MINING FREQUENT CLOSED HIGH-UTILITY SEQUENTIAL PATTERNS BASED ON A GENETIC ALGORITHM

Tran Anh Duy, Le Thi Minh Nguyen

Faculty of Information Technology, HUFLIT

tduyta@huflit.edu.vn, nguyenltm@huflit.edu.vn

Abstract - Mining Frequent Closed High-Utility Sequential Patterns (FCHUSPs) is an important problem in data mining, with many practical applications such as customer behavior analysis, supply chain optimization, and marketing. However, the mining process faces a massive search space and high computational cost, especially when the input thresholds are low or the dataset is large. In this paper, we propose a method for mining FCHUSPs using a Genetic Algorithm (GA), in which each individual is represented as a bit array to optimize memory and genetic operations. To improve efficiency, the fitness evaluation process is performed in parallel using PySpark MapReduce, with a hashtable used to verify the closeness of the patterns. The proposed algorithm, called PFCloHUS_QUANTITY_GA_SS, has been implemented and evaluated on several experimental datasets. The results show that the proposed method significantly reduces the execution time compared to traditional approaches in mining frequent closed high-utility sequential patterns.

Keywords: Sequential pattern mining, high utility, frequent closed, genetic algorithm, parallel computing, PySpark.

I. INTRODUCTION

In the context of rapidly growing data in both volume and variety, data mining has become a core field in computer science, artificial intelligence, and data science. Organizations and enterprises today are not only concerned with storing data but also with extracting hidden knowledge to support strategic decision-making. One important research direction is the mining of sequential patterns [1], which has been applied in various domains such as user behavior prediction [1], DNA sequence analysis [2], web log mining [3] [4], and particularly in e-commerce transaction analysis.

However, relying solely on occurrence frequency to find frequent sequential patterns has not truly met practical needs. Many patterns appear with high frequency but do not bring significant economic value or crucial information to the business. Conversely, some patterns are less frequent but yield high utility in terms of revenue or business efficiency. Therefore, the concept of High-Utility Sequential Patterns (HUSPs) [5] [6] was introduced, allowing for the mining of patterns that satisfy both the frequency factor and reflect the utility value associated with the data. Although HUSPs provide greater value, the mining process often generates a massive number of duplicate or redundant patterns. This leads to difficulties in analysis and may introduce noise for decision-makers. To address this issue, researchers proposed the concept of Frequent Closed High-Utility Sequential Patterns (FCHUSPs) [7] [8]. These patterns not only reduce the result set—thereby lowering the number of patterns that need to be examined—but also preserve the ability to reconstruct the original set of HUSPs. As a result, the analysis becomes easier, more efficient, and more practical. However, the problem of mining FCHUSPs still faces several major challenges.

Exponential expansion of the search space: As the number of elements and the size of the database increase, the number of potential candidates grows rapidly, making traversal and verification difficult.

High Computational Cost: Simultaneously evaluating both the utility and support of patterns requires complex calculations, especially when the input thresholds are set low.

Difficulties in Optimizing and Parallelizing Algorithms: In contrast to frequent itemset mining (FIM), which adheres to the downward closure property [1], the FCHUSP problem lacks this characteristic. This absence significantly complicates search space pruning and hinders its effective application on multi-core platforms.

A surge of research has emerged proposing optimizations for the mining of HUSPs and FCHUSPs, including the use of specialized data structures, pruning strategies, and parallel algorithms on multi-core architectures. However, these approaches still have several limitations:

Utility threshold dependency: The selection of the minimum utility threshold is often subjective and strongly influences the mining results.

Limited scalability on large datasets: Traditional sequential algorithms often struggle to handle databases containing millions of transactions.

Inefficient exploitation of parallel computing power: Some parallel solutions are limited to multithreading and do not fully leverage the data-distribution capabilities of large-scale platforms such as Spark.

To overcome the above limitations, this research proposes a method for mining FCHUSPs based on the Genetic Algorithm (GA). GA is an optimization technique inspired by natural evolution, well-known for its ability to search

for near-optimal solutions in complex spaces. In the proposed model, each individual is represented by a bit array, which helps optimize memory and accelerates crossover and mutation operations. Notably, the fitness evaluation process is implemented in parallel using PySpark MapReduce, leveraging the capability of distributed data processing on multi-core platforms . Additionally, the use of a hashtable efficiently checks the closed-frequent property of patterns, thereby significantly reducing processing costs. The main contributions of this research can be summarized as follows:

- Propose the PFCloHUS_QUANTITY_GA_SS algorithm, which combines a Genetic Algorithm with parallel computing to mine frequent closed high-utility sequential patterns.
- Apply bit array and hashtable structures to optimize memory usage and reduce processing time.
- Conduct experiments on multiple benchmark datasets to demonstrate the effectiveness of the proposed method compared to traditional algorithms.

The rest of the paper is organized as follows: Section II presents the related works. Section III describes the fundamental concepts and definitions. Section IV describes the proposed algorithm. Section V reports the experimental results and analysis. Finally, Section VI provides the conclusion and future research directions.

II. RELATED WORK

Data Mining is one of the important application fields of computer science, focusing on discovering hidden patterns and rules from databases. Among them, Sequential Pattern Mining was first introduced by Agrawal and Srikant (1995) [1], with the goal of finding recurring sequences of events in transactional data. This method quickly became the foundation for many applications in customer behavior analysis [1], web access pattern discovery [3] [4], and biological data analysis [2].

However, traditional sequential pattern mining algorithms tend to rely on the support or frequency of occurrence of patterns. This leads to situations where many discovered patterns appear frequently but do not carry practical significance. To overcome this limitation, subsequent research introduced the concept of utility [5] [6]. This value reflects the importance or economic worth of the patterns.

A. HIGH-UTILITY SEQUENTIAL PATTERN MINING (HUSPS)

In 2010, Ahmed [5] proposed a method for computing the utility value of a sequential pattern and introduced a mining approach based on sequence-weighted utility (SWU). Two algorithms were proposed, UtilityLevel (UL) and UtilitySpan (US), which use two different strategies for pattern generation. A limitation of these algorithms is that they generate too many candidates during the mining process. In 2012, Yin et al. [9] provided a comprehensive set of important definitions for the High-Utility Sequential Pattern Mining (HUSPM) problem and proposed the mining algorithm USpan. In this algorithm, the lexicographic quantitative sequence tree (LQS-tree) structure was introduced to represent the pattern search space, and the mining process is described as traversing the LQS-tree to extract the high-utility patterns. In 2020, Gan [10] proposed the utility-linked list (UL-list) structure combined with the LQS-tree, along with two new pruning strategies called Look-Ahead Removing (LAR) and Irrelevant Item Pruning (IIP) in the HUSP-ULL algorithm, which help overcome the limitations of the USpan approach. In 2023, Zhang proposed the HUSP-SP algorithm [11], which employs the seqPro structure and an upper bound called TRSU to enhance mining efficiency.

B. CLOSED HIGH-UTILITY SEQUENTIAL PATTERNS (FCHUSPS)

To limit the search space, some researchers proposed mining **closed frequent high-utility sequential patterns (FCHUSPs)**. A closed pattern ensures that no superset pattern exists that has the same support, thereby eliminating redundant patterns while retaining all necessary information to reconstruct the original set of HUSPs. In 2019, to mine FCHUSPs, Tin [8] proposed the **FMaxCloHUSM** algorithm, which utilizes a data structure called **SIDUL** to store the utility information of the patterns and three pruning strategies. These strategies include using the Apriori property to mine support, employing a breadth-first/depth-first pruning strategy, and applying a local pruning strategy on non-closed data patterns. This approach enables the finding of FCHUSPs and FMaxHUSPs faster than combining existing methods.

C. GENETIC ALGORITHM–BASED APPROACHES (GA)

To enhance mining efficiency, researchers have proposed two solution approaches: (1) applying evolutionary methods, with the genetic algorithm being the most prominent, and (2) applying parallel processing techniques on multi-core systems, GPUs, or Hadoop during high-utility pattern mining.

At first, many combinatorial optimization problems, including High-Utility Itemset Mining (HUIs)[12] [13], have successfully employed the **Genetic Algorithm (GA),** which finds near-optimal solutions in large search spaces

without needing to scan the entire dataset. While several studies have experimented with GA for HUI mining, there are not many studies applying it to the **FCHUSPs** problem, especially in big data environments.

Secondly, parallel processing techniques have been applied to the high-utility itemset mining problem in several studies, such as the 2018 work by Nguyen et al. [14] and the 2023 work by Kimura [15]. Furthermore, Fang [16] proposed a method that combines GPU-based parallel processing with evolutionary techniques to improve high-utility pattern mining efficiency. However, these studies focus only on high-utility itemset data and neglect the challenges associated with sequential data. In 2022, Nguyen et al. proposed the P-FCloHUS algorithm [7] for mining frequent closed high-utility sequences using multi-core processors. The authors demonstrated that P-FCloHUS outperforms the state-of-the-art FMaxCloHUSM algorithm [8]. However, a notable limitation of this algorithm is that it is exclusively implemented on multi-core processor systems, failing to leverage the advantages of existing distributed architectures.

D. RESEARCH GAP

Based on the findings, this paper addresses the following research gaps:

- Traditional sequential pattern mining algorithms are difficult to scale for Big Data.
- Genetic Algorithms (GA) have only been utilized at the level of High-Utility Itemset (HUI) mining and have not been fully applied to the FCHUSPs problem.
- Current parallel methods primarily focus on multi-core models and have not yet fully leveraged distributed architectures like MapReduce.

Therefore, this study proposes the **PFCloHUS_QUANTITY_GA_SS** algorithm, which combines **GA + PySpark MapReduce** to both reduce processing time and ensure accurate mining of closed frequent high-utility sequential patterns.

III. FUNDAMENTAL CONCEPTS AND DEFINITIONS

A. SEQUENTIAL TRANSACTION DATABASE

Given a set of distinct items $I = \{i_1, i_2, \ldots, i_n\}$, an itemset is a non-empty unordered set of items. A sequential sequence, denoted as $S = (e_1, e_2, \ldots, e_m)$, is an ordered list of itemsets, where each e_i $(1 \le i \le m)$ is an itemset.

A sequential database (SDB) is a list of sequential sequences, represented as $SDB = (s_1, s_2, \ldots, s|SDB|)$, where |SDB| is the number of sequential sequences in the SDB, and s_i $(1 \le i \le |SDB|)$ is the i-th sequential sequence in the SDB.

Table 1. Non-quantitative sequence database (SDB)

SID	Sequence
1	({A},{A,B},{A,B,C})
2	({A,B},{A,C},{B,C})
3	({C},{A,C},{A,B})
4	({D},{A,B,D})

Example: Table 1 describes a sequence database SDB that contains 4 sequences, |SDB| = 4, and 4 distinct items I = {A, B, C, D}

Definition 1: The support of a sequence S is defined as the number of sequences in the sequential database SDB that contain S, denoted as sup(S). From Table 1, we have sup({A, B}) = 4.

Definition 2: A sequence S is considered a frequent sequence if and only if its support is greater than or equal to the minimum support threshold, i.e., sup(S) ≥ minSup, which is a threshold set by the user. For example, given minSup = 3, the sequence {A, B} is a frequent sequence.

A quantitative sequential database is a sequential database where each item instance within a sequence is associated with its corresponding local quantity (internal utility). Furthermore, each item type is assigned an external profit value (utility value).

Table 2. Quantitative sequential database SDB D

SID	Sequence
S_1	({A:2,C:1},{A:1},{C:2,D:4})
S_2	({B:2},{A:2,B:1,C:3},{B:3,D:4},{A:1,B:2})
S_3	({A:3,D:1},{B:2,C:4,D:1},{A:1,B:1,C:2}),{A:2,D:1}
S_4	({A:1,C:2,D:3},{A:2,B:1,C:1},{A:3,B:3})

Table 3. Utility values.

Item	a	b	c	d
Utility	3	5	1	4

Definition 3: The utility of an item is defined as the product of its internal utility and external profit value, denoted as*:*

$$u(i) = iu(i) \times eu(i) \quad (1)$$

where *iu(i)* and *eu(i)* represent the internal utility and external profit, respectively.

Definition 4: The utility of an itemset *X*, denoted as *u(X)*, is the sum of the utilities of all items $i \in X$:

$$u(X) = \sum_{i \in X} u(i) \quad (2)$$

As an illustration, consider the itemset *X = {A:2, C:1}*. The external profits of items *A* and *C* are 3 and 1 respectively, the utility of *X* is computed as *u(X) = 7*.

Definition 5: The utility of a sequence $S = (s_1, s_2, ..., s_k)$ is denoted as *u(S)* and computed as.

$$u(S) = \sum_{j=1}^{k} u(s_j) \quad (3)$$

Definition 6: Let *O(sub, S)* be the set of all occurrences of subsequence *sub* in sequence *S*. The utility of *sub* in *S* is:

$$u(sub, S) = max\{u(sub, S)_i | i \in O(sub, S)\} \quad (4)$$

For instance, consider the subsequence *sub = {A, B}* in sequence S_2 from Table 2. This subsequence appears twice as $sub_1 = \{A:2, B:1\}$ and $sub_2 = \{A:1, B:2\}$. Given the utilities $u(sub_1) = 2 \times 3 + 1 \times 5 = 11$ and $u(sub_2) = 1 \times 3 + 2 \times 5 = 13$, the utility of the subsequence in S_2 is determined as $u(sub, S_2) = max(11, 13) = 13$.

Definition 7: The utility of a subsequence *sub* in the quantitative sequence database $\mathcal{D}$ is the utility value of the subsequence *sub*. Formula (2) is as follows:

$$U(sub, \mathcal{D}) = \sum_{i \in I} U(sub, S_i) \quad (5)$$

For example, consider the subsequence *sub = {A, B}* from Table 2. $U(\{A,B\}, \mathcal{D}) = U(\{A,B\}, S_2) + U(\{A,B\}, S_3) + U(\{A,B\}, S_4) = 13+8+24 = 45$.

B. GENETIC ALGORITHM

1. CHROMOSOME AND POPULATION REPRESENTATION

Table 4. Population initialization

Chromosome	Populations						
	a	b	c	d	e	f	g
C1	1	0	0	1	1	0	1
C2	0	1	0	1	0	0	1
C3	0	0	0	1	1	0	1
C4	1	0	0	0	0	0	1
C5	1	1	0	1	1	0	1
C6	1	0	1	0	0	1	1
C7	1	0	0	1	0	1	1

2. GENETIC OPERATIONS

Random Selection: Suppose there are 5 individuals with different fitness values: Table 5. Example of Random Selection

Table 5. Random Selection

Individual	Fitness
A	90
B	80
C	70
D	60
E	50

In the random selection method, all individuals have an equal chance of being chosen. Therefore, individual E can still be selected instead of A, even though it has lower fitness.

Multi-point crossover**:**

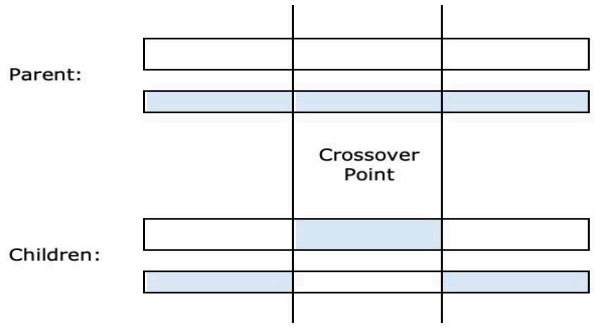

Figure 1.Example of multi-point crossover

Bit-flip mutation:

0	0	1	1	0	1	0	0	1	0	=>	0	0	1	0	0	1	0	0	1	0

Figure 2. Example of bit-flip mutation at position 4

3. PARALLEL MODEL WITH PYSPARK MAPREDUCE

PySpark is a powerful tool for distributed and parallel processing on big data.

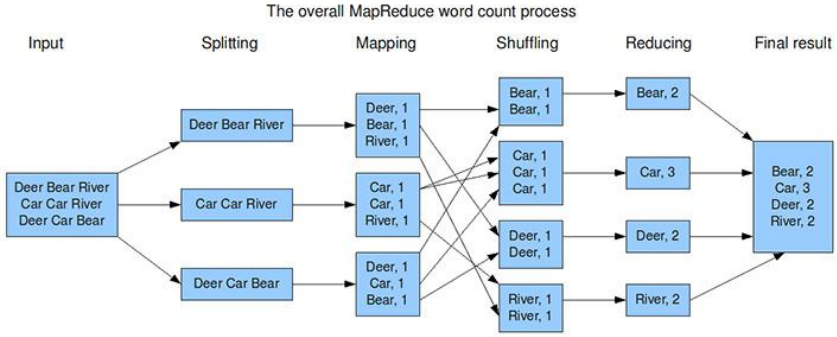

Figure 3. MapReduce execution model in Python

4. CLOSED FREQUENT PATTERNS

A pattern **P** is considered a **closed frequent pattern** if:

P is a frequent pattern, meaning its support is greater than or equal to the minimum support threshold (**min_sup**). There does **not** exist any frequent pattern **Q** such that **P ⊂ Q** and **support(P) = support(Q)**.

Table 6. Example of a transaction table

Giao dịch	Items
T_1	A, B, C
T_2	A, B
T_3	A, B, C
T_4	A, B

Given the minimum support threshold **min_sup = 3**, the pattern **{A, B}** is a closed frequent pattern because there is no larger frequent pattern that has the same support. Meanwhile, the pattern **{A}** is not a closed pattern because it is a subset of the pattern **{A, B}**.

IV. PROPOSED ALGORITHM: PFCLOHUS_QUANTITY_GA

A. PROPOSED ALGORITHM FLOWCHART

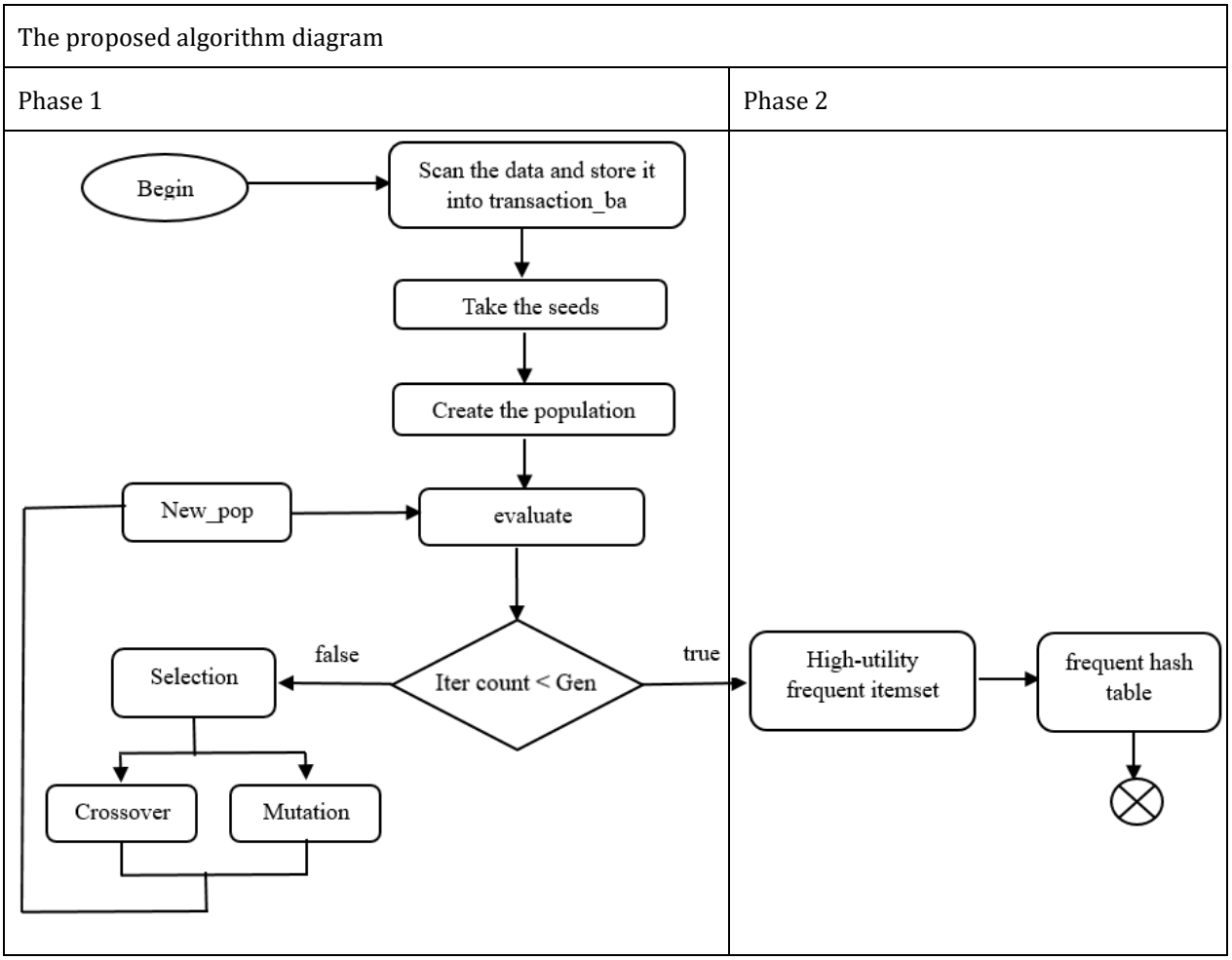

Figure 4. algorithm diagram

Phase 1: Using the Genetic Algorithm to generate Frequent High-Utility Itemsets

Phase 2: Using a hash table to filter out closed frequent itemsets.

B. TRANSACTION STORAGE REPRESENTATION

Itemset as a bit array.

Dictionary with key as the item, the value is the utility of the item.

Given the quantitative sequence database shown in Table 2, the transaction_ba_list is constructed

Table 7 Transaction_ba_list

SID	List (bitarray, dict)
S_1	[('1, 0, 1, 0',{a:6,c:1}),('1, 0, 0, 0',{a:3},('0, 0, 1, 1',{c:2,d:16})]
S_2	[('0, 1, 0, 0', {b:10}),('1, 1, 1, 0', {a:6,b:5,c:3}),('0, 1, 0, 1', {b:15,d:16}),('1, 1, 0, 0', {a:3,b:10})]
S_3	[('1, 0, 0, 1', {a:3,d:4}),('0, 1, 1, 1', {b:10,c:4,d:4}),('1, 1, 1, 0', {a:3,b:5,c:2}),('1, 0, 0, 1', {a:6,d:4})]
S_4	[('1, 0, 1, 1', {a:3,c:2,d:12}),('1, 1, 1, 0', {a:6,b:5,c:1}),('1, 1, 0, 0', {a:9,b:15})]

C. REPRESENTATION OF INDIVIDUALS AND THE POPULATION

The population is a list containing individuals. Each individual stores: a list of bitarrays, where each bitarray represents an itemset, and a tuple with two components — the fitness of the individual and its support_count.

Table 8. Initialize the population

C	Population **(bitarray_list)**	**(fit, sup) (3)**
C_1	['1, 0, 1, 0', '0, 0, 1, 1']	(25,1)
C_2	[‘1, 1, 1, 0’, ‘1, 1, 0, 0’]	(44,1)
C_3	['0, 0, 1, 1']	(40,3)
C_4	[‘1, 1, 0, 0’]	(43,4)

D. FITNESS FUNCTION

Compute the utility of each individual over the entire database, using parallel processing to divide the computation and reduce execution time. At the same time, count the number of occurrences of the individual being evaluated.

E. CALCULATE_FITNESS ALGORITHM

.Input: candidate_pattern, sequence

.Output: Utility value of candidate_pattern in sequence (or 0 if not found)

1. let pattern_len = length(candidate_pattern)

2. let i = 0, j = 0, util = 0

*3. **while** i < length(sequence) and j < pattern_len **do***

4. let current_pattern_item = candidate_pattern[j]

5. let current_seq_item = sequence[i]

*6. **if** current_pattern_item ⊂ current_seq_item **then***

7. let util = util + util_pattern_in_itemset (current_pattern_item, current_seq_item)

8. let j = j + 1

*9. **end if***

10. let i = i + 1

*11. **end while***

*12. **if** j = pattern_len **then***

13. return util

*14. **else***

15. return 0

16. end if

Explanation:

Line 1: Get the length of the candidate pattern.

Line 2: Initialize the values i, j, and util = 0.

Lines 3 to 11: loop until one of the two (string or pattern) runs out.

Line 4: Get the currently considered pattern value.

Line 5: Get the value of the current sequence.

Lines 6 to 9: If the currently considered pattern is a subsequence of the current sequence, then add the util of the current pattern and proceed to the next pattern.

Line 10: proceed to the next subsequence.

Lines 12 to 16: If all candidate patterns have been traversed, return the util value of this pattern; otherwise, return 0

F. THE PFCLOHUS_QUANTITY_GA_SS ALGORITHM

Input: Dataset Sequence, population_size, generations, min_util, min_sup

Output: highUtilityPatterns

1. let population = initialize_population(POPULATION_SIZE);

2. ***for*** *generation = 1 to GENERATIONS* ***do***

3. *sort population by fitness in descending order;*

4. *let nextGeneration = top 50% of population;*

5. ***while*** *size of nextGeneration < POPULATION_SIZE* ***do***

6. *randomly select parent1 and parent2 from population;*

7. *let (child1, child2) = crossover(parent1, parent2);*

8. *add child1, child2 to nextGeneration;*

9. *randomly select child from population;*

10. *let mutatedChild = mutate(child);*

11. *add mutatedChild to nextGeneration;*

12. ***end while***

13. *population = nextGeneration;*

14. ***end for***

15. return highUtilityPatterns;

Explanation*:*
Line 1: Initialize the population.

Lines 2 to 14: Iterate through the GENERATIONS to perform the genetic operations.

Line 3: sort the population in descending order of fitness

Line 4: Take ½ of the population for the next generation.

Lines 5 to 12: The loop performs genetic operations to create new individuals until reaching the required number of POPULATION_SIZE.

Line 13: Update the population for the next iteration

G. PHASE 2: CLOSED FREQUENT PATTERN MINING ALGORITHM

Input: pattern, fitness, support, HashTable

Output: HashTable has been updated and now contains only closed frequent patterns

1. ***if*** *support ∉ HashTable* ***then***

2. *create HashTable[support] ← ∅*

3. ***end if***

4. ***for*** *each (existing_pattern, existing_fitness) ∈ HashTable[support]* ***do***

5. ***if*** *pattern ⊂ existing_pattern or pattern = existing_pattern* ***then***

6. ***return***

7. ***else if*** *existing_pattern ⊂ pattern* ***then***

8. *remove existing_pattern from HashTable[support]*

9. *add (pattern, fitness) to HashTable[support]*

10. ***return***

11. ***end if***

12. ***end for***

13. append (pattern, fitness) to HashTable[support]

Explanation:

Lines 1, 2, 3: If the key does not exist in the hashtable, then create a new entry with HashTable[support] = ∅

Lines 1, 2, and 3: If the key does not exist in the hashtable, create a new entry with HashTable[support] = ∅.

Lines 4 to 12: Iterate through the values of hashtable[support] to perform the insertion of the (pattern, futness) pair into hashtable[support].

Lines 5 to 11: If the pattern currently being considered is a sub-pattern of or equal to the existing_pattern, do not insert the pattern into the hashtable. Otherwise, delete the existing_pattern and insert the pattern into hashtable[support].

Line 12: In the case where the pattern is neither a parent nor a child, insert the pair (pattern, fitness) into hashtable[support].

H. EXAMPLE ILLUSTRATING THE PFCLOHUS_QUANTITY_GA_SS ALGORITHM

To illustrate the operation of the PFCloHUS_QUANTITY_GA_SS algorithm, we use the sequence transaction database shown in Tables 2 and 3, stored as in Table 7, as well as the population in Table 8.

With the input parameters: minutil = 10, minsup = 2, popsize = 5, and gen = 50

The proposed algorithm consists of the following two phases:

Stage 1: Apply the genetic algorithm to find the set of high-utility frequent patterns, and store them in the hashtable as shown in the following result.

1 -1 -2 #TotalUtility: 48, sup = 10
2 -1 -2 #TotalUtility: 75, sup = 8
1 2 -1 -2 #TotalUtility: 67, sup = 5
2 3 -1 -2 #TotalUtility: 35, sup = 4
2 -1 2 -1 -2 #TotalUtility: 75, sup = 4
2 -1 1 2 -1 -2 #TotalUtility: 96, sup = 4
3 -1 2 -1 -2 #TotalUtility: 34, sup = 3
3 -1 3 -1 -2 #TotalUtility: 12, sup = 3
3 4 -1 -2 #TotalUtility: 40, sup = 3
2 4 -1 -2 #TotalUtility: 45, sup = 2
2 4 -1 2 -1 -2 #TotalUtility: 60, sup = 2
2 3 -1 4 -1 -2 #TotalUtility: 42, sup = 2
4 -1 2 3 -1 -2 #TotalUtility: 40, sup = 2
4 -1 2 3 -1 2 -1 -2 #TotalUtility: 60, sup = 2
2 -1 2 3 -1 -2 #TotalUtility: 50, sup = 2
2 -1 3 -1 -2 #TotalUtility: 40, sup = 2
3 4 -1 2 -1 -2 #TotalUtility: 32, sup = 2
2 4 -1 1 2 -1 -2 #TotalUtility: 66, sup = 2

1 2 -1 1 2 -1 -2 #TotalUtility: 59, sup = 2

Phase 2: Consideration for inclusion in the set of high utility closed frequent patterns

Table 9. Result set of high utility frequent patterns

Sup	Sequences	
	Itemsets	**#TotalUtility**
10	[1]	48
8	[2]	75
5	[1 2]	67
4	[2 3]	35
	[2], [1 2]	96
3	[3], [2]	34
	[3], [3]	12
	[3 4]	40
2	[2 3], [4]	42
	[4], [2 3], [2]	60
	[2], [2 3]	50
	[3 4], [2]	32
	[2 4], [1 2]	66
	[1 2], [1 2]	59

V. EXPERIMENTAL EVALUATION

EVALUATION OF EXECUTION TIME AND MEMORY CONSUMPTION

The experiments were performed on a computer with a 2.90 GHz CPU, 8 cores, and 32 GB of memory, running the 64-bit Microsoft Windows 10 operating system. The program was implemented in Python using the Google Colab environment. The datasets were taken from the website https://www.philippe-fournier-viger.com/spmf, as shown in Table 10. The proposed algorithm uses the basic parameters of the genetic algorithm as follows: the maximum number of generations is 20, and the population size is 100.

Table 10. Quantitative Sequential Datasets

Dataset	Sequences	Distinct itemsets	Distinct items	twu
BMS	23773	18473	497	108457438
OnlineRetails	29471	83490	16470	14910915
crimes_chicago	196003	13549	31	1593655
accidents	170042	339898	463	196141636

First, the paper compares the execution time efficiency of the proposed algorithm (PFCloHUS_QUANTITY_GA) with the P-FCloHUS [7] algorithm on the BMS dataset. The *minutil* threshold is set to 4.57e-4%, and the minsup threshold is selected within the range [4.53e-3, 9e-3]. The experimental results show that the **PFCloHUS_QUANTITY_GA** algorithm is significantly more efficient than **P-FCloHUS**. Pattern generation is performed directly during the population generation phase of the proposed algorithm. Next, the experiment compares the memory usage of P-FCloHUS and PFCloHUS_QUANTITY_GA on the BMS dataset. The results in Figure V-1 show that the proposed algorithm consumes less memory and exhibits greater stability due to its fixed number of populations and generations, even as the number of transactions increases.

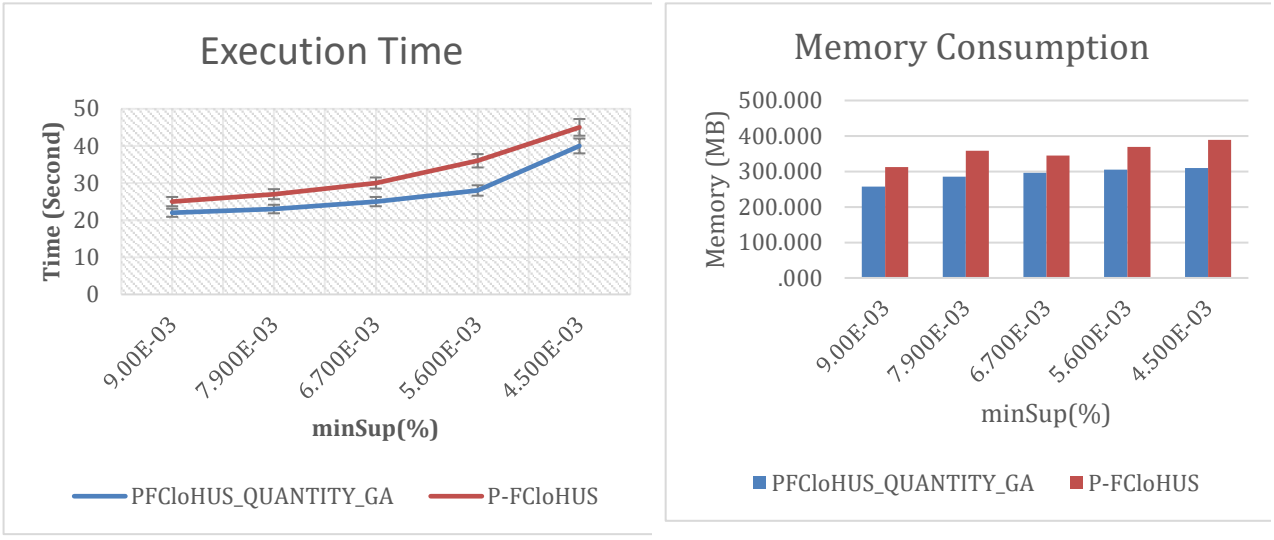

*Figure 5. Execution time and memory consumption of the BMS dataset on **P-FCloHUS** and **PFCloHUS_QUANTITY_GA***

For the Accidents dataset, with the minutil threshold of 3.57e-4 and the minsup threshold in the range [4.5e-3, 9e-3], the results consistently demonstrate that the execution time and memory consumption of **PFCloHUS_QUANTITY_GA** are significantly lower than those of **P-FCloHUS**.

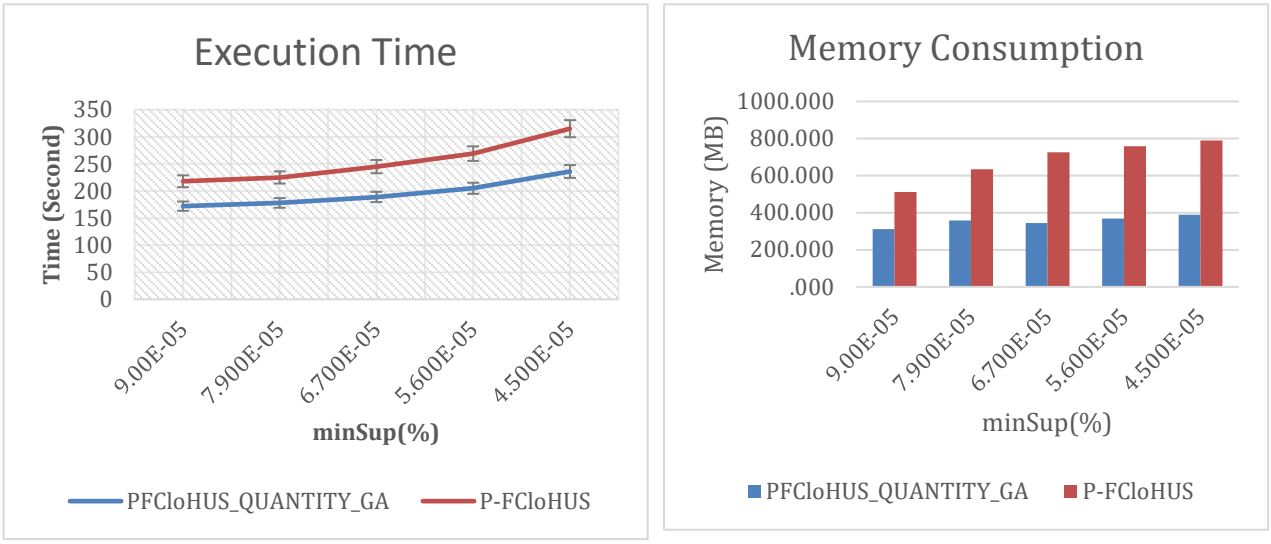

Figure 6. Execution time and memory consumption of the Accidents dataset on P-FCloHUS and PFCloHUS_QUANTITY_GA.

For the crimes_chicago dataset, with the minutil threshold of 4.52e-5 and the minsup threshold in the range [4.6e-4, 9.2e-4], the results show that both the execution time and memory consumption of **PFCloHUS_QUANTITY_GA** are significantly lower than those of **P-FCloHUS**.

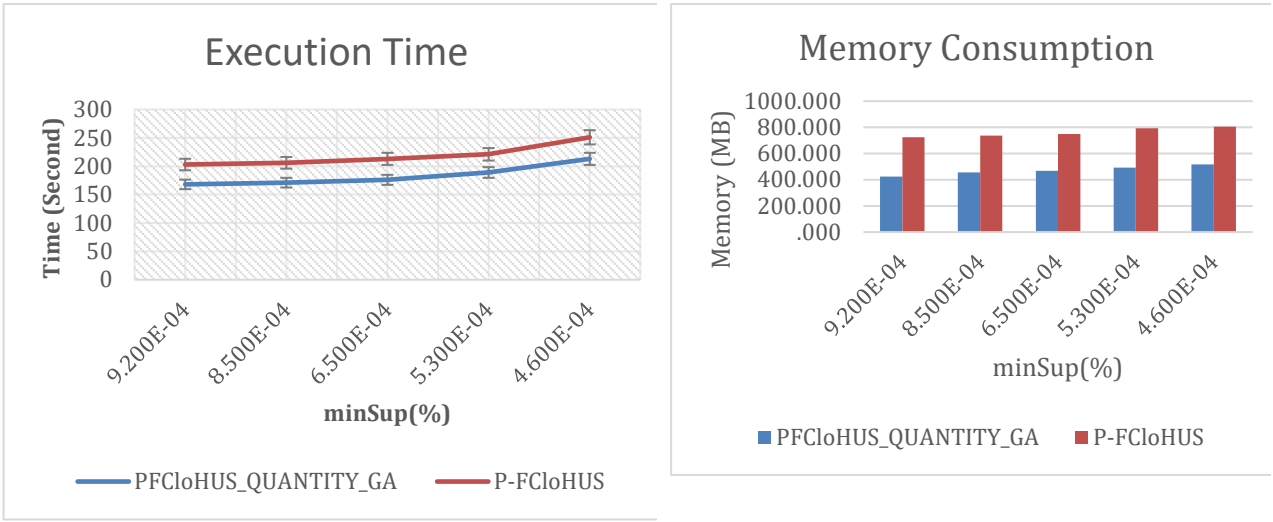

Figure 7. Execution time and memory consumption of the Crimes_Chicago dataset on P-FCloHUS and PFCloHUS_QUANTITY_GA

Finally, for the OnlineRetails dataset, with the minutil threshold of 17.9e-5 and the minsup threshold in the range [1.3e-2, 3.2e-2], the results also show that the execution time and memory usage of **PFCloHUS_QUANTITY_GA** are significantly lower than those of P**-FCloHUS**.

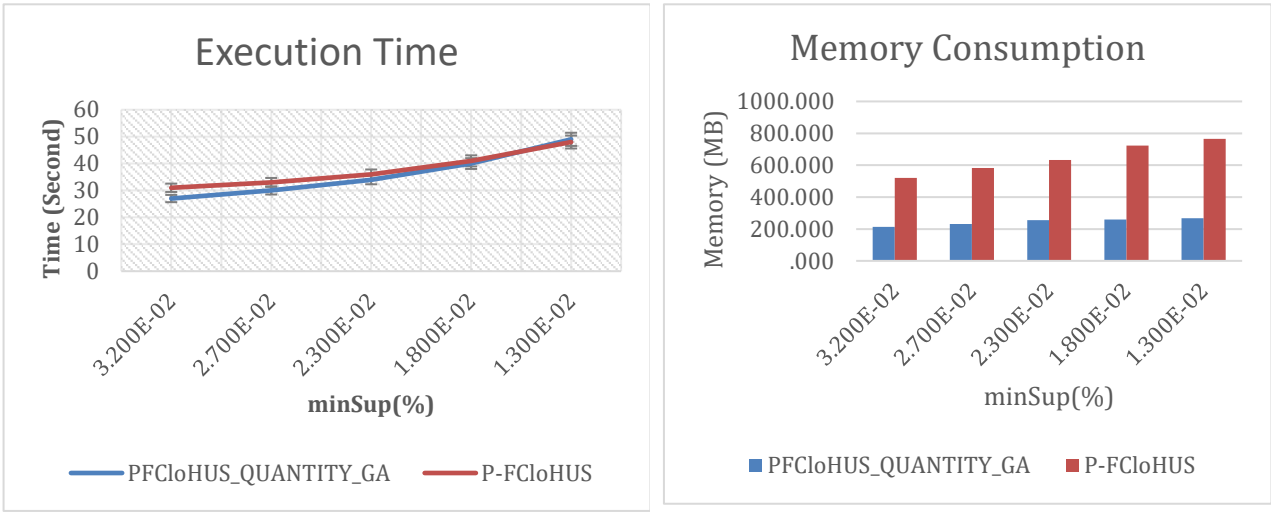

Figure 8. Execution time and memory consumption of the Retail dataset on P-FCloHUS and PFCloHUS_QUANTITY_GA

VI. CONCLUSION

The paper proposes the **PFCloHUS_QUANTITY_GA_SS** algorithm, which combines an optimization method based on the **Genetic Algorithm (GA)** with a **bit array** representation to save memory and accelerate genetic operations. The fitness evaluation process is executed in parallel using the PySpark MapReduce model, which is simultaneously coupled with a hash table structure to efficiently check the closed frequent property of the patterns. Consequently, the algorithm significantly reduces processing time while ensuring the accurate mining of the set of **Closed High-Utility Sequential Patterns (FCHUSPs)**. The main theoretical contributions include: (i) Proposing an individual structure based on the **bit array** in GA for memory optimization. (ii) Designing and implementing a novel parallel processing strategy based on PySpark MapReduce. (iii) Utilizing a **hash table** for rapid checking of the pattern's closed property (these contributions are clearly described in Section III of the paper).

The experimental results on standard datasets (**BMS**, **OnlineRetails**, **Crimes_Chicago**, **Accidents**) show that **PFCloHUS_QUANTITY_GA_SS** significantly **outperforms** the competitive algorithm **P-FCloHUS** in terms of both **execution time** and **memory consumption**. Specifically, on the BMS dataset, the proposed algorithm achieves significantly faster execution times and lower memory consumption than P-FCloHUS. Similarly, on the **OnlineRetails**, **Crimes_Chicago**, and **Accidents** datasets, **PFCloHUS_QUANTITY_GA_SS** also yields much better results regarding time and memory, demonstrating the efficiency and scalability of the method.

In terms of practical applications, the proposed algorithm holds significant promise in mining behavioral and transactional sequence data. For example, it can be used to analyze customer behavior over time, discover high-utility purchasing patterns in e-commerce transactions, or optimize supply chains through high-utility recurring transactional patterns. These applications help businesses improve marketing strategies, product planning, and logistics management by identifying high-value customer behavior patterns.

For future research directions, the fitness function of the GA can be further optimized (for example, using a multi-objective GA or a dynamic GA) to improve the quality of the generated individuals. In addition, extending the algorithm to real-time data streams (where concept drift may occur) would enable real-time data analysis, which is especially important in alert detection systems or online transaction analysis. Finally, integrating the algorithm into large-scale data analytics platforms such as Apache Spark or Hadoop MapReduce would enhance its ability to process large datasets, aligning with current data trends.

In summary, the research results not only provide an effective new method for the problem of mining closed frequent high-utility sequential patterns but also open up further development directions associated with parallel processing and large real-world datasets, promising to deliver practical value in knowledge discovery from data.

VII. REFERENCES

[1] R. Agrawal and R. Srikant, (1995), "Mining sequential patterns," in *Proceedings of the eleventh international conference on data engineering, IEEE*, DOI: 10.1109/ICDE.1995.380415.

[2] V. C.-C. Liao and M.-S. Chen, (2013), "Efficient mining gapped sequential patterns for motifs in biological sequences," *BMC systems biology,* vol. 7, DOI: 10.1186/1752-0509-7-S4-S7.

[3] S Vijayalakshmi, V Mohan and S. S. Raja, (2010), "Mining of users access behavior for frequent sequential pattern from web logs," *International Journal of Database Management System (IJDM),* vol. 2, DOI:10.5121/ijdms.2010.2304.

[4] C. F. Ahmed, S. K. Tanbeer and B.-S. Jeong, (2010), "Mining high utility web access se quences in dynamic web log data," in *11th ACIS international conference on software engineering, artificial intelligence, networking and parallel/distributed computin*, DOI: 10.1109/SNPD.2010.21.

[5] C. F. Ahmed, S. K. Tanbeer and B.-S. Jeong, (2010), "A novel approach for mining high-utility sequential patterns in sequence databases," *ETRI journal,* vol. 32, no. 5, p. 676–686, https://doi.org/10.4218/etrij.10.1510.0066.

[6] J. C.-W. Lin, Y. Li, P. Fournier-Viger and Y. Djenour, (2019), "Mining high-utility sequential patterns from big datasets," in *IEEE International conference on big data (big data)*, DOI: 10.1109/BigData47090.2019.9005996.

[7] H.-P. Nguyen and B. Le, (2022), "P-fclohus: A parallel approach for mining frequent closed high-utility sequences on multi-core processors," in *Asian Conference on Intelligent Information and Database Systems, Springer*, DOI:10.1007/978-981-19-8234-7_31.

[8] T. Truong, H. Duong, B. Le and P. Fournier-Viger, (2019), "Fmaxclohusm: An efficient algorithm for mining frequent closed and maximal high utility sequences," *Engineering Applications of Artificial Intelligence,* vol. 85, pp. 1-20, https://doi.org/10.1016/j.engappai.2019.05.010.

[9] J. Yin, Z. Zheng and L. Cao, (2012), "Uspan: An efficient algorithm for mining high utility sequential patterns," in *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*, https://doi.org/10.1145/2339530.2339636.

[10] W. Gan, J. C.-W. Lin, J. Zhang and P. Fournier-Viger, (2020), "Fast utility mining on sequence data," *IEEE transactions on cybernetics,* vol. 51, p. 487–500, DOI: 10.1109/TCYB.2020.2970176.

[11] C. Zhang, Y. Yang, Z. Du, W. Gan and P. S. Yu, (2023), "Husp-sp: Faster utility mining on sequence data," *ACM Transactions on Knowledge Discovery from Data,* vol. 18, no. 1, pp. 1-20, https://doi.org/10.1145/3597935.

[12] S Kannimuthu and K. Premalatha, (2014), "Discovery of high utility itemsets using genetic algorithm with ranked mutation," *Applied Artificial Intelligence,* vol. 28, no. 4, pp. 337-359, DOI: 10.1080/08839514.2014.891839.

[13] J. M. Luna, R. U. Kiran, P. Fournier-Viger and S. Ventura, (2023), "Efficient mining of top-k high utility itemsets through genetic algorithms," *Information Sciences,* vol. 624, p. 529–553, DOI:10.1016/j.ins.2022.12.092.

[14] T. D. Nguyen, L. T. Nguyen and B. Vo, (2018), "A parallel algorithm for mining high u tility itemsets," in *International Conference on Information Systems Architecture and Technology, Springer*.

[15] G. Kimura, Y. Hayamizu, R. U. Kiran, M. Kitsuregaw and Kazuo Goda, 2023, "Efficient parallel mining of high-utility itemsets on multicore processors," in *IEEE 39th International Conference on Data Engineering (ICDE)*, DOI: 10.1109/ICDE55515.2023.00388.

[16] W. Fang, H. Jiang, H. Lu, J. Sun, X. Wu and J. C, (2023), "Gpu-based efficient parallel heuristic algorithm for high-utility itemset mining in large transaction datasets," *IEEE Transactions on Knowledge and Data Engineering,* vol. 36, pp. 652-667, DOI: 10.1109/TKDE.2023.3290371.

KHAI THÁC CÁC MẪU TUẦN TỰ CÓ LỢI ÍCH CAO PHỔ BIẾN ĐÓNG DỰA TRÊN THUẬT GIẢI DI TRUYỀN

Trần Anh Duy, Lê Thị Minh Nguyện

TÓM TẮT—Khai thác các mẫu tuần tự có lợi ích cao phổ biến đóng (Frequent Closed High-Utility Sequential Patterns– FCHUSPs) là một bài toán quan trọng trong khai phá dữ liệu, với nhiều ứng dụng thực tiễn như phân tích hành vi khách hàng, tối ưu chuỗi cung ứng và marketing. Tuy nhiên, quá trình khai thác phải đối mặt với không gian tìm kiếm khổng lồ và chi phí tính toán cao, đặc biệt khi ngưỡng đầu vào thấp hoặc dữ liệu có quy mô lớn. Trong bài báo này, chúng tôi đề xuất một phương pháp khai thác FCHUSPs dựa trên thuật giải di truyền (Genetic Algorithm– GA), trong đó mỗi cá thể được biểu diễn bằng cấu trúc bitarray nhằm tối ưu hóa bộ nhớ và thao tác di truyền. Để nâng cao hiệu quả, quá trình tính toán độ thích nghi được thực hiện song song bằng PySpark MapReduce, kết hợp với hashtable để kiểm tra tính phổ biến đóng của mẫu. Thuật toán đề xuất, gọi là PFCloHUS_QUANTITY_GA_SS, đã được triển khai và đánh giá trên nhiều tập dữ liệu thực nghiệm. Kết quả cho thấy phương pháp đề xuất giúp giảm đáng kể thời gian thực thi so với các phương pháp truyền thống trong khai thác các mẫu tuần tự lợi ích cao đóng thường xuyên.

Từ khóa— Khai thác mẫu tuần tự, lợi ích cao, phổ biến đóng, thuật giải di truyền, tính toán song song, PySpark.

Trần Anh Duy received a Master of Science degree in Computer Science from the Ho Chi Minh City University of Science in 2017. He is currently a lecturer at the Faculty of Information Technology, Ho Chi Minh City University of Foreign Languages - Information Technology (HUFLIT). His current research interest is Data Mining.

Lê Thị Minh Nguyện received a Master of Science degree in Computer Science from the Vietnam National University Ho Chi Minh City in 2007. She is currently a lecturer at the Faculty of Information Technology, Ho Chi Minh City University of Foreign Languages - Information Technology (HUFLIT). Her current research interest is Data Mining.