

A
Major Project Report
on
**AN EXPERIMENTAL STUDY ON DYNAMIC PREFIX TREE
ALGORITHM**

Submitted in partial fulfilment of the requirements for the award of the Degree of
Bachelor of Technology

By
Alla Sai Manideep Reddy
(20EG105402)

Beerreddy Harshitha
(20EG105405)

Adithya Krishnamurthy
(20EG105417)



Under The Guidance Of

Dr. Pallam Ravi
Assistant Professor
Department of CSE

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
ANURAG UNIVERSITY VENKATAPUR (V), GHATKESAR (M),
MEDCHAL (D), T.S 500088
(2023-24)

DECLARATION

We hereby declare that the report entitled “**An Experimental Study On Dynamic Prefix Tree Algorithm**” submitted to the **Anurag University** in partial fulfilment of the requirements for the award of the degree of **Bachelor of Technology (B.Tech)** in **Computer Science and Engineering** is a record of original work done by us under the guidance of **Dr. Pallam Ravi, Assistant Professor** and this report has not been submitted to any other University or Institution for the award of any other degree or diploma.

Place: Anurag University, Hyderabad

Alla Sai Manideep Reddy

Date:

(20EG105402)

Beerreddy Harshitha

(20EG105405)

Adithya Krishnamurthy

(20EG105417)

CERTIFICATE

This is to certify that the project report entitled “**An Experimental Study On Dynamic Prefix Tree Algorithm**” being submitted by **Mr. A. Sai Manideep Reddy** bearing the Hall Ticket number **20EG105402**, **Ms. B. Harshitha** bearing the Hall Ticket number **20EG105405**, **Mr. K. Adithya** bearing the Hall Ticket number **20EG105417** in partial fulfilment of the requirements for the award of the degree of the **Bachelor of Technology in Computer Science and Engineering** to the Anurag University is a record of bonafide work carried out by them under my guidance and supervision for the academic year 2023 to 2024.

The results presented in this report have been verified and found to be satisfactory. The results embodied in this report have not been submitted to any other University or Institute for the award of any other degree or diploma.

Signature of Supervisor
Dr. Pallam Ravi
Assistant Professor
Department of CSE

Signature of Dean
Dr. G. Vishnu Murthy
Dean, CSE

External Examiner

ACKNOWLEDGEMENT

We would like to express our sincere thanks and deep sense of gratitude to project supervisor and Co-ordinator **Dr. PALLAM RAVI, Assistant Professor, Department of Computer Science and Engineering**, Anurag University for his constant encouragement and inspiring guidance without which this project could not have been completed. His critical reviews and constructive comments improved our grasp of the subject and steered us towards the fruitful completion of the work. His patience, guidance and encouragement made this project possible.

We would like acknowledge our sincere gratitude for the support extended by **Dr. G. VISHNU MURTHY, Dean, Department of Computer Science and Engineering**, Anurag University. We also express my deep sense of gratitude to

Dr. V. V. S. S. S. BALARAM, Academic co-ordinator. Project Co-ordinator and Project review committee members, whose research expertise and commitment to the highest standards continuously motivated us during the crucial stage of our project work.

We would like express our special thanks to **Dr. V. VIJAYA KUMAR, Dean School of Engineering**, Anurag University, for his encouragement and timely support in our B.Tech program.

Alla Sai Manideep Reddy
(20EG105402)

Beerreddy Harshitha
(20EG105405)

Adithya Krishnamurthy
(20EG105417)

Abstract

Frequent itemset mining is a fundamental task in data mining crucial for various applications such as recommendation systems and market analysis. Traditional algorithms like FP-growth and Apriori are effective but suffer from memory overhead and multiple passes. In this project, we used a novel approach known as Dynamic Prefix Tree (DPT), to address memory overhead and multiple passes. The DPT technique utilizes a single dynamically adaptive prefix tree to represent transactional datasets and computing frequent itemsets. This innovative method significantly reduces memory consumption and processing costs compared to existing novel approaches. By eliminating the need for multiple trees, DPT enhances processing speed, avoiding unnecessary scans and tree constructions.

We have experimented the dynamic prefix algorithm with various strategies with object-oriented programming (OOP) in Java and tried improving it further. We attempted to reduce the number of nodes in prefix tree and reduce the DPT calls, as well as to directly generate maximal itemsets using the DPT structure. However, our experiments showed that these optimization efforts were not successful, as the itemsets have multiple occurrences in the dataset but not the single occurrence. Copying and merging operations, as well as the use of the MFI (Maximal Frequent Itemset) algorithm, did not yield significant improvements in performance. This suggests that the DPT approach is already a highly efficient solution for frequent itemset mining. The DPT structure is designed to find frequently occurring itemsets efficiently within the tree, thereby minimizing memory requirements and enhancing scalability for large datasets. Unlike traditional methods, DPT offers a direct representation by generating a prefix tree consisting of all frequently occurring itemsets.

Keywords - Frequent itemset mining, Dynamic Prefix Tree (DPT), Copying and Merging, Memory consumption, Processing speed, Scalability.

TABLE OF CONTENT

S.No	Content	Page No.
1	Introduction	1
	1.1 Motivation	2
	1.2 Problem Illustration	3
	1.3 Objective	5
	1.4 Introduction to the Topics	6
	1.4.1 Classic Frequent Itemset Mining Algorithms	6
	1.4.2 Limitations of Traditional Prefix Tree based approaches	6
	1.4.3 Dynamic Prefix Tree Algorithm (DPT)	7
	1.4.4 Object Oriented Design of the DPT Algorithm	7
2	Literature Survey	8
3	Proposed Method	14
	3.1 Data Structure	14
	3.2 Construction of Prefix Tree	15
	3.3 Dynamic Prefix Tree	18
4	Implementation	21
	4.1 Functionality	21
	4.1.1 File Input /Output	21
	4.1.2 Constructing Prefix Tree	21
	4.1.2.1 Iterative Construction from Transaction Database	22
	4.1.2.2 Insertion of items into Prefix Tree	22
	4.1.2.3 Filtering infrequent items	23
	4.1.2.4 Representation of Transactional Relationships	24
	4.1.3 Dynamic Prefix Tree (DPT) Algorithm	24
	4.1.3.1 Recursive Processing of Prefix Tree	25
	4.1.3.2 Copying Subtrees	25
	4.1.3.3 Merging Subtrees for infrequent nodes	26
	4.1.3.4 Recursive Processing for relevant nodes	26
	4.1.4 Memory Management	27

	4.1.5 Support Count	28
	4.1.6 Printing Prefix Tree	28
	4.2 Attributes	29
	4.2.1 Dynamic Class	29
	4.2.2 TreeNode Class	29
	4.3 Experimental Screenshot	30
	4.4 Experimental Analysis	33
	4.4.1 Reducing the DPT calls	34
	4.4.2 Reducing the Nodes	36
	4.4.3 Finding the Maximal Frequent Itemsets using DPT	39
	4.5 Dataset	40
5	Experimental Setup	42
	5.1 Parameters	45
	5.1.1 Time Complexity	45
	5.1.2 Space Complexity	45
6	Discussion of Results	47
	6.1 Findings	51
	6.2 Advantages and Applications	52
7	Conclusion	54
8	References	55

List of Figures

Figure No.	Figure Name	Page No.
Figure 1.2.1	FP-Tree	4
Figure 1.2.2	d conditional FP-Tree	5
Figure 3.2.1	FP-Tree of First Transaction	15
Figure 3.2.2	FP-Tree of Second Transaction	16
Figure 3.2.3	FP-Tree of Third Transaction	16
Figure 3.2.4	FP-Tree of Fourth Transaction	16
Figure 3.2.5	FP-Tree of Fifth Transaction	16
Figure 3.2.6	FP-Tree of Sixth Transaction	17
Figure 3.2.7	FP-Tree of Seventh Transaction	17
Figure 3.2.8	FP-Tree of Eighth Transaction	17
Figure 3.2.9	Prefix Tree (Final FP-Tree)	18
Figure 3.3.1	Dynamic Prefix Tree	19
Figure 4.3.1	Output for Chess dataset with Optimization	30
Figure 4.3.2	Output for Chess dataset without Optimization	31
Figure 4.3.3	Output for Accidents dataset with Optimization	31
Figure 4.3.4	Output for Accidents dataset without Optimization	32
Figure 4.3.5	Output for T1OI4D100K dataset with Optimization	32
Figure 4.3.6	Output for T1OI4D100K dataset without Optimization	33
Figure 4.4.1.1	Reducing the DPT calls	35
Figure 5.1	Creating the Java Project	42
Figure 5.2	Naming the Java Project	43
Figure 5.3	Importing the Java Classes	43
Figure 5.4	FIMI Repository	44
Figure 5.5	T1OI4D100K Dataset	45

Figure 6.1	Memory consumed by T1OI4D100K Dataset	49
Figure 6.2	Time Consumed by T1OI4D100K Dataset	49
Figure 6.3	Nodes generated by Three different Datasets	50
Figure 6.4	Memory usage of Three different Datasets	51

List of Tables

Table No.	Table Name	Page No.
Table 1.2.1	Dataset for FP-Growth Algorithm	3
Table 1.2.2	Support Value of Each Item	3
Table 1.2.3	Ordered Frequent Itemset	4
Table 1.2.4	Conditional Pattern Bases	4
Table 2.1	Comparative Analysis of Pattern Mining Algorithms	9
Table 2.2	Comparison of Literature	12
Table 3.1.1	Database and Frequent Itemsets	15
Table 4.4.1.1	Global counter of each Item	35
Table 4.4.2.1	Global counter	37
Table 4.4.2.2	Global count first DPT call	37
Table 4.4.2.3	Global count second DPT call	37
Table 4.4.2.4	Global count third DPT call	37
Table 4.4.2.5	Global count fourth DPT call	37
Table 4.4.2.6	Global count fifth DPT call	37
Table 4.4.2.7	Global count sixth DPT call	37
Table 4.4.2.8	Global count seventh DPT call	38
Table 4.4.2.9	Global count eighth DPT call	38

Table 4.4.2.10	Global count ninth DPT call	38
Table 4.4.2.11	Global count tenth DPT call	38
Table 4.4.2.12	Global count eleventh DPT call	38
Table 4.4.2.13	Global count twelfth DPT call	38
Table 4.4.2.14	Global count thirteenth DPT call	39
Table 4.4.2.15	Global count Fourteenth DPT call	39
Table 6.1	Chess Dataset	47
Table 6.2	Accidents Dataset	48
Table 6.3	T10I4D100K Dataset	48

List of Abbreviations

Abbreviations	Full Form
FP	Frequent Pattern
DPT	Dynamic Prefix Tree
OOP	Object-Oriented Programming
DHP	Direct Hashing and Pruning
CARMA	Continuous Association Rule Mining Algorithm
RARM	Rapid Association Rule Mining
ELCAT	Equivalence Class Clustering and Bottom-Up Lattice Traversal
AFOPT	Ascending Frequency Ordered Prefix-Tree
DRFP	Disk-Resident Frequent pattern
COFI	Co-Occurrence Frequent Itemset
FIMI	Frequent Itemset Mining Implementations

1.INTRODUCTION

Data mining, as a fundamental aspect of modern computing, plays a pivotal role in extracting valuable insights and patterns from large datasets. It operates at the intersection of several disciplines, including statistics, machine learning, and database systems, leveraging techniques from each to extract meaningful information. The primary objective of data mining is multifaceted: it aims to uncover hidden patterns, correlations, and trends within data to facilitate informed decision-making, predict future outcomes, and gain a deeper understanding of underlying phenomena.

Frequent pattern mining has been an important subject matter in data mining from many years. Frequent itemset mining is a foundational technique in data mining, particularly within the realm of association rule mining. It involves the extraction of sets of items that frequently co-occur together in transactional datasets. These sets of items, known as frequent itemsets, are crucial for uncovering hidden patterns and relationships within the data, which can provide valuable insights for decision-making and analysis. A remarkable progress in this field has been made and lots of efficient algorithms have been designed to search frequent patterns in a transactional database. Agrawal et al. (1993) firstly proposed pattern mining concept in form of market based analysis for finding association between items bought in a market. This concept used transactional databases and other data repositories in order to extract association's casual structures, interesting correlations or frequent patterns among set of . Frequent patterns are those items, sequences or substructures that reprise in database transactions with a user specified frequency. An itemset with frequency greater than or equal to minimum threshold will be considered as a frequent pattern.

Frequent pattern mining can be used in a variety of real world applications. It can be used in super markets for selling, product placement on shelves, for promotion rules and in text searching. It can be used in wireless sensor networks especially in smart homes with sensors attached on Human Body or home usage objects and other applications that require monitoring of user environment carefully that are subject to critical conditions or hazards such as gas leak, fire and explosion .These frequent patterns can be used to monitor the activities for dementia patients. It can be seen as an important approach with the ability to monitor activities of daily life in smart environment for tracking functional decline among dementia patients .

The Apriori algorithm [1] is an effective method for frequent itemset mining. In order to obtain all frequent itemsets, Apriori iteratively scans a database. It can find out all frequent itemsets. Generating and testing a large number of candidate itemsets, Apriori and the Apriori-like algorithms are called candidate generation-and-test approaches. Although these approaches can mine all frequent itemsets from a database, they are unavoidably confronted with two problems: (1) the database is scanned many times, and the time of database scan is exactly equal to the length of the longest frequent itemset; (2) the number of candidate itemsets is very large, and generating and testing these candidates is very costly, especially when the database or all candidate itemsets cannot be completely loaded in main memory.

The issues mentioned above can be avoided and common itemsets from a database can be mined using pattern growth techniques like FP-Growth [10,11]. A database is represented in FP-Growth as an extremely compact prefix tree structure known as an FP-tree. FP-Growth builds conditional FP-trees iteratively to mine common itemsets after creating an initial FP-tree from a database.

1.1 Motivation

Many earlier studies have demonstrated the superior efficiency of pattern growth techniques based on prefix trees over candidate generation and test techniques. The issue with pattern growth approaches, however, is that they need to invest a lot of time in creating a large number of (conditional) prefix trees. There are multiple processes involved in creating a conditional prefix tree: determining which items are frequently used, creating a branch for every transaction, and adding the branch to the tree. These phases are carried out iteratively and recursively by FP-Growth-like algorithms, which takes a very long time. In order for these methods to produce a frequent itemset with k items, k conditional prefix trees must be built. Prefix tree construction is time-consuming and frequently results in poor data localization.

As an intermediary product, frequent itemsets typically have a significant function from the application perspective [4]–[7]. For instance, certain programs must determine if a given itemset is frequent or obtain the support of one that is. In actuality, there are typically a lot of common itemsets, particularly for small min_sup or dense databases.

However, regardless of the pattern growth or candidate generation-and-test techniques these algorithms belong to, earlier algorithms only produced a list of frequently occurring itemsets. This list is simply too big to use effectively. It's crucial to choose a suitable way to represent frequent itemsets before using them further.

1.2 Problem Illustration

Example:

Transaction ID	Itemset
1	{a, b, c, e}
2	{b, d}
3	{a, c, e}
4	{a, b, c, f}
5	{b, d, g}
6	{a, b, d, h}
7	{a, h}
8	{b, d, g}

Item	Frequency
a	5
b	6
c	3
d	4
e	2
f	1
g	1
h	1

Table 1.2.1 Dataset for FP-Growth Algorithm **Table 1.2.2** Support value of each item

The above Table 1.3.1 is the dataset which is used for construction of FP-Tree by using FP-Growth Algorithm. After the first Database scan we will get the by-product as items with their counts which is represented in Table 1.3.2. Now we have assumed the **Minimum Support as 3**. The items which have the count less than the Minimum Support are infrequent and are not considered for the construction of FP-Tree.

Now scan the database for the second time and order the frequent itemsets after removing the infrequent itemsets. This results in the Table 1.3.3. Then construct the FP-Tree based on the order of items in each transaction.

Transaction ID	Itemset	Ordered Frequent itemset
1	{a, b, c, e}	{b, a, c}
2	{b, d}	{b, d}
3	{a, c, e}	{a, c}
4	{a, b, c, f}	{b, a, c}
5	{b, d, g}	{b, d}
6	{a, b, d, h}	{b, a, d}
7	{a, h}	{a}
8	{b, d, g}	{b, d}

Table 1.2.3 Ordered Frequent itemset

Construct FP-Tree

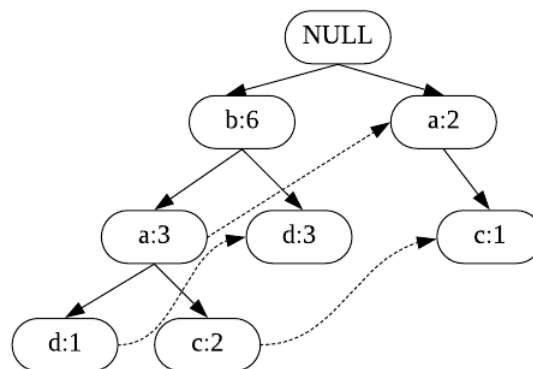


Figure 1.2.1 FP -Tree

Conditional pattern bases

Item	Conditional Pattern
b	{ }
a	b:3
d	b:3, ba:1

c	a:1, ba:2
---	-----------

Table 1.2.4 Conditional Pattern bases

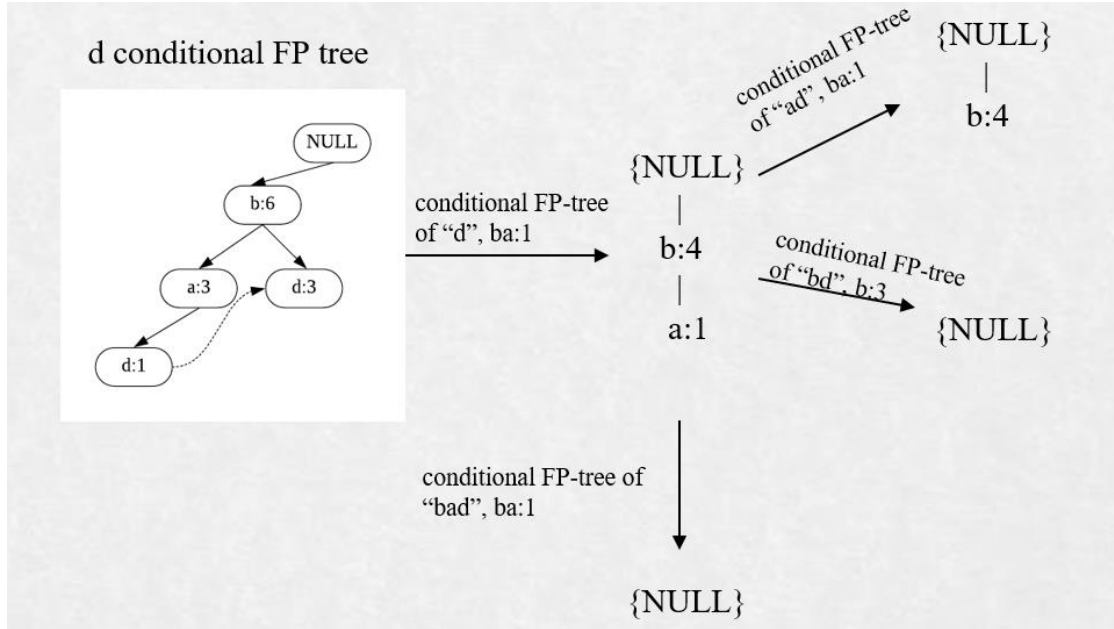


Figure 1.2.2 d conditional FP-Tree

In the above example in-order to find the frequent patterns related to 'd' , we have to construct four FP-prefix trees. In the same way if we wanted to find out the frequent patterns of all items then we need generate more prefix trees which time consuming and memory consuming. So we have used a novel method for finding the frequent patterns using only a single **Dynamic Prefix Tree**.

1.3 Objective

The primary objective of this frequent itemset mining project is to develop a novel algorithm called Dynamic Prefix Tree (DPT) that addresses the limitations of existing prefix tree-based approaches. The DPT algorithm aims to significantly improve the efficiency and scalability of frequent itemset mining by using a single, dynamically adaptive prefix tree to represent the dataset, rather than constructing multiple trees as in previous methods. Key goals of the DPT approach include minimizing memory

consumption, reducing processing time through the avoidance of unnecessary scans and constructions, and generating a comprehensive prefix tree that directly encodes all frequent itemsets. Additionally, the DPT algorithm is designed using object-oriented programming principles, such as abstraction, inheritance, and polymorphism, to ensure a modular, extensible, and maintainable implementation. The overarching goal is to provide a more efficient and scalable solution for frequent itemset mining that can better support a wide range of data mining applications.

1.4 Introduction to the topics

Frequent itemset mining is a fundamental problem in the field of data mining, with applications spanning across various domains, including recommendation systems, market analysis, and association rule mining. The goal of frequent itemset mining is to identify sets of items that occur together frequently within a given dataset. By uncovering these recurrent patterns, businesses and researchers can gain valuable insights that can drive decision-making, personalization, and optimization in their respective domains.

1.4.1 Classic Frequent Itemset Mining Algorithms

Over the years, researchers have developed several algorithms to tackle the frequent itemset mining problem. Two well-established approaches in this area are the Apriori algorithm and the FP-Growth (Frequent Pattern Growth) algorithm. These classic techniques typically rely on a prefix tree (also known as a trie) data structure to compactly represent the dataset and efficiently identify the frequent itemsets. By organizing the data in a tree-like manner, these algorithms can quickly traverse and analyze the dataset to discover the frequently occurring item sets.

1.4.2 Limitations of Traditional Prefix Tree-based Approaches

While the prefix tree-based algorithms have proven effective in many scenarios, they do have certain drawbacks that can limit their performance and scalability, especially when dealing with large-scale datasets. One key limitation is the need to construct multiple prefix trees, which can result in high memory consumption and repeated dataset scans. This can lead to prolonged processing times and make these methods less efficient for processing extensive or complex data.

1.4.3 The Dynamic Prefix Tree (DPT) Algorithm

To overcome the limitations of the traditional prefix tree-based approaches, this project presents a novel frequent itemset mining algorithm called Dynamic Prefix Tree (DPT). The DPT algorithm takes a unique approach by utilizing a single, dynamically adaptive prefix tree to represent the dataset and mine frequent itemsets. This innovative technique aims to significantly reduce memory usage and improve processing efficiency compared to the multiple tree constructions required by previous methods.

1.4.4 Object-Oriented Design of the DPT Algorithm

The DPT algorithm is designed with a focus on modularity and extensibility, leveraging object-oriented programming (OOP) principles to ensure a flexible and maintainable implementation. Key OOP concepts, such as abstraction and polymorphism, are employed to facilitate the dynamic updates and adaptations of the prefix tree structure. This object-oriented approach allows for a more modular and extensible solution, enabling the incorporation of additional features or optimizations as needed.

2. LITERATURE SURVEY

The foundations of frequent itemset mining were laid by the seminal work of Agrawal, Imieliński, and Swami in 1993 [2]. Their introduction of the concept of "mining association rules between sets of items in large databases" sparked a surge of research in this field, leading to the development of various algorithms and techniques for efficient pattern discovery. The Apriori algorithm [2] is most widely used algorithm in the history of association rule mining that uses efficient candidate generation process, such that large Itemset generated at k level are used to generate candidates at $k+1$ level. On the other hand, it scans database multiple times as long as large frequent Itemsets are generated. Apriori TID generates candidate Itemset before database is scanned with the help of Apriori-gen function. Database is scanned only first time to count support, rather than scanning database it scans candidate Itemset. This variation of Apriori performs well at higher level where as the conventional Apriori performs better at lower levels [23]. Apriori Hybrid is a combination of both the Apriori and Apriori TID. It uses apriori TID in later passes of database as it outperforms at high levels and Apriori in first few passes of database. DHP (Direct hashing and Pruning) [24] tries to maximize the efficiency by reducing the number of candidates generated but it still requires multiple scans of database. DIC [25] based upon dynamic insertion of candidate items, decrease the number of database scan by dividing the database into intervals of particular sizes. CARMA(Continuous Association Rule Mining Algorithm) proposed in [26] generates more candidate Itemset will less scan of database than Apriori and DIC, however it adds the flexibility to change minimum support threshold.

ECLAT [27] with vertical data format uses intersection of transaction ids list for generating candidate Itemset. Each item is stored with its list of Transaction ids instead of mentioning transaction ids with list of items. Sampling algorithm chokes the limitation of I/O overhead by scanning only random samples from the database and not considering whole database. Rapid Association Rule mining (RARM) proposed in [28] generates Large 1- Itemset and large 2- Itemset by using a tree Structure called SOTrieIT and without scanning database. It also avoids complex candidate generation process for large 1-Itemset and Large 2-Itemset that was the main bottleneck in Apriori Algorithm.

One of the key advancements in this domain was the introduction of the Frequent- Pattern tree (FP-tree) by Han et al. [11, 12]. The FP-tree is a compact tree-based data structure that enables the mining of frequent patterns without the need for candidate generation, a significant improvement over traditional approaches. This data structure forms the foundation for many subsequent developments in frequent itemset mining. Another accomplishment in the development of association rule mining and frequent pattern mining is FP-Growth Algorithm which overcomes the two deficiencies of the Apriori Algorithm [2]. Efficiency of FP-Growth is based on three salient features: (1) A divide-and-conquer approach is used to extract small patterns by decomposing the mining problem into a set of smaller problems in conditional databases, which consequently reduces the search space (2) FP-Growth algorithm avoid the complex Candidate Itemset generation process for a large number of candidate Itemsets, and (3) To avoid expensive and repetitive database scan, database is compressed in a highly summarized, much smaller data structure called FP tree [12].

	Apriori	RARM	ECLAT	FP-Growth
Technique	Breadth first search & Apriori property (for Pruning)	Depth first search on SOtrieIT to generate 1-Itemset & 2-Itemset.	Depth first Search & Intersection of transaction ids to generate candidate itemset.	Divide and conquer
Database Scan	Database is scanned for each time a candidate item set is generated	Database is scanned few times to construct a SOtrieIT Tree structure.	Database is scanned few times(Best case=2)	Database is scanned two times only
Time	Execution time is considerable as time is consumed in	Less execution time as compared to Apriori	Execution time is less then apriori algorithm	Less time as compared to Apriori algorithm

	scanning database for each candidate item set generation	Algorithm and FP Growth algorithm		
Drawback	Too many Candidate Itemset. Too many passes over database. Requires large memory space	Difficult to use in interactive system mining Difficult to use in incremental Mining	It requires the virtual memory to perform the transformation	FP-Tree is expensive to build Consumes more memory
Advantage	Use large itemset property. Easy to Implement.	No candidate generation. Speeds up the process for generating candidate 1- Itemset & 2- Itemset	No need to scan database each time a candidate Itemset is generated as support count information will be obtained from previous Itemset	Database is scanned only two times. No candidate generation
Data Format	Horizontal	Horizontal	Vertical	Horizontal
Storage Structure	Array	Tree	Array	Tree(FP-Tree)

Table 2.1 Comparative Analysis of Pattern Mining Algorithms

Building upon the FP-tree, researchers have explored various refinements and extensions to improve the efficiency and applicability of these tree-based methods. Gatuha and Jiang [13] proposed a "smart frequent itemsets mining algorithm" that combines the FP-tree with the DIFFset data structure, demonstrating enhanced

performance in frequent itemset mining. Shahbazi et al. [14] introduced a method for building the FP-tree on the fly, enabling single-pass frequent itemset mining, which is particularly beneficial for streaming or dynamic data environments.

The versatility of the FP-tree approach has also been showcased in various applications. Wang et al. [15] explored the mining of temporal association rules using the frequent itemsets tree, highlighting the adaptability of these tree-based structures to handle time-series data and extract meaningful patterns. The comprehensive survey by Fournier-Viger et al. [16] provides an excellent overview of the different itemset mining techniques, including the FP-tree and its various extensions. This review serves as a valuable resource for understanding the evolution and state-of-the-art in this field.

Researchers have also focused on improving the efficiency of FP-tree-based algorithms. Grahne and Zhu [17] presented fast algorithms for frequent itemset mining using FP-trees, demonstrating significant performance improvements. Liu et al. [18] introduced the AFOPT algorithm, an efficient implementation of the pattern growth approach that builds upon the FP-tree concept.

In the context of object-oriented programming (OOP), the implementation of a dynamic prefix tree (or trie) data structure can be a powerful tool for efficient frequent itemset mining. Chen et al. [18] proposed the F-miner algorithm, which employed a new frequent itemsets mining approach based on a custom data structure. El-Hajj and Zaiane [20] introduced the COFI-tree mining technique, which aimed to reduce the candidacy generation in frequent pattern mining through the use of a tree-based structure.

Adnan and Alhajj [21] developed the DRFP-tree, a disk-resident frequent pattern tree, showcasing the applicability of tree-based approaches in handling large-scale datasets. The DRFP-tree, implemented using OOP principles, can serve as a foundation for the development of a dynamic prefix tree for frequent itemset mining, leveraging the efficiency and flexibility of object-oriented design.

Author(s)	Strategies	Advantages	Disadvantages
Jun-Feng Qu, Bo Hang, Zhao Wu	Trie (prefix tress)	Efficient for exact match and prefix search and Space efficient for sets with common prefixes	Can be memory-intensive for sparse sets and may require additional storage for compression to reduce memory usage
Debabrata Datta, Atindriya De	Hash Tables	Constant-time average-case complexity for search, insertion, and deletion. Efficient for general purpose set operations. Good memory utilization for dense sets.	Lack of inherent support for prefix matching. Hash collisions can degrade performance. Not ordered, so range queries are inefficient.
Lifeng Jia, Zhe Wang, Chunguang Zhou	Suffix trees	Efficient for substring and suffix matching. Compact representation of all substrings. Supports advanced string algorithms.	Memory-intensive for large datasets. Construction can be time consuming. Complex to implement and maintain.

Mohammed J. Zaki, Jian Pei, Wei Fan	Distributed Data Structures	Scalable for large datasets in a distributed environment. Fault-tolerant and resilient to node failures.	May have increased latency due to distributed nature. Limited support for complex queries. Difficulty in maintaining global ordering
Hui Xiong, Wei Wang	Compact Dictionaries	Reduced memory usage compared to traditional structures. Suitable for scenarios with memory constraints.	Operations may be slower due to decompression. Complexity in updating and maintaining compressed structures.

Table 2.2 Comparison of Literature

3. PROPOSED METHOD

We propose a novel pattern growth algorithm for mining frequent itemsets. Different from previous pattern growth algorithms constructing many prefix trees, our algorithm needs only one prefix tree for mining all frequent itemsets with their supports. Frequently constructing prefix trees spends much time and operating on many prefix trees also leads to poor data locality, which is avoided by our algorithm using only one prefix tree. After constructing an initial prefix tree from a database, our algorithm repeatedly adjusts the tree to mine frequent itemsets. Thus, the algorithm is named as the **Dynamic Prefix Tree** algorithm (abbreviated as DPT).

Another interesting advantage of DPT is that it can directly output a prefix tree representing all frequent itemsets, while previous algorithms always output a frequent itemset list. Compared with a list representing all frequent itemsets, a prefix tree representing them is compact and can be used more efficiently and easily. For example, when an application judges whether an itemset is frequent or not or fetches the support of a frequent itemset, operations on a prefix tree are more efficient than those on a list, especially for a large number of frequent itemsets. We have done extensive experiments. Experimental results show the performance improvement of DPT over previous algorithms and the advantage of a prefix tree representing frequent itemsets over a frequent itemset list.

3.1 Data Structure

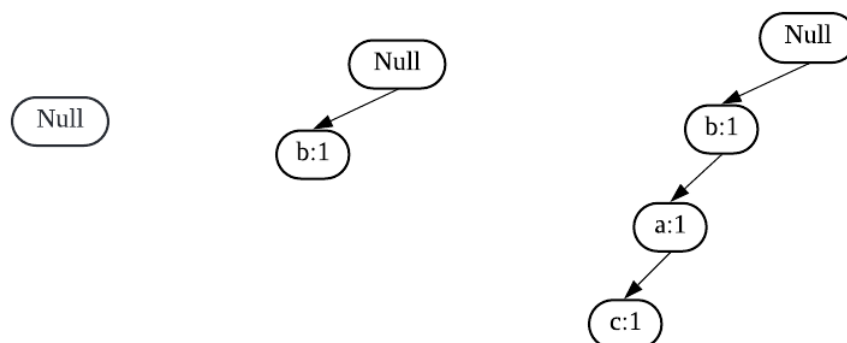
Frequent itemset mining is based on databases stored on disk. Apriori and Apriori-like algorithms have to scan a database on disk many times for mining frequent itemsets. FP-Growth and FP-Growth-like algorithms [10], [25]–[29] use a prefix tree structure to mine frequent itemsets. After an initial prefix tree is constructed from a mined database, the task of mining frequent itemsets based on prefix trees takes place in memory.

Example:

Dataset	Frequent Itemset	Sorted Frequent itemset
{a, b, c, e}	{a, b, c}	{b, a, c}
{b, d}	{b, d}	{b, d}
{a, c, e}	{a, c}	{a, c}
{a, b, c, f}	{a, b, c}	{b, a, c}
{b, d, g}	{b, d}	{b, d}
{a, b, d, h}	{a, d, b}	{b, a, d}
{a, h}	{a}	{a}
{b, d, g}	{b, d}	{b, d}

Table 3.1.1 Database and Frequent items

In the above example when the Dataset is scanned for the first time it counts the frequency of each item. Then we have eliminated the items which have frequency less than the Minimum Support Threshold. Here we have assumed the minimum support threshold as 3. After eliminating the infrequent items we have only the Frequent Itemset. Then we have sorted the items according to their support counts.

3.2 Construction of Prefix Tree**Figure 3.2.1 FP-Tree of first transaction**

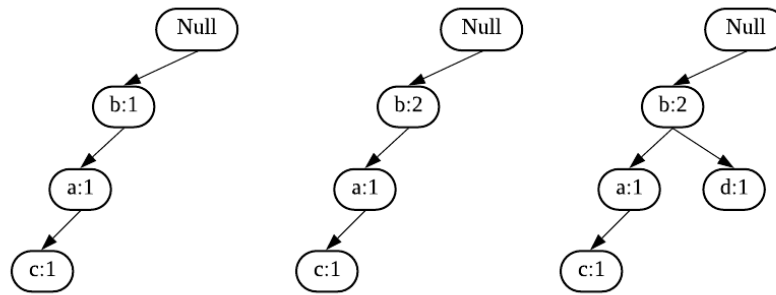


Figure 3.2.2 FP-Tree of second transaction

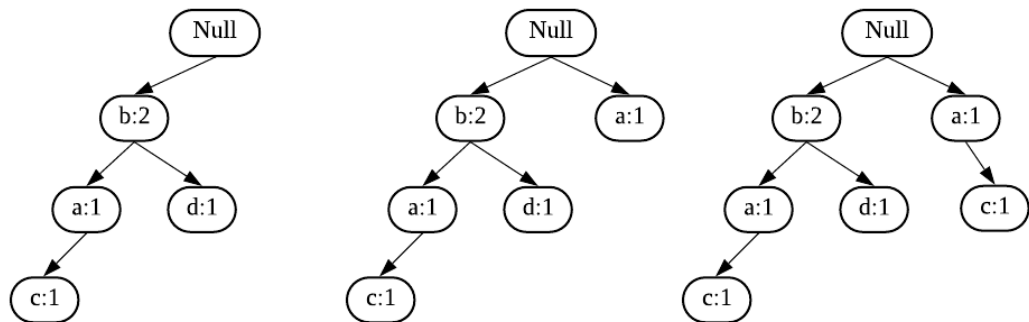


Figure 3.2.3 FP-Tree of third transaction

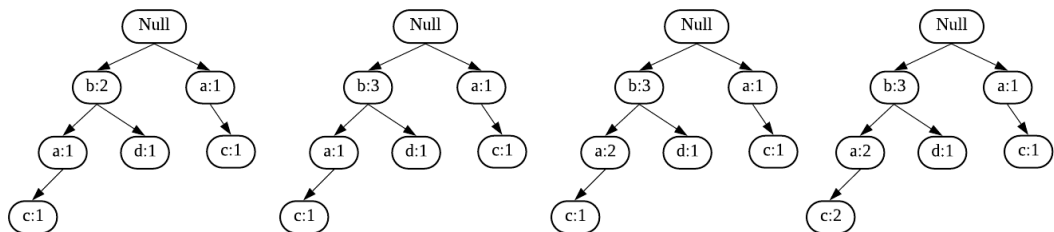


Figure 3.2.4 FP-Tree of fourth transaction

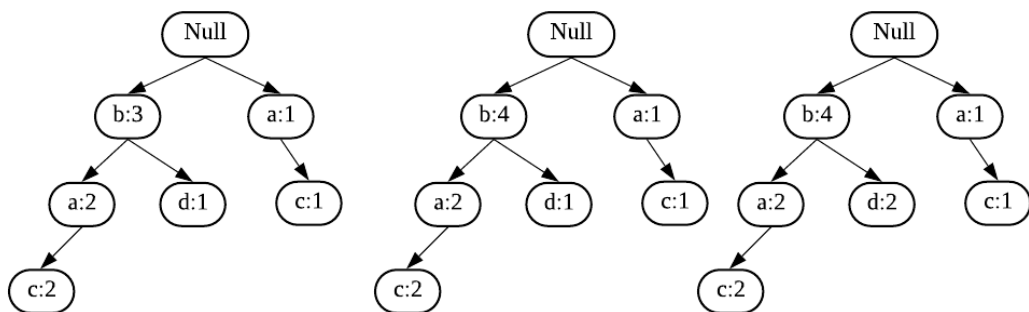


Figure 3.2.5 FP-Tree of fifth transaction

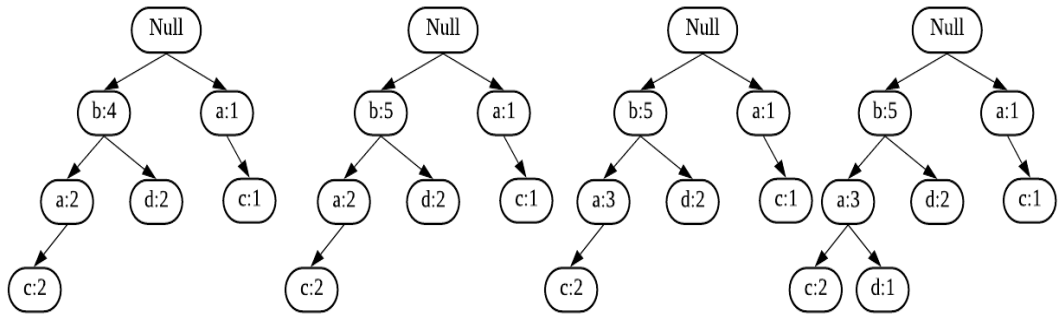


Figure 3.2.6 FP-Tree of sixth transaction

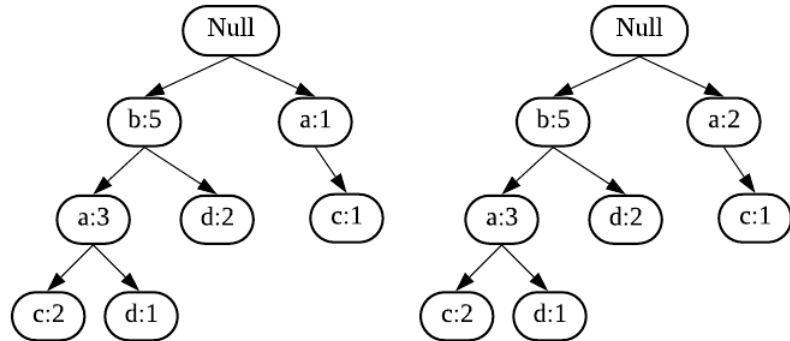


Figure 3.2.7 FP-Tree of seventh transaction

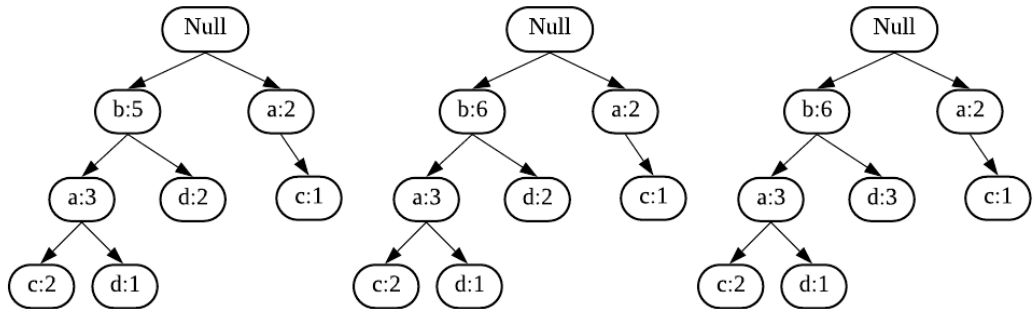


Figure 3.2.8 FP-Tree of eighth transaction

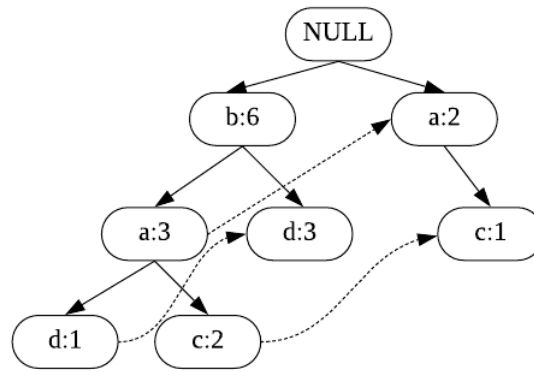


Figure 3.2.9 Prefix Tree (Final FP-Tree)

3.3 Dynamic Prefix Tree

After constructing the prefix tree from the database, the Dynamic Prefix Tree (DPT) algorithm recursively processes each node within the tree to extract frequent itemsets and manage memory efficiently. For each node (N), DPT initiates by copying the subtree rooted at (N) into its right sibling nodes. This copying process ensures that the algorithm can continue processing the prefix tree while maintaining the original structure intact.

Following the copying step, DPT evaluates the support count (N.counter) associated with node (N). If the support count is equal to or greater than the specified minimum support threshold (min_sup), DPT generates an itemset composed of the items along the path from the root node to (N), with (N.counter) representing its support. This step enables the extraction of frequent itemsets directly from the prefix tree.

Subsequently, DPT recursively processes all child nodes of (N), repeating the aforementioned steps for each child node. Once all child nodes have been processed, DPT releases the memory occupied by node (N). If the support count (N.counter) is less than the minimum support threshold (min_sup), indicating that the associated itemset is infrequent, DPT frees the entire subtree rooted at (N) after copying it. This pruning process ensures that only relevant portions of the prefix tree are retained, optimizing memory usage.

The structure of the prefix tree ensures that the support counts of all descendant nodes of (N) are less than or equal to (N.counter), and subsequently, less than the minimum support threshold (min_sup) if (N.counter) is less than (min_sup). This property enables DPT to determine that all itemsets represented by descendant nodes of (N) are infrequent if (N.counter) fails to meet the minimum support threshold. Consequently, DPT can safely free the subtree rooted at (N), eliminating unnecessary memory overhead.

The example discussed in the section 3.2.2 can be implemented by the Dynamic prefix tree algorithm as below.

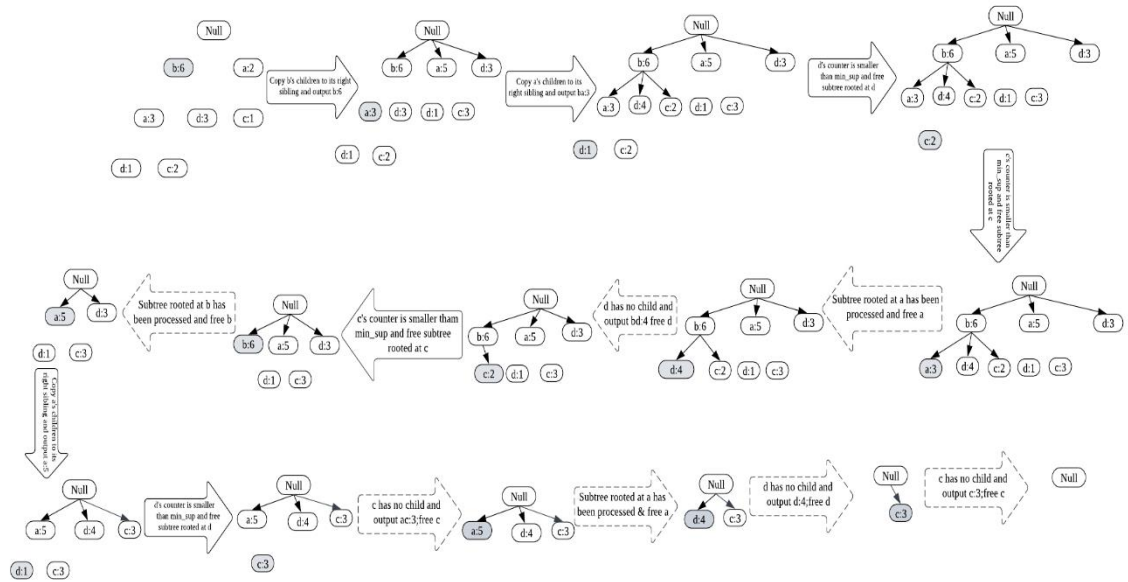


Figure 3.3.1 Dynamic Prefix Tree

The above Figure 3.3.1 depicts a process of consolidating and optimizing the structure based on a minimum support threshold. This approach is commonly used in data mining and machine learning techniques to identify the most significant decision paths while removing less impactful ones.

The minimum support threshold is set to 3 in this case. This means that any node in the decision tree with a node counter or count value less than 3 will be considered insufficient and subsequently removed or "freed" from the tree. The rationale behind

this is to focus on the more frequently occurring patterns while eliminating those that are relatively rare or insignificant.

For nodes with a count value greater than or equal to the minimum support of 3, the process involves copying the entire subtree (all connected nodes downstream) and appending it to the right sibling node. If the right sibling node already contains the same item or decision point, the counts are merged. This consolidation step helps to identify and combine duplicate or similar decision paths, streamlining the overall structure.

In the case where the right sibling node does not already contain the same item, a new node is created and added to the right sibling. This ensures that all relevant decision points are preserved, but in a more compact and efficient manner.

The final step in this optimization process is the removal or "freeing" of nodes that have a count value less than the minimum support of 3. By eliminating these less frequently occurring branches, the decision tree becomes more concise and focused, highlighting the most significant decision paths that are supported by the underlying data.

4. IMPLEMENTATION

Program file is **Dynamic.java** this consists of the code which is written using object oriented programming in Java language. This code mainly focuses on generating the Prefix Tree (FP-Tree) then extracting the frequent patterns by traversing the prefix tree by performing the copying and merging of nodes and output the patterns which has count greater than or equal to minimum support value.

Input: Dataset

Output: Frequent patterns

4.1. FUNCTIONALITY

4.1.1. File Input/Output:

The code reads input data from a file containing transactional data. It constructs the prefix tree based on the input data and the specified minimum support threshold. Optionally, the code could be extended to write the results (frequent itemsets) to an output file for further analysis or storage.

4.1.2. Constructing Prefix Tree:

The process of constructing the prefix tree involves iteratively inserting items from each transaction into the tree, filtering out infrequent items, and building paths that represent transactional relationships between items. This results in a hierarchical representation of the dataset, which forms the basis for subsequent frequent itemset mining and analysis.

Object-Oriented Approach for constructing the prefix tree:

- i. **Classes and Objects:** The prefix tree construction leverages object-oriented programming by utilizing classes and objects to represent the nodes of the tree.
- ii. **TreeNode Class:** A `TreeNode` class is defined to encapsulate the properties and behavior of a node in the prefix tree. Each node object contains attributes such as item, count, and children, representing the item associated with the node, its support count, and its child nodes, respectively.

- iii. **Encapsulation:** The `TreeNode` class encapsulates the state and behavior of individual tree nodes, providing abstraction and modularity. This encapsulation allows for easier maintenance and modification of the tree structure.

4.1.2.1. Iterative Construction from Transaction Database:

The construction of a prefix tree from a given database of transactions is achieved iteratively. In this process, each transaction within the database is sequentially processed, with the code iterating through the list of items associated with each transaction. For every item encountered, it is inserted into the prefix tree. This method invocation entails using the `TreeNode` class methods to manage the insertion process effectively. Through this iterative approach, the prefix tree is gradually built, with each transaction contributing to its structure. Each node within the prefix tree is instantiated as an instance of the `TreeNode` class dynamically during the construction process. This dynamic instantiation facilitates efficient memory allocation, adhering to object-oriented programming principles by allocating memory for each node dynamically as new instances of the `TreeNode` class are created.

The iterative construction of the prefix tree from the transaction database involves sequentially processing each transaction's list of items. Utilizing the `TreeNode` class methods, items are inserted into the prefix tree as the code progresses through the database. This incremental construction method ensures that the prefix tree evolves with each transaction, gradually forming its structure. As nodes are dynamically instantiated during this process, memory allocation adheres to object-oriented programming principles. Each node, represented by an instance of the `TreeNode` class, is dynamically allocated memory, ensuring efficient resource management and scalability in handling large transaction datasets.

4.1.2.2. Insertion of Items into the Prefix Tree:

The process of constructing a prefix tree involves the sequential insertion of items from each transaction into the tree. As the code iterates through each transaction, it meticulously adds individual items to the prefix tree, one at a time. Beginning its traversal from the root node of the tree, the code navigates through the tree's structure according to the items present in the transaction. This traversal builds a unique path

from the root node to a leaf node, effectively capturing the transaction's sequence within the prefix tree. The iterative nature of this insertion process ensures that each item from the transaction contributes to the evolving structure of the tree, incrementally forming a comprehensive representation of the transactional data.

Within the prefix tree, each node serves as a representation of either a single item or a set of items, while the edges between nodes signify the transactional relationships between these items. This representation allows for a clear depiction of item associations within transactions. By following the path dictated by the items present in the transaction, the code dynamically constructs a tailored representation of that particular transaction within the prefix tree. This approach not only facilitates efficient storage and retrieval of transactional data but also enables the identification of frequent itemsets and patterns through subsequent analysis of the constructed prefix tree.

4.1.2.3. Filtering Infrequent Items:

During the insertion process, the code incorporates conditional logic to selectively include items into the prefix tree, adhering to a predefined minimum support threshold. This threshold serves as a criterion for determining the significance of an item within the dataset. Infrequent items, characterized by a support count below the minimum threshold, undergo filtration and are consequently excluded from insertion into the prefix tree. Through this implementation of conditional logic, the code ensures that only items meeting the minimum support requirement are considered for inclusion, optimizing the efficiency and relevance of the constructed prefix tree.

This filtering mechanism guarantees that the prefix tree primarily captures relationships between items deemed significant within the dataset. By excluding infrequent items from the tree construction process, the focus remains on representing associations among the most relevant and frequently occurring items. Thus, the prefix tree effectively embodies the underlying transactional patterns and dependencies among the significant elements of the dataset, enabling subsequent analyses to yield meaningful insights into the dataset's structure and behavior.

4.1.2.4. Representation of Transactional Relationships:

Traversal through the prefix tree involves a systematic navigation from one node to another, guided by the edges that symbolize transactional relationships between items. Within this hierarchical structure, each node represents either a single item or a combination of items, while the edges between nodes denote the associations between these items. This arrangement allows for a comprehensive representation of the transactional dependencies present in the dataset, as traversal builds paths from the root to leaf nodes, effectively capturing the intricate relationships between items.

Maintaining the hierarchical structure of the prefix tree is crucial for facilitating efficient traversal and exploration of the dataset. By organizing nodes and their relationships in a hierarchical manner, the prefix tree optimizes the process of frequent itemset mining and association rule generation. This hierarchical organization enables rapid identification of frequent itemsets and patterns within the dataset, empowering subsequent analyses to uncover meaningful insights into transactional behavior and item associations. Thus, the hierarchical structure of the prefix tree serves as a fundamental framework for efficiently exploring and understanding the underlying relationships within the dataset.

4.1.3. Dynamic Prefix Tree (DPT) Algorithm:

The Dynamic Prefix Tree (DPT) algorithm efficiently manages memory and processing by recursively processing each node in the prefix tree, copying and merging subtrees as needed, and selectively retaining relevant portions of the tree. This approach leads to improved efficiency and scalability, making it suitable for frequent itemset mining in large datasets.

Optimized Memory Usage: The DPT algorithm optimizes memory usage by selectively copying and merging subtrees, ensuring that only relevant portions of the prefix tree are retained.

Modular Design: The modular design of the algorithm, implemented using object-oriented principles, allows for efficient processing of large datasets and complex itemset structures.

Abstraction and Encapsulation: Abstraction and encapsulation techniques help hide the internal details of subtree manipulation and processing, leading to improved code maintainability and scalability.

4.1.3.1. Recursive Processing of Prefix Tree:

- i. **Method Invocation:** The DPT algorithm is implemented as a method within the Dynamic class, which orchestrates the processing of each node in the prefix tree. At each node, the algorithm performs specific operations based on the support count of the node and its subtree.
- ii. **Object-Oriented Approach:** The recursive processing of nodes follows an object-oriented approach, where each node in the prefix tree is represented as an instance of the TreeNode class.

4.1.3.2. Copying Subtrees:

Encapsulation of subtree copying functionality is achieved through the implementation of the copySubtree method within the Dynamic class. Leveraging TreeNode objects to represent nodes and subtrees, this method facilitates the copying of entire subtrees rooted at specific nodes within the prefix tree. During node processing, the algorithm intelligently duplicates the subtree rooted at the current node into its right sibling nodes. This strategic copying operation allows the algorithm to preserve relevant segments of the prefix tree while optimizing memory consumption, particularly advantageous for nodes with lower support counts.

The abstraction provided by the subtree copying operation offers a modular and reusable solution for subtree manipulation within the prefix tree. By encapsulating the copying functionality, the internal complexities of the prefix tree are abstracted away, fostering a clearer separation of concerns and enhancing code maintainability. This abstraction enables developers to focus on higher-level logic without delving into the intricate details of subtree manipulation. Moreover, the decision to copy subtrees rather

than retaining them at each node ensures efficient memory utilization, especially beneficial in scenarios where memory resources are constrained or where nodes exhibit varying support counts. Through this abstraction, the algorithm achieves a balance between memory efficiency and structural integrity, enhancing the scalability and performance of the prefix tree construction process.

4.1.3.3. Merging Subtrees for Infrequent Nodes:

Encapsulation of subtree merging functionality is accomplished through the `mergeSubtree` method within the `Dynamic` class. This method encapsulates the logic for merging subtrees associated with infrequent nodes, ensuring a modular and organized approach to subtree management. If the support count of the current node falls below the minimum support threshold, the algorithm invokes the `mergeSubtree` method to consolidate the subtree with its right sibling nodes. Leveraging polymorphism, the method adapts its behavior to handle various cases of subtree merging based on the support count of the current node, thus promoting code flexibility and reusability.

The implementation of `mergeSubtree` embodies the principle of data hiding, as it conceals the intricacies of subtree merging within its confines. This abstraction fosters a clear separation of concerns, allowing developers to focus on high-level logic without being burdened by the internal details of subtree manipulation. Moreover, this encapsulation enhances code maintainability by providing a single, centralized location for subtree merging functionality. By consolidating subtrees associated with infrequent nodes, the algorithm optimizes memory usage, effectively reducing the memory footprint of the prefix tree while retaining pertinent information. This merging operation not only enhances memory efficiency but also preserves the structural integrity of the prefix tree by ensuring that relevant associations between items are maintained.

4.1.3.4. Recursive Processing for Relevant Nodes:

The `DPT` method within the `Dynamic` class employs method recursion to facilitate the recursive processing of pertinent nodes within the prefix tree. When encountering a

node whose support count meets or exceeds the minimum support threshold, the algorithm proceeds to process its child nodes recursively. This recursive approach ensures that only relevant segments of the prefix tree are retained, effectively directing computational resources towards nodes that contribute to the discovery of frequent itemsets. By iteratively traversing through the tree structure and selectively processing nodes based on their support counts, the algorithm optimizes its operations to focus on areas crucial for frequent itemset analysis.

In its recursive journey, the `dpt` method collaborates closely with objects of the `TreeNode` class to navigate child nodes and traverse the prefix tree effectively. This collaboration fosters seamless interaction between the method and the tree nodes, enabling efficient exploration of the dataset. Furthermore, dynamic memory allocation ensures that memory for each node is dynamically assigned as new instances of the `TreeNode` class are created during recursive processing. By dynamically managing memory resources, the algorithm enhances scalability and adaptability, accommodating datasets of varying sizes. Through the recursive processing of relevant nodes, the algorithm enhances efficiency and scalability by avoiding unnecessary computation on infrequent or irrelevant parts of the prefix tree. This streamlined approach contributes to improved performance and resource utilization, making the algorithm well-suited for handling large-scale datasets efficiently.

4.1.4. Memory Management:

Memory management in the prefix tree algorithm follows object-oriented principles such as encapsulation, abstraction, and modularity. Dynamic memory allocation is used to create and manage 'TreeNode' objects, while Java's garbage collection mechanism handles automatic memory deallocation. Additionally, memory usage tracking allows for performance optimization and resource monitoring.

Garbage Collection:

Automatic Memory Management: Memory management in Java is handled implicitly by the garbage collection mechanism. Objects that are no longer referenced by any part of the program are automatically identified and deallocated by the garbage collector.

Finalization: The finalize method in the `TreeNode` class serves as a hook that is called by the garbage collector before an object is reclaimed. This method can be overridden to perform cleanup operations or release resources associated with the node.

4.1.5 Support Count

The code undertakes the crucial task of computing the support count of each item within the database, a fundamental step in identifying frequent items based on a predefined minimum support threshold. These frequent items are efficiently stored in a map data structure, facilitating rapid lookup operations during the construction of the prefix tree. To ensure a clean and maintainable codebase, the logic for support count calculation is encapsulated within a dedicated method or class. This encapsulation shields the internal workings of the support count calculation from the rest of the codebase, enhancing code maintainability by allowing modifications to the calculation logic without impacting other components.

Embracing the principle of modularity, the support count calculation is encapsulated as a separate component within the codebase. This modular approach promotes easier maintenance and testing, as changes or updates to the support count calculation logic can be isolated within its designated module. Furthermore, the abstraction provided by the support count calculation logic abstracts away the intricate details of how support counts are computed, offering a simplified interface for interacting with support count data. This abstraction facilitates ease of use and comprehension for developers interacting with the support count functionality, fostering a more intuitive and manageable codebase. Overall, the code's adherence to encapsulation, modularity, and abstraction principles contributes to its robustness, maintainability, and scalability in handling support count computations for efficient prefix tree construction.

4.1.6 Printing Prefix Tree:

Printing functionality for the prefix tree is encapsulated within a method or class, abstracting the implementation details and promoting modular and reusable code. This encapsulation ensures that the printing logic remains modular and reusable, contributing to enhanced code maintainability and readability. The printing functionality can be implemented either as a method within the `Dynamic` class or as a

separate `printPrefixTree` class dedicated to handling the printing of the tree structure. By adopting this encapsulation approach, the codebase benefits from a clear separation of concerns, allowing developers to focus on distinct functionalities and facilitating independent testing and debugging of the printing logic.

The printing logic involves recursively traversing the prefix tree, commencing from the root node and progressing through each node's child nodes. At each node, the item and its associated support count are printed, followed by the recursive printing of its child nodes. This recursive traversal ensures comprehensive coverage of the prefix tree, effectively capturing its hierarchical structure and item associations. Furthermore, the printing is formatted in a hierarchical manner, with each level of the tree appropriately indented to visually represent the tree's structure. This hierarchical format enhances readability and comprehension, enabling users to discern relationships between nodes and inspect the support counts of each itemset conveniently. Overall, the encapsulation and organization of printing functionality facilitate efficient management and representation of the prefix tree, contributing to the codebase's clarity and usability.

4.2 Attributes:

4.2.1 Dynamic Class:

1. **x:** A static variable of type `TreeNode`, representing the root node of the prefix tree.
2. **minSupport:** An integer variable representing the minimum support threshold.
3. **count:** An integer variable used to count the number of nodes generated.
4. **totalNodes:** An integer variable used to track the total number of nodes generated.
5. **maxNodesDPT:** An integer variable used to track the maximum number of nodes generated in the dynamic prefix tree.

4.2.2 TreeNode Class

1. **item:** A string variable representing the item associated with the node.
2. **count:** An integer variable representing the support count of the item.

3. **children:** A map data structure storing the child nodes of the current node. The keys are strings representing item names, and the values are `TreeNode` objects representing child nodes.

4.3. Experimental Screenshot

The screenshot shows the Eclipse IDE interface. The top menu bar includes File, Edit, Source, Refactor, Navigate, Search, Project, Run, Window, and Help. Below the menu is a toolbar with various icons. The main area displays the 'Console' window, which contains the following output:

```
<terminated> Dynamic [Java Application] C:\Users\jahna.p2\pool\plugins\org.eclipse.justj.open  
Result:[null]:100  
Result:[20, null]:380  
Result:[null]:100  
Result:[47, null]:208  
Result:[null]:100  
Result:[65, null]:112  
Result:[null]:100  
Result:[28, null]:577  
Result:[null]:100  
Result:[16, null]:496  
Result:[null]:100  
Result:[32, null]:113  
Result:[null]:100  
Result:[26, null]:136  
Result:[null]:100  
Result:[18, null]:201  
Result:[null]:100  
Result:[45, null]:202  
Result:[null]:100  
Result:[null]:100  
Result:[null]:100  
Result:[null]:100  
Result:[null]:100  
Result:[null]:100  
Result:[null]:100  
Result:[null]:100  
Result:[null]:100  
Result:[null]:100  
Execution Time (DPT): 2741550900 nanoseconds  
Memory Allocated: 96062464 bytes  
Total number of nodes generated: 58682  
Maximum number of nodes generated in DPT: 128790
```

Figure 4.3.1 Output for Chess Dataset with optimization

The screenshot shows the Eclipse IDE interface with the console window open. The title bar reads "eclipse-workspace - Maximal/src/maximal/Maximal.java - Eclipse IDE". The menu bar includes File, Edit, Source, Refactor, Navigate, Search, Project, Run, Window, and Help. The console output is as follows:

```

<terminated> Maximal [Java Application] C:\Users\jahna\p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full
Result:[null]:100
Result:[20, null]:380
Result:[null]:100
Result:[47, null]:208
Result:[null]:100
Result:[65, null]:112
Result:[null]:100
Result:[28, null]:577
Result:[null]:100
Result:[16, null]:496
Result:[null]:100
Result:[32, null]:113
Result:[null]:100
Result:[26, null]:136
Result:[null]:100
Result:[18, null]:201
Result:[null]:100
Result:[45, null]:202
Result:[null]:100
Result:[null]:100
Result:[null]:100
Result:[null]:100
Result:[null]:100
Result:[null]:100
Result:[null]:100
Result:[null]:100
Result:[null]:100
Result:[null]:100
Result:[null]:100
Result:[null]:100
Result:[null]:100
Execution Time (DPT): 2939057900 nanoseconds
Memory Allocated: 101010432 bytes
Total number of nodes generated: 58682
Maximum number of nodes generated in DPT: 152766

```

Figure 4.3.2 Output for Chess Dataset without optimization

The screenshot shows the Eclipse IDE interface with the console window open. The title bar reads "eclipse-workspace - dynamicPrefix/src/dynamicPrefix/Dynamic.java - Eclipse IDE". The menu bar includes File, Edit, Source, Refactor, Navigate, Search, Project, Run, Window, and Help. The console output is as follows:

```

<terminated> Dynamic [Java Application] C:\Users\jahna\p2\pool\plugins\org.eclipse.justj.open
Result:[100, 39, 23, null]:139
Result:[49, 39, 23, null]:197
Result:[62, 39, 23, null]:238
Result:[307, 39, 23, null]:372
Result:[140, 39, 23, null]:107
Result:[36, 39, 23, null]:149
Result:[136, 39, 23, null]:183
Result:[7, 39, 23, null]:117
Result:[47, 23, null]:2692
Result:[307, 47, 23, null]:133
Result:[68, 47, 23, null]:207
Result:[44, 47, 23, null]:133
Result:[49, 47, 23, null]:122
Result:[9, 47, 23, null]:104
Result:[62, 47, 23, null]:125
Result:[108, 47, 23, null]:134
Result:[125, 47, 23, null]:105
Result:[2, 47, 23, null]:147
Result:[72, 47, 23, null]:129
Result:[36, 47, 23, null]:153
Result:[35, 47, 23, null]:288
Result:[33, 47, 23, null]:197
Result:[7, 47, 23, null]:175
Result:[385, 47, 23, null]:153
Result:[61, 47, 23, null]:146
Result:[346, 47, 23, null]:139
Result:[13, 47, 23, null]:146
Result:[35, 23, null]:3742
Result:[68, 35, 23, null]:155
Result:[44, 35, 23, null]:153
Result:[385, 35, 23, null]:139
Result:[98, 35, 23, null]:164
Result:[77, 35, 23, null]:160
Result:[136, 35, 23, null]:109
Result:[346, 35, 23, null]:113

```

Figure 4.3.3 Output for Accidents Dataset with optimization

```
eclipse-workspace - Maximal/src/maximal/Maximal.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help
Console x
Maximal [Java Application] C:\Users\jahna\p2\pool\plugins\org.eclipse.justi.openjdk.hotspot.jre.full.win32.x86_64.17.0.7.v20230425-1502\jre\bin
Result:[26, 30, 14, 63, 59, 41, 15, 28, 24, 27, 43, 29, 21, 31, 16, 18, 12, 17, null]:112
Result:[53, 26, 30, 14, 63, 59, 41, 15, 28, 24, 27, 43, 29, 21, 31, 16, 18, 12, 17, null]:101
Result:[80, 30, 14, 63, 59, 41, 15, 28, 24, 27, 43, 29, 21, 31, 16, 18, 12, 17, null]:117
Result:[10, 14, 63, 59, 41, 15, 28, 24, 27, 43, 29, 21, 31, 16, 18, 12, 17, null]:143
Result:[46, 10, 14, 63, 59, 41, 15, 28, 24, 27, 43, 29, 21, 31, 16, 18, 12, 17, null]:158
Result:[26, 46, 10, 14, 63, 59, 41, 15, 28, 24, 27, 43, 29, 21, 31, 16, 18, 12, 17, null]:137
Result:[64, 46, 10, 14, 63, 59, 41, 15, 28, 24, 27, 43, 29, 21, 31, 16, 18, 12, 17, null]:113
Result:[80, 14, 63, 59, 41, 15, 28, 24, 27, 43, 29, 21, 31, 16, 18, 12, 17, null]:116
Result:[47, 80, 14, 63, 59, 41, 15, 28, 24, 27, 43, 29, 21, 31, 16, 18, 12, 17, null]:111
Result:[38, 59, 41, 15, 28, 24, 27, 43, 29, 21, 31, 16, 18, 12, 17, null]:111
Result:[80, 38, 59, 41, 15, 28, 24, 27, 43, 29, 21, 31, 16, 18, 12, 17, null]:112
Result:[14, 59, 41, 15, 28, 24, 27, 43, 29, 21, 31, 16, 18, 12, 17, null]:425
Result:[38, 14, 59, 41, 15, 28, 24, 27, 43, 29, 21, 31, 16, 18, 12, 17, null]:276
Result:[10, 38, 14, 59, 41, 15, 28, 24, 27, 43, 29, 21, 31, 16, 18, 12, 17, null]:115
Result:[47, 10, 38, 14, 59, 41, 15, 28, 24, 27, 43, 29, 21, 31, 16, 18, 12, 17, null]:125
Result:[30, 38, 14, 59, 41, 15, 28, 24, 27, 43, 29, 21, 31, 16, 18, 12, 17, null]:180
Result:[46, 30, 38, 14, 59, 41, 15, 28, 24, 27, 43, 29, 21, 31, 16, 18, 12, 17, null]:119
Result:[10, 14, 59, 41, 15, 28, 24, 27, 43, 29, 21, 31, 16, 18, 12, 17, null]:185
Result:[30, 10, 14, 59, 41, 15, 28, 24, 27, 43, 29, 21, 31, 16, 18, 12, 17, null]:125
Result:[47, 30, 10, 14, 59, 41, 15, 28, 24, 27, 43, 29, 21, 31, 16, 18, 12, 17, null]:155
Result:[64, 30, 10, 14, 59, 41, 15, 28, 24, 27, 43, 29, 21, 31, 16, 18, 12, 17, null]:119
Result:[46, 10, 14, 59, 41, 15, 28, 24, 27, 43, 29, 21, 31, 16, 18, 12, 17, null]:132
Result:[64, 46, 10, 14, 59, 41, 15, 28, 24, 27, 43, 29, 21, 31, 16, 18, 12, 17, null]:117
Result:[47, 46, 10, 14, 59, 41, 15, 28, 24, 27, 43, 29, 21, 31, 16, 18, 12, 17, null]:130
Result:[30, 14, 59, 41, 15, 28, 24, 27, 43, 29, 21, 31, 16, 18, 12, 17, null]:156
Result:[46, 30, 14, 59, 41, 15, 28, 24, 27, 43, 29, 21, 31, 16, 18, 12, 17, null]:104
Result:[80, 46, 30, 14, 59, 41, 15, 28, 24, 27, 43, 29, 21, 31, 16, 18, 12, 17, null]:115
Result:[80, 30, 14, 59, 41, 15, 28, 24, 27, 43, 29, 21, 31, 16, 18, 12, 17, null]:154
Result:[22, 41, 15, 28, 24, 27, 43, 29, 21, 31, 16, 18, 12, 17, null]:119
Result:[14, 22, 41, 15, 28, 24, 27, 43, 29, 21, 31, 16, 18, 12, 17, null]:127
Result:[44, 14, 22, 41, 15, 28, 24, 27, 43, 29, 21, 31, 16, 18, 12, 17, null]:106
Result:[25, 15, 28, 24, 27, 43, 29, 21, 31, 16, 18, 12, 17, null]:26228
Result:[59, 25, 15, 28, 24, 27, 43, 29, 21, 31, 16, 18, 12, 17, null]:22596
```

Figure 4.3.4 Output for Accidents Dataset without optimization

```
eclipse-workspace - dynamicPrefix/src/dynamicPrefix/Dynamic.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help
Console x
<terminated> Dynamic [Java Application] C:\Users\jahna\p2\pool\plugins\org.eclipse.justi.openjdk
Result:[null]:100
Result:[null]:100
Result:[null]:100
Result:[null]:100
Result:[null]:100
Result:[null]:100
Result:[null]:100
Result:[null]:100
Result:[null]:100
Result:[null]:100
Result:[null]:100
Result:[null]:100
Result:[null]:100
Result:[null]:100
Result:[null]:100
Result:[null]:100
Result:[null]:100
Result:[null]:100
Result:[null]:100
Result:[null]:100
Execution Time (DPT): 1722755300 nanoseconds
Memory Allocated: 208638976 bytes
Total number of nodes generated: 499301
Maximum number of nodes generated in DPT: 164228
```

Figure 4.3.5 Output for T1OI4D100K Dataset with optimization

speedup. Overall, these endeavors underscored the inherent difficulty in further optimizing an already highly refined algorithm like the Dynamic Prefix Tree. Despite the setbacks encountered, the exploration process provided valuable insights into the algorithm's underlying complexities and highlighted the need for innovative solutions to address future optimization challenges in dynamic tree-based data structures.

4.4.1 Reducing the DPT calls

In the attempt to minimize the number of Dynamic Prefix Tree (DPT) calls by introducing a global counter for each item and reducing it upon emitting a pattern with its count, a significant challenge arose due to the interplay between the global counter and the minimum support condition. While the global counter initially seemed like a promising solution to track the occurrence of each item across the dataset, its decrementing nature led to unintended consequences. As patterns are emitted and their counts are subtracted from the global counter, certain items might fall below the minimum support threshold, thus violating the condition and potentially excluding valid patterns from being further explored. This scenario occurs because the global counter's decrementing behavior disregards the context in which an item contributes to the support of a pattern; it treats all occurrences of an item equally, irrespective of their significance within specific patterns or transactions.

Furthermore, the reliance on a single global counter for each item across the entire dataset overlooks the inherent variability in item support levels within different transaction contexts. Items may exhibit varying degrees of importance or relevance across transactions, and a blanket decrementing of their global counters fails to capture this nuanced relationship. Consequently, patterns that may be crucial within certain transaction subsets might be prematurely pruned due to their cumulative counts falling below the minimum support threshold at a global level. This limitation underscores the importance of context-aware support calculations and the need for a more nuanced approach to tracking item occurrences within the DPT algorithm. Ultimately, while the introduction of a global counter aimed to streamline the DPT process, its oversimplified implementation failed to adequately account for the intricate dynamics of item support within transactional data, resulting in the inability to effectively reduce the number of DPT calls.

For the example described in the section 3, the process of adding global counter goes as below:

a	b	c	d	e	f	g	h
5	6	3	4	2	1	2	2

Table 4.4.1.1: Global counter of each item

The total number of DPT calls are 14 and the Minimum Support is set to 3. Since the minimum support is 3 the items e, f, g, h are eliminated first during the construction of prefix tree. So the remaining items are a:5, b:6, c:3, d:4 and the frequent patterns obtained from the prefix tree are b:6, ab:3, bd:4, a:5, ac:3, d:4, c:3.

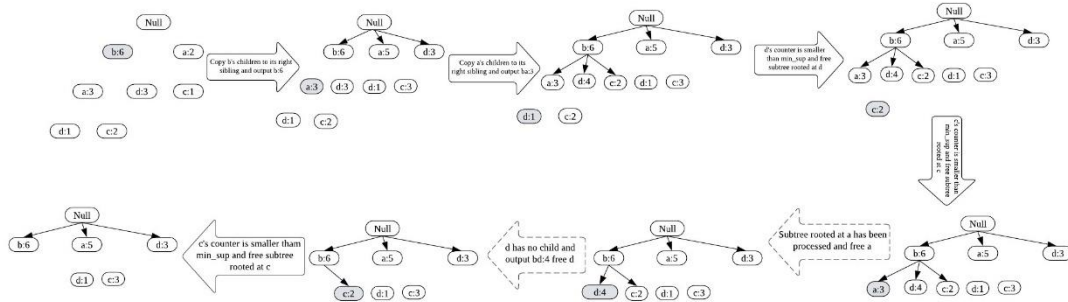


Figure 4.4.1.1 Reducing the DPT calls

When the first pattern b:6 is emitted the global counter of b is reduced to 0. Similarly when the second pattern ab:3 is emitted the global counter of a is reduced by 3. When d:1 node is freed the d's counter is reduced by 1 so the global counter of d is 3 after the node is freed. When the pattern bd:4 is emitted the condition fails as the global counter of d is 3 but the pattern which is to be emitted is bd with counter 4. So the condition fails and bd:4 will never be emitted.

We attempted to reduce the DPT calls by adding global counter so that when an infrequent node is emitted and global counter is reduced, which doesn't satisfy the condition. So our attempt to add global counter and reduce the nodes have failed as we

thought the items have single occurrence but in reality, the dataset have multiple occurrences of the same item.

4.4.2 Reducing the Nodes

In the attempt to refine the Dynamic Prefix Tree (DPT) algorithm by introducing a global counter for each item and adjusting it based on node creation and merging, a critical oversight emerged regarding the interaction between the global counter and the minimum support condition. While the approach sought to dynamically track the support of each item across the dataset, the subsequent adjustment of the global counter after node copying and merging inadvertently undermined the efficacy of the minimum support condition. By incrementing the global counter following each node operation, the condition for node removal based on minimum support became perpetually satisfied. This perpetual satisfaction of the condition rendered it ineffective in accurately identifying and removing nodes that no longer met the minimum support threshold, thus impeding the desired reduction in the number of nodes within the tree structure.

Moreover, the reliance on a post-operation adjustment of the global counter failed to capture the contextual relevance of item occurrences within specific patterns or transactions. The incrementation of the global counter across all nodes disregarded the nuanced variations in item support levels within different transaction subsets, leading to an oversimplified representation of item significance. Consequently, patterns that might have become infrequent or insignificant within certain transactional contexts were retained within the tree structure, despite their diminished relevance in the overall dataset. This oversight underscores the importance of context-aware support calculations and the need for a more nuanced approach to node management within the DPT algorithm. Ultimately, the attempted refinement, while well-intentioned, highlighted the complexity of balancing global item tracking with contextual support evaluation, necessitating further exploration of alternative strategies to optimize the DPT algorithm effectively.

a	b	c	d
5	6	3	4

Table 4.4.2.1 Global Counter

a	b	c	d
8	6	4	7

Table 4.4.2.2 Global count of first DPT call

a	b	c	d
8	6	7	9

Table 4.4.2.3 Global count of second DPT call

a	b	c	d
8	6	7	8

Table 4.4.2.4 Global count of third DPT call

a	b	c	d
8	6	5	8

Table 4.4.2.5 Global count of fourth DPT call

a	b	c	d
5	6	5	8

Table 4.4.2.6 Global count of fifth DPT call

a	b	c	d
5	6	5	4

Table 4.4.2.7 Global count of sixth DPT call

a	b	c	d
5	6	3	4

Table 4.4.2.8 Global count of seventh DPT call

a	b	c	d
5	0	3	4

Table 4.4.2.9 Global count of eighth DPT call

a	b	c	d
5	0	6	5

Table 4.4.2.10 Global count of ninth DPT call

a	b	c	d
5	0	3	4

Table 4.4.2.11 Global count of tenth DPT call

a	b	c	d
5	0	6	4

Table 4.4.2.12 Global count of eleventh DPT call

a	b	c	d
0	0	3	4

Table 4.4.2.13 Global count of twelfth DPT call

a	b	c	d
0	0	3	0

Table 4.4.2.14 Global count of thirteenth DPT call

a	b	c	d
0	0	0	0

Table 4.4.2.15 Global count of fourteenth DPT call

From the above tables , we can conclude that the global counter after each DPT call changes because of copying and merging of subtrees. As the global counter of each item changes it always satisfies the minimum support condition even after deleting the counter of freed node.

This attempt also fails because of multiple occurrences of items in the dataset.

4.4.3 Finding the Maximal Frequent Itemsets using Dynamic Prefix Tree

The attempt to integrate the Maximum Frequent Itemset (MFI) list algorithm with the Dynamic Prefix Tree (DPT) algorithm for finding maximal itemsets encountered challenges, ultimately resulting in failure. The failure can be attributed to several factors, primarily stemming from the complexity and intricacies involved in synchronizing the two algorithms seamlessly. One of the critical points of failure lies in the process of traversing the MFI list and simplifying the FP-tree. While the MFI list aims to identify maximal frequent itemsets efficiently, the integration with the DPT algorithm introduces additional complexities in managing the tree structure and support counts, leading to discrepancies in the traversal process.

Moreover, the process of copying and merging nodes within the DPT algorithm further complicates the traversal of the MFI tree. The intricate relationships between nodes in the FP-tree, coupled with the dynamic adjustments made during node copying and merging, pose significant challenges in maintaining the integrity of the MFI list. Additionally, the failure to precisely identify the reason behind the integration failure

underscores the inherent complexities and nuances involved in combining distinct algorithms to achieve a unified objective. Overall, the unsuccessful integration highlights the need for a more comprehensive understanding of the intricacies of both algorithms and a more nuanced approach to their synchronization to effectively identify maximal itemsets.

The three attempted approaches to finding maximal itemsets encountered failures stemming from different underlying assumptions and complexities. In the first two attempts, the assumption of single occurrences of items led to inaccuracies in determining support counts, as real-world datasets often contain multiple occurrences of items. This oversight compromised the effectiveness of the algorithms. In the third attempt, the incorporation of both the Maximum Frequent Itemset (MFI) list algorithm and the Dynamic Prefix Tree (DPT) algorithm introduced significant complexity. The challenge of synchronizing and managing the two algorithms within a single framework proved to be overly intricate, ultimately resulting in failure. This highlights the importance of accurately modeling real-world data and the need for a more streamlined and focused approach to algorithmic integration to achieve successful identification of maximal itemsets.

4.5 Dataset

In the realm of frequent itemset mining, the choice of dataset is paramount as it directly influences the efficacy and relevance of the mining algorithms employed. Recognizing this, the project opted to utilize datasets sourced from the FIMI repository, a reservoir of datasets specifically tailored for frequent itemset mining research and experimentation. The repository hosts a diverse array of datasets meticulously curated to represent various real-world scenarios and domains.

The datasets procured from the FIMI repository offer several advantages. Firstly, they encompass a broad spectrum of domains, including retail transactions, market basket analysis, web usage patterns, and more. This diversity ensures that the algorithms developed and evaluated in the project can be tested under a wide range of contexts, thereby enhancing their robustness and applicability. Moreover, the datasets

are typically preprocessed and formatted in a standardized manner, facilitating ease of integration into mining algorithms and enabling seamless experimentation.

To acquire the datasets, the project team navigated the FIMI repository and selected datasets pertinent to the project's objectives and research questions. These datasets were obtained in the form of text files, with each line representing a transaction and individual items within transactions delimited by whitespace or other separators. This straightforward format simplifies the process of data ingestion and preprocessing, allowing for efficient extraction of transactional data for subsequent analysis.

Upon retrieval, the datasets were imported into the project environment using Java's file input/output operations. This involved reading the dataset files and parsing the transactional data to extract relevant information. Additionally, preprocessing steps were undertaken to ensure data quality and compatibility with the frequent itemset mining algorithm implemented in the project. Infrequent items were filtered out, and the transactional data was organized into a suitable format conducive to constructing the prefix tree and conducting subsequent mining operations.

By leveraging datasets from the FIMI repository, the project benefits from standardized, well-documented datasets that have been extensively used in the research community. This not only enhances the reproducibility of the project's findings but also fosters comparability with existing literature and facilitates benchmarking against established algorithms and techniques. Furthermore, the utilization of reputable datasets reinforces the credibility and validity of the project's experimental results, enabling meaningful insights and conclusions to be drawn regarding the performance and effectiveness of the implemented mining algorithm.

5. EXPERIMENTAL SETUP

In the realm of Java development, Eclipse serves as one of the most prominent Integrated Development Environments (IDEs). An Eclipse project encapsulates a collection of resources, such as source code, configuration files, and libraries, organized within a specific directory structure. This structure facilitates seamless management and collaboration throughout the software development lifecycle.

Upon creating a new project in Eclipse, developers can import existing Java classes or create new ones. Importing classes involves incorporating external Java files or libraries into the project's workspace, enabling their utilization within the project's codebase. This allows developers to leverage pre-existing functionality, enhancing productivity and code reusability.

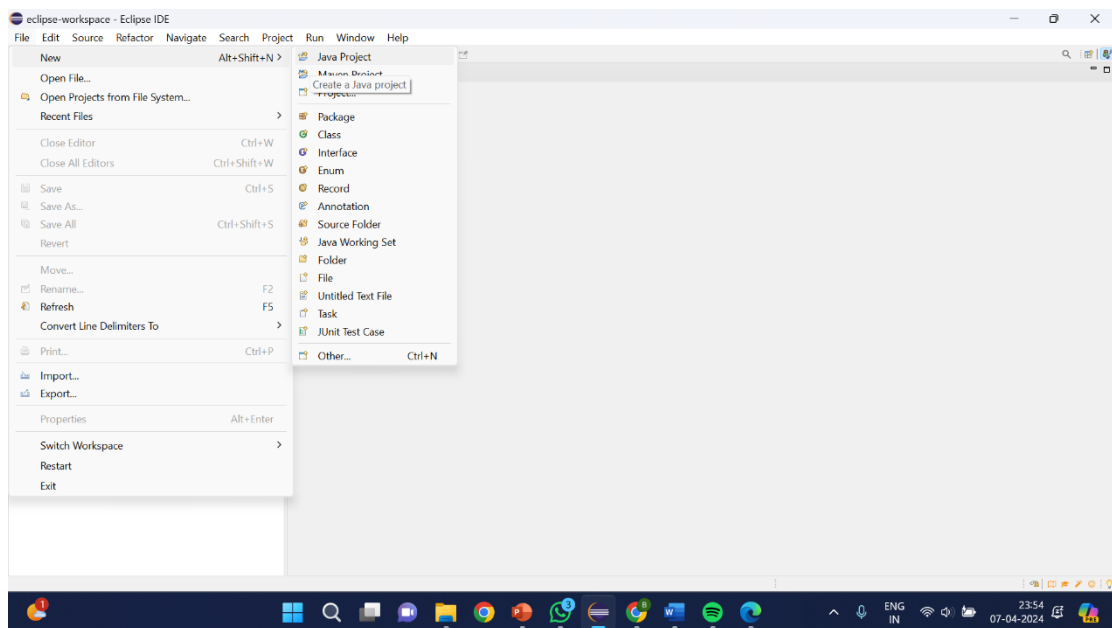


Figure 5.1 Creating the Java Project

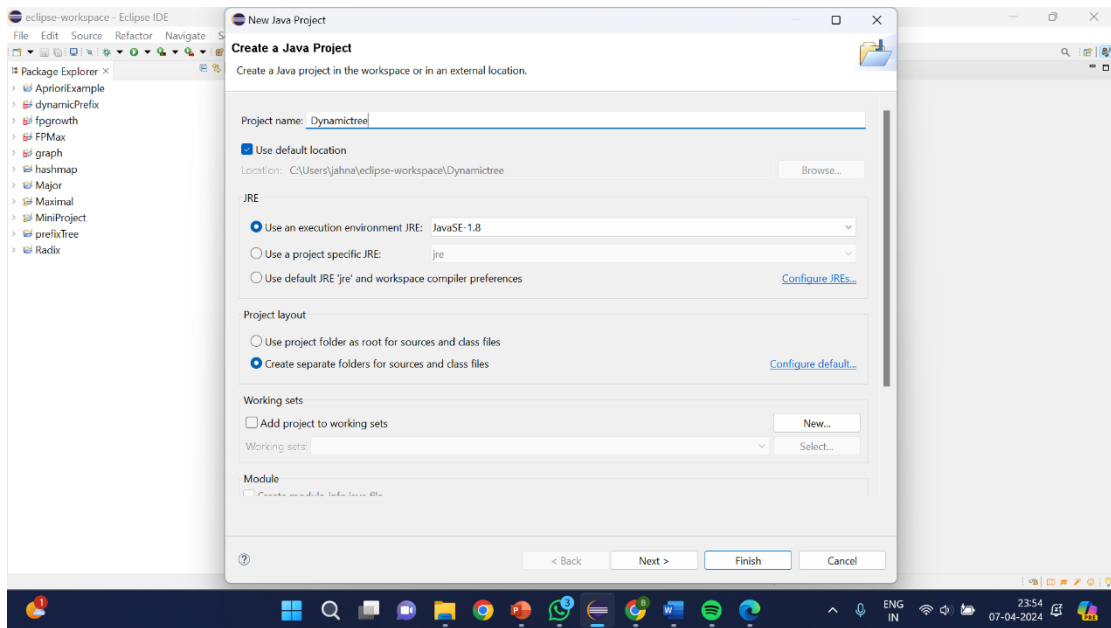


Figure 5.2 Naming the Java Project

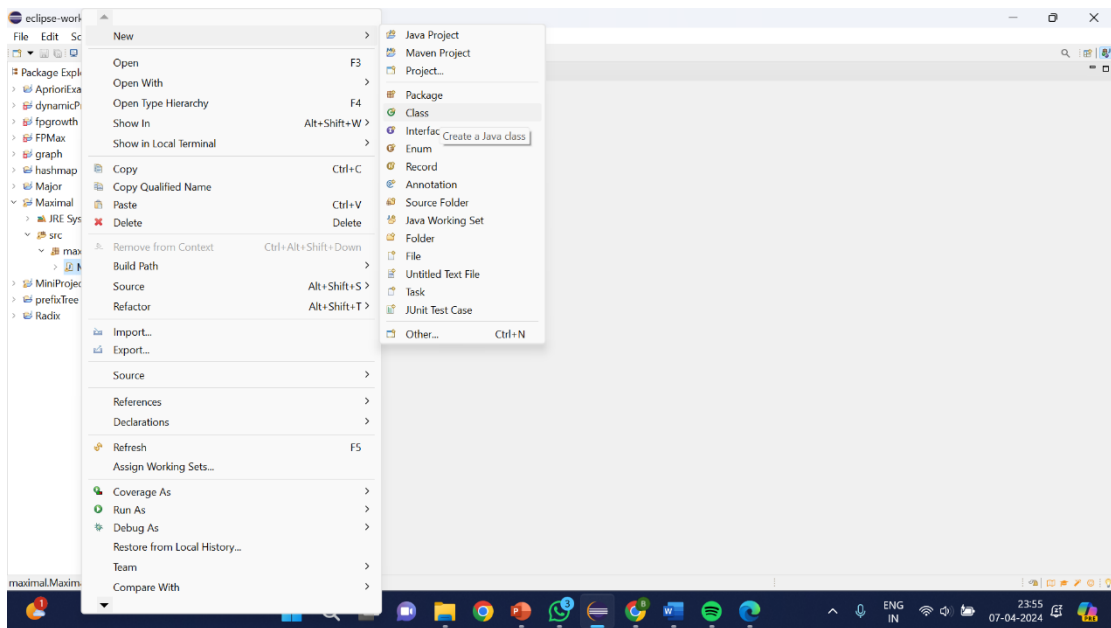


Figure 5.3 Importing the Java classes

Within the project, developers utilize the Eclipse IDE to write Java code efficiently. Eclipse provides various features such as syntax highlighting, auto-completion, and debugging tools, facilitating streamlined coding workflows. Additionally, Eclipse offers powerful refactoring capabilities, enabling developers to efficiently restructure and optimize their codebase.

Eclipse empowers developers to create and manage Java projects effectively by providing a comprehensive set of tools and features tailored to streamline the development process. From importing classes to writing and debugging code, Eclipse serves as a robust platform for Java software development.

The FIMI repository hosts datasets in text file format, which can be downloaded and imported into a Java project. These datasets typically contain structured data that can be utilized for various purposes such as machine learning research or algorithm development. By importing these text files into a Java project, developers can analyze, process, and manipulate the data using Java programming techniques. This integration facilitates seamless integration of external datasets into Java-based applications, enabling developers to leverage the rich ecosystem of Java libraries and tools for data analysis and manipulation.

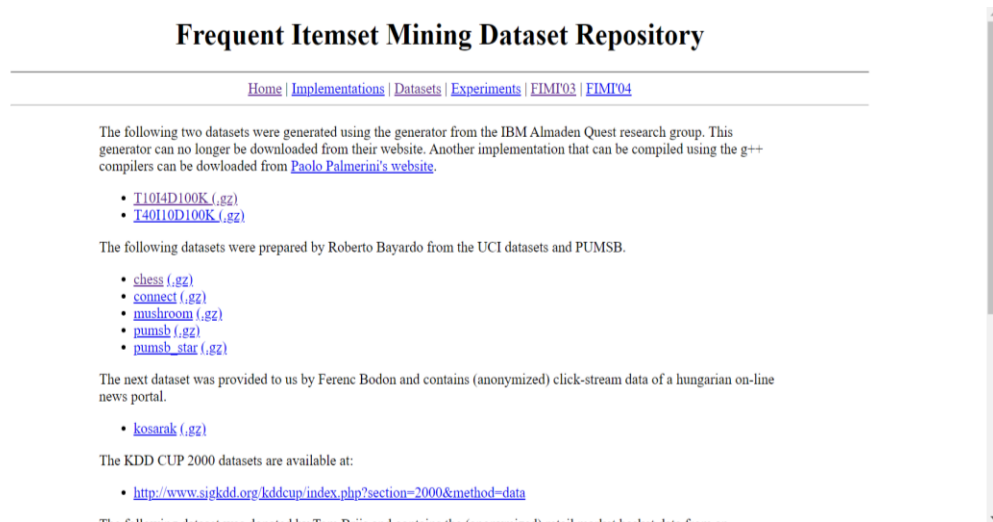
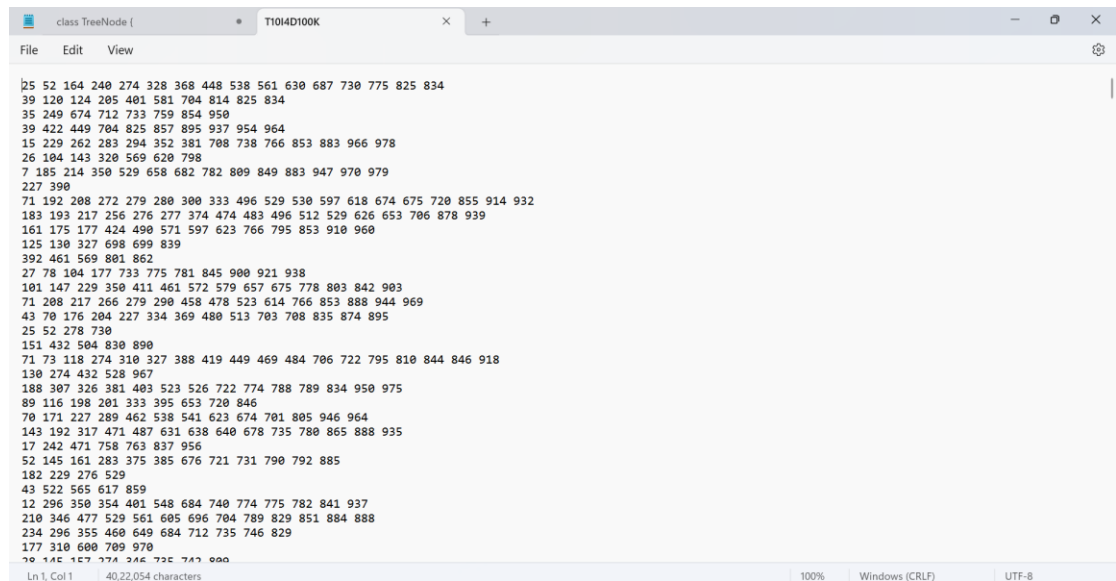


Figure 5.4 FIMI Repository



```
25 52 164 240 274 328 368 448 538 561 630 687 730 775 825 834
39 120 124 205 401 581 704 814 825 834
35 249 674 712 733 759 854 950
39 422 449 704 825 857 895 937 954 964
15 229 262 283 294 352 381 708 738 766 853 883 966 978
26 104 143 320 569 620 798
7 185 214 350 529 658 682 782 809 849 883 947 970 979
227 390
71 192 208 272 279 280 300 333 496 529 530 597 618 674 675 720 855 914 932
183 193 217 256 276 277 374 474 483 496 512 529 626 653 706 878 939
161 175 177 424 490 571 597 623 766 795 853 910 960
125 130 327 698 699 839
392 461 569 801 862
27 78 104 177 733 775 781 845 900 921 938
101 147 229 350 411 461 572 579 657 675 778 803 842 903
71 208 217 266 279 290 458 478 523 614 766 853 888 944 969
43 70 176 204 227 334 369 480 513 703 708 835 874 895
25 52 278 730
151 432 504 830 890
71 73 118 274 310 327 388 419 449 469 484 706 722 795 810 844 846 918
130 274 432 528 967
188 307 326 381 403 523 526 722 774 788 789 834 950 975
89 116 198 201 333 395 653 720 846
70 171 227 289 462 538 541 623 674 701 805 946 964
143 192 317 471 487 631 638 640 678 735 780 865 888 935
17 242 471 758 763 837 956
52 145 161 283 375 385 676 721 731 790 792 885
182 229 276 529
43 522 565 617 859
12 296 350 354 401 548 684 740 774 775 782 841 937
210 346 477 529 561 605 696 704 789 829 851 884 888
234 296 355 460 649 684 712 735 746 829
177 310 600 709 970
70 145 157 274 246 735 747 900
```

Figure 5.5 T1OL4D100K Dataset

5.1 Parameters:

5.1.1 Time Complexity:

Time complexity in Object Oriented Programming refers to the computational efficiency of an algorithm, focusing on how the execution time of an algorithm grows concerning the size of its input data. It provides an estimation of the worst-case scenario for an algorithm's runtime, often denoted using Big O notation ($O()$). Time complexity analysis involves quantifying the number of basic operations performed by the algorithm as a function of the input size. These operations could include comparisons, assignments, arithmetic operations, and iterations through data structures. By analyzing time complexity, developers can predict how an algorithm will scale with larger datasets and identify potential performance bottlenecks.

$$\text{Execution Time} = \text{End Time} - \text{Start Time}$$

5.1.2 Space Complexity:

Space complexity in OOP refers to the amount of memory space required by an algorithm or program to execute, in relation to the size of its input data. It is similar to the concept of time complexity, but instead of analyzing the execution time, space complexity focuses on the memory usage of the algorithm. The space complexity of an algorithm is often denoted using Big O notation ($O()$), which provides an upper bound

on the worst-case memory usage of the algorithm. This analysis helps developers understand how the memory usage of the algorithm scales as the input size increases. In the context of OOP, space complexity involves analyzing the memory consumption of various elements, such as the variables used by the algorithm, including primitive data types and reference types, the data structures employed by the algorithm, such as arrays, linked lists, trees, or graphs, whose space complexity can vary significantly based on their implementation and the specific operations performed on them, any additional data or resources allocated by the algorithm during its execution, such as temporary variables, buffers, or caches, the memory consumed by the creation and management of object instances, including the object's properties and methods, and the memory implications of inheritance hierarchies and polymorphic behavior in OOP, as the algorithm may need to handle different types of objects. Understanding the space complexity of an algorithm is crucial for memory optimization, resource management, scalability and performance, and debugging and profiling, as it provides insights into how the algorithm's memory usage scales with increasing input size, enabling developers to write more efficient and well-performing algorithms, optimize the use of system resources, and ensure the scalability and robustness of their software applications.

$$\text{Memory} = \text{Total Runtime Memory} - \text{Free Runtime Memory}$$

6. DISCUSSION OF RESULTS

The Dynamic Prefix Tree (DPT) algorithm with optimization demonstrates superior performance compared to other algorithms, showcasing the efficiency and effectiveness of the optimization techniques employed. The reduced memory consumption and faster execution time of DPT with optimization highlight its capability to handle large datasets more adeptly. Additionally, the comparison with alternative algorithms such as Apriori and FP-Growth underscores the competitive advantage of DPT in terms of both time and space complexity. These findings emphasize the importance of optimizing algorithms for enhanced performance in real-world applications, particularly when dealing with substantial datasets where efficiency is paramount.

Method	Memory Allocated	Execution Time
Apriori	118310344 bytes	1388430400 nanoseconds
FPGrowth	192346616 bytes	391595800 nanoseconds
DPT Optimization	96062464 bytes	2741550900 nanoseconds
DPT	101010432 bytes	2939057900 nanoseconds

Table 6.1 Chess Dataset

From the above table , we can observe out of the four algorithms tested - Apriori, FP Growth, Dynamic Prefix Tree (DPT), and DPT with optimization - the latter proved to be the most efficient, consuming significantly less memory and time. With optimization techniques, DPT outperformed the others by consuming only 96,062,464 bytes of memory and completing the task in 2,741,550,900 nanoseconds. Consequently, DPT with optimization emerges as the preferred choice for handling large datasets and complex processing tasks.

Method	Memory Allocated	Execution Time
Apriori	592447488 bytes	2423542600 nanoseconds
FPGrowth	72571368 bytes	27753853500 nanoseconds
DPT Optimization	716702440 bytes	2013028000 nanoseconds
DPT	1274542592 bytes	418222068800 nanoseconds

Table 6.2 Accidents Dataset

From the above table , we can observe out of the four algorithms tested - Apriori, FP Growth, Dynamic Prefix Tree (DPT), and DPT with optimization - the latter proved to be the most efficient, consuming significantly less memory and time. With optimization techniques, DPT outperformed the others by consuming only 716702440 bytes of memory and completing the task in 2013028000 nanoseconds. Consequently, DPT with optimization emerges as the preferred choice for handling large datasets and complex processing tasks.

Method	Memory Allocated	Execution Time
Apriori	103286784 bytes	866487400 nanoseconds
FPGrowth	17153736 bytes	10858916200 nanoseconds
DPT Optimization	200369184 bytes	2005798800 nanoseconds
DPT	245338112 bytes	2754644900 nanoseconds

Table 6.3 T10L4D100K Dataset

From the above table , we can observe out of the four algorithms tested - Apriori, FP Growth, Dynamic Prefix Tree (DPT), and DPT with optimization - the latter proved to be the most efficient, consuming significantly less memory and time. With optimization techniques, DPT outperformed the others by consuming only 200369184 bytes of memory and completing the task in 2005798800 nanoseconds. Consequently, DPT with optimization emerges as the preferred choice for handling large datasets and complex processing tasks.

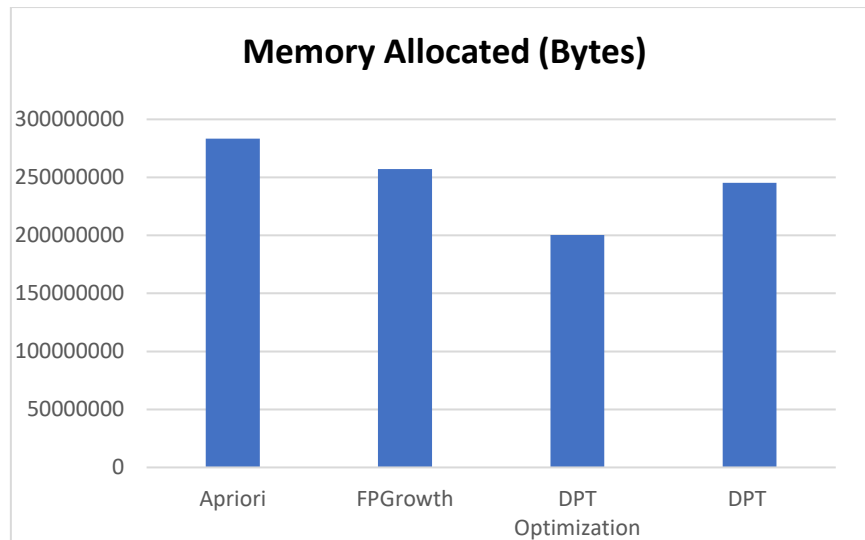


Figure 6.1 Memory Consumed by T10I4D100K dataset

From the above diagram we can draw the conclusions such as both DPT and FPGrowth can be optimized for efficiency, the provided diagram suggests DPT with optimization shines in terms of memory usage. This is evident because the line representing DPT with optimization consistently stays below the line for FPGrowth with optimization, signifying DPT's ability to handle the data while requiring less memory throughout the process.

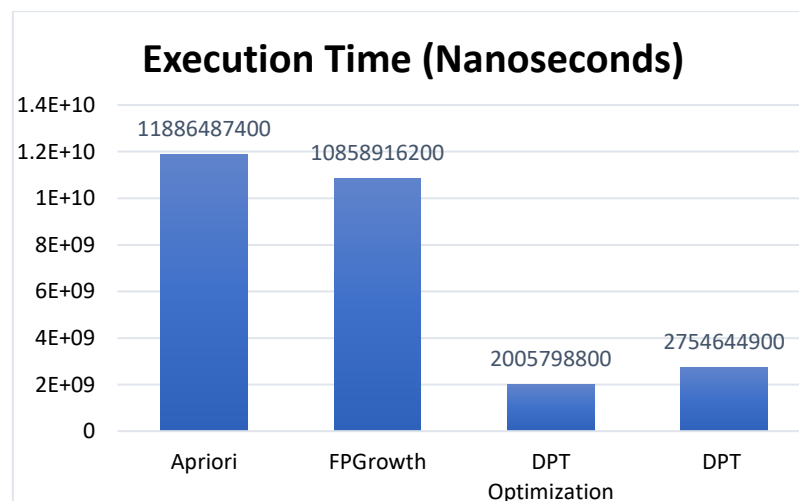


Figure 6.2 Time Consumed by T10I4D100K dataset

From the above diagram , we can obtain the conclusions as DPT with Optimization stands out as the undisputed leader in terms of speed. Its optimized execution time, depicted by the shortest line on the graph, approximately at 200 billion nanoseconds, surpasses Apriori (1.2 trillion nanoseconds), FPGrowth (1.1 trillion nanoseconds), and even the non-optimized DPT (2.7 trillion nanoseconds).

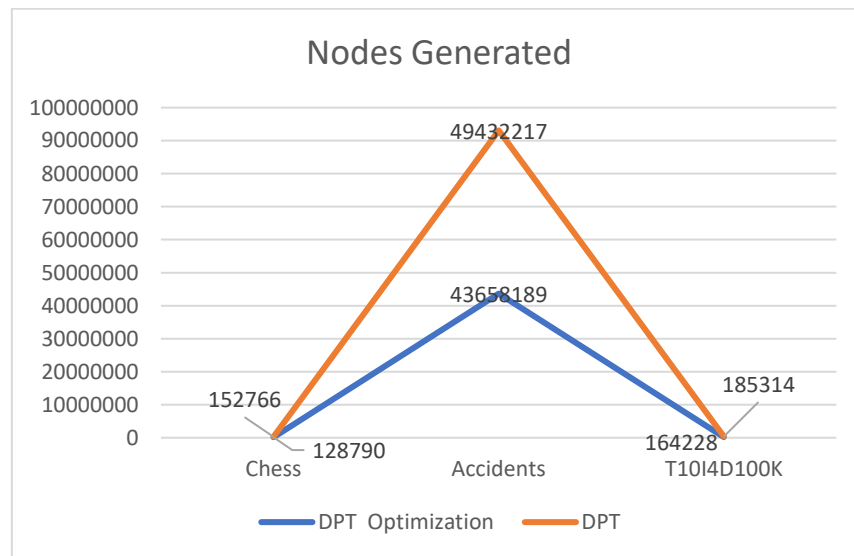


Figure 6.3 Nodes Generated by three different datasets

From the above diagram, we can obtain the conclusions as the graph compares the number of nodes generated by three datasets Chess, Accidents, and T10I4D100K under two conditions, DPT Optimization and DPT. Overall, all the datasets generates significantly more nodes with the DPT condition. Chess, Accidents and T10I4D100K datasets DPT Optimization generates fewer nodes than DPT.

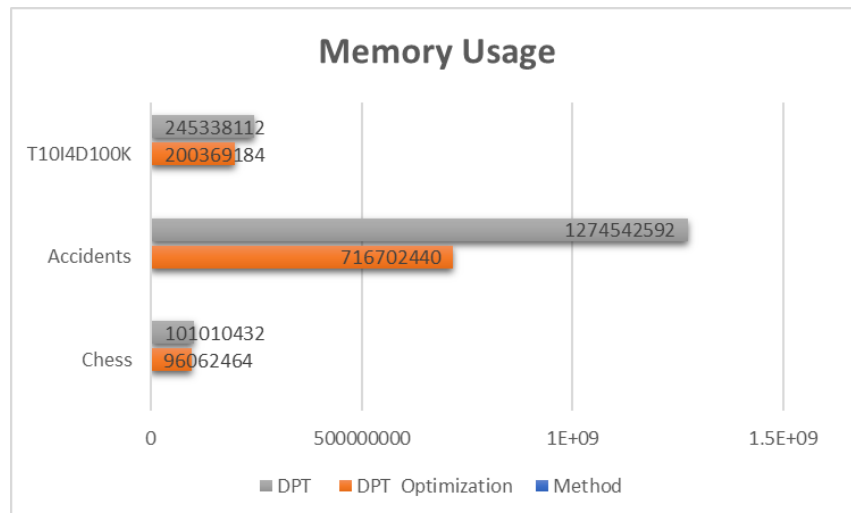


Figure 6.4 Memory Usage of three different datasets

From the above diagram, we can obtain the conclusions that it compares memory usage between two algorithms, likely DPT (Dynamic Prefix Tree) with and without optimization. The y-axis shows memory usage and the x-axis likely represents different datasets. For each dataset, the memory usage of DPT with optimization is lower than DPT without optimization.

6.1 Findings

1) Dynamic prefix tree is easily implemented in object-oriented programming. There are many advantages like:

Firstly, OOP facilitates modularity by organizing the DPT implementation into smaller, manageable classes, simplifying development and maintenance. Secondly, encapsulation hides the internal details of the tree structure, exposing only the necessary interfaces for interaction, thus reducing complexity. Thirdly, inheritance enables the creation of specialized versions of the DPT, promoting code reuse and extensibility. Additionally, polymorphism allows flexibility by treating objects of different classes as instances of a common superclass, enabling seamless integration of various functionalities.

2) The drawback of assuming single occurrences of items in the dataset undermines the reliability of support counts and may lead to inaccurate pattern identification and

premature pruning, ultimately compromising the effectiveness of the DPT algorithm in analyzing the dataset.

3) The complexity introduced by merging and copying subtrees within the Dynamic Prefix Tree (DPT) algorithm, coupled with the complexity of managing multiple algorithms within a unified framework, poses significant challenges in finding maximal itemsets. The process of synchronizing these algorithms becomes difficult to deal, especially considering the dynamic adjustments made during subtree copying and merging. The complexity inherent in managing these operations within a single framework renders the task of finding maximal itemsets using the DPT algorithm unfeasible. This underscores the need for alternative approaches or simpler algorithms to achieve this objective effectively.

6.2 Advantages and Applications:

Advantages:

1. Superior Performance: The DPT algorithm with optimization demonstrated significantly better performance compared to other algorithms like Apriori and FP-Growth. It showed reduced memory consumption and faster execution times, especially when handling large datasets.

2. Efficient Memory Utilization: The optimized version of DPT was able to consume much less memory compared to the non-optimized version and the other algorithms tested. This highlights its ability to efficiently utilize system resources.

3. Faster Processing: The execution time of the DPT algorithm with optimization was substantially faster than the non-optimized version and the other algorithms. This improved speed can be crucial in real-world applications where timely processing of large datasets is required.

Applications:

1. Large-scale Data Processing: The DPT algorithm with optimization is well-suited for handling large datasets and complex processing tasks. Its efficiency in terms of memory usage and execution time makes it a suitable choice for applications dealing with big data.

2. Frequent Itemset Mining: The DPT algorithm is designed for efficient frequent itemset mining, which is a fundamental task in areas like market basket analysis, recommendation systems, and pattern recognition.

3. Association Rule Discovery: The DPT algorithm can be leveraged for discovering association rules between items in large databases, with potential applications in areas like customer behavior analysis, product recommendations, and fraud detection.

7. CONCLUSION

The Dynamic Prefix Tree (DPT) approach presented in this project demonstrates significant improvements over traditional frequent itemset mining algorithms. By utilizing a single dynamically adaptive prefix tree to represent datasets, DPT is able to effectively reduce the memory overhead and processing costs associated with methods like FP-growth and Apriori, which require multiple data structures and multiple passes through the data. This innovative technique enhances the overall speed and efficiency of the frequent itemset mining process by eliminating the need for unnecessary scans and tree constructions. Despite attempts to further optimize the DPT algorithm through methods such as reducing the number of nodes and DPT calls, as well as trying to directly generate maximal itemsets using the DPT structure, the experiments showed that the DPT approach was already highly optimized and efficient. Reducing the nodes and the DPT calls have failed due to the multiple occurrences of items in the dataset. The copying, merging, and integration of the MFI (Maximal Frequent Itemset) algorithm did not yield any substantial performance improvements, indicating that the DPT algorithm has been carefully designed and engineered to maximize its capabilities.

The strength of the DPT structure lie in its ability to efficiently encode frequently occurring itemsets within the dynamic prefix tree. This minimizes memory requirements and enhances the scalability of the approach, making it well-suited for handling large datasets common in real-world applications. Moreover, the DPT's direct representation of all frequent itemsets, encapsulated within the comprehensive prefix tree, facilitates immediate access to the data for subsequent processing, analysis, and visualization tasks.

8. REFERENCES

- [1] Jun-Feng Qu, Bo Hang, Zhao Wu, Zhong Bo Wu, Qiong Gu and Bo Tang, “Efficient Mining of Frequent Itemsets Using Only One Dynamic Prefix Tree”, January 2020 ,IEEE Access 8:183722183735,DOI:10.1109/ACCESS.2020.3029302
- [2] R. Agrawal, T. Imieliński, and A. Swami, “Mining association rules between sets of items in large databases,” in Proc. ACM SIGMOD Int. Conf. Manage. Data, 1993, pp. 207–216.
- [3] Y. Djenouri, D. Djenouri, A. Belhadi, and A. Cano, “Exploiting GPU and cluster parallelism in single scan frequent itemset mining,” Inf. Sci., vol. 496, pp. 363–377, Sep. 2019.
- [4] F. Guil, “Associative classification based on the transferable belief model,” Knowl.-Based Syst., vol. 182, Oct. 2019, Art. no. 104800.
- [5] S. Gao, M. Zhou, Y. Wang, J. Cheng, H. Yachi, and J. Wang, “Dendritic neuron model with effective learning algorithms for classification, approximation, and prediction,” IEEE Trans. Neural Netw. Learn. Syst., vol. 30, no. 2, pp. 601–614, Feb. 2019.
- [6] X. Luo, M. Zhou, Y. Xia, Q. Zhu, A. C. Ammari, and A. Alabdulwahab, “Generating highly accurate predictions for missing QoS data via aggregating nonnegative latent factor models,” IEEE Trans. Neural Netw. Learn. Syst., vol. 27, no. 3, pp. 524–537, Mar. 2016.
- [7] J. Huang, S. Li, and Q. Duan, “Constructing multicast routing tree for inter-cloud data transmission: An approximation algorithmic perspective,” IEEE/CAA J. Automatica Sinica, vol. 5, no. 2, pp. 514–522, Mar. 2018.

- [8] S. B. Prusty, S. Seshagiri, U. C. Pati, and K. K. Mahapatra, “Sliding mode control of coupled tank systems using conditional integrators,” *IEEE/CAA J. Automatica Sinica*, vol. 7, no. 1, pp. 118–125, Jan. 2020.
- [9] J. M. Luna, P. Fournier-Viger, and S. Ventura, “Frequent itemset mining: A 25 years review,” *Wiley Interdiscip. Rev. Data Mining Knowl. Discovery*, vol. 9, no. 6, p. e1329, 2019.
- [10] R. Aggrawal and R. Srikant, “Fast algorithm for mining association rules in large databases,” in *Proc. VLDB*, 1994, pp. 487–499.
- [11] J. Han, J. Pei, and Y. Yin, “Mining frequent patterns without candidate generation,” in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2000, pp. 1–12.
- [12] J. Han, J. Pei, Y. Yin, and R. Mao, “Mining frequent patterns without candidate generation: A frequent-pattern tree approach,” *Data Mining Knowl. Discovery*, vol. 8, no. 1, pp. 53–87, Jan. 2004.
- [13] G. Gatuha and T. Jiang, “Smart frequent itemsets mining algorithm based on FP-tree and DIFFset data structures,” *TURKISH J. Electr. Eng. Comput. Sci.*, vol. 25, pp. 2096–2107, 2017.
- [14] N. Shahbazi, R. Soltani, J. Gryz, and A. An, “Building FP-tree on the fly: Single-pass frequent itemset mining,” in *Proc. Int. Conf. Mach. Learn. Data Mining Pattern Recognit.* Cham, Switzerland: Springer, 2016, pp. 387–400.
- [15] L. Wang, J. Meng, P. Xu, and K. Peng, “Mining temporal association rules with frequent itemsets tree,” *Appl. Soft Comput.*, vol. 62, pp. 817–829, Jan. 2018.
- [16] P. Fournier-Viger, J. C.-W. Lin, B. Vo, T. T. Chi, J. Zhang, and H. B. Le, “A survey of itemset mining,” *Wiley Interdiscipl. Rev., Data Mining Knowl. Discovery*, vol. 7, no. 4, p. e1207, Jul. 2017.

- [17] G. Grahne and J. Zhu, "Fast algorithms for frequent itemset mining using FP-trees," *IEEE Trans. Knowl. Data Eng.*, vol. 17, no. 10, pp. 1347–1362, Oct. 2005.
- [18] G. Liu, H. Lu, J. X. Yu, W. Wei, and X. Xiao, "AFOPT: An efficient implementation of pattern growth approach," in *Proc. ICDM Workshop FIMI*, 2003, pp. 1–10.
- [19] X. Chen, L. Li, Z. Ma, S. Bai, and F. Guo, "F-miner: A new frequent itemsets mining algorithm," in *Proc. IEEE Int. Conf. e-Bus. Eng. (ICEBE)*, Oct. 2006, pp. 466–472.
- [20] M. El-Hajj and O. R. Zaiane, "COFI-tree mining: A new approach to pattern growth with reduced candidacy generation," in *Proc. ICDM Workshop FIMI*, 2003. [Online]. Available: <http://www.ceur-ws.org/Vol-90/>.
- [21] M. Adnan and R. Alhajj, "DRFP-tree: Disk-resident frequent pattern tree," *Int. J. Speech Technol.*, vol. 30, no. 2, pp. 84–97, Apr. 2009.
- [22] Frequent Pattern Mining Algorithms for Finding Associated Frequent Patterns for Data Streams: A Survey Shamila Nasreen^a, Muhammad Awais Azamb , Khurram Shehzada , Usman Naeemc , Mustansar Ali Ghazanfara, *The 5th International Conference on Emerging Ubiquitous Systems and Pervasive Networks (EUSPN-2014)*.
- [23] H. Mannila, R. Srikant, H. Toivonen, and A. Inkeri R. Agrawal, "Fast Discovery of Association Rules," in *Advances in Knowledge Discovery and Data Mining*, 1996, pp. 307-328.
- [24] M. Chen, and P.S. Yu J.S. Park, "An Effective Hash Based Algorithm for Mining Association Rules," in *ACM SIGMOD Int'l Conf. Management of Data*, May, 1995.

- [25] R. Motwani, J.D. Ullman, and S. Tsur S. Brin, "Dynamic Itemset Counting And Implication Rules For Market Basket Data," ACM SIGMOD, International Conference on Management of Data, vol. 26, no. 2, pp. 55–264, 1997.
- [26] C. Hidber, "Online Association Rule Mining," ACM SIGMOD International Conference on Management of Data, vol. 28, no. 2, pp. 145–156, 1999.
- [27] Mohammed J. Zaki, "Scalable Algorithms for Association Mining," IEEE TRANSACTIONS ON KNOWLEDGE AND DATA ENGINEERING, pp. 372-390, 2002.
- [28] WeeKeong, YewKwong Amitabha Das, "Rapid Association Rule Mining," in Information and Knowledge Management, Atlanta, Georgia, 2001, pp. 474-481.