. (2) Efficient List Construction (ELC) the construct procedure is stopped as soon as there are not enough transactions left in $L(X).tids$ to attain $L(Z).tids \geq minsup$.

5 Experimental evaluation

LCIM is implemented in Java. For the experiments, we take the LCIM algorithm without the two optimizations and consider it as a baseline since there is no prior algorithm. The performance of the baseline is compared with LCIM on four benchmark datasets, namely Chess, Mushroom, Accidents and E-Learning. The source code of LCIM and datasets can be downloaded from the SPMF data mining library [5]. Chess has 3,196 transactions, 37 distinct items, and an average transaction length of 75 items. Mushroom contains 8,416 transactions,

119 distinct items, and the average transaction length is 23 items. Accidents contains 340,183 transactions with 468 distinct items, and an average transaction length of 34 items. But only 10% of Accidents was used. E-Learning contains 54 transactions, 11 distinct items, and the average transaction length is 3.8 items. The experiments were conducted on a laptop with an Intel Celeron processor and 16 GB of RAM running 64-bit Windows 10.

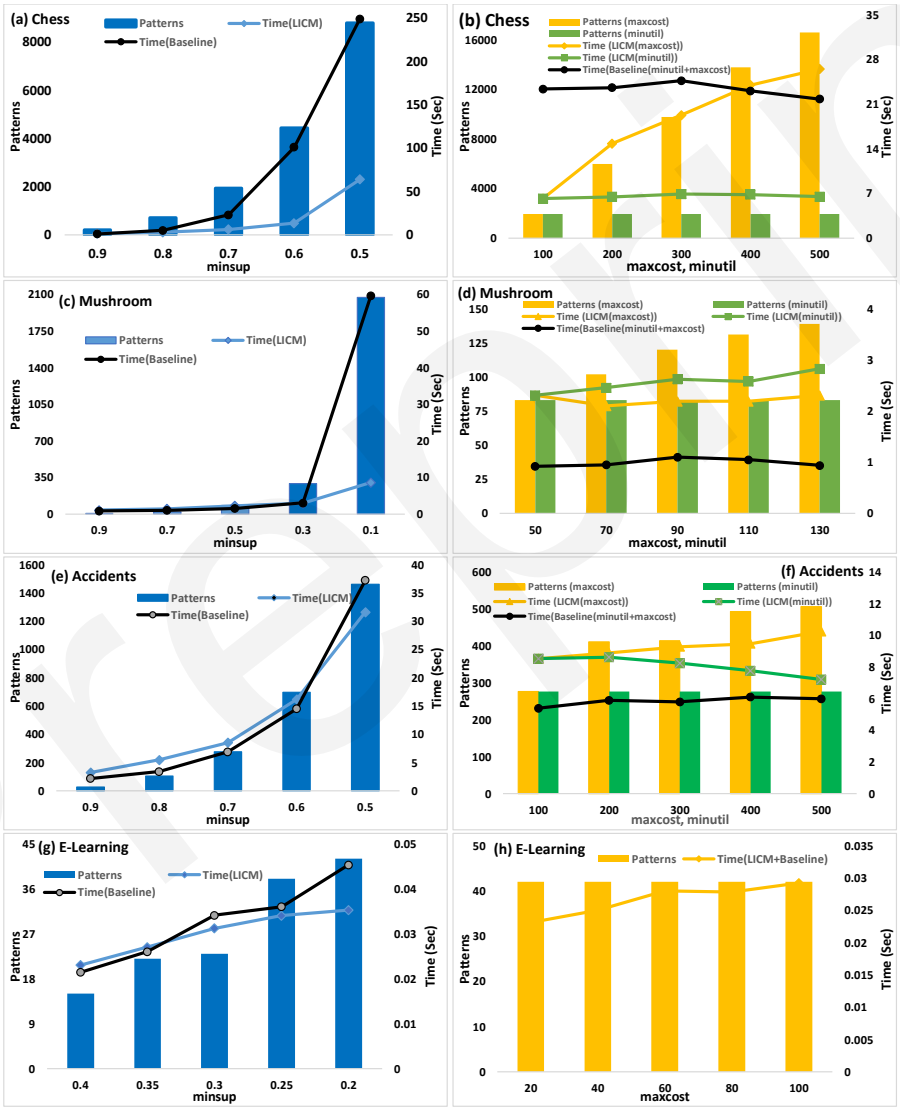

Fig. 1. Runtime of the baseline and LCIM for varying *minsup*, *maxcost* and *minutil*

Runtime and pattern count. In the first experiment, the performance of LCIM was evaluated on four datasets in terms of execution time and number of patterns found. Fig 1 (a, c, e and g) show the execution time and pattern count on Chess, Mushroom, Accidents and E-Learning, respectively, for various *minsup* values. For the Chess, Mushroom, Accidents and E-Learning datasets, the *maxcost* (*minutil*) values were set to 100 (100), 50 (50), 100 (100) and 100 (10), respectively. Fig 1 (b, d, f and h) show the execution time and discovered patterns on Chess, Mushroom, Accidents and E-Learning, respectively, for various *maxcost* and *minutil* values. For the Chess, Mushroom, Accidents and E-Learning datasets, *minsup* was set to 0.7%, 0.5%, 0.7% and 0.2%, respectively. It is observed that the baseline and LCIM are fast. The execution time and pattern count for LCIM increased with a decrease in *minsup* and an increase in *maxcost* values, which show the effectiveness of pruning properties. For *minutil*, it has a negligible effect on the execution time as it is not used for pruning. Interestingly, the pattern count remained the same for various *minutil* values. This is why the execution time and pattern count results are omitted for various *minutil* values on the E-learning dataset. Compared to LCIM, the baseline algorithm takes more time, particularly on the Chess and Mushroom datasets, when *minsup* is decreased. For *maxcost* and *minutil*, the baseline algorithm takes the same amount of time. Interestingly, the baseline algorithm is slightly faster, on overall, for experiments of varying *maxcost* and *minutil*.

Pattern analysis for e-learning. The Chess, Mushroom and Accidents datasets have synthetic values for cost and utility, while E-learning has real cost and utility values. Hence, to evaluate the quality of patterns found, we look at patterns found in E-learning. It contains sets of online activities done by 54 students during Session 4 of an e-learning environment. Each transaction is the list of activities performed by a student. Each activity has a cost representing the time spent by the student on the activity. Moreover, each transaction has a utility value that is the score of the student at a test at the end of the session. Table 3 lists the patterns obtained in that dataset by setting *minutil*, *maxcost* and *minsup* to 10, 100 and 0.2% respectively.

Table 3. Cost-effective patterns discovered in events sequences in Session 4

No	Patterns	Avg. Util	Avg. Cost	Sup	Trade-off
1	*Deeds_Es_4_3, Deeds_Es_4_2*	12.53	21.23	13	1.69
2	*Deeds_Es_4_4, Deeds_Es_4_1*	14.07	25.21	14	1.81
3	*Deeds_Es_4_4, Deeds_Es_4_5*	12.42	28.92	14	2.38
4	*Deeds_Es_4_1*	14.21	13.82	23	0.98
5	*Deeds_Es_4_1, Deeds_Es_4_2*	13.64	26.71	14	1.96
6	*Deeds_Es_4_1, Deeds_Es_4_2, Deeds_Es_4_5*	12.76	60.0	13	4.76
7	*Deeds_Es_4_1, Deeds_Es_4_5*	12.5	34.12	16	2.77
8	*Deeds_Es_4_2*	13.26	10.60	23	0.80
9	*Deeds_Es_4_2, Deeds_Es_4_5*	12.47	27.68	19	2.22
10	*Deeds_Es_4_5*	11.62	17.20	24	1.49

A manual analysis of activities confirmed that the patterns are reasonable for learners. But to compare the efficiency of these patterns, we can further check the *trade-off* between cost and utility. A pattern efficiency's can be calculated as a *trade-off* value. The *trade-off* of a pattern (p) is the ratio of its average cost to its average utility [4]. For example, patterns 6 and 7 (4 and 8) in Table 3 have

the highest (lowest) *trade-off* values. A pattern with a low *trade-off* is especially interesting as it provides utility (high grades) at a low cost (time). Thus, students could be more efficient at studying by carefully selecting learning activities from patterns with low trade-off (it is not mandatory for students to do all activities in that e-learning environment).

6 Conclusion

This paper presented a novel problem of low cost high utility itemset mining (finding patterns that have a high average utility but a low average-cost). An efficient algorithm named LCIM (Low Cost Itemset Miner) was presented to solve this problem efficiently. It introduces a lower bound on the average cost called Average Cost Bound (ACB) to reduce the search space, and a cost-list data structure. Experiments have shown that LCIM is efficient and can find interesting patterns in e-learning data. In future work, alternative ways of integrating utility and cost will be studied.

References

1. Agrawal, R., Srikant, R.: Fast algorithms for mining association rules. In: Proceedings of VLDB. vol. 1215, pp. 487–499 (1994)
2. Chan, R., Yang, Q., Shen, Y.: Mining high utility itemsets. In: Proceedings of ICDM. pp. 19–26 (2003)
3. Duong, Q., Fournier-Viger, P., Ramampiaro, H., Nørvåg, K., Dam, T.: Efficient high utility itemset mining using buffered utility-lists. Applied Intelligence **48**(7), 1859–1877 (2018)
4. Fournier-Viger, P., Li, J., Lin, J.C., Truong-Chi, T., Kiran, R.U.: Mining cost-effective patterns in event logs. Knowledge Based Systems **191**, 105241 (2020)
5. Fournier-Viger, P., Lin, J.C.W., Gomariz, A., Gueniche, T., Soltani, A., Deng, Z., Lam, H.T.: The SPMF open-source data mining library version 2. In: Proceedings of ECML PKDD. pp. 36–40 (2016)
6. Fournier-Viger, P., Lin, J.C.W., Truong-Chi, T., Nkambou, R.: A survey of high utility itemset mining. In: High-Utility Pattern Mining, pp. 1–45. Springer (2019)
7. Fournier-Viger, P., Lin, J.C.W., Vo, B., Chi, T.T., Zhang, J., Le, B.: A survey of itemset mining. WIREs Data Mining and Knowledge Discovery **7**(4), e1207 (2017)
8. Fournier-Viger, P., Wu, C.W., Zida, S., Tseng, V.S.: FHM: Faster high-utility itemset mining using estimated utility co-occurrence pruning. In: Proceedings of ISMIS. pp. 83–92 (2014)
9. Kim, H., Yun, U., Baek, Y., Kim, J., Vo, B., Yoon, E., Fujita, H.: Efficient list based mining of high average utility patterns with maximum average pruning strategies. Information Sciences **543**, 85–105 (2021)
10. Nawaz, M.S., Fournier-Viger, P., Yun, U., Wu, Y., Song, W.: Mining high utility itemsets with hill climbing and simulated annealing. ACM Transactions on Management Information Systems **13**(1) (2022)
11. Peng, A.Y., Koh, Y.S., Riddle, P.: mhuiminer: A fast high utility itemset mining algorithm for sparse datasets. In: Proceedsings of PAKDD. pp. 196–207 (2017)

12. Qu, J., Fournier-Viger, P., Liu, M., Hang, B., Wang, F.: Mining high utility itemsets using extended chain structure and utility machine. Knowledge Based Systems **208**, 106457 (2020)
13. Qu, J.F., Liu, M., Fournier-Viger, P.: Efficient algorithms for high utility itemset mining without candidate generation. In: High-Utility Pattern Mining, pp. 131–160. Springer (2019)
14. Verma, A., Dawar, S., Kumar, R., Navathe, S., Goyal, V.: High-utility and diverse itemset mining. Applied Intelligence **51**(7), 4649–4663 (2021)