

# A Genetic Algorithm for Efficient Descriptive Pattern Mining

M. Zohaib Nawaz<sup>1,2</sup>[0000–1111–2222–3333], M. Saqib Nawaz<sup>1</sup>[1111–2222–3333–4444],  
Philippe Fournier-Viger<sup>1</sup>[2222–3333–4444–5555], and Nazha  
Selmaoui-Folcher<sup>3</sup>[0000–0003–1667–3819]

<sup>1</sup> Shenzhen University, Shenzhen, China

<sup>2</sup> University of Sargodha, Sargodha, Pakistan

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

**Abstract.** Traditional pattern mining algorithms can find too many patterns in large datasets, including many redundant patterns. Minimum Description Length (MDL)-based methods address this issue through data compression to capture small sets of patterns that capture significant information. However, MDL-based pattern mining techniques, are computationally expensive. As a solution, this paper investigates the efficacy of integrating genetic algorithms (GAs) with MDL-based methods for improved pattern compression in a novel approach called GA4PC (GA for MDL-based pattern compression). By relying on GAs, GA4PC offers a dynamic and adaptable framework for compression tasks. Through experiments, this study reveals that using a GA for MDL-based pattern compression surpasses previous methods in compression ratios, achieves reduced computational time, and produces compressed representations that capture more representative itemsets within the data.

**Keywords:** Genetic Algorithm · Data Compression · MDL · Itemsets.

## 1 Introduction

Frequent itemset mining (FIM) [1], introduced in the early 1990s, is a data mining task aimed at identifying sets of items —such as values, events or symbols— that appear together frequently within a database. Initially, FIM was employed for market basket analysis in transaction databases to uncover co-occurring items, a process initially called large itemset mining [2].

In a transaction database, which consists of a collection of transactions where each transaction is a set of co-occurring items, the goal is to find itemsets that appear frequently. For instance, an itemset found in a customer transaction database might be  $\{bread, butter, cheese\}$ . An itemset is deemed frequent if it occurs often within a database, a frequency quantified by the *support* measure—the count of transactions containing the itemset. For example, if the itemset  $\{bread, butter, cheese\}$  occurs in 10 transactions, its *support* is 10. The process

of FIM aims at finding only those itemsets with a support that meet or surpass 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. The task of FIM is to find all those frequent itemsets. FIM is used in data mining to discover not only interesting patterns in databases, such as association rules, correlations, sequential patterns but also for tasks such as clustering and classification.

A major challenge in FIM and pattern mining in general is the phenomenon of *pattern explosion* [5]. This occurs when a low minimum support ( $min\_sup$ ) threshold is set, which results 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 [6], which entails that each distinct itemset that fulfills the support constraint is included in the result set independently of other itemsets that have already been selected. Because of this, redundant sets of patterns are generally presented to the user, many of which basically describe the same area of the database. To address this issue, many solutions have been proposed such as to mine closed frequent itemsets [7], free-sets [8], non-derivable itemsets [9], and imposing constraints [10]. While these approaches mitigate the issue to some extent, they are not without drawbacks, and redundancy remains a concern.

An emerging approach to tackle the redundancy problem is to search for the best compact set of patterns that succinctly describe the data in a database. The principle of minimum description length (MDL) [11] can be used to discover such compact and compressed patterns. Algorithms such as KRIMP [12], SLIM [13] and CSPM [14] have demonstrated the effectiveness of using the MDL, particularly in classification tasks. However, their application faces several challenges such as long computational time, the inability to guarantee the identification of an optimal pattern set, and limited suitability for multi-class classification (or distinguishing between classes). The search space for finding all patterns (itemsets) that result in optimal compression is prohibitively large. Given the combinatorial explosion of possible combinations, exhaustively evaluating every pattern's contribution to compression is often computationally infeasible, especially as the number of patterns increases. To address this challenge, a heuristic approach is proposed in this paper, leveraging a Genetic Algorithm (GA) [15] for pattern selection. A GA provides a scalable and efficient method to explore a vast solution space by iteratively evolving a population of patterns through crossover and mutation operations. This allows for a more manageable exploration of the space of potential compressed frequent itemsets, offering a practical compromise between computational complexity and solution quality.

The proposed approach, called GA4PC (GA for MDL-based pattern compression), serves as a heuristic to direct the search towards promising patterns that contribute to improved compression ratios without the need to exhaustively evaluate all possible combinations. This approach is especially beneficial in situations where the sheer size of the pattern space makes exhaustive exploration impractical. Moreover, the approach is not reliant on various costly functions

used in traditional MDL-based pattern compression, and it provides an adaptive and efficient framework to find a set of frequent itemsets that offers a superior lossless compression of the database.

The next sections present related work, preliminaries, the GA4PC approach, experimental results, and the conclusion.

## 2 Related Work

Using the MDL for frequent itemset mining is based on the argument that the best set of itemsets compresses the overall size of the database best. KRIMP [12], the first MDL-based algorithm for pattern-based compression, follows two steps: (1) mining frequent itemsets, and (2) selecting a subset of the mined frequent patterns that minimizes the overall description length for improved compression, by exploring combinations in a static order. Because KRIMP requires that all frequently occurring itemsets be generated, mining itemsets is costly. Moreover, KRIMP achieves poor compression results when the frequency threshold is increased. Besides, KRIMP sometimes rejects itemsets that it may have used later since it only evaluates them once and in a fixed order. SLIM [13], an improved version of KRIMP, overcomes these problems. SLIM discovers optimal itemsets gradually and it is not dependent on generating all frequent itemsets in advance. However, the encoding scheme used in both algorithms has various limitations. A decision tree in combination with a refined version of MDL was used in the PACK algorithm [16] for pattern-based compression.

DIFFNORM [17], an extension of SLIM, used a better encoding and can be used to compare datasets in terms of itemsets. COMPRESX [18], also based on the MDL, was proposed to overcome problems with scalability. This algorithm works on categorical data and is more resistant to the pattern explosion problem. KRIMP was extended as REALKRIMP [19] to process real-valued data. REALKRIMP requires data discretization, relies on various heuristics and cannot jointly evaluate the set of generated hyper-rectangles that represent patterns. MINT [20], also based on MDL, was used to find useful patterns in numerical datasets. MINT generates a small set of non-redundant informative patterns. The SHRIMP algorithm [21] builds a FP-tree-like data structure to examine the schema selection impact on a database. This approach allows the fast computation of the quality of the mined itemsets. RSCO [22] was used as a heuristic for ranking itemsets discovered by KRIMP. Another study [23] proposed two extensions of KRIMP, called SeqKRIMP and GoKRIMP, to efficiently mine compressing sequential patterns.

In mining compressing patterns, the focus till 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, 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 as-

sumptions about the data or the model. This is the first study that proposes to use an evolutionary based technique for the efficient mining of compressed patterns. More precisely, a GA is combined with the existing MDL-based approach to reduce the number of resulting itemsets.

### 3 Preliminaries

#### 3.1 Notation

Transactional databases are considered in this paper. Let  $I$  represents a set of items. A transaction  $t \in P(I)$  is a set of items (an itemset). A database, denoted as  $D$ , over items in  $I$  is a list of transactions. A transaction  $t \in D$  supports an itemset  $X \subseteq I$ , iff  $X \subseteq t$ . In  $D$ , the support of  $X$  is computed as the total count of transactions in  $D$  that contain  $X$ .

The model  $M$  that minimizes the compression size among a set of models is considered the best model:

#### 3.2 MDL-based Compression

The MDL is defined as follows: Given a set of models, the model  $M$  that best minimizes the compression size is considered the best model:

$$L(D, M) = L(D|M) + L(M)$$

whereas  $L(D|M)$  represents the compression size, in bits, of the database, with the assumption that  $M$  is used in the encoding process, and  $L(M)$  represents  $M$ 's description size in bits. For FIM with MDL, a *code table* is used for compression. A code table, denoted as  $CT$ , works as a dictionary that contains two columns: the left column contains itemsets and the right column provides a code for each itemset. The  $CT$  typically comprises non-zero usage itemsets, ordered in descending order of cardinality, then by decreasing support, and then in ascending lexicographical order. This defines a total order called the *Standard Cover Order*. In the right column, the lengths of the codes is important.

A code table  $CT \subseteq \{(X, code(X)) \mid X \in CS\}$ , represents a compressing model. The notation  $code(X)$  denotes a code assigned to an itemset  $X \in CT$ . The coding set ( $CS$ ) represents the set of all itemsets  $X$ . A *cover* is defined to encode each transaction  $t$  from  $D$  over  $I$  with a  $CT$ . The *cover function*,  $cover(t)$ , denotes the pattern set from the  $CS$  that, when applied, encodes the transaction  $t$ . For an itemset  $X \in CT$ , its *usage*( $X$ ) is the count of transactions  $t \in D$  having  $X$  in their cover.

$L(D, CT)$  denotes the overall compressed size, in bits, of the encoded database  $D$  and the  $CT$ , and is defined as:

$$L(CT, D) = L(CT|D) + L(D|CT)$$

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

$$\begin{aligned}
L(CT|D) &= \sum_{\substack{X \in CT \\ usage(X) \neq 0}} L(X|ST) + L(X|CT) \\
L(D|CT) &= \sum_{t \in D} L(t|CT) \\
L(t|CT) &= \sum_{X \in cover(t)} L(X|CT)
\end{aligned}$$

The size of the right column of a  $CT$  is the summation of all the different code lengths. For the size of the left column of a  $CT$ , the single items make up the most basic valid code table, that is called *Standard code Table* (ST). The problem of finding the minimal coding set is defined as follows:

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

In our GA-based pattern compression approach, the overall size of the compression is determined by summing the size of the  $CT$  and the size of  $D$  after deleting itemsets  $X \in CT$  from  $D$ . Formally it is written as:

$$\begin{aligned}
L(CT) &= \sum_{X \in CT} L(X) \\
L(D) &= \sum_{\substack{X \in D \\ X \notin CT}} L(X)
\end{aligned}$$

where  $L(CT)$  and  $L(D)$  denote the size of the  $CT$  and that of  $D$  respectively. The total compression size is:

$$L(CT, D) = L(CT) + L(D)$$

In our approach, we did not use the  $ST$  that is used in previous approaches, as the GA selects the itemsets randomly from the database and then apply crossover and mutation operators on the randomly selected itemsets. If the compression improves, then the evolved itemsets are put in the  $CT$ , if not present previously. Note that  $L(CT|D)$  is used as fitness function. The single point crossover and standard mutation operators are used to evolve the randomly selected itemsets. More details about the proposed GA for pattern compression are presented next.

## 4 Proposed GA-based Compression

The flowchart for the proposed GA-based approach for pattern compression, called GA4PC, is shown in Figure 1.

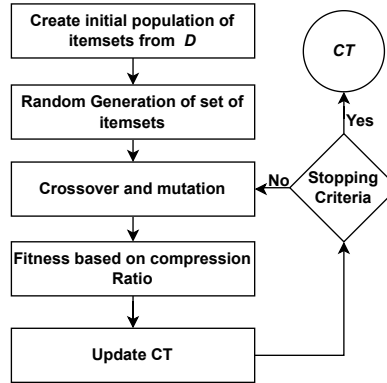


Fig. 1: Flowchart of GA4PC for GA based pattern compression

The GA for pattern compression works by first generating an initial population of itemsets, followed by a random generation of itemsets from the population, applying the crossover and mutation operators to the generated itemsets, and computing the fitness (compression) of evolved itemsets. After the stopping criterion is met, that is, the required number of generations are completed, the GA stores those itemsets in the  $CT$  that compresses the database best. More details are provided next.

Algorithm 1 provides the pseudo-code of GA4PC: GA for MDL-based approach to provide heuristic solutions to the problem of finding the minimal coding set from data. GA4PC first initializes a population with unique itemsets from the dataset (line 1), finds the length of the longest itemset(s) in the population (line 2), generates random itemsets (line 3), and then do a loop drawing pairs of distinct itemsets (called parents) randomly and without repetition (line 4). For each pair, the algorithm applies crossover (6) and mutation operators (lines 7, 8), resulting in evolved itemsets, and calculates the updated code table  $CT_c$  with those itemsets (line 9). Then, GA4PC calculates the total compressed size of  $CT_c$  (line 10). If compression with  $CT_c$  is more than with  $CT$  (line 10), the code table  $CT$  is replaced by  $CT_c$  (line 11). For each generated pair of itemsets, the process (lines 5-11) is repeated for multiple generations to refine the results. Similar to the  $ST$  in traditional MDL-based compression approaches, a population provides an optimal encoding of the database.

Algorithm 2 provides the pseudo-code for the crossover operation applied to a unique pair of random itemsets  $P_1$  and  $P_2$ . Single point crossover [24] is used where 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. The symbol  $\parallel$  represents the concatenation of sub-itemsets. A simple example is provided next to explain the crossover operators. Let  $P_1$  and  $P_2$  be:

$$P_1 = \text{apple, bread, banana, cake}$$

**Algorithm 1** GA for Compression-based FIM

---

**Input:** Dataset  $\mathcal{D}$   
**Output:** Compressed dataset in the form of  $CT$

- 1:  $population \leftarrow$  itemsets from  $\mathcal{D}$
- 2:  $MIL \leftarrow$  length of the longest itemset from  $population$
- 3:  $CT \leftarrow$  Generate random unique itemsets (length  $[2, MIL]$ )
- 4: **for** each pair of itemsets  $P_1, P_2$  drawn randomly and without repetition from  $CT$   
     **do**
- 5:     **repeat**
- 6:          $C_1, C_2 \leftarrow$  Crossover( $P_1, P_2$ )
- 7:          $C'_1 \leftarrow$  Mutation( $C_1, population$ )
- 8:          $C'_2 \leftarrow$  Mutation( $C_2, population$ )
- 9:          $CT_c \leftarrow (CT \oplus C'_1, C'_2)$
- 10:        **if**  $L(\mathcal{D}, CT_c) < L(\mathcal{D}, CT)$  **then**
- 11:             $CT \leftarrow CT_c$
- 12:        **end if**
- 13:     **until** (max. number of generations is attained)
- 14: **end for**
- 15: **Return**  $CT$

---

**Algorithm 2** Crossover

---

**Input:**  $X_1, X_2$ : Two parent itemsets,  
**Output:** Two child itemsets

- 1: **procedure** CROSSOVER( $X_1, X_2$ )
- 2:      $s \leftarrow \min(\text{len}(X_1), \text{len}(X_2))$
- 3:      $cop \leftarrow \text{randomInt}(1, s)$   $\triangleright (1 \leq cop \leq size)$
- 4:      $X_1 \leftarrow X_1[1, cop] \parallel X_2[cp + 1, \text{len}(X_2)]$
- 5:      $X_2 \leftarrow X_2[1, cop] \parallel X_1[cp + 1, \text{len}(X_1)]$
- 6: **end procedure**
- 7: **Return**  $X_1$  and  $X_2$

---

$$P_2 = \text{apple, butter, diaper, milk, orange}$$

Let  $n$  represents the length of the largest itemset. A random  $cp$  ( $(1 \leq cp \leq n)$ ) is randomly selected. For  $cp = 3$ , single point crossover generates the two child sequences as:

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

Single point mutation [25] (Algorithm 3) is then performed on  $C_1$  and  $C_2$  using the current population as a reference. The mutated itemsets are denoted as  $C'_1$  and  $C'_2$ . The mutation introduces diversity by replacing the selected random itemset with another itemset from the population. This helps prevent being trapped in local optima by introducing random information into the search process. For example, a mutation of  $C_1$  and  $C_2$  is:

$$C'_1 = \text{apple, bread, coffee, milk, orange}$$

$$C'_2 = \text{candy, butter, diaper, cake}$$

---

**Algorithm 3** Mutation

---

**Input:**  $C$ : An itemset and *population*: set of all itemsets

**Output:** A mutated itemset

```

1: procedure MUTATION( $C$ )
2:    $ind \leftarrow \text{randomInt}(1, \text{length}(C))$ 
3:    $alter \leftarrow \text{randomSelect}(\text{population}, 1)$  ▷ (singleton itemset)
4:    $C[ind] \leftarrow alter$  ▷ ( $P_1[ind] \neq alter$ )
5:   Return  $C$ 
6: end procedure

```

---

The entire process is repeated for a fixed number of iterations, or generations, for each pair in  $CT$ . At the end of the iterations, for each pair in  $CT$ , the set of itemsets  $CT$  is returned, representing potentially compressing frequent itemsets obtained through the GA. Note that GABPM does not use the *Standard Candidate order*, the *Standard Cover Order*, nor the *cover* and *usage* functions that play important roles in the total compression size in traditional MDL-based approaches.

## 5 Results

*GA4PC* is implemented in *Java*<sup>4</sup>. Experiments were performed on a fifth generation Core i5 processor with 8 GB of RAM. Next we provide the details for the datasets used in the experiments, followed by the compression ratios and finally, the running times respectively.

### 5.1 Datasets

While previous studies utilized up to 27 datasets, we intentionally limited our scope to five carefully chosen datasets for the sake of brevity and clarity, focusing on demonstrating the potential of our proposed GA-based compression technique. From the FIMI repository<sup>5</sup>, we used the two datasets (Accidents and Chess). The Heart and Wine datasets were obtained from the SPMF repository<sup>6</sup>. Each dataset contains transactions involving sets of items. These datasets and their statistical details are presented in Table 1. For each dataset,  $|D|$  denotes the number of transactions,  $|I|$  is the count of unique items,  $|A|$  is the average count of items in each transaction, and  $\text{Density} = \frac{|A|}{|I|} \times 100$ .

<sup>4</sup> The code can be found at: [github.com/MuhammadzohaibNawaz/GABFIM](https://github.com/MuhammadzohaibNawaz/GABFIM)

<sup>5</sup> <http://fimi.uantwerpen.be/data/>

<sup>6</sup> <http://philippe-fournier-viger.com/spmf/index.php?link=datasets.php>

Table 1: Description of datasets and statistics

Dataset	$ D $	$ I $	$ A $	Density
Accidents	340,183	468	33.8	7.22
Chess (k-k)	3,196	75	37	49.33
Connect-4	67,557	129	43	33.33
Pen Digits	10,992	86	17	19.80
Pumsbstar	49,046	2,088	50	3.50

## 5.2 Compression

Firstly, we investigate how well GA4PC describes a dataset by examining the key quality measure, that is the compression ratio. The algorithm is run for 20 iterations and initially  $CT$  contains 20 random unique itemsets. Lower compression ratio values indicate superior performance. The relative compression ratio, denoted as  $L\%$ , of size  $D$  is calculated as:

$$\frac{L(D, CT)}{L(D, Pop)} \%$$

wherever  $D$  is clear from the context and  $Pop$  represents the population.

Table 2: Comparison of compression ratios

Dataset	Compression ratio (L%)				
	GA4PC	KRIMP	Avg. Gain	SLIM	Avg. Gain
Accidents	<b>11.5</b>	55.1	43.6	31.1	19.6
Chess (k-k)	<b>10.2</b>	30.0	19.8	14.7	4.5
Connect-4	<b>8.7</b>	42.9	34.2	12.3	3.6
Pen Digits	<b>9.6</b>	42.4	32.7	39.4	29.7
Pumsbstar	44.8	56.0	11.2	<b>25.1</b>	-19.7

The proposed GA4PC algorithm demonstrates competitive compression ratios across various datasets, indicating its effectiveness in reducing dataset size. It consistently achieves superior compression ratios compared to KRIMP and SLIM, with the exception of the *Pumsbstar* dataset, where it exhibited a slightly lower performance than SLIM. Its performance on this dataset deviates from its favorable results on other datasets, prompting an examination of its characteristics as presented in Table 1. This dataset comprises 49,046 transactions, 2,088 unique itemsets, and an average of 50 items in each transaction. With a low density of 3.50%, indicating a sparse distribution of items, GA4PC, being a random-based algorithm, faces challenges in efficiently discovering meaningful patterns in such sparse datasets. The algorithm's strategy of seeking optimal candidate patterns may encounter difficulties in the presence of a large number of unique itemsets,

potentially leading to suboptimal compression. Additionally, the high average count of items in each transaction could contribute to increased complexity in pattern mining.

Overall, GA4PC achieved better performance with average gain in compression ratio of approximately 28% compared to KRIMP and approximately 7% compared to SLIM. The exceptional performance of our proposed methodology can be observed in case of the *Accidents* dataset, where it outperforms KRIMP by 43% and SLIM by 19%, in terms of compression ratio gain. This emphasizes the potential of GA4PC in minimizing dataset size efficiently.

### 5.3 Running time and Convergence

We also examined the runtimes as well as the convergence. GA4PC demonstrates superior runtime efficiency across multiple datasets (see results in Table 3). On average, GA4PC exhibits approximately a 58 times speed improvement over KRIMP across the five datasets. In comparison to SLIM, GA4PC demonstrates a 117 times speed enhancement approximately. Notably, GA4PC showcases exceptional performance on the *Pen Digits* dataset, where it is approximately 590 times faster than KRIMP. Note that the runtime of the proposed algorithm on five datasets is low as it is run for 20 iterations. It would be interesting to see the runtime for a larger number of iterations.

Table 3: Runtime comparison

Dataset	GA4PC (sec.)	KRIMP (sec.)	SLIM (sec.)
Accidents	<b>1280.5</b>	58,000	96,000
Chess (k-k)	44.67	14,100	<b>22</b>
Connect-4	<b>121.4</b>	11,300	9,900
Pen Digits	<b>16.6</b>	9,800	2,900
Pumsbstar	<b>280.7</b>	8,900	97,000

To examine the convergence behavior of GA4PC, we analyze the algorithm’s performance for five dataset across 20 generations in Figure 2. The x-axis represents the generations (1 to 20), while the y-axis illustrates the compression achieved in each generation for five datasets. The figure provides insights into the algorithm’s convergence and compression behavior over successive generations, offering a dynamic view of its performance across various datasets. The data illustrates fluctuations in compression values over generations the *Pumsbstar* dataset and a constant varying trends observed in the *Accidents* dataset, highlighting the dataset-specific nature of the algorithm’s performance. The lower compression value in the initial generation for the *Pumsbstar* dataset can be attributed to its sparse nature, as discussed in Section 5.2. The sparse distribution of items in this dataset poses a challenge for the GA, making it more challenging to identify optimal compression patterns in the early generations.

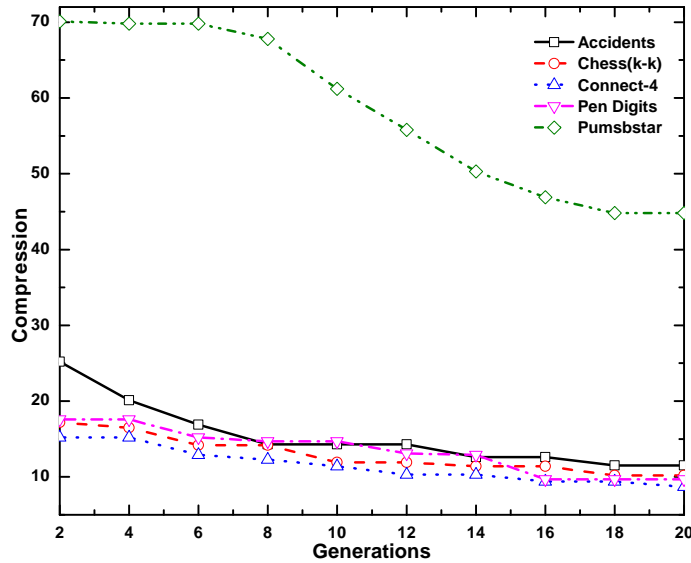


Fig. 2: Comparison of compression vs generations

## 6 Conclusion

GA4PC demonstrated promising results in terms of compression ratio and run-time efficiency compared to traditional methods like KRIMP and SLIM. The approach leverages a genetic algorithm for pattern generation, offering a novel perspective on itemset-based compression. However, it is important to acknowledge certain limitations. GA4PC's focuses on subsets rather than complete candidate patterns within transactions leading to potential overlooks. The algorithm's randomness introduces variability, and some candidates added to the final solution may have limited impact on compression. Additionally, the comparison of candidate solutions with the database multiple times contributes to computational overhead. Additionally, GA4PC faces challenges when dealing with sparse datasets, indicating the necessity for optimization and further exploration to enhance its effectiveness in managing similar data scenarios. Despite these limitations, GA4PC presents an innovative direction for itemset-based compression with encouraging outcomes. Further refinement and exploration of its parameters could enhance its effectiveness and address some of the identified limitations.

In the future, we plan to improve the proposed algorithm and compare it with other recent algorithms for MDL-based pattern compression.

## References

1. Luna, J.M., Fournier-Viger, P., Ventura, S.: Frequent itemset mining: A 25 years review. *WIREs Data Min. Knowl.* **9**(6), e1329 (2019)
2. Agrawal, R., Imieliński, T., Swami, A. Mining association rules between sets of items in large databases. In: *SIGMOD*, pp. 207-216 (1993)

3. Chee, C-H., Jaafar, J., Aziz, I.B., Hasan, M.H., Yeoh, W.: Algorithms for frequent itemset mining: A literature review. *Artif. Intell. Rev.* **52**(4), 2603-2621 (2019)
4. Thomas M. Cover and Joy A. Thomas, *Elements of Information Theory*, 2nd Edition, Wiley Series in Telecommunications and Signal Processing, Wiley, 2006.
5. Makhalova, T., Kuznetsov, S. ), Napoli, A.: Likely-occurring itemsets for pattern mining. In: *FCA4AI@IJCAI*, pp 39-50 (2021)
6. Belaid, M-B., Lazaar, N.: Frequent itemset mining with multiple minimum supports: A constraint-based approach. *CoRR abs/2109.07844* (2021)
7. Pasquier, N., Bastide, Y., Taouil, R., Lakhal, L.: Discovering frequent closed itemsets for association rules. In: *ICDT*, pp. 398-416 (1999)
8. Boulicaut, J-F., Bykowski, A., Rigotti, C.: Free-sets: A condensed representation of boolean data for the approximation of frequency queries. *Data Min. Knowl. Discov.* **7**(1), 5–22 (2003)
9. Calders, T., Goethals, B.: Non-derivable itemset mining. *Data Min. Knowl. Discov.* **14**(1), 171–206 (2007)
10. Boulicaut, J.-F., De Raedt, L., Mannila, H. (eds.): *Constraint-Based Mining and Inductive Databases*. LNCS (LNAI) (2005)
11. Grünwald, P.: *The Minimum Description Length Principle*. MIT Press, 2007
12. Vreeken, J., Van Leeuwen, M., Siebes, A.: KRIMP: mining itemsets that compress. *Data Min. Knowl. Discov.*, **23**(1), 169-214 (2011).
13. Smets, K., Vreeken, J.: SLIM: Directly mining descriptive patterns. In: *SDM* , pp. 236-247 (2012)
14. Liu, J., Fournier-Viger, P., Zhou, M., He, G., Nouioua, M.: CSPM: Discovering compressing stars in attributed graphs. *Inf. Sci.* **611**. 126-158 (2022)
15. Holland, J.H.: *Adaptation in natural and artificial systems*. MIT Press (1975)
16. Tatti, N., Vreeken, J.: Finding Good Itemsets by Packing Data. In: *ICDM*, pp. 588-597, 2008.
17. Budhathoki, K, Vreeken, J.: The difference and the norm—characterising similarities and differences between databases. In: *ECML/PKDD*, pp 206–223 (2015)
18. Akoglu, L., Tong, H., Vreeken, J., Faloutsos, C.: Fast and reliable anomaly detection in categorical data. In: *CIKM*, pp 415–424 (2012)
19. Witteveen, J., Duivesteijn, W., Knobbe, A., Grünwald, P.: RealKrimp — Finding hyperintervals that compress with MDL for real-valued data. In: *IDA*, pp 368–379 (2014)
20. Makhalova, T., Kuznetsov, S. O., Napoli, A.: Mint: MDL-based approach for mining interesting numerical pattern sets. *Data Min. Knowl. Discov.* **36**(1), 108-145 (2022)
21. Hess, S., Piatkowski, N., Morik, K.: Shrimp: Descriptive patterns in a tree. In: *LWA*, pp. 181-192 (2014)
22. Sampson, O., Berthold, M.R.: Widened KRIMP: Better performance through diverse parallelism. In: *IDA*, pp. 276–285 (2014)
23. Lam, H.T., Morchen, F., Fradkin, D., Calders, T.: Mining compressing sequential patterns. *Stat. Anal. Data Min.* **7**(1): 34-52 (2014)
24. Nawaz, M.S., Nawaz, M.Z., Hasan, O., Fournier-Viger, P., Sun, M.: An evolutionary/heuristic-based proof searching framework for interactive theorem prover. *Applied Soft Computing*, 104, 107200 (2021).
25. Nawaz, M.Z., Hasan, O., Nawaz, M.S., Fournier-Viger, P., Sun, M.: Proof searching in HOL4 with genetic algorithm. In: *SAC*, 513-520 (2020)