Applied Intelligence
https://doi.org/10.1007/s10489-023-04913-w

A new algorithm using integer programming relaxation for privacy-preserving in utility mining

Duc Nguyen[1,2] · Minh-Thai Tran[3] · Bac Le[1,2]

Accepted: 22 July 2023

Abstract
High-utility itemset mining (HUIM) is an effective technique for discovering significant information in data. However, data containing sensitive and private information may cause privacy concerns. Therefore, privacy preserving utility mining (PPUM) has recently become a critical research area. PPUM is the process of transforming a quantitative transactional database into a sanitised one, thus ensuring that utility mining algorithms cannot discover sensitive information. The sanitisation process can have several side effects, including the loss of non-sensitive information and the introduction of redundant information. Additionally, the running times of heuristic algorithms for sanitising data are high. To minimise negative effects and lower the execution time of the hiding process, we propose the G-ILP algorithm with a GPU parallel programming method for preprocessing and a new efficient constraint satisfaction problem for hiding data. The experimental evaluations of G-ILP show the algorithm's efficiency in terms of running time and its ability to minimise side effects in large datasets.

Keywords Privacy preserving · Data mining · Utility mining · Sanitization

1 Introduction

Discovering knowledge from databases, also known as data mining, focuses mainly on extracting the information hidden in large and complex data. However, data mining can reveal confidential information concerning individuals or organisations. Therefore, privacy preserving in data mining (PPDM) is an important issue.

Utility mining is a common and widely used data mining technique that considers item's unit profits and quantities in transactions to find high-utility itemsets (HUIs). Despite its high applicability, utility mining also has several privacy problems. In particular, sensitive patterns discovered in businesses's data can be used to infer confidential information and personal identities. There are also hidden patterns that can uncover companies non-public relationships or trade secrets. For these reasons, sensitive high-utility itemsets must be hidden before data is shared for trading and research purposes. Privacy preserving utility mining (PPUM) emerged and became a necessary step in utility mining [1] to hide sensitive patterns while retaining critical information.

In the majority of published studies, sensitive high-utility itemsets (SHUIs) are hidden by removing them or reducing their utilities in the database. The first approach used to hide sensitive high-utility patterns was designed by Yeh et al. [2] and involved two algorithms, HHUIF and MSICF. Each algorithm found prior victim transactions and items using its own criteria, then decreased item quantities to lower the SHUIs utilities. Two additional approaches were proposed by Lin et al. [3, 4], inserting new transactions in the database and removing current transactions from the database. Lin et al. [5] also introduced two new hiding algorithms and three PPUM evaluation measures. All these methods use heuristic ideas to hide all sensitive itemsets. Consequently, their results have low generality and high rates of side effects. To solve these problems, a new approach was proposed by Li et al. [6] named PPUM-ILP. They designed a preprocessing mechanism to

✉ Bac Le
lhbac@fit.hcmus.edu.vn

Duc Nguyen
nnduc@fit.hcmus.edu.vn

Minh-Thai Tran
minhthai@huflit.edu.vn

1 Faculty of Information Technology, University of Science, Ho Chi Minh City, Vietnam

2 Vietnam National University, Ho Chi Minh City, Vietnam

3 Faculty of Information Technology, University of Foreign Languages -Information Technology, Ho Chi Minh City, Vietnam

Published online: 03 August 2023

identify the high-utility itemsets that must be constrained. Then, integer programming techniques are applied to change the states of these itemsets in a more efficient, precise and general way.

Thus far, new algorithms [7–10] have been proposed. However, almost all the algorithms only used a heuristic approach for sanitising data. Those approaches require multiple database modifications or multiple structure updates during the hiding process. That leads to a high execution time in large datasets. In this paper, we introduce a new algorithm that utilises an integer linear programming (ILP) solution to perturb data and hide sensitive information. The main contributions of our proposed algorithm, G-ILP, are summarised as follows.

1. We introduced a parallel preprocessing and designed a new table structure called GIT for faster preprocessing.
2. We proposed a mathematical model for finding a relaxation of the hiding problem.
3. The experiments show that our proposed algorithm is quite promising, and the weakness of the ILP approach, which is not mentioned in previous publications, is also clarified.

The remainder of this paper is organised as follows. Some important related works are reviewed in Section 2. Section 3 provides some related preliminaries for the PPUM problem. The proposed algorithm's implementation is described in detail in Section 4. Experiments are reported in Section 5. Section 6 concludes the paper and discusses future research.

2 Related works

2.1 Utility mining

Recently, utility mining has emerged as an important research topic in data mining. This method considers the profit values and item quantities in transactions. Moreover, quantities are not just represented in binary values as in traditional transaction databases. A minimum utility threshold δ must be chosen to find high-utility itemsets. If δ is too low, many itemsets will be discovered at the cost of memory and computation time. However, if δ is high, useful itemsets with low utilities could be ignored. Often, high-utility itemsets need to be located in the medical field, such as in analyses of biomedical data and DNA. Different genes have different levels of significance, and no data point from a single gene is limited to binary values. When combinations of the high-utility genes are located, genes closely related to an illness can be discovered, forming the basis for the study of the illness's treatment. High-utility itemsets are also used in web mining. By considering the importance of a website and user usage times, high-utility itemsets that show the relationship between internet users and websites can be discovered. Other applications include analyses of financial data, predictions of stock prices, examination of business transactions, the detection of fraudulent transactions, security and so forth.

Yao et al. [11] introduced the first work about utility mining. They developed a high-utility itemset mining algorithm by using the information about items quantities and profits to find useful HUIs. Because HUIs do not have the downward closure property, a transaction-weighted utilisation (TWU) [12] model was proposed to speed up the HUI discovery process by finding itemsets with high TWUs (HTWUIs). This method follows a level-wise approach.

To resolve the drawbacks of the level-wise approach, Lin et al. [13] designed a tree structure, HUP-Tree. HTWUIs containing single items are first discovered when building the HUP-Tree, which is then used to discover all HUIs. Tseng et al. [14] also introduced a tree structure, UP-Tree. Based on UP-Tree, two algorithms, UP-growth and UP-growth+ [14, 15], were proposed. In addition, Liu et al. [16] introduced the HUI-Miner algorithm. They designed the novel utility-list structure to find HUIs directly without candidate generation. FHM [17] and HUP-Miner [18] also used utility-list to store itemset information during the mining process. Zida et al. [19] developed a database projection and transaction merging technique with the EFIM algorithm. EFIM relies on two upper bounds, revised sub-tree utility and local utility, to prune the search space effectively. Extensive issues were also presented. Lin et al. [20] proposed the HUI-MMU algorithm for mining high-utility itemsets with multiple minimum utility thresholds. For mining up-to-date patterns, Gan et al. [21] introduced a mining approach which relies on the information trend. Because transactional data profit values change over time, Vo et al. [22] designed the MCH-Miner algorithm to mine HUIs in dynamic transaction databases. Also, within the dynamic database context, Yun et al. [23] proposed the HUIPRED algorithm to mine HUIs in databases that have continuously deleted transactions.

2.2 Privacy preserving utility mining

With the development of high-utility itemset mining algorithms, privacy concerns have become increasingly serious. Thus, PPUM has also arisen as a critical research issue in protecting information. Yeh et al. [2] published the first work on the PPUM problem and introduced two heuristic algorithms, HHUIF and MSICF. The main idea behind both algorithms is to hide SHUIs by decreasing the quantities of victim items in selected transactions. To improve the efficiency of the HHUIF algorithm, Yun et al. [24] designed a tree structure called FPUTT. FPUTT achieves the perturbation process with only three required database scans. Although FPUTT is faster than previous algorithms, its tree may not be compact

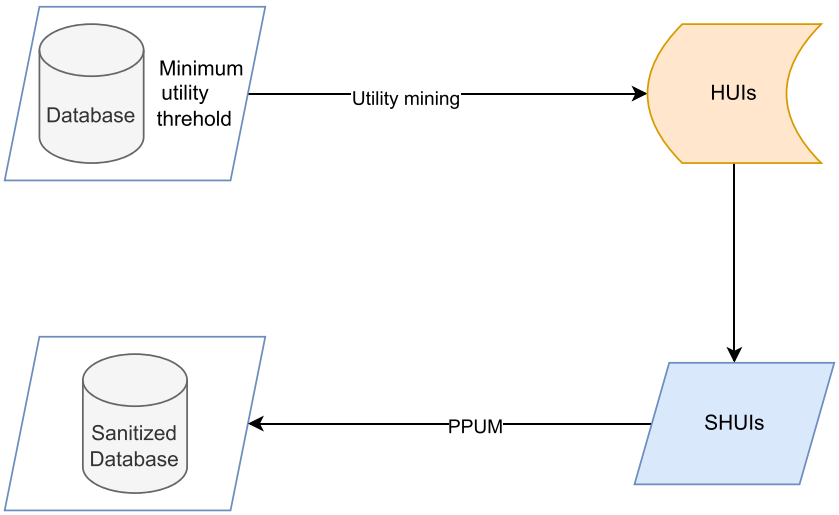

Fig. 1 The general process of the PPUM approach

on a large dataset. In addition, memory usage and computational costs are high.

Lin et al. [5] introduced two algorithms, MSU-MAU and MSU-MIU. They used the maximum and minimum utility concepts to remove or decrease item quantities, thus efficiently reducing the utilities of the SHUIs. They also proposed three criteria similar to those used in PPDM to evaluate the performance of the sanitisation method and applied some meta-heuristic methods to solve PPUM. Lin et al. [3, 4] designed two algorithms based on the genetic algorithm PPUMGA+insert and PPUMGAT. PPUMGA+insert inserts transactions into the database, while PPUMGAT removes transactions from the original database. However, the transaction insertion/deletion process changes the number of transactions in the database, and the sanitisation process produces unreal itemsets. Additionally, the mining algorithm needs to be modified to find the pre-large concept, which is impractical.

The challenge of minimising the side effects of the sanitisation process remained. To address this challenge, Liu et al. [7] proposed an improved version of the MSICF algorithm, IMSICF. IMSICF computes the conflict count for each sensitive item dynamically during the sanitisation process; hence, the computational cost is higher, and execution is slower than in other algorithms. The results also show that its ability to minimise side effects is inefficient. In the worst case, nearly 80% of the non-sensitive information is lost when perturbing the database. In 2020, Liu et al. [8] proposed three sanitisation algorithms, SMAU, SMIU and SMSE. Those algorithms avoid scanning databases multiple times by constructing a T-table and an HUI-table. Each algorithm uses its own heuristic to find victim items for modifications. However, when the T-table and the HUI-table are updated with each modified item, their execution times are still high.

Recently, Jangra et al. [9] proposed three methods for selecting victim items. The difference lies in dataset sorting techniques and transaction length is used to select victim transactions. After that, three more algorithms utilising sorting techniques are introduced [10]. The RISU concept and the Weighted Sorting technique are applied to select victim items and transactions. Experimental results show that those algorithms have fewer side effects than previous heuristic algorithms. Despite this, missing information in PPUM is still a challenge.

Li et al. [6] introduced a more general approach. Instead of using a heuristic, they formulated the sanitisation process as a constraint satisfaction problem (CSP) and then solved the problem with integer programming techniques. PPUM-ILP [6] has better overall performances in running time and minimising side effects in the hiding process. However, in [25], Duc et al. pointed out that without the right strategies for preprocessing and establishing the hiding problem, the ILP approach can lead to a significant running time. They redesigned the data structures and CSP formulation process and found a faster implementation of PPUM-ILP, called FILP, for hiding sensitive information. In many cases, the heuristic approach for finding relaxation of the ILP problem in PPUM-ILP can lead to a significant running time. Therefore, we proposed a new algorithm also based on ILP with a mathematical method for finding relaxation. We also introduce a parallel preprocessing which utilises GPU.

3 Preliminaries

This section describes some related preliminaries and states the PPUM problem. Suppose that a quantitative transactional database D consists of N transactions: $D = \{T_1, T_2, ..., T_N\}$. Furthermore, suppose that transaction $T_n \in D (1 \leq n \leq N)$ contains one or more items and that $I = \{i_1, i_2, \ldots, i_M\}$ is the set of distinct items in D, $T_n \subset I$. Let each transaction

Table 1 A quantitative transactional database

tid	1	2	3	4	5
1	1	9	0	1	9
2	6	0	3	0	0
3	0	0	0	7	5
4	0	0	6	0	8
5	5	3	0	8	0
6	0	0	0	3	5
7	1	5	0	0	0
8	8	0	8	7	0
9	9	0	1	0	0
10	4	0	1	0	0

have a unique identifier, tid. An example of a quantitative transactional database is provided in Table 1.

Definition 1 *(Internal Utility) The internal utility of item i_m in transaction T_n is denoted as $\texttt{In}(i_m, T_n)$ and is the quantity of i_m in transaction T_n.*

For example, in Table 1, the internal utility of item 1 in transaction T_1 is 3, i.e., $\texttt{In}(1, T_1) = 1$.

Definition 2 *(External Utility) The external utility of item i_m denoted as $\texttt{Ex}(i_m)$ represents the importance of item i_m.*

For example, in Table 2, the external utility of item 2 is 1.42, i.e, $\texttt{Ex}(B) = 1.42$.

Definition 3 *(Utility of item i_m in transaction T_n) The utility of item i_m in transaction T_n is denoted as $u(i_m, T_n)$ and calculated as follows:*

$$u(i_m, T_n) = \texttt{In}(i_m, T_n) \times \texttt{Ex}(i_m)$$

For example, in Table 1, the utility of item 2 in transaction T_1 is: $u(2, T_1) = \texttt{In}(2, T_1) \times \texttt{Ex}(2) = 9 \times 1.42 = 12.78$.

Definition 4 *(Utility of itemset X in transaction T_n) The utility of itemset X in transaction T_n ($X \subseteq T_n$) is denoted as $u(X, T_n)$ and is the sum of the utilities of all the items in X in transaction T_n:*

$$u(X, T_n) = \sum_{i_m \in X} u(i_m, T_n)$$

Table 2 An external utility table

Item	External Utility
1	2.48
2	1.42
3	1.9
4	2.58
5	4.3

Note that we only compute $u(X, T_n)$ if $X \subseteq T_n$; otherwise, $u(X, T_n)$ will be zero. For example, in Table 1, the utility of itemset $\{1, 4\}$ in transaction T_1 is computed as follows: $u(\{1, 4\}, T_1) = u(1, T_1) + u(4, T_1) = 1 \times 2.48 + 1 \times 2.58 = 5.06$. Because $\{1, 4\} \nsubseteq 1, 4\}, T_2) = 0$.

Definition 5 *(Utility of itemset X in database D) The utility of itemset X in database D is computed by summing the utility of X in all transactions that support X: $u(X) = \sum_{T \in D, X \subseteq T_n} u(X, T_n)$.*

For example, the utility of itemset $\{1, 3\}$ in the database represented in Table 1 is calculated as follows: $u(\{1, 3\}) = u(\{1, 3\}, T_2) + u(\{1, 3\}, T_8) + u(\{1, 3\}, T_9) + u(\{1, 3\}, T_{10}) = (6 \times 2.48 + 3 \times 1.9) + (8 \times 2.48 + 8 \times 1.9) + (9 \times 2.48 + 1 \times 1.9) + (4 \times 2.48 + 1 \times 1.9) = 91.66$.

Definition 6 *(Minimum utility threshold) The minimum utility threshold is a criterion used in utility mining to define which itemsets yield high utility. Commonly, the minimum utility threshold δ, is a constant determined by the user, i.e., $\delta = c$ ($0 \leq c \leq \texttt{total utility}$).*

Definition 7 *(High-utility itemset - HUI) An itemset X is called a high-utility itemset when its utility in the database is greater or equal to the minimum utility threshold δ.*

Problem statement 1 *Given a database D, $\mathcal{H}$ is the set of all high-utility itemsets. Let $\mathcal{S}$ be the set that contains sensitive high-utility itemsets ($\mathcal{S} \subset \mathcal{H}$), as defined by the user. The main problem confronting PPUM is finding a suitable way to transform the database D into database D' in such a manner that all itemsets in $\mathcal{S}$ are concealed while minimising the negative effects on database D and the non-sensitive knowledge in $\mathcal{H}$ (Fig. 1).*

3.1 Side effects

A non-sensitive pattern that will not raise privacy concerns should be discoverable for the better benefit of the mining process. However, PPUM is a trade-off problem. The sanitisation process affects not only sensitive information but also non-sensitive information and creates a difference between the original database and the sanitised one. Lin et al. [5] proposed three evaluations to measure the extent of negative effects. Given the original database D and the sanitised database D', let $\mathcal{H}$ be the set of HUIs in D and $\mathcal{H}'$ be the set of HUIs in D'. Furthermore, let $\mathcal{N}$ be the set of NHUIS in D and $\mathcal{S}$ be the set of SHUIs in D.

Definition 8 *(Hiding Failure) The hiding failure signifies that SHUIs in the database can still be found as HUIs. Formally, let α denote the ratio of SHUIs in the database before and after the sanitisation process. Then α can be calculated as follows:*

$$\alpha = \frac{|\mathcal{S} \cap \mathcal{H}'|}{|\mathcal{S}|}$$

Definition 9 *(Missing Cost) The missing cost represents the ratio of certain non-sensitive high-utility itemsets (NSHUIs) in original database D that lose their utilities through the sanitisation process and cannot be discovered in perturbed database D'. Let β denote the ratio of the NSHUIs before and after the sanitisation. β can be calculated as follows:*

$$\beta = \frac{|\mathcal{N} - \mathcal{N} \cap \mathcal{H}'|}{|\mathcal{N}|}$$

Definition 10 *(Artificial Cost) The artificial cost represents the ratio of low-utility itemsets in D that become HUIs in D' after the sanitisation. Let γ be the ratio of NHUIs before and after sanitisation. γ can be calculated as follows:*

$$\gamma = \frac{|\mathcal{H}' - \mathcal{H} \cap \mathcal{H}'|}{|\mathcal{N}|}$$

In addition, [6] proposed an evaluation and used it to evaluate the PPUM-ILP algorithm in their publication.

Definition 11 *(Hiding Cost [6]) Let ϵ denoted the hiding cost, ϵ can be calculated as follows:*

$$\epsilon = \frac{|\mathcal{N} - \mathcal{H}'|}{|\mathcal{N}|}$$

The formula shows that the hiding cost cannot give us any information about the false HUIs (redundant HUIs) in D'. Therefore, in this paper, we did not evaluate results by hiding cost.

4 The proposed G-ILP algorithm

The key idea of PPUM based on integer programming is formulating the sanitisation process as a CSP problem to achieve two principal objectives: hiding the SHUIs and minimising the side effects. Our proposed algorithm has three main steps, as in PPUM-ILP [6]. In the first step, data structures for storing the relationships between itemsets and transactions are constructed. In the second step, the scale of the problem is reduced by keeping only the itemsets that will be affected by the perturbation process. Last, the CSP is established and solved with integer programming techniques, and then the CSP solution is used to perturb the original database. The overall process used in our G-ILP algorithm is described in Fig. 2.

4.1 Tables construction

The itemset selected as a sensitive itemset must be a SHUI. The G-ILP algorithm's input consists of the original database D, the set of SHUIs ($\mathcal{S}$) and the set of HUIs ($\mathcal{H}$). For preserving the mapping between the items in itemsets and the transactions supporting them. We designed a table structure that is similar to HI [6] and IT [25] called GIT. Additionally, the GIT keep track of the sizes and utilities of itemsets for parallel preprocessing and efficiently establish the CSP constraints. The sensitive GIT (S-GIT) table is constructed by parallel scanning of the original database D; for each SHUI s_i in $\mathcal{S}$, we store s_i itself, the tidset of the transactions supporting s_i, and its size and utility (Table 3).

The S-GIT table is used to speed up the process of establishing the constraints and finding special HUIs. It also helps avoid unnecessary scans of the database as HI tables. The non-sensitive high-utility GIT (N-GIT) table is also

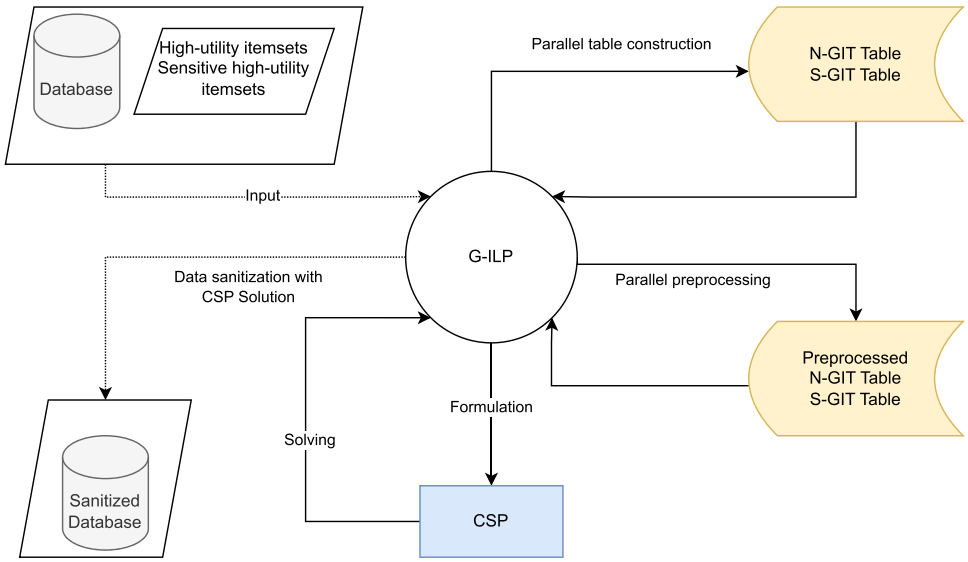

Fig. 2 Overall process of the proposed algorithm

Table 3 The GIT table structure

High-utility itemset	Tidset	Itemset size	Itemset utility
a_1	t_1	s_1	u_1
a_2	t_2	s_2	u_2
a_3	t_3	s_3	u_3

constructed as the same database scan stores information on NSHUIs.

Itemset sizes are helpful when used in the parallel preprocessing. Itemset utilities help compute the remaining utilities of itemsets when establishing the constraints as suggested by [25]. A specific description of the sequential table construction is represented in Algorithm 1.

4.2 Preprocessing strategy

In the sanitisation process, hiding the SHUI also affects the NSHUIs, because they share items and transactions. As a result, non-sensitive information can be lost or redundant information created. By filtering out itemsets that are affected by the hiding process, the hiding strategy will become more effective, and the size of the problem will be reduced. Our algorithm uses the same three-phase preprocessing strategy used in PPUM-ILP [6]. The first phase involves identifying the NSHUIs that will be affected by the hiding process with the filter mechanism described in the following.

Definition 12 *(Filter mechanism [6]) Let $\mathcal{S}$ denote the set of SHUIs and $\mathcal{N}$ denote the set of NSHUIs in the original database D, let the transactions that support itemset X be denoted by T_X. Itemset $n_i \in \mathcal{N}$ will not be removed from $\mathcal{N}$ if n_i meets the following requirements in conjunction with at least one sensitive itemset $s_i \in \mathcal{S}$:*

1. *Itemset n_i and itemset s_i have at least one item in common, i.e, $n_i \cap s_i \neq \emptyset$*
2. *Tidset T_{n_i} and tidset T_{s_i} have at least one transaction in common, i.e, $T_{n_i} \cap T_{s_i} \neq \emptyset$.*

Let $\mathcal{N}'$ be the set of remaining NSHUIs in $\mathcal{N}$ after filtering. The next phase consists of finding itemsets in $\mathcal{N}'$ that are certain to be hidden during the hiding process (bound-to-lose itemsets) to avoid conflicts when establishing inequalities.

Definition 13 *(High-utility closed itemset (HUIC) [26]) Given itemset X and the transaction set T_X that supports X, X is called a high-utility closed itemset if X satisfies the following conditions:*

1. *X is high-utility: $u(X) \geq \delta$.*
2. *X has no superset that supports by the same transaction set as X: $T_X \neq T_Y, \forall Y \supset X$.*

Definition 14 *(Bound-to-lose itemset [6]) Given the NSHUI X and the transaction set T_X that supports X, X is a bound-to-lose itemset if $\exists Y \in \mathcal{S}$, X is not closed to Y.*

Let $\mathcal{N}''$ be the set of remaining NSHUIs. The final phase involves removing itemsets that are unnecessary for the establishment of the constraints process. In particular, if itemset $n_{i_1} \in \mathcal{N}''$ is not closed to another itemset $n_{i_2} \in \mathcal{N}''$, n_{i_2} can be removed from $\mathcal{N}''$ without increasing the missing cost. Finally, we obtained $\mathcal{R}$ as the preprocessing output. Sequential preprocessing is described in Algorithm 2.

Algorithm 1 Table Construction.

Input: Original database D; the set of HUIs I_H; the set of user-defined SHUIs I_S
Output: N-GIT, S-GIT tables.

```
for each transaction T_n ∈ D do
    for each itemset X ∈ I_H do
        if X ⊆ T then
            if X ∈ I_S then
                Move X, TID of T_n, X.size, u(X) to S-GIT.
            else
                Insert X, TID of T_n, X.size, u(X) into N-GIT.
            end if
        end if
    end for
end for
```

Algorithm 2 Preprocessing.

Input: N-GIT, S-GIT tables
Output: Modified N-GIT, S-GIT tables.

```
for each itemset n_i in N-GIT do
    L ← ∅
    for each itemset s_i in S-GIT do
        if n_i ∩ s_i ≠ ∅ and T_{n_i} ∩ T_{s_i} ≠ ∅ then
            if n_i ⊆ s_i and T_{n_i} = T_{s_i} then
                Remove n_i from N-GIT
            else
                Insert s_i into L
            end if
        end if
        if L == S then
            Remove n_i from N-GIT
        end if
    end for
    for each itemset n_{i_2} in N-GIT do
        for each itemset n_{i_1} in N-GIT do
            if n_{i_1} ≠ n_{i_2} and n_{i_1} ⊆ n_{i_2} and T_{n_{i_1}} = T_{n_{i_2}} then
                Remove n_{i_2} from N-GIT
                Break
            end if
        end for
    end for
end for
```

4.3 Parallel preprocessing implementation

GPUs can be used to speed up a compute-intensive process, but most of them need to be treated as a coprocessor, and their asynchronous nature makes them relatively hard to program. Since development for GPU accelerators works with low-level languages and lacks abstractions, it requires parallel programming experience and domain knowledge to map the problems [27]. This section will describe important parts of the implementation of the proposed parallel preprocessing method. First, the data structures representing the itemsets (and tidsets) are redefined for parallel computing.

4.3.1 Data structure

To utilise a GPU in the preprocessing stage, we change the data structures storing the itemsets and tidsets. Each itemset is represented by a BitList [28] of size M, where M is the number of distinct items in the database D. Each tidset is also represented by a BitList size N, where N is the number of transactions in the database. In BitList representations, the number of items in the intersection of two sets is calculated via a dot product of two bit vectors. That is, given that $\mathbf{a}$ is the bit vector of itemset A and $\mathbf{b}$ is the bit vector of itemset B, the number of items in the intersection of the two itemsets A and B is $\mathbf{a} \cdot \mathbf{b}$. Furthermore, given that $P = \{p_1, p_2, ..., p_x\}$ contains x distinct itemsets, it can be represented by a BitTable [28] that includes x BitLists of size g. Additionally, let $Q = \{q_1, q_2, \ldots, q_y\}$ include y distinct itemsets and be represented by a BitTable including y BitLists of size g. Then the number of items in the intersection of the itemsets in P and Q is found via the matrix multiplication of the bit matrix of P and the transposed bit matrix of Q. Formally, let $\mathbf{P}$ $(x \times g)$ be the bit matrix of P, $\mathbf{Q}$ $(y \times g)$ be the bit matrix of Q and $\mathbf{R}$ $(x \times y)$ be the result of the matrix multiplication between $\mathbf{P}$ and $\mathbf{Q}^T$: $\mathbf{R} = \mathbf{P}\mathbf{Q}^T$. Then r_{ij} represents the number of items in the intersection of the two itemsets p_i and q_j $(p_i \in P, q_j \in Q)$.

For example:

$$\mathbf{P} = \begin{bmatrix} 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 \end{bmatrix}, \mathbf{Q} = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 \end{bmatrix}$$

$$\mathbf{R} = \mathbf{P}\mathbf{Q}^T = \begin{bmatrix} 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 1 \\ 0 & 1 \\ 1 & 0 \\ 0 & 0 \end{bmatrix} = \begin{bmatrix} 2 & 1 \\ 1 & 1 \end{bmatrix}$$

The BitTable structure allows parallel computing of the cardinality of intersection sets between GIT tables and can be combined with parallel data accessing methods to significantly speed up preprocessing.

4.3.2 N-GIT and S-GIT tables construction

Theorem 1 *Given two sets A and B, I is the intersection of A and B: $I = A \cap B$. If $|A| = |I|$, then $A \subseteq B$.*

Proof Because $I = A \cap B$, we have $i \in A \wedge i \in B, \forall i \in I$. If $|A| = |I|$, then $A = I$. Since $I \subseteq B$, we also have $A \subseteq B$.

With GPU threads organised in a block, for all preprocessing phases, each thread in a block will be identified using two dimensions with size 16 for each dimension. For each preprocessing phase, the cardinality of the intersection of itemsets or transactions are computed by matrix multiplication and stored in an array in GPU memory. Based on Theorem 1, a parallel algorithm was carefully designed for constructing the N-GIT and S-GIT tables. Let I_D be the BitTable representing the itemsets in D. First, we initialise the N-GIT table with all the itemsets in $\mathcal{H}$. Then, we compute the number of items in the intersection of the itemsets in I_D with each itemset in $\mathcal{H}(I_{DH})$ as well as the itemsets in $\mathcal{H}$ with each itemset in $\mathcal{S}(I_{HS})$ via matrix multiplications on the GPU. After that, we parallel scan I_{DH} and I_{HS} to store the tidsets of the itemsets and move sensitive itemsets from the N-GIT table to the S-GIT table. In addition, for this phase, we use a three-dimensional grid with sizes $\left\lceil \frac{|\mathcal{H}|}{16} \right\rceil, \left\lceil \frac{|D|}{16} \right\rceil, |\mathcal{S}|$. This setting is chosen specifically for its importance in I/O in GPU memory. As shown in Algorithm 3, GPU threads are retrieved mostly with tids as indexes. With these settings, each block loads data (by tids) in an orderly manner. Moreover, with the block index acting as the sIdx, data in $\mathcal{S}$ are loaded efficiently in each block.

The specific process with recommended settings (dimensions of GPU arrays for data storing and retrieval) for the kernel function is described in Algorithm 3.

Algorithm 3 GPU Table Construction.

Input: I_D, BitTable representing the itemsets in database D; $\mathcal{H}$, the set of HUIs discovered in D; $\mathcal{S}$, the set of SHUIs to be hidden
Output: N-GIT, S-GIT tables.

```
Initialize N-GIT with itemsets in H
I_DH = matmul(I_D, H)
I_HS = matmul(H, S)
for threadIdx in threads do
    hIdx = threadIdx.x
    tid = threadIdx.y
    sIdx = threadIdx.z
    if I_DH[tid][hIdx] == H[hIdx].size then
        Insert tid to N-GIT[hIdx]
    end if
    if I_HS[hIdx][sIdx] == H[hIdx].size == S[sIdx].size then
        Move N-GIT[hIdx] to S-GIT
    end if
end for
```

Algorithm 4 GPU Preprocessing.

Input: N-GIT, S-GIT tables
Output: Modified N-GIT, S-GIT tables.

```
1: I_NS = matmul(N, S)
2: T_NS = matmul(T_N, T_S)
3: for threadIdx in threads do
4:     nIdx = threadIdx.x
5:     sIdx = threadIdx.y
6:     if T_NS > 0 and I_NS > 0 then
7:         if I_NS[nIdx][sIdx] == N [nIdx].size and T_N[nIdx].size ==
   T_S[sIdx].size == T_NS[nIdx][sIdx] then
8:             Remove N[nIdx] from N-GIT
9:         else
10:            L[nIdx] += 1
11:        end if
12:        if L[nIdx] == number of sensitive itemsets then
13:            Remove N[nIdx] from N-GIT
14:        end if
15:    end if
16: end for
17: I_NN = matmul(N', N')
18: T_NN = matmul(T'_N, T'_N)
19: for threadIdx in threads do
20:    n_1 = threadIdx.x
21:    n_2 = threadIdx.y
22:    if n_1 ≠ n_1 then
23:        if I_NN[n_1][n_2] == N'[n_1].size and T_NN[n_1][n_2] ==
   T'_N[n_1].size == T'_N[n_2].size then
24:            Remove N'[n_2] from N-GIT
25:        end if
26:    end if
27: end for
```

4.3.3 Preprocessing

Based on Theorem 1, we also designed an algorithm for the parallel preprocessing of the data using a GPU (details in Algorithm 4. First, intersection between BitTables of $\mathcal{N}$ and $\mathcal{S}$ are computed by GPU matrix multiplications. Then, we use GPU threads to parallel access the intersection sizes that are stored in GPU memory and determine which itemsets will not be affected and should not be constrained in the CSP problem. Second, intersection cardinalities between BitTables of $\mathcal{N}$ and itself are also computed and parallel scanned to determine redundant itemsets indexes. Finally, the indexes are used to modify the G-IT tables and obtain the G-IT table for $\mathcal{R}$.

To identify which itemset will not be affected by the hiding process, a set of itemsets in $\mathcal{S}$ needs to be kept for each non-sensitive itemset. Given that n_i is an NHUIS, a set of SHUIs that does not share any common items and transactions with n_i is denoted as L. If $L = \mathcal{S}$, n_i will be removed from $\mathcal{N}$. Storing and comparing a set of itemsets in GPU memory is slow, complex and hard. A better method needs to be proposed. Let L_{n_i} denote the number of SHUIs that do not have items in common with the NSHUI n_i. If L_{n_i} contains the number of itemsets in $\mathcal{S}$, we remove n_i from $\mathcal{N}$. L_{n_i} is stored in GPU shared memory to improve the performance, and GPU threads are used for parallel counting L_{n_i}. However, the GPU execution is asynchronous, and multiple threads can write data to L_{n_i} simultaneously, which causes errors or unexpected results. The "atomic add" operator must be used to write back results to GPU memory without interference from other threads. The details of parallel preprocessing are described in Algorithm 4.

4.4 CSP formulation

The goal of the PPUM algorithm is to hide SHUIs. To hide the SHUIs efficiently, we change the internal utility values. The CSP is designed to find the optimal internal utility value for hiding all the SHUIs and minimising the unwanted effect on the NSHUIs. First, we replace the internal utilities of all the items in the SHUIs in $\mathcal{S}$ with integer variables and create a hash table to store variable positions and utilities. Let V be the set of integer internal utility of item m in transaction n. To remove SHUI X we have the following constraint:

$$\sum_{T_n \in D, X \subseteq T_n} \sum_{i_m \in X} v_{nm} Ex(i_m) < \delta \tag{1}$$

Let U be the set of internal utilities of items in $\mathcal{R}$:

$$U = \left\{ u_{n'm'} | n' \in [1, N], m' \in [1, M] \right\}.$$

where $u_{n'm'}$ represents the internal utility of each item in $\mathcal{R}$ in two states, i.e.:

$$u_{n'm'} = \begin{cases} v_{n'm'}, & \texttt{If } v_{n'm'} \in V, \\ \texttt{In}(i_{m'}, T_{n'}), & \texttt{Otherwise} \end{cases}$$

To keep itemset $Y \in \mathcal{R}$, we have the following constraint:

$$\sum_{T_{n'} \in D, Y \subseteq T_{n'}} \sum_{i_{m'} \in Y} u_{n'm'} Ex(i_{m'}) \geq \delta \tag{2}$$

The minimum value of an integer variable should be one. The reason for this suggestion is quite simple. If we change the internal utility of an item in transaction T_n to zero, the itemset containing this item will lose its utility in transaction T_n since it will no longer be supported by T_n. In addition, in a dense dataset, one item can be supported by many transactions. If its internal utility is high, many LUIs may become HUIs, increasing the artificial cost and leading to misleading information. Therefore, in a dense dataset, we suggest that the upper bound of the integer variables should be set to the maximum value of internal utilities in the original database. The objective function then minimises the sum of the integer variables. Because the brute force formulation strategy makes the ILP approach impossible to run in large datasets [25], in

this paper, we use the same implementation of the CSP formulation with [25].

4.5 Relaxation

Although some NSHUIs that can create conflict constraints are removed during preprocessing, the CSP does not always have a feasible solution. Therefore, we need an efficient way to modify the CSP to create a feasible solution. A feasibility relaxation is a solution to the original model that minimises the number of violated constraints. In this paper, we designed a new relaxed model. In particular, the constraints created with the N-GIT table can be violated. Let c be the number of itemsets in the N-GIT table. The penalty for violating the constraint is 1.0. The objective of feasibility relaxation is to minimise the sum of the weighted magnitudes of the constraint violations. That is, if a constraint is violated by 2.0, it will contribute 2.0×1.0 to the feasibility relaxation objective.

Formally, let c_Y be the violated value of the constraint that corresponds to itemset $Y \in \mathcal{R}$. The relaxed model can be described as follows:

$$min \sum_{v_{nm} \in V} v_{nm} + \sum_{Y \in R} c_Y \tag{3}$$

$$s.t. \sum_{i_m \in X} \sum_{T_n \in T_X} v_{nm} Ex(i_m) \leq \delta, \qquad \forall X \in \mathcal{S} \tag{4}$$

$$\sum_{i_{m'} \in Y} \sum_{T_{n'} \in T_Y} u_{n'm'} Ex(i_{m'}) + c_Y \geq \delta, \qquad \forall Y \in \mathcal{R} \tag{5}$$

$$1 \leq v_{nm} \leq max(\texttt{internal utilities}), \ \forall v_{nm} \in V \tag{6}$$

The original database is perturbed according to the relaxation. This new model helps reduce the execution time since the model can be updated only once and solved mathematically instead of using heuristic to iteratively remove constraints, update and resolve the CSP model as presented in [6].

5 Experimental results

5.1 Experimental environment and datasets

We conducted the experiments on an Intel(R) Core(TM) i7-6700 CPU @ 3.40GHz, RAM 32GB; NVIDIA GEFORCE RTX2080, VRAM 8GB. The number of transactions is denoted as N, and I is the set of items. The maximum itemsets of a dataset is $2^{|I|}$, and the maximum number of tidsets is 2^N. The small datasets used to evaluate algorithm performances are mushrooms and foodmart. The large datasets are t25i10d10k1 (maximum 2^{9976} tidsets) and t20i6d100k1 (maximum 2^{99922} tidsets).

The datasets characteristics are described in Table 4. The density of a dataset is computed by the average transaction length m_t divided by the number of unique items: $\texttt{Density} = \frac{m_t}{|I|}$. The external utility of each item is generated in the interval $\mathbb{R} \cap [1, 10]$ with the log-normal distribution, whereas the internal utilities of the items in the transactions are generated to be between 1 and 10 using the uniform distribution. In this paper, we used the d2HUP algorithm [29] as the mining technique. The performance of the proposed algorithm was evaluated and compared with other algorithms across different datasets in terms of running time and ability to minimise side effects. The algorithms based on modifying the original data approach (Fig. 1), HHUIF and MSICF [2], were selected for baseline algorithms; MSU-MAU and MSU-MIU [4] are the fastest executing heuristic algorithms [10]; and FILP [25] is an efficient reimplementation of PPUM-ILP. All these algorithms were implemented with the high-performance language Julia version 1.6.6. The CSP was solved with the Gurobi solver for integer programming [30].

5.2 Running time

The experiments analysing running time were conducted under two scenarios: increasing the minimum utility threshold δ and increasing the sensitive information percentage (SIP). Observing the results in Figs. 3, 4, 5, 6, we see that the algorithm execution time does not always decrease as the minimum utility threshold δ increases. The higher minimum utility threshold with a fixed percentage of sensitive information causes the number of HUIs to decrease and may lead to lower running times. However, the higher minimum utility threshold also means more frequent SHUIs appear in the database, which makes the hiding problem harder to solve and slows the sanitisation. With a fixed percentage of sensitive information, the number of SHUIs also decreases. The advantage of the proposed algorithm G-ILP is in large and sparse datasets, as we can see in Fig. 10. The reason for this advantage is that heuristic algorithms need to scan the database multiple times to hide the SHUIs iteratively.

Table 4 Datasets characteristics

Dataset	N	$\|I\|$	Density(%)
Mushrooms	8124	119	19.3
Foodmart	4141	1559	0.25
T25I10D10K	9976	893	2.66
T20I6D100K	99922	929	2.22

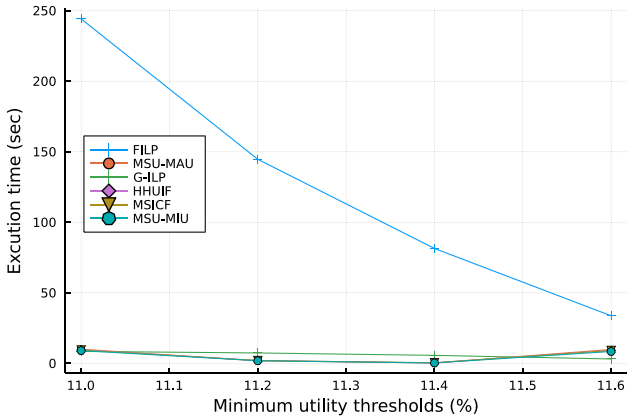

Fig. 3 Running times of the algorithms under different minimum utility thresholds on mushrooms dataset

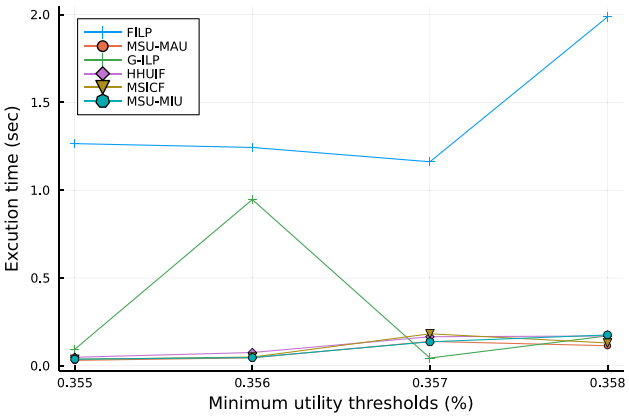

Fig. 5 Running times of the algorithms under different minimum utility thresholds on t25i10d10k dataset

Consequently, the larger the dataset, the greater the execution time. G-ILP and FILP only scan the database twice, once to construct the GIT tables and once to hide sensitive itemsets. Additionally, t20i6d100k is a sparse dataset in which a particular itemset does not appear in many transactions. Therefore, the constraints on the CSP are less likely to collide, resulting in shorter solving times. From the results in t20i6d100k, the FILP and G-ILP algorithms can be faster than the others. In general, the proposed algorithm G-ILP always has a lower execution time. The difference between the G-ILP and FILP algorithms can be seen clearly in t25i10d10k1. When the minimum utility threshold is set at 0.356%, G-ILP's execution time increases while FILP's execution time decreases. Even for a small and dense dataset like mushrooms, G-ILP remains superior.

The graphs in Figs. 7, 8, 9, 10 show the execution times of the algorithms for the four datasets as the percentage of sensitive information is increased. From the results, we can observe that the execution times of all the algorithms tend to increase in most cases. When we increase the percentage of sensitive information while maintaining a fixed minimum utility threshold, the number of SHUIs increases. Consequently, the computational work must increase, especially for heuristic algorithms. Heuristic algorithms have to scan a maximum 2^N tidset to hide each itemset. The more sensitive itemset and the more transactions in the dataset, the more execution time is needed. G-ILP and FILP scan the database a fixed number of times, so the time consumed by preprocessing mainly depends on the number of HUIs. Thus, the execution times do not change much, so ILP approaches have better scalability than heuristic approaches. Furthermore, in most cases, the CSP does not have an optimal solution. With the mathematical relaxation, the time consumed by G-ILP does not increase much. Beyond that, as shown in Fig. 6, the proposed algorithm also achieves better results than heuristic algorithms for large and sparse datasets.

5.3 Side effects

To analyse the side effects created by the sanitisation process, we ran the algorithms with different percentages of sensitive information (SIPs) and used three evaluation mea-

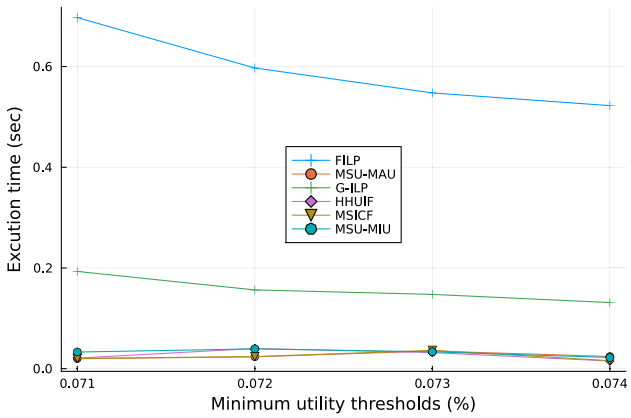

Fig. 4 Running times of the algorithms under different minimum utility thresholds on foodmart dataset

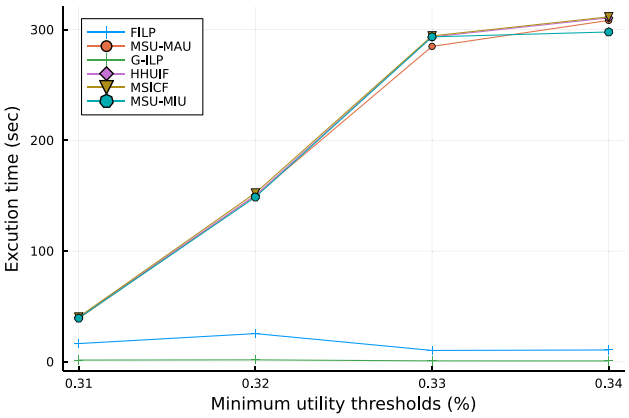

Fig. 6 Running times of the algorithms under different minimum utility thresholds on t20i6d100k dataset

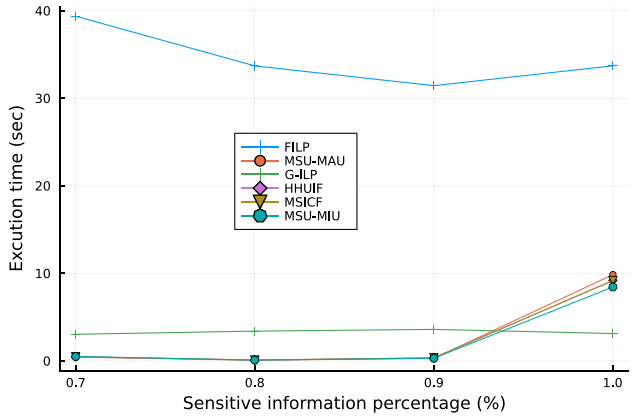

Fig. 7 Running times of the algorithms under different percentages of sensitive patterns on mushrooms dataset

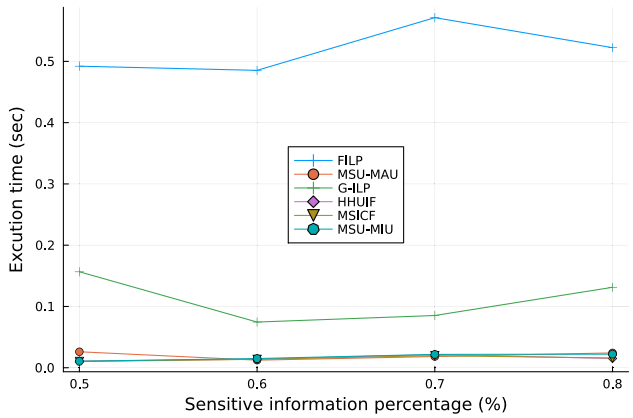

Fig. 8 Running times of the algorithms under different percentages of sensitive patterns on foodmart dataset

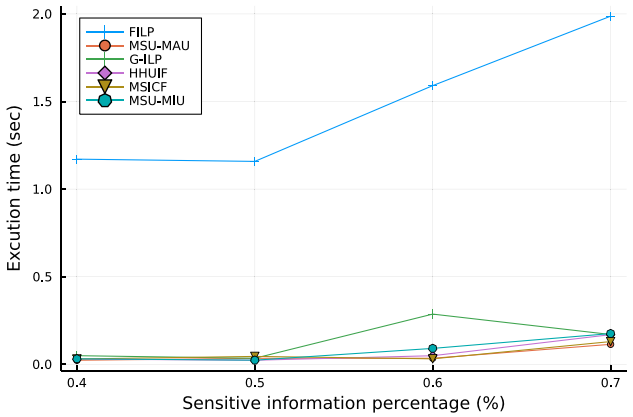

Fig. 9 Running times of the algorithms under different percentages of sensitive patterns on t25i10d10k dataset

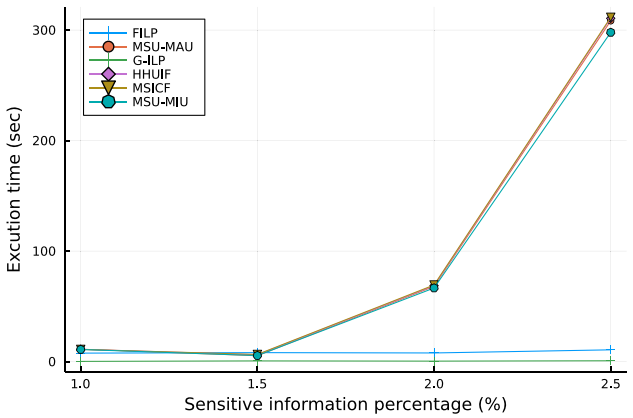

Fig. 10 Running times of the algorithms under different percentages of sensitive patterns on t20i6d100k dataset

Table 5 Side effects of the HHUIF, MSICF, MSU-MAU and MSU-MIU algorithms in different percentages of sensitive information

Dataset	SIP (%)	HHUIF β(%)	MSICF β(%)	MSU-MAU β(%)	MSU-MIU β(%)
mushrooms	0.7	86.7	85.5	84.6	91.2
	0.8	94.3	82.9	95.1	93.0
	0.9	88.6	90.6	89.6	92.3
	1.0	99.7	52.5	99.7	98.4
foodmart	0.5	44.3	44.4	44.3	24.3
	0.6	46.3	44.3	46.3	36.0
	0.7	44.5	44.3	44.5	34.9
	0.8	44.6	44.1	44.6	34.8
t25i10d10k	0.4	12.8	13.1	15.2	9.3
	0.5	3.0	3.0	3.0	3.0
	0.6	28.1	23.4	30.8	21.3
	0.7	24.6	21.6	17.4	18.0
t20i6d100k	1.0	0.0	0.0	0.0	0.0
	1.5	17.8	17.8	14.9	12.1
	2.0	26.6	11.1	24.3	16.2
	2.5	48.3	48.3	41.9	33.1

Table 6 Side effects of the two versions of G-ILP in different percentages of sensitive information

Dataset	SIP (%)	FILP		G-ILP	
		β(%)	γ(%)	β(%)	γ(%)
mushrooms	0.7	78.7	950.5	53.0	160.7
	0.8	74.1	462.3	46.5	210.6
	0.9	79.6	382.7	49.8	130.8
	1.0	57.4	810.0	58.6	238.8
foodmart	0.5	26.1	10.5	19.3	4.4
	0.6	6.6	99.7	7.0	16.8
	0.7	14.9	58.2	12.0	45.4
	0.8	20.1	43.3	7.3	35.0
t25i10d10k	0.4	0.0	695.5	0.0	695.5
	0.5	0.0	0.0	0.0	0.0
	0.6	3	72.2	48.5	150.6
	0.7	29.3	150.3	29.3	108.4
t20i6d100k	1.0	0	0	0	0
	1.5	29.9	282.8	0.6	270.7
	2.0	0	71.7	0	71.1
	2.5	33.1	475.0	5.2	196.5

sures: hiding failure, missing cost and artificial cost. The FILP algorithm used the same CSP model and the method for finding relaxation with PPUM-ILP. The side effects of FILP can also be considered as the side effects of PPUM-ILP. Because all the algorithms were successful in hiding all SHUIs, we will focus on the missing cost and artificial cost. In Table 5 there are no artificial costs attached to the four algorithms HHUIF, MSICF, MSU-MAU and MSU-MIU because these algorithms only reduce the internal utilities of items or delete items in transactions.

In general, the missing costs for the G-ILP and FILP algorithms are much lower than those for the other algorithms in most cases because of the specific constraints (corresponding to inequality Section 5) to retain NHUIs. As for the dense dataset mushrooms, all the algorithms lost more than 80% of non-sensitive information after the sanitisation algorithm. From Table 5, we can see that MSICF is slightly better than the other heuristic algorithms in this dataset. With 1.0% of sensitive information, MSICF only had a 57.4% loss in the non-sensitive pattern. For the ILP approaches, no constraints were used to limit the amount of redundant information except the objective function in the CSP model. As a result, the ILP algorithms introduced many false HUIs. However, the proposed algorithm always had a lower artificial cost than the FILP algorithm.

The results of the algorithms on the foodmart dataset in Figs. 8, 4, Table 5 and Table 6 show that the ILP approaches performed well. However, we can observe a phenomenon with the 0.7% SIP. The number of missing patterns in G-ILP is slightly lower than in FILP, but in contrast, G-ILP adds a large number of redundant patterns to the database.

In the t25i10d10k1 dataset, there is one case where the hiding problem has an optimal solution. The ILP approaches found the solution and had a perfect result. Furthermore, we see that increasing the percentage of sensitive patterns does not always make the problem harder due to the sensitive patterns being chosen randomly. As with the large and sparse dataset t20i6d100k, G-ILP not only surpassed the other algorithms in execution time, Table 6 also shows that G-ILP is superior to FILP in missing cost and artificial cost.

In conclusion, the ILP approaches have a good ability to retain non-sensitive information but introduce much false information. The proposed G-ILP algorithm had the lowest running time and the best ability to minimise the negative effects of the hiding process in all privacy-preserving algorithms based on ILP. Although the heuristic approaches tend to run faster than the ILP approaches in small datasets, G-ILP can still achieve equivalent performance.

6 Discussion

In this paper, we proposed a new algorithm for privacy-preserving utility mining based on integer programming G-ILP. Experiments on typical datasets demonstrated the strength of our algorithm. However, the parallel preprocessing has some limitations. The data structure used for parallel preprocessing is inefficient in terms of memory, especially for sparse datasets, since there are too many zero-valued bits in the BitLists it generates. In some cases, the internal utilities of items in sensitive itemsets are not higher than one, and the only way to hide that itemset is by removing it from the database. Since the lower bound of integer variables in the proposed model is one, the hiding problem and its relaxed problem cannot be solved. For this reason, we should design a new relaxed problem for integer variables. Additionally, the artificial costs of ILP algorithms are high, and we need specific constraints to reduce the artificial cost. Another potential approach for future work is to apply metaheuristic optimisation methods to solve the hiding problem.

Acknowledgements This research is funded by University of Science, VNU-HCM under grant number CNTT 2021-05

References

1. Yun U, Kim D (2017) Analysis of privacy preserving approaches in high utility pattern mining. In: Park, J.J.J.H., Pan, Y., Yi, G., Loia, V. (eds.) Advances in Computer Science and Ubiquitous Computing, Singapore, pp. 883–887. https://doi.org/10.1007/978-981-10-3023-9_137

2. Yeh J-S, Hsu P-C (2010) HHUIF and MSICF: Novel algorithms for privacy preserving utility mining. Expert Syst Appl 37(7):4779–4786. https://doi.org/10.1016/j.eswa.2009.12.038
3. Lin C-W, Hong T-P, Wong J-W, Lan G-C, Lin W-Y (2014) A GA-based approach to hide sensitive high utility itemsets. Sci World J 2014:2356–6140. https://doi.org/10.1155/2014/804629
4. Lin JC-W, Hong T-P, Fournier-Viger P, Liu Q, Wong J-W, Zhan J (2017) Efficient hiding of confidential high-utility itemsets with minimal side effects. J Exp Theoretical Artif Intell 29(6):1225–1245. https://doi.org/10.1080/0952813X.2017.1328462
5. Lin JC-W, Wu T-Y, Fournier-Viger P, Lin G, Zhan J, Voznak M (2016) Fast algorithms for hiding sensitive high-utility itemsets in privacy-preserving utility mining. Eng Appl Artif Intell 55:269–284. https://doi.org/10.1016/j.engappai.2016.07.003
6. Li S, Mu N, Le J, Liao X (2019) A novel algorithm for privacy preserving utility mining based on integer linear programming. Eng Appl Artif Intell 81:300–312. https://doi.org/10.1016/j.engappai.2018.12.006
7. Liu X, Chen G, Wen S, Song G (2020) An improved sanitization algorithm in privacy-preserving utility mining. Mathematical Problems in Engineering 2020:1–14. https://doi.org/10.1155/2020/7489045
8. Liu X, Wen S, Zuo W (2020) Effective sanitization approaches to protect sensitive knowledge in high-utility itemset mining. Appl Intell 50(1):169–191. https://doi.org/10.1007/s10489-019-01524-2
9. Jangra S, Toshniwal D (2022) Efficient algorithms for victim item selection in privacypreserving utility mining. Future Gener Comput Syst 128:219–234. https://doi.org/10.1016/j.future.2021.10.008
10. Ashraf M, Rady S, Abdelkader T, Gharib TF (2023) Efficient privacy preserving algorithms for hiding sensitive high utility itemsets. Comput Sec 103360. https://doi.org/10.1016/j.cose.2023.103360
11. Yao H, Hamilton HJ, Butz CJ (2004) A foundational approach to mining itemset utilities from databases. In: Proceedings of the 2004 SIAM International Conference on Data Mining, pp. 482–486. https://doi.org/10.1137/1.9781611972740.51
12. Liu Y, Liao W-K, Choudhary A (2005) A two-phase algorithm for fast discovery of high utility itemsets. In: Ho TB, Cheung D, Liu H (eds.) Advances in Knowledge Discovery and Data Mining, Berlin, Heidelberg, pp. 689–695. https://doi.org/10.1007/1143091979
13. Lin C-W, Hong T-P, Lu W-H (2011) An effective tree structure for mining high utility itemsets. Expert Syst Appl 38(6):7419–7424. https://doi.org/10.1016/j.eswa.2010.12.082
14. Tseng VS, Wu C-W, Shie B-E, Yu PS (2010) Up-growth: An efficient algorithm for high utility itemset mining. In: Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. KDD-10, pp. 253–262. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/1835804.1835839
15. Tseng VS, Shie B, Wu C, Yu PS (2013) Efficient algorithms for mining high utility itemsets from transactional databases. IEEE Trans Knowl Data Eng 25(8):1772–1786. https://doi.org/10.1109/TKDE.2012.59
16. Liu M, Qu J (2012) Mining high utility itemsets without candidate generation. In: Proceedings of the 21st ACM International Conference on Information and Knowledge Management. CIKM-12, pp. 55–64, New York, NY, USA. https://doi.org/10.1145/2396761.2396773
17. Fournier-Viger P, Wu C-W, Zida S, Tseng VS (2014) FHM: Faster high-utility itemset mining using estimated utility co-occurrence pruning. In: Andreasen T, Christiansen H, Cubero J-C, Ra's ZW (eds.) Foundations of Intelligent Systems, Cham, pp. 83–92. https://doi.org/10.1007/978-3-319-08326-19
18. Krishnamoorthy S (2015) Pruning strategies for mining high utility itemsets. Expert Syst Appl 42(5):2371–2381. https://doi.org/10.1016/j.eswa.2014.11.001
19. Zida S, Fournier Viger P, Lin C-W, Wu C-W, Tseng V (2016) EFIM: a fast and memory efficient algorithm for high-utility itemset mining. Knowl Inf Syst 51:595–625. https://doi.org/10.1007/s10115-016-0986-0
20. Lin JC-W, Gan W, Fournier-Viger P, Hong T-P (2015) Mining high-utility itemsets with multiple minimum utility thresholds. In: Proceedings of the Eighth International C* Conference on Computer Science & Software Engineering. C3S2E-15, pp. 9–17, New York, NY, USA. https://doi.org/10.1145/2790798.2790807
21. Gan W, Lin JC-W, Chao H-C, Fournier-Viger P, Wang X, Yu PS (2020) Utility-driven mining of trend information for intelligent system. ACM Trans Manag Inf Syst 11(3). https://doi.org/10.1145/3391251
22. Vo B, Nguyen LTT, Nguyen TDD, Fournier-Viger P, Yun U (2020) A multi-core approach to efficiently mining high-utility itemsets in dynamic profit databases. IEEE Access 8:85890–85899. https://doi.org/10.1109/ACCESS.2020.2992729
23. Yun U, Nam H, Kim J, Kim H, Baek Y, Lee J, Yoon E, Truong T, Vo B, Pedrycz W (2020) Efficient transaction deleting approach of pre-large based high utility pattern mining in dynamic databases. Futur Gener Comput Syst 103:58–78. https://doi.org/10.1016/j.future.2019.09.024
24. Yun U, Kim J (2015) A fast perturbation algorithm using tree structure for privacy preserving utility mining. Expert Syst Appl 42(3):1149–1165. https://doi.org/10.1016/j.eswa.2014.08.037
25. Nguyen D (2022) Le B (2022) A fast algorithm for privacy-preserving utility mining. J Inf Technol Commun 1:12–22. https://doi.org/10.32913/mic-ict-research.v2022.n1.1026
26. Wu C, Fournier-Viger P, Gu J, Tseng VS (2015) Mining closed+ high utility itemsets without candidate generation. In: 2015 Conference on Technologies and Applications of Artificial Intelligence (TAAI), pp. 187–194. https://doi.org/10.1109/TAAI.2015.7407089
27. Li X, Shih P-C, Overbey J, Seals C, Lim A (2016) Comparing programmer productivity in openacc and cuda: An empirical investigation. Intern J Comput Sci, Eng Appl (IJCSEA) 6(5):1–15
28. Dong J, Han M (2007) BitTableFI: An efficient mining frequent itemsets algorithm. Knowl-Based Syst 20(4):329–335. https://doi.org/10.1016/j.knosys.2006.08.005
29. Liu J, Wang K, Fung BC (2012) Direct discovery of high utility itemsets without candidate generation. In: 2012 IEEE 12th International Conference on Data Mining, pp. 984–989. https://doi.org/10.1109/ICDM.2012.20. IEEE
30. Gurobi Optimization L (2020) Gurobi Optimizer Reference Manual. http://www.gurobi.com