

# GRIMP: A Genetic Algorithm for Compression-based Descriptive Pattern Mining

M. Zohaib Nawaz<sup>1,2</sup>[0000–0001–9205–912X], M. Saqib  
Nawaz<sup>1</sup>[0000–0001–9856–2885], Philippe Fournier-Viger<sup>1</sup>[0000–0002–7680–9899], and  
Nazha Selmaoui-Folcher<sup>3</sup>[0000–0003–1667–3819]

<sup>1</sup> College of Computer Science and Software Engineering, Shenzhen University, China

<sup>2</sup> Faculty of Computing and Information Technology, Department of Computer  
Science, University of Sargodha, Pakistan

<sup>3</sup> ISEA, University of New Caledonia, New Caledonia  
zohaib.nawaz@uos.edu.pk, {nawazmuhammadzohaib2022, msaqibnawaz,  
philfv}@szu.edu.cn, nazha.selmaoui@unc.nc

**Abstract.** Traditional frequent pattern mining algorithms often report an overwhelming number of patterns in large datasets, many of which are redundant. To address this issue, Minimum Description Length (MDL)-based methods have been employed, which use data compression to capture a smaller yet significant set of patterns. However, finding a good set of patterns according to MDL involves a very large search space and current MDL-based techniques often suffer from long runtimes and find suboptimal solutions. To discover better sets of patterns in less time, this paper introduces GRIMP (a Genetic algoRithm for coMpression-based descriptive Pattern mining), a novel framework that combines a genetic algorithm with MDL-based pattern selection. Multiple genetic algorithm variants are explored within the GRIMP framework, and their effectiveness is compared using a large number of datasets. Experimental results demonstrate that GRIMP consistently outperforms previous methods by achieving higher compression ratios, generating more representative itemsets, and requiring less time. Additionally, the extracted patterns improve downstream classification tasks, highlighting the ability of GRIMP to find more representative patterns within the data.

**Keywords:** Pattern Mining · Genetic Algorithm · Data Compression · Minimum Description Length.

**Funding.** The authors declare that no funds, grants, or other support were received during the preparation of this manuscript.

**Conflicts.** The authors have no relevant financial or non-financial interests to disclose.

**Author contributions.** All authors contributed to the study conception and design. Experiments were performed by M. Zohaib Nawaz. All authors wrote the manuscript and approved the final manuscript.

**Data and code availability.** The datasets used in this study and the code of GRIMP are openly available in the GRIMP repository at <http://github.com/MuhammadzohaibNawaz/GRIMP/>.

**Ethics statement.** This study does not involve humans, patients, ani-

mals or clinical trials. It is solely conducted on public data.

**Permission to reproduce.** All content of this paper is original.

## 1 Introduction

Frequent itemset mining (FIM) (Luna et al., 2019), also referred to as frequent pattern mining or large itemset mining, was introduced in the early 1990s. It is a data mining task aimed at identifying sets of items - such as values, events, or symbols - that co-occur frequently within a database. FIM was initially employed for market basket analysis in transactional databases to uncover frequently co-occurring items (Agrawal et al., 1993). A transaction database is a collection of transactions where each transaction is a set of items. The goal of FIM is to find set of items that appear frequently (the frequent itemsets). An itemset is considered frequent if it occurs in the database with a frequency that meets or surpasses a user-specified threshold called the minimum support (*min\_sup*). For example, if the user sets *min\_sup* to 5, then only the itemsets that are contained in at least 5 transactions are reported. Beyond market basket analysis, FIM is used in numerous fields where data can be represented as records (transactions) described by binary attributes (items). FIM not only allows the discovery of interesting patterns in databases but also supports other tasks such as clustering and classification (Aggarwal, 2014).

In the last few decades, researchers have introduced many variations of FIM to select itemsets based on various properties and measures, either as alternatives or in addition to the support measure (Luna et al., 2019; Farghaly and El-Hafeez, 2022). These variations include identifying representative itemsets (Seno, 2023), closed frequent itemsets (Pasquier et al., 1999), free-sets (Boulicaut et al., 2003), non-derivable itemsets (Calders and Goethals, 2007), itemsets that respect various user-defined constraints (Belaid and Lazaar, 2021; Bonchi et al., 2006), local dependency itemsets (Ni et al., 2020), and statistically significant itemsets (Fournier-Viger et al., 2018). Besides, many studies have investigated alternative approaches to explore the search space to improve the performance of FIM (Cao et al., 2023).

Despite these advancements, a major challenge in FIM remains the *pattern explosion* problem (Luna et al., 2019; Vreeken et al., 2011). This occurs when a low minimum support threshold is set, resulting in an overwhelming number of patterns, often exceeding the total number of transactions in the database by several orders of magnitude. This problem arises due to the locality of the minimal support constraint (Belaid and Lazaar, 2021), where each distinct itemset fulfilling the support constraint is included in the result set independently of other already selected itemsets. Because of this, redundant sets of patterns are generally presented to the user, many of which describe the same area of the database and vary only slightly. A promising approach to address this issue is to change the focus from evaluating each pattern individually to evaluating itemsets collectively (to identify a good collection of patterns), through processes such as the iterative removal of redundancy (Darrab et al., 2021; Liu et al., 2022b).

While these approaches mitigate the issue to some extent, they are not without drawbacks. Redundancy still remains a concern and removing information inappropriately may lead to the loss of important details.

An emerging approach to address the pattern explosion problem involves removing redundancy while ensuring that a small yet informative set of itemsets is discovered. This is accomplished by searching for the best compact set of patterns that succinctly describe the data within a database. The principle of Minimum Description Length (MDL) (Grünwald, 2007) can be used to discover such compact and compressed patterns by selecting the model (a set of itemsets) that minimizes the overall length of the data encoding, effectively balancing complexity and information retention. Based on MDL, several algorithms were designed, such as KRIMP (Vreeken et al., 2011) and SLIM (Smets and Vreeken, 2012), for itemsets mining. These algorithms utilize heuristic-based search methods that primarily focused on local optimizations. They manage the vast search space by selectively evaluating itemsets according to specific heuristic criteria.

KRIMP, for instance, employs a sequential selection strategy to minimize the description length of individual itemsets, adhering closely to a greedy approach that prioritizes local improvements. Conversely, SLIM enhances KRIMP by dynamically adjusting its itemsets during the mining process. Both KRIMP and SLIM tend to overlook global optimal sets due to their narrow search perspective and are slow due to their heuristic-based direct search methods. SLIM’s speed advantage is counterbalanced by its high algorithmic complexity, especially with large datasets, as it requires an exhaustive exploration of the search space due to the limitations of its heuristic approach. KRIMP and SLIM also face several other challenges, such as their inability to guarantee the identification of an optimal pattern set, the imposition of arbitrary restrictions on parameter spaces, and limited suitability for multi-class classification (Bouritsas et al., 2021; Proença and van Leeuwen, 2020; Galbrun, 2022).

The search space for finding all patterns (itemsets) that result in optimal compression is extremely vast, encompassing  $2^{(2^n-1)}$  possible combinations of itemsets, where  $n$  represents the number of unique items in the dataset. The combinatorial explosion of these combinations makes it computationally infeasible to exhaustively evaluate every pattern’s contribution to compression. A promising approach to address this challenge is to adopt evolution-based heuristic search techniques for selecting optimal patterns. These techniques provide a good balance or trade-off between performance and completeness (Yu and Gen, 2010; Nawaz et al., 2021a). They have the ability to explore large search spaces to find near-optimal solutions based on fitness functions while adhering to various constraints. In this paper, we propose the use of a genetic algorithm (GA) (Kramer, 2017; Holland, 1975) for MDL-based selection of optimal patterns. Inspired by biological evolution, GAs iteratively refine a population of candidate solutions by applying genetic operators such as selection, crossover, and mutation. Moreover, GA for compression-based pattern mining does not rely on various costly functions used in traditional MDL-based pattern compression,

thereby providing an adaptive and efficient way to find a set of frequent itemsets that offers superior lossless compression of the database.

Leveraging the capability of GA to adeptly navigate through complex solution spaces, we propose a novel framework called GRIMP (a **G**enetic algo**R**ithm for co**M**pression-based descriptive **P**attern mining). This framework utilizes the inherent strengths of GAs for effective data compression and pattern discovery. The process begins by generating a random population of patterns, each representing a potential solution. These patterns are evaluated based on a compression metric, which assesses their ability to represent data efficiently. Higher-performing patterns are selected for reproduction, following the principle of “survival of the fittest”. Through crossover, pairs of selected patterns exchange information to create offspring solutions. Mutation introduces random changes to promote diversity and prevent stagnation in the population. This iterative process of selection, crossover, and mutation continues for a set number of generations, allowing the population to evolve towards better solutions. After a defined number of iterations, the best-performing patterns in the final population are output.

An early version of this work was published in a workshop paper (Nawaz et al., 2024). Initially, we focused exclusively on single-point crossover and mutation operators. In the current paper, we also explore multi-point and uniform crossover techniques, as well as a modified pairwise interchange mutation, to provide a more detailed exploration of the GA’s capabilities. More extensive experiments are also presented to investigate how these methods can enhance the handling of large datasets and the reduction of redundancy. The main contributions of this paper are outlined as follows:

1. GRIMP is presented with six variants resulting from using three crossover operators (single-point, multi-point, and uniform) and two mutation operators (standard mutation and modified pairwise interchange mutation).
2. An optimization called Dynamic Dataset Reduction (DDR) is devised to enhance runtime efficiency of GRIMP. DDR reduces the dataset size dynamically during pattern extraction by removing identified patterns from the dataset as they are added to the final set of solution. This accelerates future pattern lookups, streamlining computational processes and improving overall efficiency.
3. To broaden the investigation into the performance of the proposed framework, this study incorporates 21 datasets, compared to only five in the initial study. This expanded dataset collection provides a more comprehensive evaluation of GRIMP’s efficacy across diverse scenarios and problem instances, thereby enhancing the understanding of its capabilities.
4. Extensive experiments are performed to evaluate GRIMP across various metrics, including compression ratio, runtime, the number of generations required for achieving compression, the number of patterns needed by GRIMP to achieve the same compression as previous methods and classification. This comprehensive analysis offers deeper insights into GRIMP and highlights its efficiency and effectiveness in real-world applications.

5. The headless chicken test is conducted to evaluate the effectiveness of the crossover operators within the GRIMP.

The rest of the paper is organized as follows. Section 2 reviews the existing research on MDL-based pattern mining algorithms. Section 3 provides an overview of the fundamentals related to using MDL and GA for selecting itemsets. Section 4 introduces the GRIMP framework. Section 5 presents extensive experimental results validating GRIMP, as well as an evaluation of the choices made in its design. Section 6 summarizes the paper with some final remarks.

## 2 Related Work

This section is divided into two parts. It first reviews related work on MDL for pattern mining and then gives a brief introduction to GA.

### 2.1 MDL for Pattern Mining

Using the MDL for FIM is based on the argument that the best set of itemsets will provide the most effective compression of the database. KRIMP (Vreeken et al., 2011), the first MDL-based algorithm for pattern-based compression, followed two steps: (1) mining frequent itemsets, and (2) selecting a subset of the mined frequent patterns that minimize the overall description length for improved compression, by exploring combinations in a static order. Because KRIMP required that all frequently occurring itemsets be generated and mining itemsets was costly, it had long runtimes. Moreover, KRIMP achieved poor compression results when the frequency threshold was increased. Besides, KRIMP sometimes rejected itemsets that it may have used later since it only evaluated them once and in a fixed order. SLIM (Smets and Vreeken, 2012), an improved version of KRIMP, overcame these problems. SLIM discovered optimal itemsets gradually and was not dependent on generating all frequent itemsets in advance. The encoding scheme employed by both algorithms had limitations, including the use of a static, predefined code table that might not be optimal for compressing databases with diverse characteristics and distributions. Additionally, the order-dependent encoding process, where itemsets were encoded in a fixed sequence, could lead to suboptimal coding sets if certain combinations were initially overlooked. Similarly, a decision tree in combination with a refined version of the MDL was used in the PACK (Tatti and Vreeken, 2008) for pattern-based compression. DIFFNORM (Budhathoki and Vreeken, 2015), an extension of SLIM, used a better encoding and could be utilized to compare datasets in terms of itemsets. The SHRIMP algorithm (Hess et al., 2014), built a FP-tree-like data structure to examine the schema selection impact on a database. This approach allowed the fast computation of the quality of the mined itemsets. Likewise, RSCO (Sampson and Berthold, 2014) presented an enhanced version of the KRIMP that improved performance by employing diverse parallel processing techniques.

In pattern mining, the MDL principle was initially applied for itemset mining but researchers have also extended its usage to other pattern mining tasks.

These alternative applications indicate that the MDL is flexible and can be adapted to diverse mining contexts, from sequence and graph mining to real-valued and numerical datasets. For example, CSPM (Liu et al., 2022a) extended the application of the MDL principle to the domain of graph mining, specifically focusing on identifying and compressing star structures within attributed graphs. (Lam et al., 2014) introduced SEQKRIMP and GOKRIMP, extensions of KRIMP for mining compressing sequential patterns. SEQKRIMP adapted KRIMP for sequence data using a two-phase approach to reduce redundancy. Whereas, GOKRIMP increased efficiency by greedily mining patterns and using a dependency test to enhance pattern quality and reduce runtime.

Similarly, KRIMP was extended as REALKRIMP (Witteveen et al., 2014) to process real-valued data. REALKRIMP required data discretization, relied on various heuristics, and could not jointly evaluate the set of generated hyper-rectangles that represented patterns. MINT (Makhalova et al., 2022), also based on the MDL, was used to find useful patterns in numerical datasets. MINT generated a small set of non-redundant informative patterns. COMPREX (Akoglu et al., 2012), also based on the MDL, was proposed to overcome scalability problems. This algorithm worked on categorical data and was more resistant to the pattern explosion problem. DRIM (Vanetik and Litvak, 2018) utilized the MDL principle to generate a diverse summary of itemsets efficiently, optimizing the selection process to capture the most informative patterns rapidly and succinctly. Additionally, the VOUW algorithm (Faas and van Leeuwen, 2020) applied the MDL principle to geometric pattern mining, focusing on the extraction of significant spatial patterns by minimizing the description length, thus efficiently encoding spatial data. Moreover, the SWIFT algorithm (Yan et al., 2018) employed the MDL principle to streamline the extraction of highly representative patterns from event streams, ensuring that the identified patterns were not only statistically significant but also offered a compact description of the data, which was critical for real-time analysis and decision-making processes.

In a related study, (Zhang et al., 2019) explored an innovative approach to identifying succinct and meaningful patterns within datasets where items were interconnected through graph structures. This study leveraged the properties of graph connectivity to enhance the relevance and efficiency of pattern discovery, focusing on extracting concise itemsets that provided clear insights into the structural and relational dynamics of the data. (Mantuan and Fernandes, 2018) introduced a method that integrated the spatial context into the mining of closed itemsets, enhancing the discovery process by considering the geographical or spatial relationships between data points. This approach improved the relevance and applicability of the extracted patterns in spatially oriented datasets. Furthermore, they utilized the MDL principle to verify the descriptiveness of the mined patterns. (Liang et al., 2021) utilized the MDL principle to identify regular patterns and detect anomalies within financial records. By simplifying complex financial data into more manageable patterns, this study facilitated accurate and efficient anomaly detection and contributed to robust bookkeeping

audits. A summary of the discussed MDL-based algorithms along with their key parameters are presented in Table 1.

Table 1: Overview of related work on MDL-based pattern mining algorithms

Algorithm	DMT	Strengths	Limitations
KRIMP	IM	Improved compression efficiency	Long runtimes; poor results with high frequency thresholds
SLIM	IM	Gradual discovery of optimal itemsets	Still dependent on the quality of frequent itemsets
PACK	IM	Effective for pattern-based compression	Limited applicability to certain data types
DIFFNORM	IM	Improved itemset comparison	May not generalize well across all datasets
SHRIMP	IM	Fast quality computation of itemsets	Potential issues with schema selection impact
RSCO	IM	Enhanced performance	Complexity of implementation
DRIM	IM	Efficient pattern summary	May miss some less frequent patterns
SEQKRIMP	SP	Redundancy reduction	Requires thorough parameter tuning
REALKRIMP	RvD	Handles real-valued data	Requires data discretization
MINT	NP	Generates non-redundant patterns	Limited to specific pattern types
CSPM	GM	Focus on star structures	May not generalize to other graph types
MZ <sup>1</sup>	GM	Enhances relevance and efficiency	Dependent on graph structure
CompreX	CD	Addresses scalability issues	May be less effective for high-dimensional data
VOUW	GeoM	Focuses on spatial patterns	Performance may vary with dataset size
SWIFT	ES	Real-time pattern extraction	May require extensive computational resources
SCIM	SM	Integrates spatial context	Limited to spatially oriented datasets
TG-SUM	FR	Simplifies complex data	May overlook certain anomalies

DMT: Data Mining Technique, IM: Itemset Mining, SP: Sequential Patterns, RvD: Real-valued Data, NP: Numerical Patterns, GM: Graph Mining, CD: Categorical Data, GeoM: Geometric Mining, ES: Event Streams, SM: Spatial Mining, FR: Financial Records

<sup>1</sup>: Method of (Zhang et al., 2019)

In mining compressing patterns, the focus until now has been to improve the existing MDL-based approaches by proposing various heuristics, data structures, and encoding schemes, and extending the existing methods to databases of various types. However, developed algorithms have many limitations, such as extensive computational time, requiring scanning all or a major part of the dataset, their inability to guarantee the identification of the optimal pattern set, imposing arbitrary restrictions on parameter spaces, and dependency on statistical assumptions about the data or the model.

## 2.2 Genetic Algorithm

GA, a stochastic global search method, is inspired by natural evolution and follow Darwin’s principle of “survival of the fittest” to refine potential solutions iteratively (Gen and Lin, 2023; Kramer, 2017; Holland, 1975). These algorithms work with a population of potential solutions, termed chromosomes, by applying genetic operators such as selection, crossover, and mutation (Nawaz et al., 2021c). During selection, chromosomes are chosen for reproduction based on their fitness, while crossover combines genetic material from two parent chromosomes to create new child chromosomes. Mutation, which typically occurs with a low probability, alters chromosome values to introduce variability. Over generations, this process enhances the population’s adaptation to its environment. This process results in the evolution of populations of individuals (chromosomes)

that are more suited to their environment than the individuals from which they were derived.

Leveraging this evolutionary nature, this paper addresses two significant limitations of current MDL-based methods for mining compressing patterns: (1) prohibitively long runtimes and (2) the tendency to identify suboptimal patterns, resulting in low compression efficiency. To overcome these challenges, the proposed framework integrates GA with the MDL principle to optimize the itemset selection process and reduces the number of itemsets required for achieving high compression. To the best of our knowledge, this is the first instance of combining evolutionary algorithm with MDL-based compression, effectively mitigating the high computational demands and enhancing pattern identification.

### 3 Preliminaries

This section first provides important preliminaries on FIM and MDL-based approaches for mining itemsets. It then explains how the GA is tailored for MDL-based compression, facilitating their integration into the proposed framework.

#### 3.1 Frequent Itemset Mining

The aim of FIM (Agrawal et al., 1993) is to identify all sets of values (items) that appear in at least a minimum number of records (transactions) in a transactional database.

Formally, for a transactional database  $\mathcal{D}$ , let  $\mathcal{I}$  represent the set of all distinct items (values) that appear in  $\mathcal{D}$ . A database  $\mathcal{D}$ , defined over items in  $\mathcal{I}$  and containing  $n$  transactions is denoted as  $\{t_1, t_2, \dots, t_n\}$ . A transaction  $t \in \mathcal{D}$  is a non-empty set of items (an itemset), i.e.  $t \in P(\mathcal{I})^1$  and  $t \neq \emptyset$ . An itemset  $\mathcal{X} \subseteq \mathcal{I}$  containing a single item is called a singleton itemset.

A transaction  $t$  in  $\mathcal{D}$  is said to support an itemset  $\mathcal{X} \subseteq \mathcal{I}$ , if and only if  $\mathcal{X} \subseteq t$ . In  $\mathcal{D}$ , the support of an itemset  $\mathcal{X}$  is computed as the total count of transactions in  $\mathcal{D}$  that contain  $\mathcal{X}$ .

#### 3.2 MDL for Itemset Mining

MDL-based itemset mining algorithms are designed with the goal of finding the model  $\mathcal{M}$  that best compresses a transactional database  $\mathcal{D}$ , where a model is a set of itemsets (Vreeken et al., 2011). To find a good model, MDL-based algorithms searches the space of possible models, and assess each model that is encountered during this search to keep the best model.

To assess a model  $\mathcal{M}$  based on the MDL principle, the following sum is calculated:

$$L(\mathcal{D}, \mathcal{M}) = L(\mathcal{D}|\mathcal{M}) + L(\mathcal{M}) \quad (1)$$

---

<sup>1</sup> The notation  $P(I)$  denotes the powerset of  $I$ .



where  $L(\mathcal{D}|\mathcal{M})$  is the compressed size in bits of the database encoded using the model  $\mathcal{M}$ , while  $L(\mathcal{M})$  is the size of  $\mathcal{M}$  in bits. A model with a smaller  $L(\mathcal{D}, \mathcal{M})$  is considered a better model.

In the context of FIM with MDL, a structure called *code table* is utilized by KRIMP and SLIM for representing a compression model. A code table, denoted as  $CT$ , is a dictionary that comprises two columns: the first column contains the itemsets from the model, and the second column provides a unique code for each itemset that is used for encoding the database using that itemset. A  $CT$  generally comprises itemsets with non-zero usage, ordered firstly by descending cardinality, then by decreasing support, and finally in ascending lexicographical order. This establishes a total order called the *Standard Cover Order*. This order is used for encoding the database on the basis that shorter codes are assigned to itemsets that appear before others according to that order. The lengths of itemset codes are important as they impact the compression of the database.

A  $CT$  is represented as a set of pairs  $CT \subseteq \{(\mathcal{X}, code(\mathcal{X})) \mid \mathcal{X} \in CS\}$ . The notation  $code(\mathcal{X})$  denotes a code assigned to an itemset  $\mathcal{X} \in CT$ . The coding set ( $CS$ ) of a  $CT$  is the set of all itemsets  $\mathcal{X}$  that it contains. Figure 1 presents an example of a  $CT$ . The first column lists the itemsets included in the  $CT$ , while the second column provides information about their respective codes, visualized as colored bars whose widths represent their lengths. For illustrative purposes, a third column called ‘usage’ is also included, which is not an actual part of the  $CT$ , and will be explained subsequently. In this example, the set of items from the database is  $\mathcal{I} = \{Apple, Banana, Cheese, Date\}$ . It can be observed that all singleton itemsets are included in the  $CT$ , albeit *Date* has no code. The number of itemsets in the  $CT$  is 6, denoted as  $|CT| = 6$ , while the number of itemsets from the  $CT$  that are not included in  $\mathcal{I}$  is 2, which is denoted as  $|CT \setminus \mathcal{I}| = 2$ .







Code Table Itemsets	Code	Usage
Apple Banana Date		3
Apple Banana		2
Apple		2
Banana		1
Cheese		1
Date		0

Fig. 1: An example code table for a database where  $\mathcal{I} = \{Apple, Banana, Cheese, Date\}$ . Codes are visually represented as bars, with the widths of the bars indicating the lengths of the codes. Note that the usage column is not an actual part of the code table but is included for illustrative purposes. For optimal compression, shorter codes are assigned to the itemsets that are more frequently used (have a greater usage count).

To compress a database  $\mathcal{D}$  with a  $CT$ , a *cover* function is utilized to encode each transaction  $t$ . The *cover function*,  $cover(t)$ , denotes the pattern set from the  $CS$  that is selected for encoding the transaction  $t$ . Figure 2 displays an example database. The dataset contains a total of nine transactions, with three of them being identical to each other, and two other transactions also being identical. The figure also illustrates how this database is covered by the example  $CT$ , referenced in Figure 1. Notably, in this example, each transaction is represented by exactly one itemset from the  $CT$ . For instance, the first three transactions of the database are encoded using the code assigned to the itemset  $\{Apple, Banana, Date\}$  in the  $CT$  of Figure 1.










Database	Encoded Database
Apple Banana Date	
Apple Banana Date	
Apple Banana Date	
Apple Banana	
Apple Banana	
Apple	
Apple	
Banana	
Cheese	

Fig. 2: An example database (left) and its encoding using the code table of Figure 1 and the cover function (right).

To encode a database  $\mathcal{D}$  using a  $CT$ , each transaction  $t \in \mathcal{D}$  is replaced by the codes of the itemsets in the cover of  $t$ :

$$t \mapsto \{\text{code}(\mathcal{X}) \mid \mathcal{X} \in \text{cover}(t)\} \quad (2)$$

In Figure 3, the encoding of a second database is depicted, illustrating how some transactions can be encoded using multiple codes from the  $CT$  in Figure 1. For instance, the second transaction  $\{Apple, Banana, Cheese\}$  is encoded using the codes of both the itemsets  $\{Apple, Banana\}$  and  $\{Cheese\}$  from the  $CT$ .

Because the aim of MDL-based pattern mining is to identify the model that most effectively compresses a database, the codes within the  $CT$  should be selected such that the more frequently an itemset is used, the shorter its code should be. Note that, in the context of MDL-based pattern mining, the actual codes are irrelevant for the MDL calculations; only the lengths of the codes associated with the itemsets in the  $CT$  are needed to calculate the encoded database



Fig. 3: Another example database (left) and its encoding using the code table of Figure 1 (right). Some transactions are encoded using multiple codes.

size and model size. As there exists a clear correspondence between code lengths and probability distributions (Li and Vitányi, 2019), the optimal code lengths can be calculated through the Shannon entropy.

The optimality property of code lengths ensures that no bias is introduced when using the code length. For MDL-based itemset mining, the probability distribution induced by a cover function is simply given by the relative usage frequency of each of the itemsets in the  $CT$ . To determine this, the usage count of an itemset  $\mathcal{X} \in CT$  is defined as the number of transactions  $t$  from  $\mathcal{D}$  where  $\mathcal{X}$  is part of the cover. Normally, this frequency represents the probability that the code is used in the encoding of an arbitrary transaction  $t \in \mathcal{D}$ . The optimal code length (Li and Vitányi, 2019) is then the  $-\log$  of this probability, and a  $CT$  is optimal if all its codes have an optimal length. Fractional lengths, as opposed to integer-valued lengths of materialized codes, are used to ensure that the length of a code accurately represents its usage probability. Since materialized codes are not the focus, only relative lengths are of importance. More formally, we have the following definition for *usage*.

Let  $\mathcal{D}$  be a transaction database over a set of items  $\mathcal{I}$ , a cover function *cover*, and a code table  $CT$  over  $\mathcal{I}$  and  $\mathcal{C}$ . The usage count of an itemset  $\mathcal{X} \in CT$  is defined as:

$$\text{usage}(\mathcal{X}) = |\{t \in \mathcal{D} \mid \mathcal{X} \in \text{cover}(t)\}| \quad (4)$$

This defines a probability distribution of  $\mathcal{X} \in CT$  for  $\mathcal{D}$ , given by:

$$\mathcal{P}(\mathcal{X} \mid \mathcal{D}) = \frac{\text{usage}(\mathcal{X})}{\sum_{\mathcal{Y} \in CT} \text{usage}(\mathcal{Y})} \quad (5)$$

The  $\text{code}(\mathcal{X})$  for  $\mathcal{X} \in CT$  is optimal for  $\mathcal{D}$  if and only if:

$$L(\text{code}(\mathcal{X})) = |\text{code}(\mathcal{X})| = -\log(\mathcal{P}(\mathcal{X} \mid \mathcal{D})) \quad (6)$$

A  $CT$  is code-optimal for  $\mathcal{D}$  if and only for each itemset  $\mathcal{X} \in CT$ , the  $\text{code}(\mathcal{X})$  is optimal. From now onward, it is assumed that the  $CT$ s are code-optimal for the database they are induced on, unless stated differently.

For instance, from Figure 1, the itemset  $\{Apple, Banana, Cheese\}$  appears three times in the cover of the database. For  $\mathcal{X} = \{Apple, Banana, Cheese\}$ :

$$\mathcal{P}(\mathcal{X} | \mathcal{D}) = \frac{3}{9}$$

$$L(\text{code}(\mathcal{X})) = -\log\left(\frac{3}{9}\right) = 1.58$$

Similarly, for  $\mathcal{Y} = \{Cheese\}$ :

$$\mathcal{P}(\mathcal{Y} | \mathcal{D}) = \frac{1}{9}$$

$$L(\text{code}(\mathcal{Y})) = -\log\left(\frac{1}{9}\right) = 3.16$$

Thus,  $\{Apple, Banana, Cheese\}$  is assigned a code length of 1.58 bits, while  $\{Cheese\}$  is assigned code lengths of 3.16 bits each.

As mentioned earlier in Eq. (1), the total encoded size in bits for compressing a  $\mathcal{D}$  with a  $CT$  is denoted as  $L(CT, \mathcal{D})$  and calculated as:

$$L(CT, \mathcal{D}) = L(CT|\mathcal{D}) + L(\mathcal{D}|CT) \quad (7)$$

where  $L(CT|\mathcal{D})$  and  $L(\mathcal{D}|CT)$  respectively represent the size of  $CT$  and the encoded size of  $\mathcal{D}$  in bits, and are defined as:

$$L(CT|\mathcal{D}) = \sum_{\substack{\mathcal{X} \in CT \\ \text{usage}(\mathcal{X}) \neq 0}} (L(\mathcal{X}|ST) + L(\mathcal{X}|CT)) \quad (8)$$

$$L(\mathcal{D}|CT) = \sum_{t \in \mathcal{D}} L(t|CT) \quad (9)$$

$$L(t|CT) = \sum_{X \in \text{cover}(t)} L(X|CT) \quad (10)$$

$ST$  represents the most fundamental and valid  $CT$ , that is the *Standard Code Table* ( $ST$ ), which is the  $CT$  that contains only single items from  $I$  in its coding set, i.e.,  $\text{code}(ST) = I$ . Calculating the size of a  $CT$  is done by summing all its code lengths from its second column.

Based on the above definitions, the task of MDL-based itemset mining can be formalized as the Minimal Coding Set Problem (Vreeken et al., 2011). The goal is to optimize coding efficiency for a database by determining the ‘best’ coding set (model) from a collection of itemsets. This problem seeks to identify the minimal coding set necessary to efficiently represent the data while minimizing the total code length. The problem of finding the minimal coding set is defined as follows:

**Problem of Minimal Coding Set:** Suppose  $\mathcal{I}$  represents the set of items,  $\mathcal{D}$  is a database over items from  $\mathcal{I}$ ,  $\text{cover}$  is a cover function, and  $F \subseteq P(\mathcal{I})$  represents candidate itemsets (or patterns). The problem is to determine the smallest set of itemsets  $P \subseteq F$  such that the overall compressed size, denoted as  $L(CT, \mathcal{D})$ , is minimal for the corresponding  $CT$ .

### 3.3 GA for MDL-based Itemset Mining

In this paper, a framework is presented, which follows the general approach presented in the previous subsection for MDL-based itemset mining but some modifications are made to render this approach more appropriate for the integration with a GA. These changes are explained next.

The proposed framework, which is presented in the next section, first counts the occurrences of itemsets  $\mathcal{X} \in CT$  within a dataset  $\mathcal{D}$ . Subsequently, it computes the sizes of the  $CT$  and dataset  $\mathcal{D}$  in bits. The approach employs the Shannon entropy discussed in the previous subsection to assign a minimum bit size to an  $\mathcal{X}$  in  $CT$  that appears the most frequently in  $\mathcal{D}$ , optimizing the efficiency of compression based on the frequency of occurrence.

The total encoded size in bits for compressing a database with a  $\mathcal{D}$  with a  $CT$ , denoted as  $L_G(CT, \mathcal{D})$ , is calculated in the proposed framework as as:

$$L_G(CT, \mathcal{D}) = L_G(CT|\mathcal{D}) + L_G(\mathcal{D}|CT) \quad (11)$$

where  $L_G(CT|\mathcal{D})$  and  $L_G(\mathcal{D}|CT)$  denote the size of the  $CT$  and that of the compressed encoded database  $\mathcal{D}$  w.r.t.  $CT$ , respectively. Those terms are defined as:

$$L_G(CT|\mathcal{D}) = \sum_{\substack{\mathcal{X} \in CT \\ usage(\mathcal{X}) \neq 0}} (L(\mathcal{X}|CT) + L(code(\mathcal{X}))) \quad (12)$$

$$L_G(\mathcal{D}|CT) = \sum_{t \in \mathcal{D}} L_G(t|CT) \quad (13)$$

$$L_G(t|CT) = \sum_{\mathcal{X} \in cover(t)} L(\mathcal{X}|CT) + \sum_{i \in t \setminus cover(t)} L(i|CT) \quad (14)$$

Here,  $cover(t)$  represents the set of itemsets from the  $CT$  that are used to cover transaction  $t$ , and  $t \setminus cover(t)$  is the set of items in  $t$  not covered by these itemsets. For each itemset  $\mathcal{X}$ ,  $L(\mathcal{X}|CT)$  is the encoded length of  $\mathcal{X}$  according to the  $CT$ , and  $L(i|CT)$  is the encoded length of the non-covered item  $i$  according to the same  $CT$ . In the rest of this paper, the above definition of  $L_G(CT, \mathcal{D})$  is used, which is slightly different from KRIMP and SLIM. The main difference is that the standard code table  $ST$  is not used in the calculations.

The reason for not using the  $ST$  is the following. GRIMP selects itemsets randomly from the database and then applies crossover and mutation operators to these selections to generate new itemsets. Then, the compression effectiveness is calculated relative to those itemsets. More precisely, any items or itemsets not covered by the  $CT$  remain in the encoded database. Therefore, calculating the compression size based on the encoded database does not require every item or itemset to be explicitly covered, enhancing flexibility in our compression strategy. The decision to deviate from using the  $ST$  is well-motivated, as the minimal hitting set generation problem (Gainer-Dewar and Vera-Licona, 2017), which is closely related to the minimal coding set problem, does not necessarily require complete coverage of all items/itemsets. As discussed in (Gainer-Dewar and Vera-Licona, 2017), the goal is to find the inclusion-minimal “hitting sets”

(analogous to the minimal coding sets in the case of this paper) from a given collection of sets, without the strict requirement of covering every element.

Additionally, we utilize the count of occurrences to guide GRIMP. Specifically, if the count of an itemset in the  $CT$  increases as a result of the evolutionary process, GRIMP updates the itemset in the  $CT$ . Conversely, if the count does not increase, GRIMP keeps the previously evolved itemset, ensuring only beneficial modifications are preserved. This method strategically uses occurrence metrics to optimize the evolution of itemsets for more effective compression. In the GRIMP framework, we have enhanced the diversity of genetic operations by employing three crossover operators and two mutation operators. This variety allows for a more dynamic and robust evolution of the randomly selected itemsets, providing different mechanisms to explore and optimize the solution space effectively. More details about the proposed GRIMP framework for pattern compression are presented next.

## 4 GRIMP

This section presents the proposed GRIMP framework for mining representative compression-based patterns using a GA. An overview of the algorithm is first presented and then each step is explained in detail.

The algorithm operates on two primary inputs: a transaction dataset  $\mathcal{D}$  and a threshold specifying the maximum size of the  $CT$  ( $maxCodeTableSize$ ), which serves as the stopping condition. This  $maxCodeTableSize$  threshold ensures that the framework terminates once the  $CT$  reaches the specified size, providing a practical limit to the search space and computation time. Using this threshold can also be interesting for the user as it allows to limit the number of itemsets that are output so as to not be overwhelmed by a potentially large number of patterns, an issue known as *pattern explosion* in the field of itemset mining (Luna et al., 2019). The output of GRIMP is a  $CT$ , which contains representative patterns that effectively compress the dataset. Additionally, GRIMP can be configured to operate without the  $maxCodeTableSize$  parameter, making it fully parameter-free. In such a configuration, the algorithm could terminate based on the stability of the compression ratio; specifically, if the compression ratio does not improve for a predefined number of generations, the algorithm would stop. However, for this study and to facilitate direct comparisons with KRIMP and SLIM in the experimental evaluation, we employ a fixed threshold size for the  $CT$ . This approach aligns with the comparative framework and provides consistent benchmarks across different algorithms.

The flowchart of GRIMP, highlighting its main steps, is shown in Figure 4. The GRIMP framework first generates an initial population of itemsets, which represent potential solutions for compressing the dataset. It then enhances these itemsets through an iterative GA process, encompassing selection, crossover, mutation, and evaluation:

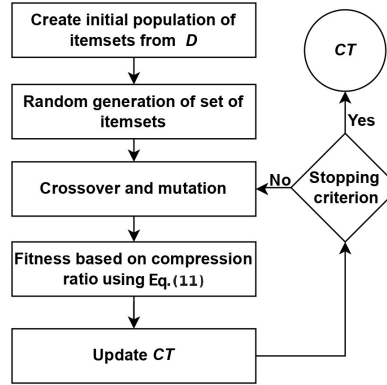


Fig. 4: Flowchart of GRIMP for pattern compression

1. **Selection:** Randomly selects itemsets from the current population, based on their fitness, which is assessed by their dataset compression capability.
2. **Crossover:** Applies crossover operators to pairs of selected itemsets to combine their features, generating new itemsets. This step simulates genetic recombination and aims to explore new solution spaces.
3. **Mutation:** Introduces random changes to the itemsets, altering one or more elements to enhance population diversity.
4. **Evaluation:** Computes the fitness of the newly generated itemsets by evaluating their compression efficiency on the dataset.
5. **Stopping Criteria:** GRIMP terminates when a predefined condition is met, that is adding itemsets in the  $CT$  till achieving a satisfactory compression ratio.

The process continues until the stopping criterion is satisfied, which is determined by  $CT$  reaching the size specified by the threshold,  $maxCodeTableSize$ . At this moment, GRIMP returns the  $CT$  containing the itemsets that optimally compress the dataset. The implementation of GRIMP includes three variants of crossover (single, multi, and uniform) and two variants of mutation (single and modified pairwise interchange), details of which are provided subsequently.

Algorithm 1 provides the pseudo-code of GRIMP. It first initializes a population with items from the dataset (line 1) and determines the maximum transaction length (MTL) in the dataset (line 2). GRIMP then initializes the  $CT$  as an empty set (line 3). A while loop runs, which continues until the length of  $CT$  is not equal to the specified  $maxCodeTableSize$  threshold. Inside this loop, the algorithm creates two random parent itemsets (called  $P_1$  and  $P_2$ ) of a length between 2 and the  $MTL$  (line 5), and counts its occurrences w.r.t. the dataset (lines 6, 7). It then applies the crossover (line 9) and mutation operators (lines 10, and 11), resulting in evolved child itemsets (called  $C_1$  and  $C_2$ ). The algorithm then calculates the number of occurrences of  $C_1$  and  $C_2$  in the database (lines 12, 13). These occurrences are compared with those of the original parent

itemsets. If the updated child itemsets have more occurrences, they replace the respective original itemsets in the  $CT$  along with their occurrences (lines 14-21). This process is iteratively repeated across a specified number of generations to refine the result. The output of this algorithm is the  $CT$ —a refined set of itemsets that optimally compresses the database (line 26).

To improve the runtime of the GRIMP algorithm, we propose an optimization strategy named Dynamic Dataset Reduction (DDR), which leverages the dynamic nature of the dataset during pattern extraction to reduce the time required to calculate the compression ratio. The DDR is implemented by Line 24 in Algorithm 1. During a given iteration, as patterns are identified and added to the  $CT$ , the DDR consists of replacing the occurrences of these patterns in the dataset by their corresponding codes from the  $CT$ . This means that as more patterns are added to the  $CT$ , the dataset becomes increasingly smaller, and thus the algorithm can scan the dataset more quickly. Consequently, with each new iteration, the time required for counting pattern occurrences in the dataset decreases progressively. This optimization has a compounding effect: each successful pattern that is added to the  $CT$  not only contributes to the compression but also reduces the cost of subsequent compression ratio calculations, thereby enhancing GRIMP’s efficiency. The full extent of DDR’s impact on performance is thoroughly examined and quantified in the results section (see section 5.3).

A notable characteristic of GRIMP is its behavior regarding solution quality over iterations. It is designed to either improve the solution or maintain its current quality, ensuring that it never degrades (see Algorithm 1, lines 14-21). Because of this, the compression ratio, a key metric in our optimization, either decreases or remains constant throughout the iterations. GRIMP achieves this by employing a selective update mechanism. Specifically, if a child itemset provides a better compression ratio than the current parent itemset, the parent is updated with this child itemset. Conversely, if the child itemset does not offer an improvement, the parent itemset remains unchanged. This approach guarantees that the itemset quality does not deteriorate over time. This phenomenon is discussed further in detail in the result section (see section 5.4).

Algorithms 2, 3 and 4 provide the pseudo-code for the three crossover operators that are used within GRIMP. The symbol  $\parallel$  represents the concatenation of sub-itemsets. The three crossover operators are explained with simple examples. Let  $P_1$  and  $P_2$  be two itemsets, which are sorted and each represented by a different color for presentation purposes:

$$\begin{aligned} P_1 &= \text{apple, bread, banana, cake} \\ P_2 &= \text{apple, butter, diaper, milk, orange} \end{aligned}$$

For the single-point crossover (SPC) (Nawaz et al., 2021b) operator (Algorithm 2), one crossover point ( $cp$ ) is randomly selected and the itemsets to the left (or right) of  $cp$  are swapped to obtain two new child itemsets  $C_1$  and  $C_2$  from parent itemsets. Let  $n$  represent the length of the smallest itemset. A random  $cp$  ( $1 \leq cp \leq n$ ) is randomly selected. For  $cp = 3$ , applying SPC on  $P_1$  and  $P_2$  generates the two child itemsets  $C_1$  and  $C_2$  as: For  $cp = 3$ , applying SPC on  $P_1$  and  $P_2$  generates the two child itemsets  $C_1$  and  $C_2$  as:



---

**Algorithm 1** GRIMP

---

**Input:** Dataset  $\mathcal{D}$ ,  $maxCodeTableSize$   
**Output:**  $CT$  that best compresses the dataset

- 1:  $population \leftarrow \mathcal{I} \in \mathcal{D}$
- 2:  $MTL \leftarrow$  length of the longest transaction from dataset  $\mathcal{D}$
- 3:  $CT \leftarrow \emptyset$
- 4: **while**  $len(CT) \neq maxCodeTableSize$  **do**
- 5:    $P_1, P_2 \leftarrow$  Generate two random unique itemsets (length  $[2, MTL]$ )
- 6:    $Pocc_1 \leftarrow CountOccurrences(P_1, \mathcal{D})$
- 7:    $Pocc_2 \leftarrow CountOccurrences(P_2, \mathcal{D})$
- 8:   **repeat**
- 9:      $C_1, C_2 \leftarrow Crossover(P_1, P_2)$
- 10:     $C_1 \leftarrow Mutation(C_1, population)$
- 11:     $C_2 \leftarrow Mutation(C_2, population)$
- 12:     $Cocc_1 \leftarrow CountOccurrences(C_1, \mathcal{D})$
- 13:     $Cocc_2 \leftarrow CountOccurrences(C_2, \mathcal{D})$
- 14:    **if**  $Cocc_1 > Pocc_1$  **then**
- 15:      $P_1 \leftarrow C_1$
- 16:      $Pocc_1 \leftarrow Cocc_1$
- 17:    **end if**
- 18:    **if**  $Cocc_2 > Pocc_2$  **then**
- 19:      $P_2 \leftarrow C_2$
- 20:      $Pocc_2 \leftarrow Cocc_2$
- 21:    **end if**
- 22:   **until** (max. number of generations is attained)
- 23:    $CT \leftarrow (CT \cup \{P_1, P_2\})$
- 24:   Replace patterns from  $CT$  with codes in  $\mathcal{D}$   $\triangleright$  DDR
- 25: **end while**
- 26: **return**  $CT$

---

---

**Algorithm 2** Single-point crossover

---

**Input:**  $P_1, P_2$ : Two parent itemsets,**Output:** Two child itemsets  $C_1$  and  $C_2$ 

```

1: procedure SPC( $P_1, P_2$ )
2:    $s \leftarrow \min(\text{len}(P_1), \text{len}(P_2))$ 
3:    $cp \leftarrow \text{randomInt}(1, s)$   $\triangleright (1 \leq cp \leq s)$ 
4:    $C_1 \leftarrow P_1[1, cp] \parallel P_2[cp + 1, \text{len}(P_2)]$ 
5:    $C_2 \leftarrow P_2[1, cp] \parallel P_1[cp + 1, \text{len}(P_1)]$ 
6:   Return  $C_1$  and  $C_2$ 
7: end procedure

```

---

$C_1 = \text{apple, bread, banana, milk, orange}$   
 $C_2 = \text{apple, butter, diaper, cake}$

For the multi-point crossover (MPC) (Nawaz et al., 2021c) operator, shown in Algorithm 3, two crossover points ( $cp1$ ) and ( $cp2$ ) are randomly selected and the itemsets to the left (or right) of  $cp1$  and  $cp2$  are swapped to obtain two new child itemsets  $C_1$  and  $C_2$  from parent itemsets. Let  $n$  represent the length of the smallest itemset. Two random crossover points  $cp1$  and  $cp2$  are randomly selected ( $cp1 < cp2 \leq n$ ). For  $cp1 = 1$  and  $cp2 = 3$ , applying MPC on  $P_1$  and  $P_2$  generates the two child itemsets  $C_1$  and  $C_2$  as:

$C_1 = \text{apple, butter, diaper, cake}$   
 $C_2 = \text{apple, bread, banana, milk, orange}$

---

**Algorithm 3** Multi-point crossover

---

**Input:**  $P_1, P_2$ : Two parent itemsets,**Output:** Two child itemsets  $C_1$  and  $C_2$ 

```

1: procedure MPC( $P_1, P_2$ )
2:    $s \leftarrow \min(\text{len}(P_1), \text{len}(P_2))$ 
3:    $cp1, cp2 \leftarrow \text{randomInt}(1, s), \text{randomInt}(cp1 + 1, s)$   $\triangleright (cp1 < cp2 \leq s)$ 
4:    $C_1 \leftarrow P_1[1, cp1] \parallel P_2[cp1 + 1, cp2] \parallel P_1[cp2 + 1, \text{len}(P_1)]$ 
5:    $C_2 \leftarrow P_2[1, cp1] \parallel P_1[cp1 + 1, cp2] \parallel P_2[cp2 + 1, \text{len}(P_2)]$ 
6:   Return  $C_1$  and  $C_2$ 
7: end procedure

```

---

In uniform crossover (UC) (Algorithm 4), each item of the itemsets is assigned to the child itemsets with a probability value  $p$ . UC evaluates each item in the itemsets and selects the value from one of the itemsets with the probability  $p$ . If  $p$  is 0.5, then the child has approximately half of the items from the first parent

itemset and the other half from the second parent itemset. For  $P_1$  and  $P_2$ , two newly generated itemsets ( $C_1$  and  $C_2$ ) after UC with  $p = 0.5$  are:

$$\begin{aligned} C_1 &= \text{apple, butter, banana, milk} \\ C_2 &= \text{apple, bread, diaper, cake, orange} \end{aligned}$$

Here, the items at positions 2 and 4 within each parent's itemsets are swapped to create the child itemsets. UC is stochastic, thus the resulting child itemsets may vary based on the selection probability.

---

**Algorithm 4** Uniform crossover

---

**Input:**  $P_1, P_2$ : Two parent itemsets,  $p$ : crossover probability

**Output:** Two child itemsets  $C_1$  and  $C_2$

```

1: procedure UC( $P_1, P_2, p$ )
2:   for  $i \leftarrow 1$  to  $\min(\text{len}(P_1), \text{len}(P_2))$  do
3:     if  $\text{random}() \leq p$  then ▷ {random() generates a value in [0,1]}
4:        $C_1[i] \leftarrow P_2[i]$ 
5:        $C_2[i] \leftarrow P_1[i]$ 
6:     else
7:        $C_1[i] \leftarrow P_1[i]$ 
8:        $C_2[i] \leftarrow P_2[i]$ 
9:     end if
10:  end for
11:  return  $C_1, C_2$ 
12: end procedure

```

---

The mutation operation is applied after the crossover operation. Mutation is applied to the itemsets  $C_1$  and  $C_2$  separately. In GRIMP, two mutation operators are implemented, presented in Algorithms 5 and 6. The first operator is known as the Standard Mutation (SM) operator, which introduces random information into the search process to prevent convergence to local optima (Nawaz et al., 2020). In SM, a selected item is altered from its original value to one of the randomly selected items from the population. To demonstrate how mutation works, consider the itemsets  $C_1$  and  $C_2$  after applying the SPC. The mutated itemsets of  $C_1$  and  $C_2$ , denoted as  $C'_1$  and  $C'_2$ , are:

$$\begin{aligned} C'_1 &= \text{apple, bread, biscuit, milk, orange} \\ C'_2 &= \text{beer, butter, diaper, cake} \end{aligned}$$

In this example, the mutation occurs at position 3 in the first itemset ( $C_1$ ), while in the second itemset ( $C_2$ ), the mutation takes place at position 1.

The Pairwise Interchange Mutation (PIM) operator selects and swaps two random items within an itemset. However, in our empirical observations of itemset searching using a GA, we found that the GA struggled to locate the desired

---

**Algorithm 5** Standard mutation

---

**Input:**  $C$ : An itemset and  $population$ : set of all items**Output:** A mutated itemset  $C'$ 

```

1: procedure SM( $C$ )
2:    $ind \leftarrow \text{randomInt}(1, \text{length}(C))$ 
3:    $alter \leftarrow \text{randomSelect}(population, 1)$   $\triangleright$  (singleton itemset)
4:    $C'[ind] \leftarrow alter$   $\triangleright (C'[ind] \neq alter)$ 
5:   Return  $C'$ 
6: end procedure

```

---



---

**Algorithm 6** Modified Pairwise Interchange mutation

---

**Input:**  $C$ : An itemset,  $population$ : set of all items**Output:** A mutated itemset  $C'$ 

```

1: procedure MPIM( $C$ )
2:    $ind1 \leftarrow \text{randomInt}(1, \text{length}(C))$ 
3:    $ind2 \leftarrow \text{randomInt}(1, \text{length}(C))$   $\triangleright ind1 \neq ind2$ 
4:    $alter1, alter2 \leftarrow \text{randomSelect}(population, 1)$ 
5:    $C'[ind1] \leftarrow alter1$   $\triangleright C'[ind1] \neq alter1$ 
6:    $C'[ind2] \leftarrow alter2$   $\triangleright C'[ind2] \neq alter2$ 
7:   Return  $C'$ 
8: end procedure

```

---

target items with the traditional PIM approach. This was attributed to the limited effect of simply exchanging the values between two items in the random itemset. To overcome this challenge, we enhanced the PIM procedure by modifying it to replace the values of the two selected items with random items from the population, rather than solely interchanging their values. We refer to this modified version of PIM as Modified Pairwise Interchange Mutation (MPIM), which is presented in Algorithm 6. To illustrate the application of MPIM, consider the itemsets  $C_1$  and  $C_2$  previously obtained by applying the *SPC* operator. The mutated itemsets produced by MPIM, denoted as  $C'_1$  and  $C'_2$ , are:

$$\begin{aligned}
C'_1 &= \text{apple, bread, biscuit, milk, fruit} \\
C'_2 &= \text{beer, butter, pen, cake}
\end{aligned}$$

In this example, the mutation occurs at positions 3 and 5 in the  $C_1$ , while in the  $C_2$ , the mutation takes place at positions 1 and 3.

The GRIMP framework terminates when the stopping condition of the GA is satisfied, that is when the number of itemsets in the  $CT$  is equal to the *maxCodeTableSize* threshold. This threshold can be set by the user according to their needs in terms of result size. In the experimental evaluation presented in the next section, where GRIMP is compared with KRIMP and SLIM, this threshold

is set to the smallest number of itemsets found by either KRIMP and SLIM so as to compare the compression achieved for the same number of patterns.

To analyze the time complexity of the GRIMP algorithm, we first consider the initialization of the population, which involves collecting all unique items from the dataset  $\mathcal{D}$ . This operation requires  $O(|\mathcal{I}|)$  time, where  $|\mathcal{I}|$  is the number of unique items in  $\mathcal{D}$ . Next, the longest transaction length  $MTL$  is determined by scanning through the dataset, which takes  $O(|\mathcal{D}|)$  time, where  $|\mathcal{D}|$  is the number of transactions. The main *While* loop continues until the size of the  $CT$  reaches  $maxCodeTableSize$ . Let  $n$  represent the maximum number of iterations in the outer loop, which is limited by the  $maxCodeTableSize$ . In each iteration, two random unique itemsets  $P_1$  and  $P_2$  are generated, that requires  $O(1)$  time.

GRIMP then calls the *CountOccurrences* function for both  $P_1$  and  $P_2$ , which takes  $O(|\mathcal{D}| \times |P|)$  time, where  $|P|$  is the average size of the itemsets. This operation is performed twice, contributing a total of  $O(2 \times |\mathcal{D}| \times |P|)$ . Next, the three crossover operations have a complexity of  $O(\min(\text{len}(P_1), \text{len}(P_2)))$ . Since SPC and MPC involve copying elements from the two parent itemsets into the child itemsets, the time complexity can be considered as  $O(|P|)$  in the worst case. For UC, the complexity is also  $O(|P|)$  as it iterates through all elements of  $P_1$  and  $P_2$ . The SM, which replaces a randomly selected element in the itemset, has a complexity of  $O(1)$  for selecting the index and a complexity of  $O(1)$  for replacing the element. Thus, the overall complexity for this mutation is  $O(1)$ . Similarly, the MPIM involves selecting two distinct indices and replacing their values, which also takes  $O(1)$  time.

After the mutation operations, the evolved itemsets are added to the  $CT$ . This operation is effectively a constant-time operation since it involves adding a fixed number of items to a collection, resulting in  $O(1)$  time complexity for each addition. After that, as patterns are identified and added to the  $CT$ , they are simultaneously replaced in the dataset (DDR optimization). Specifically, since we are replacing only two evolved itemsets from  $\mathcal{D}$ , the time complexity for this removal is  $O(|\mathcal{D}|)$  as it requires scanning through the transactions to eliminate occurrences of these patterns. However, because the dataset is progressively reduced with each successful extraction, the average time complexity for subsequent pattern lookups decreases progressively.

Compiling all these contributions, the overall time complexity of GRIMP is as follows:

$$O(|\mathcal{I}| + |\mathcal{D}| + n \times (2 \times |\mathcal{D}| \times |P| + |P| + |P| + 1) + |\mathcal{D}|)$$

Assuming  $n$  is proportional to  $maxCodeTableSize$ , we simplify the above complexity to:

$$O(maxCodeTableSize \times |\mathcal{D}| \times |P| + |\mathcal{I}| + |\mathcal{D}|).$$

Given that  $maxCodeTableSize$  is small (i.e.,  $maxCodeTableSize \ll |\mathcal{D}|$ ), we can consider this size as a constant, leading to the final simplified worst-case complexity of:

$$O(|\mathcal{D}| \times |P| + |\mathcal{I}|).$$

## 5 Experimental evaluation

This section presents the experimental evaluation of the GRIMP. First, the experimental settings are described, including the compared methods, execution environment, and datasets. Then, experiments are presented to evaluate the quality of the resulting itemsets in the *CTs*, in terms of compression ratios, runtimes, convergence, headless chicken test and classification.

### 5.1 Experimental settings

**Algorithms:** GRIMP is implemented in *C++*<sup>2</sup>. The original source code of both KRIMP and SLIM was acquired from the website of J. Vreeken<sup>3</sup>, and are also implemented in *C++*. To ensure a fair comparison of runtimes, the executable versions of GRIMP, KRIMP, and SLIM were run on the same machine. This methodological consistency allows for an accurate assessment of the efficiency of each framework across various datasets.

**Environment:** Experiments were performed on a machine equipped with a 13th generation Core i5 processor and 32 GB of RAM, running Windows 11 Professional. The execution of each framework on a dataset was constrained by a time limit of two hours.

**Datasets:** We utilized a comprehensive set of 21 publicly available datasets. This wide-ranging dataset collection ensures that our findings are generalizable for various real-world scenarios where data have different characteristics. We analyze a diverse set of benchmark and real datasets. In each dataset, the data is modeled as transactions (sets of items). We selected some of the largest and dense databases from the LUCS/KDD dataset repository<sup>4</sup>. Additionally, we utilized the *Accidents*, *Chess*, and *Pumsb* datasets from the FIMI<sup>5</sup> repository. The *Chess (kr-k)* dataset is sourced from the UCI repository<sup>6</sup>. The *Heart* and *Wine* datasets were obtained from the SPMF repository<sup>7</sup>. For convenience, all the 21 datasets have also been uploaded to the GitHub repository of GRIMP. These datasets and their statistical details are presented in Table 2. For each dataset,  $|\mathcal{D}|$  denotes the number of transactions,  $|\mathcal{I}|$  is the count of unique items, *ATL* (average transaction length) is the average count of items in each transaction, and Density =  $\frac{ATL}{|\mathcal{I}|} \times 100$ .

<sup>2</sup> [github.com/MuhammadzohaibNawaz/GRIMP](https://github.com/MuhammadzohaibNawaz/GRIMP)

<sup>3</sup> [vreeken.eu/prj/krimp/](http://vreeken.eu/prj/krimp/)

<sup>4</sup> [cgi.csc.liv.ac.uk/~frans/KDD/Software/LUCS-KDD-DN/](http://cgi.csc.liv.ac.uk/~frans/KDD/Software/LUCS-KDD-DN/)

<sup>5</sup> [fimi.uantwerpen.be/data/](http://fimi.uantwerpen.be/data/)

<sup>6</sup> [archive.ics.uci.edu/](http://archive.ics.uci.edu/)

<sup>7</sup> [philippe-fournier-viger.com/spmf/index.php?link=datasets.php](http://philippe-fournier-viger.com/spmf/index.php?link=datasets.php)

Table 2: Description of datasets and their statistics

Dataset	$ \mathcal{D} $	$ \mathcal{I} $	<i>ATL</i> Density	
Accidents	340,183	468	33.80	7.22
BMSWV1	59,602	497	2.53	0.51
Adult	48,842	97	35.60	15.33
Chess (k-k)	3,196	75	37	49.33
Chess (kr-k)	28,056	58	7.00	12.07
Connect-4	67,557	129	43	33.33
Heart	303	50	13.98	27.96
Ionosphere	351	157	35.01	22.29
Iris	150	19	5.01	26.32
Led7	3,200	24	7.99	33.33
Letter Recognition	20,000	102	17.17	16.67
Mushroom	8,124	119	23.01	19.33
Nursery	12,960	32	9.01	28.13
Page Blocks	5,473	44	11.00	25
Pen Digits	10,992	86	17.00	19.77
Pima	768	38	9.06	23.68
Pumsb	49,046	2,113	73.95	3.50
Pumsbstar	49,046	2,088	50.52	2.42
Retail	88,162	16,470	9.88	0.06
Tic-tac-toe	958	29	9.99	34.48
Wine	178	68	14.00	20.59

## 5.2 Compression ratio

In the first experiment, we assess the effectiveness of GRIMP in describing a dataset by focusing on the key metric, the compression ratio. It is to be noted that the number of iterations in Algorithm 1 was empirically fixed at five, as this value consistently yielded optimal compression ratio results on different datasets. This empirically determined iteration count was maintained throughout the study to ensure consistency and comparability of results across different experimental conditions. We analyze six variants of GRIMP, each employing a combination of one of the three crossover operators with one of the two mutation operators. For each variant, the results are also averaged across all datasets to provide a comprehensive view of their performance. This allows identifying the best-performing variant. Lower compression ratio values indicate superior performance. The relative compression ratio, denoted as  $L\%$ , is calculated as:

$$\frac{L_G(\mathcal{D}, CT)}{L_G(\mathcal{D}, Pop)}\% \quad (15)$$

where  $\mathcal{D}$  is clear from the context and  $Pop$  represents the population.

The results are structured to first present the compression performance followed by the computational time for each dataset. The details of the result are presented in Table 3, where the numerator represents the compression achieved,

while the denominator indicates the time taken to achieve this compression. For example, the first entry  $\frac{42.70}{6653.60}$  indicates that the GA variant using SPC with SM achieved 42.70 percent compression in a time span of 6653.60 seconds. The values highlighted in bold represent the best-performing variant in terms of compression ratios and/or runtime. Any variant that failed to complete within the time limit of two hours was omitted marked as “-”) from the results. By excluding excessively long runs, we ensure that the evaluation is focused on solutions that are feasible within reasonable operational parameters.

Table 3: Compression and runtime for all GRIMP variants on all datasets. Each entry in the table indicates the compression achieved (numerator) by a variant and the required time to achieve this compression (denominator).

Dataset	# of itemsets	SM			MPIM		
		SPC	MPC	UC	SPC	MPC	UC
Accidents	1583	42.70 6653.60	<b>6.50</b> <b>6408.80</b>	44.11 6742.20	29.50 6516.40	8.80 6430.40	—
Adult	1201	15.70 133.90	<b>3.50</b> 128.40	18.90 147.80	15.40 148.50	5.40 <b>112.10</b>	19.10 163.30
BMSWV1	718	90.10 1276.40	48.40 <b>903.40</b>	94.70 1382.40	91.10 1426.30	<b>33.40</b> 1024.50	92.80 1137.90
Breast	30	25.10 0.21	<b>22.90</b> <b>0.17</b>	31.60 0.23	31.70 0.22	24.80 0.25	37.60 0.28
Chess (k-k)	292	23.00 13.00	<b>9.90</b> <b>10.34</b>	25.90 21.16	26.00 24.82	11.90 16.34	27.20 17.44
Chess (kr-k)	1060	12.30 64.51	<b>11.10</b> <b>63.31</b>	11.40 624.90	15.10 129.60	14.70 79.54	16.20 560.34
Connect-4	1671	28.90 1392.30	<b>3.30</b> <b>12.00</b>	29.30 1121.80	31.20 1378.40	6.20 1146.50	31.60 <b>1090.40</b>
Heart	79	35.70 0.28	<b>35.70</b> <b>0.22</b>	37.70 0.30	33.60 0.31	30.20 0.29	35.40 0.29
Ionosphere	170	57.90 1.18	<b>24.00</b> 1.17	55.20 1.11	51.20 1.12	31.20 <b>1.00</b>	53.50 1.05
Iris	13	34.50 0.03	29.60 <b>4.90</b>	40.60 0.02	35.30 0.01	<b>28.80</b> 0.02	39.00 <b>0.01</b>
Led7	152	16.60 3.58	<b>4.90</b> <b>3.60</b>	19.50 3.51	22.70 3.64	10.20 <b>3.94</b>	24.90 3.47
Letter Recognition	1599	8.20 200.40	<b>3.26</b> <b>5.40</b>	8.51 179.54	10.20 180.20	<b>3.94</b> <b>169.93</b>	11.40 174.81
Mushroom	442	30.60 38.35	<b>5.40</b> 30.52	39.70 28.81	32.20 30.23	12.20 <b>25.61</b>	34.20 28.39
Nursery	260	17.90 2.45	<b>2.20</b> 1.92	19.70 1.54	19.00 <b>1.48</b>	5.30 1.59	23.10 2.14
Page Blocks	53	19.22 1.32	<b>2.56</b> <b>1.04</b>	19.86 1.28	19.35 1.82	10.06 1.52	21.18 2.17
Pen Digits	1247	57.10 69.82	18.40 <b>56.96</b>	61.60 154.90	55.00 75.09	<b>16.60</b> 61.19	57.10 160.04
Pima	58	30.60 0.43	<b>16.70</b> 13.70	27.60 0.52	21.10 0.48	26.60 <b>10.80</b>	26.10 0.49
Pumsbstar	331	46.20 2765.20	13.70 523.70	48.40 4814.50	26.40 1178.80	<b>0.42</b> <b>377.50</b>	28.30 1320.30
Retail	6264	—	<b>56.40</b> 1099.40	—	—	<b>56.40</b> <b>1062.10</b>	—
Tic-tac-toe	160	26.30 1.12	<b>19.60</b> 1.09	29.90 1.12	27.80 1.36	27.10 <b>1.07</b>	29.00 1.18
Wine	63	47.50 0.11	44.60 0.11	49.20 0.09	43.30 0.09	<b>41.10</b> <b>0.08</b>	45.90 0.10

The analysis of compression performance across various datasets reveals that the MPC/SM variant generally achieves the lowest compression ratios among the tested variants (with few exceptions like *BMSWV1*, *Iris*, *Pen Digits*, *Pumsbstar*, *Retail*, and *Wine*), underscoring its superior effectiveness in data compression. Notably, for these exceptions, the differences in compression ratios are minimal, indicating that the effectiveness of MPC/SM remains consistently high across diverse datasets. Remarkably, this variant demonstrates exceptional performance, almost two times more than the second-best variant, on datasets such



as *Connect-4*, *Heart*, *Led7*, *Letter Recognition*, *Mushroom*, *Nursery*, *Page Blocks*, *Pima*, and *Tic-tac-toe*. After the MPC/SM variant, the second most effective variant is MPC/MPIM. This variant also shows high compression efficiency, often closely following the performance of MPC/SM. It consistently secures low compression ratios on a variety of datasets, making it another reliable choice for effective data compression across diverse data scenarios. In contrast, the UC/SM variant consistently produces higher compression ratios, indicating a lower efficiency in data compression compared to the other variants.

Next, we present average results in Table 4. It is important to note that datasets where all variants of GRIMP could not complete the task within two hours, such as *Accident* and *Retail*, are excluded from this average result analysis. Specifically, the average is calculated by summing the total compression ratios and execution times for each variant across all datasets. This sum is then divided by the number of datasets included in the analysis. Including datasets with incomplete results would unfairly increase the average time for variants that completed the task within the allotted time, potentially skewing the efficiency comparison. This exclusion ensures that the results reflect only those scenarios where GRIMP successfully completed its compression tasks within the allotted timeframe, thereby providing a more accurate comparison.

Table 4: Average compression and time for each GRIMP variant

GRIMP variant	Avg. Compression	Avg. Time (in sec.)
SPC/SM	32.71	313.93
MPC/SM	<b>15.57</b>	159.34
UC/SM	35.28	460.15
SPC/MPIM	31.98	228.74
MPC/MPIM	18.51	<b>156.13</b>
UC/MPIM	34.40	254.06

The analysis of GRIMP’s variants reveals distinct performance characteristics, with the MPC/SM configuration emerging as the most effective, achieving a significantly lower average compression ratio of 15.57, indicating superior compression efficiency. Conversely, the UC/SM configuration records the highest average compression ratio of 35.28 and highest average time of 460.15 seconds, highlighting its relative ineffectiveness in compressing data. Moreover, the MPC/MPIM configuration demonstrates good computational efficiency, requiring an average time of 156.13, which is among the lowest in the table. Overall, the ranking of variants based on the average compression achieved is in the order MPC/SM > MPC/MPIM > SPC/MPIM > SPC/SM > UC/MPIM > UC/SM.

Next, we compare GRIMP with KRIMP and SLIM. For this comparison, we take the best-performing variant of GRIMP, namely MPC/SM, since this variant gives the best result for compression ratio. From this point onward, we use this variant for comparison. Table 5 presents the results of this comparison. The  $\Delta$  symbol indicates the difference between algorithms, with  $\Delta_{\text{KRIMP}}^{\text{GRIMP}}$  representing

Table 5: Comparison of compression ratios and runtimes of GRIMP, KRIMP, and SLIM, with results presented using the notation  $\frac{\text{ratio}}{\text{time}}$ .

Dataset	GRIMP	KRIMP	$\Delta_{\text{KRIMP}}^{\text{GRIMP}}$	SLIM	$\Delta_{\text{SLIM}}^{\text{GRIMP}}$
Accidents	<u>6.50</u> <b>6408.80</b>	<u>55.10</u> —	<u>48.60</u> —	<u>31.10</u> —	<u>24.60</u> —
Adult	<u>3.50</u> <b>128.40</b>	<u>24.40</u> 205.00	<u>20.90</u> 76.60	<u>22.80</u> 240.45	<u>19.30</u> 112.05
BMSWV1	<u>48.40</u> <b>903.40</b>	<u>85.90</u> —	<u>37.50</u> —	<u>84.00</u> —	<u>35.60</u> —
Breast	<u>22.90</u> <b>0.17</b>	<u>17.00</u> 0.39	<u>−5.90</u> 0.22	—	—
Chess (k-k)	<u>9.90</u> <b>10.34</b>	<u>30.00</u> —	<u>20.10</u> —	<u>14.70</u> 22.70	<u>4.80</u> 12.36
Chess (kr-k)	<u>11.10</u> <b>63.31</b>	<u>61.60</u> 68.38	<u>50.50</u> 5.27	<u>57.50</u> 86.46	<u>46.40</u> 23.15
Connect-4	<u>9.30</u> <b>1021.80</b>	<u>42.90</u> —	<u>39.60</u> —	<u>12.30</u> —	<u>9.00</u> —
Heart	<u>12.00</u> <b>0.22</b>	<u>57.70</u> 4.81	<u>45.70</u> 4.59	—	—
Ionosphere	<u>24.10</u> <b>1.17</b>	<u>59.80</u> —	<u>35.70</u> —	<u>49.70</u> 65.51	<u>25.60</u> 64.34
Iris	<u>29.60</u> <b>0.02</b>	<u>48.20</u> 0.06	<u>18.60</u> 0.04	—	—
Led7	<u>4.90</u> <b>3.26</b>	<u>28.60</u> 1.20	<u>23.70</u> −2.06	—	—
Letter Recognition	<u>3.00</u> <b>179.54</b>	<u>35.70</u> 2796.01	<u>32.60</u> 2616.47	<u>33.40</u> —	<u>30.30</u> —
Mushroom	<u>5.40</u> <b>30.52</b>	<u>20.50</u> —	<u>15.10</u> —	<u>18.50</u> 45.21	<u>13.10</u> 14.69
Nursery	<u>2.20</u> <b>1.92</b>	<u>45.50</u> 12.16	<u>43.30</u> 10.24	—	—
Page Blocks	<u>2.56</u> <b>1.04</b>	<u>5.00</u> 1.32	<u>1.90</u> 0.22	—	—
Pen Digits	<u>18.40</u> <b>56.96</b>	<u>42.20</u> 3370.80	<u>23.80</u> 3313.84	<u>39.40</u> 1364.87	<u>21.00</u> 1307.91
Pima	<u>16.70</u> <b>0.44</b>	<u>34.80</u> 0.81	<u>18.10</u> 0.37	—	—
Pumsbstar	<u>13.70</u> <b>523.70</b>	<u>56.00</u> —	<u>42.30</u> —	<u>25.10</u> —	<u>11.40</u> —
Retail	<u>65.60</u> <b>1099.40</b>	<u>97.50</u> 4031.54	<u>31.90</u> 2932.14	—	—
Tic-tac-toe	<u>19.60</u> <b>1.09</b>	<u>56.00</u> 2.00	<u>36.40</u> 0.91	—	—
Wine	<u>44.60</u> <b>0.11</b>	<u>77.40</u> 2.01	<u>32.80</u> 1.90	—	—

the difference between GRIMP and KRIMP. Similarly,  $\Delta_{\text{SLIM}}^{\text{GRIMP}}$  represents the difference between GRIMP and SLIM. For datasets whose compression results were not available in the original SLIM paper, we did not include them in comparison and marked these instances with a “—”. This approach ensures a fair comparison across all methodologies within practical operational constraints and maintains consistency in our reporting.

GRIMP demonstrates superior performance in most cases, achieving better compression ratios often in less time than its counterparts. This advantage is particularly evident for larger datasets such as *Accidents*, *BMSWV1*, and *Connect-4*. However, it is important to note that GRIMP’s performance is not uniformly superior across all datasets. In the case of the *Breast* dataset, KRIMP outperforms GRIMP, achieving a compression ratio of 17.00 compared to GRIMP’s 22.90. This exception highlights areas where GRIMP could potentially be improved. For smaller datasets like *Iris*, *Pima*, and *Wine*, the performance gap between GRIMP and the other two narrows. While GRIMP still generally performs better, the difference in compression ratios is less pronounced compared to larger datasets. This suggests that GRIMP’s advantages become more significant as dataset size increases. A notable strength of GRIMP is its consistent ability to complete the compression task within the two-hour time limit. In several in-

Table 6: Comparison of  $CT$  length of GRIMP with KRIMP and SLIM to achieve equivalent compression ratios

Dataset	GRIMP	$\frac{\text{KRIMP}}{\text{SLIM}}$	Comparison (%)
Accidents	209	1583	86.80
Adults	240	2018	88.14
BMSWV1	180	1303	86.18
Breast	190	1201	84.18
Chess (k-k)	419	718	41.64
Chess (kr-k)	560	965	41.97
Connect-4	34	30	88.67
Heart	N/A	N/A	N/A
Ionosphere	98	275	64.36
Iris	155	292	46.92
Led7	209	1684	87.55
Letter Recognition	212	1060	80.00
Mushroom	199	2036	90.23
Nursery	434	1670	74.01
Page Blocks	18	79	77.22
Pen Digits	N/A	N/A	N/A
Pima	71	170	58.24
Pumsbstar	83	240	65.42
Retail	10	15	76.92
Tic-tac-toe	N/A	N/A	N/A
Wine	27	152	82.24
Average	N/A	N/A	N/A
	152	1780	91.46
	141	1599	91.18
	169	689	75.47
	131	442	70.36
	19	260	92.69
	N/A	N/A	N/A
	36	53	32.07
	N/A	N/A	N/A
	698	1247	44.02
	911	1347	32.36
	32	58	44.83
	N/A	N/A	N/A
	106	331	67.98
	1043	4274	75.62
	3200	6264	48.91
	N/A	N/A	N/A
	76	232	67.24
	N/A	N/A	N/A
	40	76	47.37
	N/A	N/A	N/A
	285.80	906.33	68.35
	372.72	1373.45	72.86

stances where KRIMP or SLIM failed to complete within the timeframe, GRIMP successfully finished the task, often achieving impressive compression ratios.

To further explore Grimp’s capabilities, we now present the results detailing the number of patterns GRIMP required to achieve the same compression ratios as KRIMP and SLIM. This analysis illustrates the efficiency of GRIMP in attaining competitive compression ratios, potentially with fewer patterns. Table 6 provides the results for the number of patterns required by GRIMP, KRIMP, and SLIM to achieve similar levels of data compression. The second column quantifies the number of patterns GRIMP requires, represented as a fraction where the numerator reflects the number of patterns used to match KRIMP’s compression ratio and the denominator for SLIM’s. Similarly, the third column shows the number of patterns needed by KRIMP and SLIM to achieve their respective compression ratios, also in fractional form. The fourth column quantifies the percentage reduction in the number of patterns used by GRIMP compared to KRIMP (numerator) and SLIM (denominator). For instance, in the *Accidents* dataset, the value  $\frac{86.80}{88.14}$  indicates that GRIMP used 86.80% fewer patterns than KRIMP and 88.14% fewer than SLIM to achieve a similar compression ratio. With an average numerator of 68.35% and a denominator of 72.86% in the comparison column,

it is evident that GRIMP requires significantly fewer patterns than both KRIMP and SLIM for all datasets.

### 5.3 Runtime

In terms of processing speed, as shown in Table 4, GRIMP, particularly with the MPC/MPIM variant, shows exceptionally low processing times across datasets including *Heart*, *Ionosphere*, *Iris*, *Pima*, and *Wine*. Overall, the ranking of variants based on the average time is in the order MPC/MPIM > MPC/SM > SPC/MPIM > UC/MPIM > SPC/SM > UC/SM.

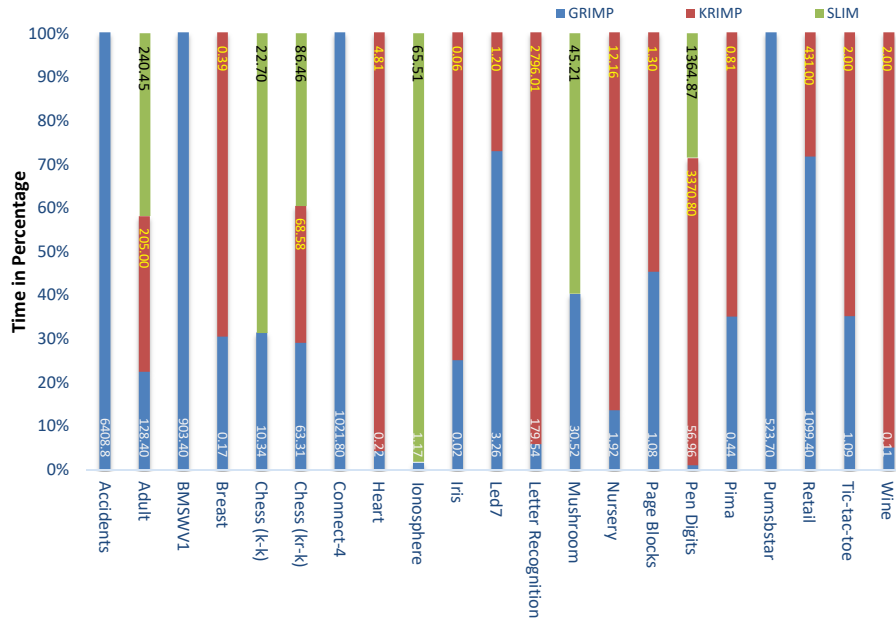


Fig. 5: Runtime comparison of GRIMP, KRIMP and SLIM

The runtime of GRIMP, KRIMP and SLIM as a percentage of the total observed runtime is presented in Figure 5. GRIMP consistently outperforms both KRIMP and SLIM in terms of speed, with the sole exceptions being the *Led7* and *Retail* datasets, where it falls short of KRIMP’s performance. It is important to note that this comparison includes datasets where runtime values for at least one of KRIMP or SLIM were available. Notably, only GRIMP was able to complete the compression within the allocated timeframe for the *Accidents*, *BMSWV1*, *Connect-4*, and *Pumsbstar* datasets.

Next, we illustrate the impact of DDR on the overall time reduction for the *Accident* dataset, analyzed across all six variants of GRIMP. Figure 6 showcases these effects, where the x-axis represents the length of the *CT*, and the y-axis

indicates the time in milliseconds. These lines exhibit significant fluctuations, a result of the inherent randomness in the crossover and mutation processes. These processes interact with itemsets whose lengths vary randomly—larger itemsets typically take longer to process due to more extensive searches required in the dataset.

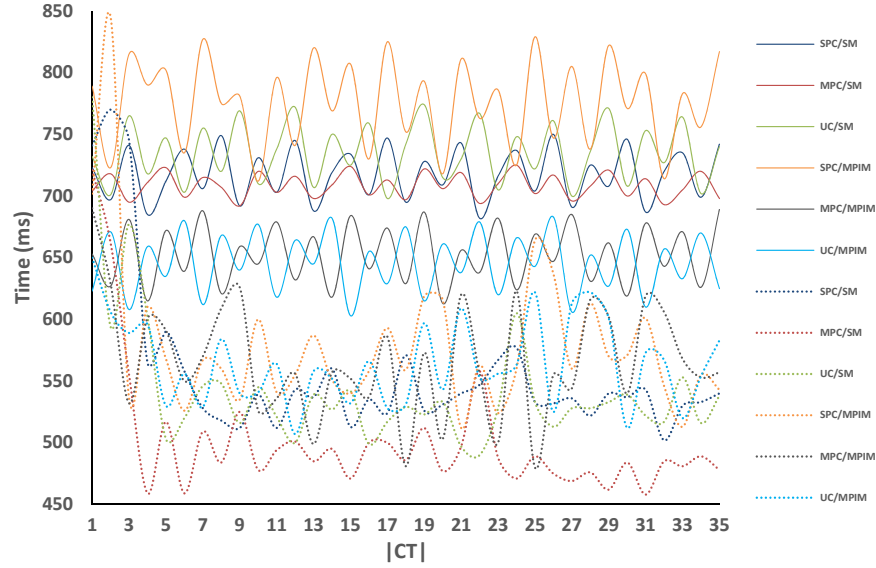


Fig. 6: Effect of the DDR optimization on the execution time of GRIMP for the *Accident* dataset. Solid and dashed lines represent the performance of variants without and with the optimization, respectively.

The dotted lines demonstrate the notable reduction in execution times post-optimization across all variants. The optimization, by systematically decreasing the size of the dataset as patterns are added to the *CT*, reduces the overhead of searching for patterns. Each successful addition to the *CT* not only contributes to compression efficiency but also expedites subsequent operations by reducing the dataset size. This leads to a smoother and generally lower trajectory in the graph, indicating a more consistent and reduced processing time.

#### 5.4 Convergence

Another experiment was carried out to observe the convergence behavior of GRIMP, that is how it systematically refines the compression ratio through successive iterations to enhance pattern selection for optimal compression. Convergence of the compression ratio is assessed with respect to two factors, namely the number of patterns that have been added to the *CT*, and the number of generations that are used to evolve a set of patterns before it is added to the

*CT*. The results are depicted in Figure 7. The bottom x-axis represents the number of patterns that have been added to the *CT*, while the top x-axis, indicates the number of generations that GRIMP utilized to evolve each set of patterns. The y-axis quantifies the effectiveness of the compression by measuring the compression ratio. To provide concise results as a figure, the number of patterns and number of generations are respectively observed in the range of [1,20] and [1,5] but the overall convergence behavior of GRIMP remains the same for other ranges of values.

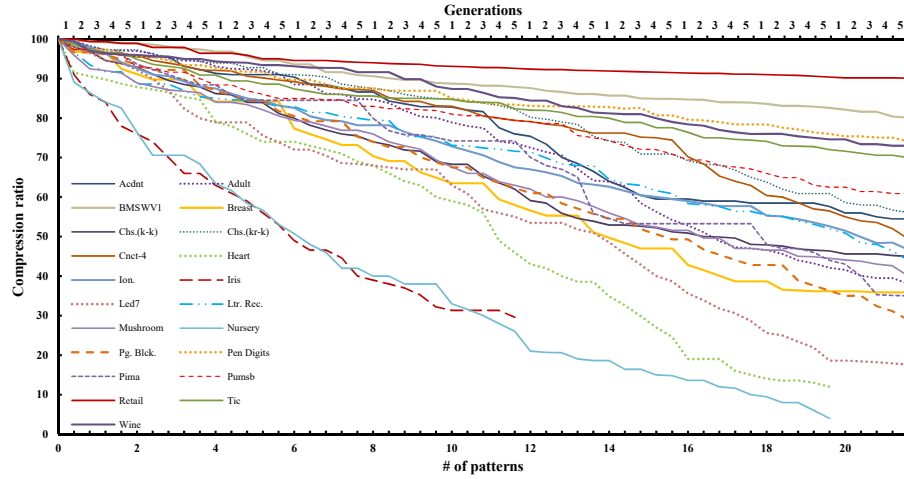


Fig. 7: Comparison of compression ratios with respect to number of itemsets and generations

On most datasets, GRIMP shows a trend where the compression ratio initially remains high but gradually declines as more patterns are added to the *CT*. This suggests that early additions contribute more significantly to data representation than subsequent ones. Different datasets exhibit unique convergence behaviors, which reflects their inherent complexities and the varying effectiveness of GRIMP in compressing them. *Heart*, for example, starts with a higher compression ratio that declines sharply as more patterns are added, indicating a quick gain in compression that stabilizes as the pattern count increases. Similarly, *Breast* and *Page Blocks* show moderate compression efficiencies with a steady decline, highlighting a consistent difficulty in achieving optimal compression. Looking at the example of *Pima* dataset, we can see that there is no improvement in compression particularly from patterns 5 to 8 and 14 to 18. This is because GRIMP could not improve the compression in this period of time. Overall, *Pima* exhibits an unusual decline, stabilizing at certain iterations before experiencing sudden decreases, a pattern that continues until it reaches optimal compression. A similar phenomenon is evident in the *Iris* dataset analysis, where the objective was to

identify compressing patterns within a limit of 13 patterns. Here too, we observe periods where GRIMP failed to yield improved compression results, followed by phases where new patterns were generated, enhancing the compression efficiency. These observations underscore a common characteristic inherent in evolutionary framework like GRIMP: periods of stagnation where it is trapped at local optima, temporarily unable to make further progress. Given the random nature of the underlying operations, such behavior is to be expected. This randomness can sometimes hinder continuous improvement, leading to plateaus in performance until new, effective variations emerge through the evolutionary process.

As discussed in Section 4, GRIMP is strategically engineered to either enhance the solution or preserve its current quality, thereby ensuring no degradation over time. Figure 7 clearly illustrates this principle across all datasets. Here, we observe a consistent pattern where the compression ratios either decrease or stabilize as the number of generations and itemsets increases. This behavior underscores GRIMP’s effectiveness in refining or maintaining data compression efficiency through successive iterations.

### 5.5 Headless Chicken Test

In GAs, the concept of crossover involves selecting two chromosomes (solutions) and merging their components to create more robust offspring. (Holland, 1975) utilized building blocks from schema theory to formalize the concept of crossover. The crossover mechanics offer a way for the implementation of this concept. Therefore, while all crossover types share the same underlying principle, the methods or mechanics employed to execute the principle concept can vary significantly. For instance, SPC and MPC utilize a single and two (or more) crossing points, respectively. Generally, the effectiveness of crossover can be accessed (or tested) by comparing the performance of a GA with crossover to one without it. If the GA incorporating crossover demonstrates better performance, it provides evidence supporting the effectiveness of the crossover operation. To distinguish between the concept of crossover and its mechanics, (Jones, 1995) proposed the headless chicken test (HCT) to evaluate the usefulness of the crossover operation. It was argued that crossover mechanics could effectively drive search and evolutionary processes independently of the crossover idea.

The HCT is employed to assess the effectiveness of the three crossover operators in GRIMP. In the test, GRIMP with normal SPC, MPC and UC is contrasted against a version using randomized SPC, MPC and UC. Rather than recombining two parents, a random crossover generates two random individuals that are used in the crossover process with the original parents (Figure 8). This replaces the strategic recombination of parents with a mechanical, randomized process, removing the fundamental element of crossover - direct interaction between parents. If a GA does not demonstrate additional benefits from the crossover concept, it may be equally effective to utilize macromutations instead. The inferior performance of the GA with standard crossover in comparison to the GA with random crossover suggests a lack of well-defined building blocks. Table 7 presents the comparative results between GRIMP with conventional and

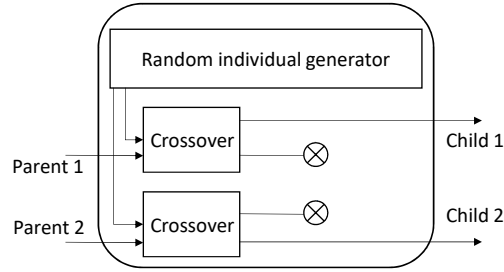


Fig. 8: Headless chicken test with random crossover

randomized crossover operators. The ‘ $\times$ ’ sign next to the names of the crossover operators indicates their randomized versions. When examining the average compression ratio, it is evident that the original GRIMP with normal crossover operators performs similarly to GRIMP with randomized crossover operators without any significant differences. This suggests that well-defined building blocks exist for the crossover operators in GRIMP.

Table 7: Results of the headless chicken test for GRIMP

GA Variant	Avg. Compression	Avg. Time
SPC/SM	32.71	313.93
MPC/SM	15.57	159.34
UC/SM	35.28	460.15
SPC/MPIM	31.98	228.74
MPC/MPIM	18.51	156.13
UC/MPIM	34.40	254.06
SPC $\times$ /SM	32.52	395.35
MPC $\times$ /SM	16.41	213.56
UC $\times$ /SM	34.31	523.55
SPC $\times$ /MPIM	30.12	408.61
MPC $\times$ /MPIM	20.62	232.97
UC $\times$ /MPIM	33.51	409.71

## 5.6 Classification

In previous experiments, we observed that GRIMP describes data more succinctly than KRIMP and SLIM. In a subsequent experiment, we independently validate how well GRIMP captures data distributions through classification. We employ a classification scheme similar to KRIMP and SLIM for this evaluation.

For classification, we need to create a *CT* for each class. This process begins by splitting the database according to class labels. After splitting, we remove



the class labels from all transactions within each subset. Next, we apply GRIMP to each of these class-specific databases, thereby generating a *CT* for each class. Once the *CT*s are constructed, classifying a transaction  $t$  becomes straightforward: we assign the class label associated with the *CT* that yields the minimal encoded length for  $t$ . We measure performance using accuracy, which represents the percentage of true positives on the test data. All the reported results have been obtained using 10-fold cross-validation to ensure robust and reliable performance estimates. By comparing the results to those of KRIMP and SLIM, we can evaluate GRIMP’s relative performance in generating meaningful and discriminative *CT*s.

Table 8: Comparison of classification accuracy (%).

Dataset	GRIMP	KRIMP	SLIM
Adult	78	79	<b>80</b>
Chess (kr-k)	54	50	<b>56</b>
Connect-4	<b>71</b>	68	69
Ionosphere	<b>95</b>	91	88
Letter Recognition	<b>95</b>	<b>95</b>	94
Mushroom	94	<b>100</b>	<b>100</b>
Pen Digits	73	93	<b>94</b>
Waveform	<b>89</b>	72	74

The classification results, as presented in Table 8, demonstrate the effectiveness of GRIMP’s *CT*s in capturing meaningful patterns for classification tasks. While GRIMP was primarily designed for compression rather than classification, its performance in this domain provides valuable insights into the quality of the patterns it discovers. GRIMP shows competitive performance across various datasets, outperforming both KRIMP and SLIM on several datasets. Notably, GRIMP achieves the highest accuracy for the *Connect-4*, *Ionosphere*, and *Waveform* datasets, with a particularly significant improvement in the *Waveform* dataset (89% compared to 72% for KRIMP and 74% for SLIM). For the *Letter Recognition* dataset, GRIMP matches the best performance, achieving 95% accuracy alongside KRIMP. In cases where GRIMP does not surpass KRIMP or SLIM, such as in the *Adult* and *Chess (kr-k)* datasets, its performance remains comparable. For instance, in the *Adult* dataset, GRIMP’s accuracy (78%) is very close to KRIMP’s (79%) and SLIM’s (80%).

It is important to note that GRIMP does not always outperform KRIMP and SLIM in classification. This is not a major issue because the primary objective of GRIMP is superior compression, and classification is used here merely as a mean to evaluate the quality of the discovered patterns. The *Mushroom* dataset presents an interesting case where both KRIMP and SLIM achieve perfect accuracy, while GRIMP reaches 94%. This slight difference might be attributed to GRIMP’s more aggressive compression, which could lead to some loss of dis-

criminative information in favor of a more concise representation. Overall, these results demonstrate that GRIMP, while optimized for compression, generates *CTs* that capture meaningful and discriminative patterns in the data. Its ability to compete with, and often outperform, KRIMP and SLIM in classification tasks underscores the effectiveness of GRIMP’s pattern discovery approach.

To summarize the results of the overall experimental evaluation of GRIMP, it is observed that it distinctly surpasses both KRIMP and SLIM in several key aspects of data compression and processing efficiency. Notably, GRIMP’s ability to deliver the same compression ratio with around 70% fewer patterns than its competitors highlights its efficiency in pattern representation. GRIMP also excels in runtime performance, processing data more rapidly than both KRIMP and SLIM. This efficiency is especially valuable in large-scale data environments where quick processing is critical. This runtime efficiency is largely attributable to the DDR optimization technique, which accelerates processing by reducing the size of the dataset. Additionally, the convergence analysis of GRIMP highlights its consistent progression towards optimal compression ratios, confirming its reliability and effectiveness. While GRIMP is primarily designed to identify patterns that best capture the characteristics of the data, its applicability in classification tasks was also explored, where it proved to be competitive. This indicates that the *CTs* generated by GRIMP are not only useful for finding the best set of patterns that capture the data most effectively but also possess significant discriminative properties for classification tasks.

## 6 Conclusion

This paper introduced GRIMP, a GA for compression-based descriptive itemset mining. GRIMP addresses the limitations of traditional FIM by integrating the principles of pattern compression with evolutionary algorithm. It efficiently explores the search space of potential itemsets, focusing on those that offer the most significant compression benefits. GRIMP has been evaluated through comprehensive experiments demonstrating its ability to achieve better compression ratios and runtime performance compared to the KRIMP and SLIM. Moreover, despite being optimized primarily for compression, GRIMP has also shown promising results in classification tasks. Obtained results indicated that GRIMP not only reduced the redundancy in the extracted patterns but also enhanced the interpretability and usefulness of the mining results.

It is important to acknowledge certain limitations of GRIMP. Due to the inherent randomness in how itemsets are initially generated and the stochastic nature of GA operators, there can be significant variability in performance. Occasionally, this randomness may lead to suboptimal itemsets, including low compression results. Additionally, some candidate itemsets added to the final code table may have a limited impact on enhancing compression, and the repetitive comparison of candidate itemsets against the database contributes significantly to the computational overhead. Some future research directions are: Future research can explore enhancing GRIMP with additional data mining tasks, such

as sequential, graph, and contrast pattern mining. Potential developments could also focus on increasing the scalability of GRIMP for handling even larger datasets and adapting them to dynamic data environments, thereby extending its utility and impact across various real-world applications. Another direction is to replace GA with alternative evolutionary and heuristic algorithms such as particle swarm optimization (Wang et al., 2018), ant colony optimization (Dorigo and Stützle, 2019), bat algorithm (Yang and He, 2013), and simulated annealing (Nikolaev and Jacobson, 2010).

## Bibliography

- Aggarwal, C. C. (2014). Applications of frequent pattern mining. In Aggarwal, C. C. and Han, J., editors, *Frequent Pattern Mining*, pages 443–467, Cham. Springer International Publishing.
- Agrawal, R., Imieliński, T., and Swami, A. (1993). Mining association rules between sets of items in large databases. In *Proceedings of 19th International Conference on Management of Data (SIGMOD'93)*, pages 207–216.
- Akoglu, L., Tong, H., Vreeken, J., and Faloutsos, C. (2012). Fast and reliable anomaly detection in categorical data. In *Proceedings of the 21st ACM International Conference on Information and Knowledge Management (CIKM'12)*, pages 415–424.
- Belaid, M.-B. and Lazaar, N. (2021). Frequent itemset mining with multiple minimum supports: A constraint-based approach. *arXiv preprint*, abs/2109.07844.
- Bonchi, F., Giannotti, F., and Pedreschi, D. (2006). A relational query primitive for constraint-based pattern mining. In Boulicaut, J.-F., De Raedt, L., and Mannila, H., editors, *Constraint-Based Mining and Inductive Databases*, pages 14–37. Springer.
- Boulicaut, J.-F., Bykowski, A., and Rigotti, C. (2003). Free-sets: A condensed representation of boolean data for the approximation of frequency queries. *Data Mining and Knowledge Discovery*, 7(1):5–22.
- Bouritsas, G., Loukas, A., Karalias, N., and Bronstein, M. (2021). Partition and code: Learning how to compress graphs. In *Proceedings of 35th Annual Conference on Neural Information Processing Systems (NeurIPS 2021)*, pages 18603–18619.
- Budhathoki, K. and Vreeken, J. (2015). The difference and the norm — characterising similarities and differences between databases. In *Proceedings of European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML PKDD 2015)*, pages 206–223.
- Calders, T. and Goethals, B. (2007). Non-derivable itemset mining. *Data Mining and Knowledge Discovery*, 14(1):171–206.
- Cao, Y., Xu, J., Yang, C., Wang, J., Zhang, Y., Wang, C., Chen, L., and Yang, Y. (2023). When to pre-train graph neural networks? from data generation perspective! In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDE'23)*, pages 142–153.
- Darrab, S., Broneske, D., and Saake, G. (2021). Modern applications and challenges for rare itemset mining. *International Journal of Machine Learning*, 11(3):208–218.
- Dorigo, M. and Stützle, T. (2019). Ant colony optimization: Overview and recent advances. In Gendreau, M. and Potvin, J.-Y., editors, *Handbook of Metaheuristics*, pages 311–351. Springer.
- Faas, M. and van Leeuwen, M. (2020). Vouw: geometric pattern mining using the mdl principle. In *Proceedings of 18th International Symposium on Intelligent Data Analysis (IDA 2020)*, pages 158–170.

- Farghaly, H. M. and El-Hafeez, T. A. (2022). A new feature selection method based on frequent and associated itemsets for text classification. *Concurrency and Computation: Practice and Experience*, 34(25):e7258.
- Fournier-Viger, P., Li, X., Yao, J., and Lin, J. C.-W. (2018). Interactive discovery of statistically significant itemsets. In *Proceedings of 31st International Conference on Industrial Engineering and Other Applications of Applied Intelligent Systems (IEA/AIE 2018)*, pages 101–113.
- Gainer-Dewar, A. and Vera-Licona, P. (2017). The minimal hitting set generation problem: algorithms and computation. *SIAM Journal on Discrete Mathematics*, 31(1):63–100.
- Galbrun, E. (2022). The minimum description length principle for pattern mining: A survey. *Data Mining and Knowledge Discovery*, 36(5):1679–1727.
- Gen, M. and Lin, L. (2023). Genetic algorithms and their applications. In Pham, H., editor, *Springer Handbook of Engineering Statistics*, pages 635–674. Springer, London.
- Grünwald, P. D. (2007). *The Minimum Description Length Principle*. MIT Press.
- Hess, S., Piatkowski, N., and Morik, K. (2014). Shrimp: Descriptive patterns in a tree. In *Proceedings of 16th Workshops on Learning, Knowledge, Adaptation, LWA 2014: Knowledge Discovery, Data Mining and Machine Learning, KDML 2014, Information Retrieval, IR 2014 and Knowledge Management, FGWM 2014*, pages 181–192.
- Holland, J. H. (1975). *Adaptation in Natural and Artificial Systems*. MIT Press.
- Jones, T. (1995). Crossover, macromutation, and population-based search. In *Proceedings of 6th International Conference on Genetic Algorithms*, pages 73–80.
- Kramer, O. (2017). *Genetic Algorithm Essentials*. Springer.
- Lam, H. T., Morchen, F., Fradkin, D., and Calders, T. (2014). Mining compressing sequential patterns. *Statistical Analysis and Data Mining*, 7(1):34–52.
- Li, M. and Vitányi, P. (2019). *An introduction to Kolmogorov complexity and its applications*, volume 4. Springer.
- Liang, P. J., Wang, A., Akoglu, L., and Faloutsos, C. (2021). Pattern recognition and anomaly detection in bookkeeping data. *Asian Bureau of Finance and Economic Research Working Papers*.
- Liu, J., Fournier-Viger, P., Zhou, M., He, G., and Nouioua, M. (2022a). Cspm: Discovering compressing stars in attributed graphs. *Information Sciences*, 611:126–158.
- Liu, J., Ye, Z., Yang, X., Wang, X., Shen, L., and Jiang, X. (2022b). Efficient strategies for incremental mining of frequent closed itemsets over data streams. *Expert Systems with Applications*, 191:116220.
- Luna, J. M., Fournier-Viger, P., and Ventura, S. (2019). Frequent itemset mining: A 25 years review. *WIREs Data Mining and Knowledge Discovery*, 9(6):e1329.
- Makhalova, T., Kuznetsov, S. O., and Napoli, A. (2022). Mint: Mdl-based approach for mining interesting numerical pattern sets. *Data Mining and Knowledge Discovery*, 36(1):108–145.

- Mantuan, A. and Fernandes, L. (2018). Spatial contextualization for closed itemset mining. In *Proceedings of International Conference on Data Mining (ICDM 2018)*, pages 1176–1181.
- Nawaz, M. S., Fournier-Viger, P., Yun, U., Wu, Y., and Song, W. (2021a). Mining high utility itemsets with hill climbing and simulated annealing. *ACM Transactions on Management Information System*, 13(1).
- Nawaz, M. S., Nawaz, M. Z., Hasan, O., Fournier-Viger, P., and Sun, M. (2021b). An evolutionary/heuristic-based proof searching framework for interactive theorem prover. *Applied Soft Computing*, 104:107200.
- Nawaz, M. S., Nawaz, M. Z., Hasan, O., Fournier-Viger, P., and Sun, M. (2021c). Proof searching and prediction in HOL4 with evolutionary/heuristic and deep learning techniques. *Applied Intelligence*, 51:1580–1601.
- Nawaz, M. Z., Hasan, O., Nawaz, M. S., Fournier-Viger, P., and Sun, M. (2020). Proof searching in HOL4 with genetic algorithm. In *Proceedings of the 35th Annual ACM Symposium on Applied Computing (SAC'20)*, page 513–520.
- Nawaz, M. Z., Nawaz, M. S., Fournier-Viger, P., and Selmaoui-Folcher, N. (2024). A genetic algorithm for efficient descriptive pattern mining. In *6th International Workshop on Utility-Driven Mining and Learning (UDML) @ PAKDD*.
- Ni, L., Luo, W., Lu, N., and Zhu, W. (2020). Mining the local dependency itemset in a products network. *ACM Transactions on Management Information Systems*, 11(1):1–31.
- Nikolaev, A. G. and Jacobson, S. H. (2010). Simulated annealing. In Gendreau, M. and Potvin, J.-Y., editors, *Handbook of Metaheuristics*, pages 1–39. Springer.
- Pasquier, N., Bastide, Y., Taouil, R., and Lakhal, L. (1999). Discovering frequent closed itemsets for association rules. In *Proceedings of 7th International Conference on Database Theory (ICDT'99)*, pages 398–416.
- Proença, H. M. and van Leeuwen, M. (2020). Interpretable multiclass classification by mdl-based rule lists. *Information Sciences*, 512:1372–1393.
- Sampson, O. and Berthold, M. R. (2014). Widened krimp: Better performance through diverse parallelism. In *Proceedings of 13th International Symposium on Intelligent Data Analysis (IDA 2014)*, volume 8819, pages 241–252. Springer.
- Seno, G. (2023). Representative itemsets mining: A clustering approach. Master's thesis, University of Padua.
- Smets, K. and Vreeken, J. (2012). Slim: Directly mining descriptive patterns. In *Proceedings of 12th SIAM International Conference on Data Mining (SDM 2012)*, pages 236–247.
- Tatti, N. and Vreeken, J. (2008). Finding good itemsets by packing data. In *Proceedings of 8th International Conference on Data Mining (ICDM 2008)*, pages 588–597.
- Vanetik, N. and Litvak, M. (2018). Drim: Mdl-based approach for fast diverse summarization. In *Proceedings of 17th International Conference on Web Intelligence (WI 2018)*, pages 660–663.
- Vreeken, J., Van Leeuwen, M., and Siebes, A. (2011). Krimp: mining itemsets that compress. *Data Mining and Knowledge Discovery*, 23(1):169–214.

- Wang, D., Tan, D., and Liu, L. (2018). Particle swarm optimization algorithm: an overview. *Soft Computing*, 22:387–408.
- Witteveen, J., Duivesteijn, W., Knobbe, A., and Grünwald, P. (2014). Realkrimp - finding hyperintervals that compress with mdl for real-valued data. In *Proceedings of 13th International Symposium on Intelligent Data Analysis (IDA 2014)*, volume 8819, pages 321–332. Springer.
- Yan, Y., Cao, L., Madden, S., and Rundensteiner, E. A. (2018). Swift: Mining representative patterns from large event streams. *Proceedings of the VLDB Endowment*, 12(3):265–277.
- Yang, X.-S. and He, X. (2013). Bat algorithm: literature review and applications. *International Journal of Bio-Inspired Computation*, 5(3):141–149.
- Yu, X. and Gen, M. (2010). *Introduction to Evolutionary Algorithms*. Springer.
- Zhang, D., Zhang, Y., Niu, Q., and Qiu, X. (2019). Mining concise patterns on graph-connected itemsets. *Neurocomputing*, 336:27–35.