DATA MINING ON ARCHITECTURE SIMULATION

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

ENGİN MADEN

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

FEBRUARY 2010

Approval of the thesis:

DATA MINING ON ARCHITECTURE SIMULATION

submitted by **ENGİN MADEN** in partial fulfilment of the requirements for the degree of **Master of Science in Computer Engineering Department, Middle East Technical University** by,

Prof. Dr. Canan Özgen ______________________
Dean, Graduate School of **Natural and Applied Sciences**

Prof. Dr. Müslim Bozyiğit ______________________

Head of Department, **Computer Engineering**

Asst.Prof. Dr. Pınar Şenkul ______________________

Supervisor, **Computer Engineering Dept., METU**

Examining Committee Members:

Assoc. Prof. Dr. Ali Doğru ______________________
Computer Engineering Dept., METU

Asst. Prof. Dr. Pınar Şenkul ______________________
Computer Engineering Dept., METU

Asst. Prof. Dr. Tolga Can, ______________________
Computer Engineering Dept., METU

Asst. Prof. Dr. Tuğba Taşkaya Temizel ______________________
Informatics Institue, METU

Dr. Ayşenur Birtürk ______________________
Computer Engineering Dept., METU

Date: ______________________

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last name : ENGİN MADEN

Signature **:**

ABSTRACT

DATA MINING ON ARCHITECTURE SIMULATION

Maden, Engin
M.S., Department of Computer Engineering
Supervisor : Asst. Prof. Dr. Pınar Şenkul

February 2010, 120 pages

Data mining is the process of extracting patterns from huge data. One of the branches in data mining is mining sequence data and here the data can be viewed as a sequence of events and each event has an associated time of occurrence. Sequence data is modelled using episodes and events are included in episodes.

The aim of this thesis work is analysing architecture simulation output data by applying episode mining techniques, showing the previously known relationships between the events in architecture and providing an environment to predict the performance of a program in an architecture before executing the codes. One of the most important points here is the application area of episode mining techniques. Architecture simulation data is a new domain to apply these techniques and by using the results of these techniques making predictions about the performance of programs in an architecture before execution can be considered as a new approach. For this purpose, by implementing three episode mining techniques which are WINEPI approach, non-overlapping occurrence based approach and MINEPI approach a data mining tool has been developed. This tool has three main components. These are data pre-processor, episode miner and output analyser.

Keywords: Sequential Pattern Mining, Frequent Episode Mining, Architecture Simulation

ÖZ

MİMARİ BENZETİMİ ÜZERİNDE VERİ MADENCİLİĞİ

Maden, Engin
Yükseklisans, Bilgisayar Mühendisliği Bölümü
Tez Yöneticisi : Yrd. Doç. Dr. Pınar Şenkul

Şubat 2010, 120 sayfa

Veri madenciliği, büyük verilerden örüntüler çıkartma sürecidir. Veri madenciliği dallarından birisi de ardışık veriler üzerinde işlem yapılmasıdır ve burada veri olaylar dizisi olarak görülür ve her bir veri bir meydana gelme zamanı değerine sahiptir. Ardışık veri, bölümler kullanılarak modellenir ve olaylar bu bölümler içinde yer alır.

Bu tezin amacı, bölüm madenciliği tekniklerini kullanarak mimari benzetimi çıktılarını analiz etmek, mimarideki olaylar arasında bilinen ilişkileri göstermek ve bir mimari üzerinde bir programın kodlarını çalıştırmadan önce performansı ile ilgili tahmin yapmak için bir ortam hazırlamaktır. Buradaki en önemli noktalardan birisi bölüm madenciliği yöntemlerinin uygulama alanıdır. Mimari benzetimi verileri, bu yöntemlerin uygulanması için yeni bir alandır ve bu yöntemlerle elde edilen sonuçlar kullanılarak bir mimari üzerinde bir programı çalıştırmadan önce performansı hakkında tahminler yapılması yeni bir yaklaşım olarak değerlendirilebilir. Bu amaçla bölüm madenciliğine ilişkin üç yöntem olan WINEPI yaklaşımı, örtüşmeyen oluş temelli yaklaşım ve MINEPI yaklaşımının gerçekleştirilmesiyle bir veri madenciliği aracı geliştirilmiştir. Bu aracın üç temel bileşeni mevcuttur ve bunlar veri ön işleyicisi, bölüm madencisi ve çıktı analizcisidir.

Anahtar Kelimeler: Ardışık Örüntü Madenciliği, Sık Görülen Bölüm Madenciliği ,
Mimari Benzetimi

To My Father

ACKNOWLEDGEMENTS

First of all, I would like to thank to my thesis advisor, Asst. Prof. Dr. Pınar Şenkul, for her guidance, support and motivation she provided throughout my research. Also, I would like to thank to Dr. Soner Önder and Dr. Nilüfer Önder from MTU (Michigan Technological University, Department of Computer Science) for their suggestions, comments and contributions. I would like to thank to TÜBITAK (The Scientific and Technological Research Council of Turkey) for their support. I also very much like to thank to my family.

TABLE OF CONTENTS

ABSTRACT....iv
ÖZ....vi
ACKNOWLEDGEMENTS....ix
TABLE OF CONTENTS....x
LIST OF FIGURES....xii
CHAPTERS
1 INTRODUCTION....1
2 AN OVERVIEW ABOUT DATA MINING....4
2.1 About Data Mining....4
2.1.1 Data, Information and Knowledge....4
2.1.2 Definition of Data Mining....5
2.2 Association Rule Mining....7
2.3 FIM (Frequent Itemset Mining)....9
2.4 Apriori Algorithm....11
3 BACKGROUND....13
3.1 Sequential Pattern Mining....13
3.2 Frequent Episode Mining....15
3.2.1 Event Sequences and Episodes....17
3.2.1.1 Event Sequences....17
3.2.1.2 Episodes....19
3.2.2 Algorithms for WINEPI Approach....21
3.2.3 A Fast Algorithm For Finding Frequent Episodes In Event Streams (Non-overlapping Approach)....33
3.2.3.1 Algorithm For Serial Episodes....36
3.2.3.2 Algorithm For Parallel Episodes....37
3.2.4 Minimal Occurrences Approach (MINEPI)....38
3.2.4.1 Finding Minimal Occurrences of Episodes....40
3.2.4.2 Finding Confidences of Rules....41
3.3 Microarchitecture and Branch Prediction....42
3.3.1 Microarchitecture....42
3.3.2 Basic Microcomputer Design....42
3.3.3 Clock and Instruction Execution Cycle....45
3.3.4 Branch Prediction....47
3.3.5 IPC , ILP and Performance....49
3.3.5.1 How Programs Run - Load and Execute Process....49
3.3.5.2 IPC (Instruction Per Cycle)....50
3.3.5.3 ILP (Instruction-level Parallelism)....51
4 EPISODE MINING ON ARCHITECTURE SIMULATION DATA....56
4.1 General Overview....56
4.2 Implementation Details of The Algorithms....57

4.3 Experiments and Results....64
4.3.1 Experiment 1....65
4.3.2 Experiment 2....70
4.3.3 Experiment 3....74
4.3.4 Experiment 4....77
4.3.5 Experiment 5....82
4.3.6 Experiment 6....85
5 EPISODE MINING TOOL (EMT)....90
5.1 General Properties....90
5.2 Components of EMT....91
5.3 Usage....92
6 CONCLUSION AND FUTURE WORK....93
REFERENCES....96
APPENDICES
A- USAGE OF EMT IN GUI MODE....99
B- USAGE OF EMT IN COMMAND LINE MODE....119

LIST OF FIGURES

Figure 1: The example event sequence and two windows of width 5........................17
Figure 2: Episodes α , β and γ..19
Figure 3: Block Diagram of a Microcomputer..44
Figure 4: One clock cycle...45
Figure 5: Before reading a new event from sequence 59
Figure 6: After reading a new event from sequence 59
Figure 7: Processing multiple events in a single cycle..62
Figure 8: Input file for experiment-1...65
Figure 9: Outputs for frequency of the episodes in experiment-1.............................67
Figure 10: Outputs for confidence of the rules in experiment-1................................67
Figure 11: Input file for experiment-2...72
Figure 12: Outputs for frequency of the episodes in experiment-2............................73
Figure 13: Outputs for confidence of the rules in experiment-2................................73
Figure 14: Input file for experiment-3...74
Figure 15: Outputs for frequency of the episodes in experiment-3............................76
Figure 16: Outputs for confidence of the rules in experiment-3................................77
Figure 17: Input file for experiment-4...78
Figure 18: Outputs for frequency of the episodes in experiment-4............................80
Figure 19: Outputs for confidence of the rules in experiment-4................................80
Figure 20: The first 10 episodes with length 3 according to their frequencies in the experiment-4...81
Figure 21: The first 5 rules with length 3 according to their confidences in the experiment-4...82
Figure 22: Episode mining results for episodes and frequency values.......................88
Figure 23: Episode mining results for rules and confidence values...........................89
Figure 24: Pre-processing operation in EMT...99
Figure 25: Opening input file for pre-processing...100
Figure 26: Selecting file from the browser for pre-processing................................100
Figure 27: Ignore events in the given dataset...101
Figure 28: Selecting the pre-processing method..101
Figure 29: Message given to the user after pre-processing.....................................102
Figure 30: Selecting processing unique sequences with IPC changes.....................102
Figure 31: Message given to the user to select the input datasets............................103
Figure 32: Selecting the datasets for two architectures..103
Figure 33: Selecting the directory including the rules for unique sequences...........104
Figure 34: Getting the name of the output event sequence.....................................104
Figure 35: Getting an IPC level to express the changes..105
Figure 36: Message given to the user after pre-processing of two datasets is completed...105
Figure 37: Selecting filter operation for pre-processing..106

Figure 38: Specifying input file for filtering..106
Figure 39: Episode mining operation in EMT..107
Figure 40: Algorithm selection in episode miner..107
Figure 41: Giving window width parameter in episode miner..108
Figure 42: Giving minimum frequency parameter in episode miner..108
Figure 43: Giving minimum confidence parameter in episode miner..109
Figure 44: Giving maximum episode length parameter in episode miner..109
Figure 45: Giving ignored event types parameter in episode miner..110
Figure 46: Selecting file from the browser for episode mining..110
Figure 47: Specifying output file for episode mining..111
Figure 48: Message given to the user after episode mining..111
Figure 49: Output analysing operation in EMT..112
Figure 50: Open single file in output analyser..112
Figure 51: Selecting file from the browser for output analysing..113
Figure 52: Analyse type selection for a single output..113
Figure 53: Rule length selection for output analysing..114
Figure 54: Chart type selection for output analysing..114
Figure 55: Rule count and type selection for output analysing..115
Figure 56: Result for single output analysis..115
Figure 57: Selecting directory for multiple output file analysis..116
Figure 58: Analyse episodes/rules..116
Figure 59: Selecting analyse type..117
Figure 60: Grouping the output files 117
Figure 61: Resulting chart after grouping the output files 118
Figure 62: Runing pre-processor from command line..119
Figure 63: Runing episode miner from command line..120

CHAPTER 1

INTRODUCTION

The aim of this thesis work is firstly analysing architecture simulation output data by applying episode mining techniques, showing the previously known relationships between the events in architecture and generating an environment to predict the performance of a program in an architecture before executing the codes. For this purpose a data mining tool has been developed that has three main components.

1. First component is the data pre-processor that transforms the raw output data of the architecture simulation into processable input data for the second component of the tool,the episode miner.
2. Episode miner component takes the inputs and by applying the episode mining algorithms with the given options by the user it produces an output containing the frequent episodes and rules generated from these episodes. This component includes the implementations of three algorithms, window based episode mining algorithm of Mannila et. al. [1], minimal occurrence based episode mining algorithm of Mannila et. al. [1] and non-overlapping occurrence based algorithm of Laxman et. al. [2].
3. The last component of the tool is the analyser and it gets the output of the episode miner as input and produces visual outputs as several types of charts about the frequent episodes and the generated rules.

Event logs taken from the architecture simulation is the input to this study. These log data are generally very huge in terms of gigabytes consisting of various type of events. In order to analyse the data feasibly and to be able to generate interpretable patterns, the number of event types is limited to four in most of the experiments.

Several different analyses have been performed on the architecture event logs;

1. The relations among the event types have been analysed.
2. The performance of a program execution on a processor can be measured with the IPC (Instruction Per Cycle) values and this value depends on how the particular software being run interacts with the processor. Therefore, as another analysis, we search for relationships between event types and IPC changes in benchmark data. For this analysis, the data set is preprocessed to generate a new dataset consisting conventional event types and two new event types as "IPC increases" and "IPC decreases" on the basis of the changes in IPC values. The interesting rules containing the events such as "IPC increase" and "IPC decrease" have been generated. In this analysis,we partitioned the data into program blocks containing these rules for further analysis.
3. Here, in the previous analysis we have noticed that there are unique sequences of events corresponding to execution blocks and these blocks are repeated several times in benchmark data. Therefore, we decided to analyse these unique sequences as individual input datasets. Input datasets were obtained for two different architectures and the unique sequences have been analysed with respect to the generated rules and the IPC differences between two architectures. We generated rules from program blocks and obtained IPC changes for these blocks during the execution of a program. Afterwards, we generated patterns showing the relationships between these rules and IPC change values.

As a result the main objective of this work is applying techniques of episode mining, to a new domain, processor architecture simulation datasets, which is different from the classical samples such as telecommunication alarm networks. With the help of the generated results, it was expected to generate facilities to make a strong prediction about the performance of the programs on different architectures before running them.

Here, analysing architecture simulation data by using episode mining techniques and

using the results of this analysis to make predictions about the performance of a program in an architecture before executing the codes can be considered as a new approach for computer architecture and data mining.

CHAPTER 2

AN OVERVIEW ABOUT DATA MINING

2.1 About Data Mining

2.1.1 Data, Information and Knowledge

From the IT (Information Technologies) point of view data are any facts, numbers or text that can be processed by a computer. This data can be:

- Operational or transactional data such as, sales, cost, inventory, payroll, and accounting
- Non-operational data, such as industry sales, forecast data and macro economic data
- Meta data - (data about the data) such as logical database design or data dictionary definitions

The patterns, associations, or relationships among all this data can provide information. For example, analysis of retail point of sale transaction data can give information on which products are selling and when. Therefore this information can be used to analyse the data about the sales. Information can be converted into knowledge about historical patterns and future trends. For example, summary information on retail supermarket sales can be analysed in light of promotional efforts to provide knowledge of consumer buying behaviour. Thus, a manufacturer or retailer could determine which items are most susceptible to promotional efforts [3].

2.1.2 Definition of Data Mining

If we look at the various definitions of data mining , first of all we can see that Hand, Mannila and Smyth define the term “data mining” as follows:

“Data mining is the analysis of (often large) observational data sets to find unsuspected relationships and to summarize the data in novel ways that are both understandable and useful to the data owner” [4].

According to another definition, data mining is the process of looking for new knowledge in existing data. The main idea of data mining can be given as transforming a low-level data which is frequently too large and too complex to understand, into a higher form that can be considered as knowledge or information. Here the basic properties of this information or knowledge which is obtained from the raw data are:

- It is more compact. For example it can be a summary.
- It is more abstract. For example it can be a descriptive model.
- It is more useful. For example it can be a predictive model.

Data mining is usually defined as searching, analysing and sifting through large amounts of data to find relationships, patterns or any significant statistical correlations. With the advent of computers, large databases and the internet, it is easier than ever to collect millions, billions and even trillions of pieces of data that can then be systematically analysed to help look for relationships and to seek solutions to difficult problems. The use of data mining is increasing and becoming more and more common in both private and public sectors. Industries such as banking, insurance, medicine and retailing commonly use data mining to reduce costs, enhance research and increase sales. In the public sector, data mining applications initially were used as a means to detect fraud and waste, but have grown to also be used for purposes such as measuring and improving program performance. [5].

The data stored in files databases and other repositories has reached an enormous quantity in information technologies and such a huge amount of data makes it really essential to develop powerful means to for analysis. In addition to analysis of data, it has also become important to extract the interesting knowledge which will help in decision-making. Data mining, whose another name can be given as KDD (Knowledge Discovery in Databases) , basically explains a set of processes consisting of non-trivial extraction of implicit , previously unknown and potentially useful information from databases. In general KDD and data mining are considered as synonyms. However, actually data mining should be evaluated as a part of KDD process [6].

Data mining commonly involves four classes of task [7]:

- Classification: It arranges the data into predefined groups. For example an email program might attempt to classify an email as legitimate or spam. Common algorithms include Decision Tree Learning, Nearest Neighbour, Naive Bayesian Classification and Neural network.
- Clustering: It is like classification but the groups are not predefined, so the algorithm will try to group similar items together.
- Regression: It attempts to find a function which models the data with the least error.
- Association Rule Mining: It searches for relationships between variables. For example a supermarket might gather data on customer purchasing habits. Using association rule learning, the supermarket can determine which products are frequently bought together and use this information for marketing purposes. This is sometimes referred to as "market basket analysis".

2.2 Association Rule Mining

The proposed work can be considered as a association rule mining task. For this reason, in this section , a brief overview of association rule mining is presented.

The formal definition for association rules can be given as follows:

Let $I=\{I_1,I_2,...,I_m\}$ be a set of m distinct attributes, T be a transaction containing a set of items such that $T \subseteq I$ and D be a database with different transaction records T_s. An association rule is an implication in the form of $X \Rightarrow Y$, where X, $Y \subset I$ are sets of items called itemsets, and $X \cap Y = \emptyset$. X is called antecedent while Y is called consequent, the rule means "X implies Y". For association rules, there are two basic measures: support and confidence. Support and confidence values are predefined by users to drop the rules that are not interesting or useful because the database is generally too large and users are interested in only frequent or interesting itemsets or rules. Support of an association rule is defined as the percentage or fraction of record containing $X \cup Y$ to the total number of records in the database. For example, let's assume that the support of an association rule is 0.5%. This means that just 0.5 percent of the transaction contain this item. Confidence of an association rule can be defined as the percentage or fraction of the number of transactions that contain $X \cup Y$ to the total number of records that contain X. Confidence is a measure of strength of the association rules. Let's assume that the confidence of the association rule $X \Rightarrow Y$ is 75%, it means that 75% of the transactions that contain X also contain Y together [8].

A transaction can be defined as a set of items and discovering association rules from databases of transactions can be considered as one of the most important problems in data mining. Here, the most time consuming operation in this discovery process is the computation of the frequency of the occurrences of interesting subset of items which are called as candidates in the database of transactions. Association rule mining has a wide range of applicability such market basket analysis, medical diagnosis research, website navigation analysis, homeland security and so on.

Association rules are used to identify relationships among a set of items in database. These relationships are not based on inherent properties of the data themselves (as with functional dependencies), but rather based on co-occurrence of the data items. Association rule and frequent itemset mining became a widely researched area and hence faster and faster algorithms have been presented. Many of these algorithms are based on Apriori or they are the modifications of Apriori. The scheme of Apriori algorithm is not only used in association rule mining but also in other data mining fields such as sequential pattern mining, episode mining, functional dependency discovery and here frequent pattern mining techniques can also be used to solve many other problems such as iceberg cube computation and classification. Therefore, it has become an interesting research problem to find out new methods and approaches for effective and efficient frequent pattern mining [9].

Many algorithms for generating association rules were presented over time. Some well known algorithms are Apriori, Eclat and Frequent Pattern Growth (FP-Growth) [10] , but they only do half the job, since they are algorithms for mining frequent itemsets. Another step needs to be done after to generate rules from frequent itemsets found in a database.

- Apriori Algorithm: The best-known algorithm to mine association rules is the Apriori. It uses a breadth-first search strategy to counting the support of itemsets and uses a candidate generation function which exploits the downward closure property of support.
- Eclat Algorithm: Eclat is a depth-first search algorithm using set intersection.
- FP-growth Algorithm: FP-growth uses an extended prefix-tree (FP-tree) structure to store the database in a compressed form. FP-growth adopts a divide-and-conquer approach to decompose both the mining tasks and the databases. It uses a pattern fragment growth method to avoid the costly process of candidate generation and testing used by Apriori.
- Zero-attribute-rule: The zero-attribute-rule, or ZeroR, does not involve any attribute in the condition part, and always returns the most frequent class in the training set. This algorithm is frequently used to measure the

classification success of other algorithms.

- One-attribute-rule: The one-attribute-rule, or OneR, is an algorithm for finding association rules. According to Ross, very simple association rules, involving just one attribute in the condition part, often work well in practice with real-world data. The idea of the OneR (one-attribute-rule) algorithm is to find the one attribute to use to classify a novel data point that makes fewest prediction errors. For example, to classify a car you haven't seen before, you might apply the following rule: If Fast Then Sportscar, as opposed to a rule with multiple attributes in the condition: If Fast And Softtop And Red Then Sportscar. The algorithm is as follows:

```
For each attribute A:
    For each value V of that attribute, create a rule:
            1. count how often each class appears
            2. find the most frequent class, c
            3. make a rule "if A=V then C=c"
    Calculate the error rate of this rule
Pick the attribute whose rules produce the lowest error rate
```

2.3 FIM (Frequent Itemset Mining)

The aim of frequent itemset mining is finding interesting patterns from databases. A FIM algorithm scans the database for several times to find itemsets that occur in transactions more frequently than a given threshold. Here the support is the number of occurrences and the threshold is the minimum support. Two of the most popular FIM algorithms are Apriori and FP-Growth. According to a report from the first Workshop on Frequent Itemset Mining Implementations (FIMI'03), FP-growth implementations were generally an order of magnitude faster than Apriori ;however, on several datasets, an Apriori implementation, apriori borgelt, was slightly faster when the support was high [11].

There are various types of applications on mining frequent patterns or itemsets. Some of them are the discovery of association rules, strong rules, correlations, sequential rules, episodes, multi-dimensional patterns and many other important discovery tasks. The problem in mining frequent patterns can be given as: Assuming a large data base of item transactions is given, find all frequent itemsets, where a frequent itemset is one that occurs in at least a user-specified percentage of the database. Most of the proposed pattern-mining algorithms are a variant of Apriori. Apriori employs a bottom up, breadth-first search that enumerates every single frequent itemset. Also Apriori uses the downward closure property of itemset support in order to prune the search space with the idea that the property that all subsets of a frequent itemset must themselves be frequent. Thus only the frequent itemsets are used to construct candidate (k + 1)-itemsets. A pass over the database is made at each level to find the frequent itemsets among the candidates. Apriori-inspired algorithms show good performance with sparse datasets such as market basket data, where the frequent patterns are relatively short [12].

Generally Aprori approaches were used based on the downward closure property before the FP-tree based mining method was developed. That is, if any length k pattern is not frequent in a transaction database, superset patterns can not be frequent. Aprori based algorithms can prune the candidate itemsets using this characteristic. But as a negative point Apriori based algorithms should generate and test all candidates. Moreover, they must repeatedly scan a large amount of the original database in order to check if a candidate is frequent or not and this approach is inefficient and ineffective. Pattern growth based approaches were developed to solve this problem. Using a divide and conquer method, FP-tree based methods mine the complete set of frequent patterns to reduce the search space without generating all candidates. Typically an association rule mining algorithm firstly generates frequent patterns and then it makes association rules satisfying a minimum support. There is a limitation for traditional model about mining frequent itemsets. Here, all the items are treated uniformly but in real the importance of items are different. So that weighted frequent itemset mining algorithms have been developed. Different weight values are given to items in the transaction database. The main idea of weighted

frequent itemset mining is to concern satisfying the downward closure property. This is broken when different weights are applied to the items according to their significance. Generally, weighted association rule mining algorithms have adopted an Apriori algorithm based on the downward closure property. They have suggested the weighted closure property, the sorted closure property or other techniques in order to satisfy the downward closure property. But the algorithms that are based on Apriori, use candidate set generation and test techniques. Here, it is really a remarkable cost to generate and test all candidates. It is shown in performance analyses that frequent pattern growth algorithms are more efficient at mining large databases and more scalable than Apriori based techniques. On the other hand, there is not an approach of weighted association rule mining using the pattern growth algorithm. Because, here when you apply the FP-growth methods the downward closure property is broken [13].

2.4 Apriori Algorithm

The problem of association rule mining can be divided into two sub-problems:

- Find all combinations of items in a set of transactions that occur with a given minimum frequency. These combinations are called frequent itemsets.
- Calculate rules that express the probable co-occurrence of items within frequent itemsets.

Apriori calculates the probability of an item being present in a frequent itemset, given that another item or items is present. Association rule mining is not recommended for finding associations involving rare events in problem domains with a large number of items. Apriori discovers patterns with frequency above the minimum support threshold. Therefore, in order to find associations involving rare events, the algorithm must run with very low minimum support values. However, doing so could potentially explode the number of enumerated itemsets, especially in cases with a large number of items. This could increase the execution time

significantly. Classification or anomaly detection may be more suitable for discovering rare events when the data has a high number of attributes [14].

Apriori algorithm can be considered as an example of a level-wise algorithm for association discovery. It passes for several times over the input data to find the frequent itemsets. Let L_k denote the set of frequent itemsets of size k and let C_k denote the set of candidate itemsets of size k. Before making the k-th pass, Apriori generates C_k using L_{k-1}. Its candidate generation process ensures that all subsets of size k-1 of C_k are all members of the set L_{k-1}. In the k-th pass, it then counts the support for all the itemsets in Ck. At the end of the pass all itemsets in C_k with a support greater than or equal to the minimum support form the set of frequent itemsets L_k. The method of pruning the C_k set using L_{k-1} results in a much more efficient support counting phase for Apriori when compared to the earlier algorithms. In addition, the usage of a hash-tree data structure for storing the candidates provides a very efficient support-counting process [15].

Except for Aprori , other algorithms are designed for finding association rules in data having no transactions (WINEPI and MINEPI), or having no timestamps (DNA sequencing). As common in association rule mining, given a set of itemsets (for instance, sets of retail transactions, each listing individual items purchased), the algorithm attempts to find subsets which are common to at least a minimum number C of the itemsets. Apriori uses a "bottom up" approach, where frequent subsets are extended one item at a time (a step known as candidate generation), and groups of candidates are tested against the data. The algorithm terminates when no further successful extensions are found. Apriori, while historically significant, suffers from a number of inefficiencies or trade-offs, which have spawned other algorithms. Candidate generation generates large numbers of subsets (the algorithm attempts to load up the candidate set with as many as possible before each scan). Bottom-up subset exploration (essentially a breadth-first traversal of the subset lattice) finds any maximal subset S only after all 2 | S | − 1 of its proper subsets [14].

CHAPTER 3

BACKGROUND

3.1 Sequential Pattern Mining

We can call a database that consists of sequences of values or events that change with time as a time-series database. In this type of database, the valid time of each dataset is recorded. For instance, in a time-series database that records the sales transaction of a supermarket, each transaction includes an extra attribute indicate when the transaction happened. Basically there are four kinds of patterns we can get from various types of time-series data:

1-Trend Analysis: Finding the evolution patterns over the time can be called as trend analysis. These can be cyclic movements or variations, seasonal movements, long-term trend movements and irregular or random movements. If we give an example, we can model the changes in price into a function Y=F(t) by using the historical stock price records of a company. Also this function can be given as a time-series graph. Tuesday the price will increase by 8 % and every Thursday the price will drop by around 4.5%. This method is widely used in the analysis of stock market [16].

2-Similarity Search: The process here is a matching process that can tolerate some differences within a certain threshold. According to the length of sequences we are trying to match , sequence matching can be classified as:

- Subsequence Matching
- Whole Sequence Matching

Assume that the data of stock prices is transformed into curves , those curves include many different shapes such as up, sharp up, down, big down. The typical example of similarity search is about the curve and shapes of one stock and the idea is to find other stocks that have the similar curves and shapes [16].

3-Sequential Patterns: In sequential pattern mining the relationships between the occurrences of sequential events are searched and the aim is to find if there exist any specific order of the occurrences. The sequential patterns of specific individual items can be found in addition to sequential patterns cross different items. Sequential pattern mining is widely used in analysing of DNA sequence. An example of sequential patterns is that every time company-A stock drops 8% , company-B stock will also drops at least 3.5% within three days [16].

4-Periodical Patterns**:** These patterns are recurring patterns in the database and the periods can be a day, a week, a month, a season or a year. If we take the periodical sequences as a set of sequences in sequential pattern mining, this can be considered as periodical pattern mining. While operating periodical pattern mining in a customer transaction database where each sequence represents a sequence of items bought by a customer, we have a long sequence of data. However, this sequence can be partitioned into periods and thus, we can obtain a set of sequences. An example of periodical pattern is that if a restaurant received many coffee customers during 4:00-6:00 pm, dinner will sell well between 9:00-10:00 pm [16].

After the introduction of sequential pattern mining, its popularity has remarkably increased with its broad area of applications like market and customer analysis, web log analysis, pattern discovery in protein sequences and mining XML query access patterns for caching. We can define a sequential pattern as a sequence of itemsets occurring in a specific order.

Sequential pattern is a sequence of itemsets that frequently occurred in a specific order, all items in the same itemsets are supposed to have the same transaction time value or within a time gap. In recent years many studies have presented convincing

arguments that for mining frequent patterns for itemsets and sequences. As different from mining frequent itemsets, there are not so many methods proposed for mining closed sequential patterns. The major reason for this situation can be given as the complexity of the problem. Today, CloSpan is known as the only algorithm for this process. As in most of the frequent closed itemset mining algorithms, it proceeds a candidate maintenance-and-test approach. For example, it needs to maintain the set of already mined closed sequence candidates which can be used to prune search space and check if a newly found frequent sequence is promising to be closed. However, this closed pattern mining algorithm has a poor scalability in the number of frequent closed patterns. Because a large number of frequent closed patterns or just candidates need much memory space and large search space for the closure checking of new patterns [17].

For each dataset, it is clear that timestamp value is an important attribute. Also this is important in in the process of data mining and it can give us more accurate and useful information. For instance, association rule mining does not take the timestamp into account, the rule can be "Buy A ⇒ Buy B". If we consider timestamp then we can get more accurate and useful rules such as: Buy A implies Buy B within a week or usually people Buy A every week. As we can see with the second kind of rules, business organizations can make more accurate and useful prediction and consequently make more sound decisions [16].

3.2 Frequent Episode Mining

The data can be analysed consists of sequence of events and there are several data mining and machine learning application areas processing this type of data. Alarms in a telecommunication network, user interface actions, crimes committed by a person and occurrences of recurrent illnesses can be given as examples of such data. Here, each event in the sequence has an associated time of occurrence. In Figure 1, [1] an example of event sequence is represented. Here A, B, C, D, E and F are event

types such as different types of alarms from a telecommunication network or different types of user actions and they have been marked on a time line [1].

Generally , machine learning and data mining techniques are focused on the analysis of unordered collections of data. Examples of this type of data can be given as transaction databases and sequence databases. While analysing a sequence the typical point is to find the frequent episodes such as collections of events occurring frequently together. Here, the main idea of episode mining is to find relationship between events. After that these relationships will be used to explain the problems that cause an event or predict the possible result. There are several applications of episode mining such as biomedical data analysis, drought risk management in climatology and internet anomaly intrusion detection. The task of mining frequent episodes was originally defined on a sequence of events where the events are sampled regularly as proposed by Mannila [1].

Episodes, in general, are partially ordered sets of events. From the sequence in the Figure 1 one can make, for instance, the observation that whenever A and B occur, in either order, C occurs soon. When discovering episodes in a telecommunication network alarm log, the goal is to find relationships between alarms. Such relationships can then be used in the on-line analysis of the incoming alarm stream, for instance to better explain the problems that cause alarms, to suppress redundant alarms and to predict severe faults [1].

3.2.1 Event Sequences and Episodes

3.2.1.1 Event Sequences

We consider the input as a sequence of events, where each event has an associated time of occurrence. Given a set E of event types, an event is a pair (A,t) , where $A \in E$ is an event type and t is an integer, the (occurrence) time of the event. The event type can actually contain several attributes; for simplicity we consider here just the case where the event type is a single value. An event sequence s on E is a triple (s, Ts , Te), where $s = \langle (A_1, t_1), (A_2,t_2) ,...,(A_n,t_n) \rangle$ is an ordered sequence of events such that $A_i \in E$ for all i = 1,...,n, and $t_i \leq t_{i+1}$ for all i = 1,...,n-1. Further on, Ts and T_e are integers: T_s is called the starting time and T_e the ending time, and $T_s \leq t_i < T_e$ for all i = 1,...,n [1].

Example 3.2.1.1.1: Figure 1 [18] presents the event sequence s=(29; 68), where s=<(E , 31) , (D , 32) , (F , 33) , (A , 35) , (B , 37) , (C , 38) , . . . , (D , 67) >

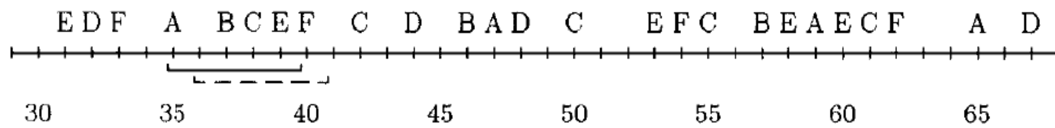


Figure 1: The example event sequence and two windows of width 5

The sample sequence here starts with time 29 to just before time 68. In this sample for each event that occurred in the time interval [29 , 68), the event type and the time of occurrence have been recorded [1].

While analysing event sequences, the aim is to find all frequent episodes from a class of episodes. An episode's events must occur close enough in time for us to consider this episode as an interesting one.

Here, the borders of being close for events are specified by the user. A window is a slice of event sequence and the event sequence can be considered as a sequence of

partially overlapping windows. By determining the width of the window , the user gives the measures for deciding if the events occur closely or not.

In addition to the width of the window, the user specifies in how many windows an episode has to occur to be considered frequent. “Formally, a window on an event sequence $s = (s , T_s , T_e)$ is an event sequence $w = (w , t_s , t_e)$, where $t_s < T_e$ and $t_e > T_s$, and w consists of those pairs (A , t) from s where $t_s \leq t < t_e$. The time span $t_e - t_s$ is called the width of the window w, and it is denoted width(w). Given an event sequence s and an integer win, we denote by W(s , win) the set of all windows w on s such that width(w)=win” [1]

For an event sequence, the first and the last window extend outside the sequence. Here, the first window contains only the first time point of the sequence and the last window contains only the last time point of the sequence. Given an event sequence $s = (s , T_s , T_e)$ and a window width win, the number of windows in W(s , win) is $T_e - T_s + win – 1$.

Example 3.2.1.1.2: In Figure 1 [1] there are also two windows of width 5 on the sequence s. The window starting at time 35 is

$$(\langle (A , 35) , (B , 37) , (C , 38) , (E , 39)\rangle , 35 , 40)$$

Note that the event (F , 40) that occurred at the ending time is not in the window. The window starting at 36 is similar to this one; the difference is that the first event. (A, 35) is missing and there is a new event (F,40) at the end. The set of the partially overlapping windows of width 5 constitutes W (s, 5) the first window is (∅, 25, 30), and the last is (<(D, 67)>, 67, 72). Event (D, 67) occurs in 5 windows of width 5, as for instance event (C ,50) does [1].

3.2.1.2 Episodes

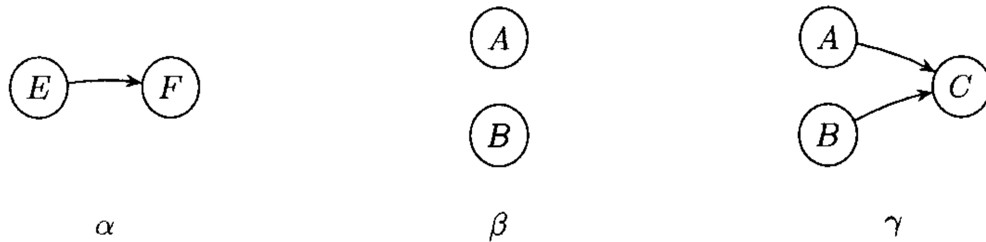


Figure 2: Episodes α , β and γ

We can define an episode as a partially ordered collection of events occurring together. We can illustrate the episodes in directed acyclic graphs and episodes can be classified into three categories such as serial episodes, parallel episodes and non-serial and non-parallel episodes. If we look at the episodes given in Figure 2 [1], episode α is a serial episode: it occurs in a sequence only if there are events of types E and F that occur in this order in the sequence. Episode β is a parallel episode: no constraints on the relative order of A and B are given. Episode γ is an example of non-serial and non-parallel episode: it occurs in a sequence if there are occurrences of A and B and these precede an occurrence of C; no constraints on the relative order of A and B are given.

The formal definition of episodes can be given as in the following:

" An episode α is a triple (V , ≤ , g) where V is a set of nodes, ≤ is a partial order on V, and $g : V \rightarrow E$ is a mapping associating each node with an event type. The interpretation of an episode is that the events in g(V) have to occur in the order described by ≤. The size of α, denoted |α|, is |V|. Episode α is parallel if the partial order ≤ is trivial (i.e., $x \not\leq y$ for all $x , y \in V$ such that $x \neq y$). Episode α is serial if the relation ≤ is a total order (i.e., $x \leq y$ or $y \leq x$ for all $x , y \in V$). Episode α is injective if the mapping g is an injection, i.e., no event type occurs twice in the episode." [1]

Example 3.2.1.2.1: If we consider episode $\alpha=(V, \leq, g)$ in Figure 2 episode α is injective, because it does not contain duplicate event types. Here, we only compute the number of windows where α occurs at all and we do not consider the number of occurrences per window [1].

Another important term about episode mining is subepisodes. The definition of subepisode can be given as :

"An episode $\beta = (V', \leq', g')$ is a subepisode of $\alpha=(V', \leq', g')$ denoted $\beta \preceq \alpha$, if there exists an injective mapping $f : V' \rightarrow V$ such that $g'(v)=g(f(v))$ for all $v \in V'$,and for all $v, w \in V'$ with $v \leq' w$ also $f(v) \leq f(w)$. An episode α is a superepisode of β if and only if $\beta \preceq \alpha$. We write $\beta \prec \alpha$ if $\beta \preceq \alpha$ and $\alpha \npreceq \beta$" [1].

Example 3.2.1.2.2: In Figure 2 [1] the episode β is a subgraph of the episode γ. Here, the nodes in episode β are not ordered, therefore the corresponding nodes in γ do not need to be ordered[1].

To consider an episode to occur in a sequence, the nodes of the episode need to have corresponding events in the sequence such that the event types are the same and the partial order of the episode is respected.

"Formally, an episode $\alpha=(V, \leq, g)$ occurs in an event sequence $s=\langle(A_1,t_1), (A_2, t_2), ..., (A_n,t_n)\rangle$ if there exists an injective mapping $h : V \rightarrow \{1, ... , n\}$ from nodes of α to events of s such that $g(x) = A_{h(x)}$ for all $x \in V$, and for all $x, y \in V$ with $x \neq y$ and $x \leq y$ we have $t_{h(x)} < t_{h(y)}$." [1].

Example 3.2.1.3: In Figure 1 [2] the given window (w ,35 ,40) contains events A , B , C and E. Here, episodes β and γ of Figure 2 [1] occur in the window, but α does not.

The frequency of an episode is the fraction of windows in which the episode occurs. If the event sequence is s and the window width is win , then the frequency of an episode α in s is

$$fr(\alpha, \mathbf{s}, win) = \frac{|\{\mathbf{w} \in \mathcal{W}(\mathbf{s}, win) \mid \alpha \text{ occurs in } \mathbf{w}\}|}{|\mathcal{W}(\mathbf{s}, win)|}$$

Here episode α is frequent if fr (α ,s, win) ≥ min_fr. The min_fr is the threshold value specified by the user. Our task is to discover all frequent episodes from a given class ε of episodes and the collection of frequent episodes with respect to s, win and min_fr is denoted by F (s, win, min_fr).

After the frequent episodes are discovered, these episodes will be used to generate the rules describing the relationships between the events in the given sequence. For example, if we know that the episode β of Figure 2 occurs in 4.2% of the windows and that the superepisode γ occurs in 4.0% of the windows, we can estimate that after seeing a window with A and B, there is a chance of about 0.95 that C follows in the same window. Formally, an episode rule is an expression β→ γ , where β and γ are episodes such that $\beta \preceq \gamma$. The fraction fr(γ,s,win) / fr (β,s,win) is the confidence of the episode rule. The confidence can be interpreted as the conditional probability of the whole of γ occurring in a window, given that β occurs in it. Episode rules show the connections between events more clearly than frequent episodes alone [1].

3.2.2 Algorithms for WINEPI Approach

For episode mining with WINEPI approach,there are five algorithms. Mainly the algorithms starts generating candidates by starting with 1-length episodes and finds the frequent episodes among these candidates. Then the rules are generated by using these frequent episodes and their sub-episodes.

If we consider the whole procedure, firstly, we find the frequencies of the episodes with length-1, determine the frequent episodes by using the frequency threshold given and then generate the 2-length candidates. After that we will compute the frequencies of these candidate episodes and specify which of them are frequent. And the procedure will go on until the set of frequent episodes is empty. Finally, we will generate the rules from the frequent episodes by using the given confidence threshold parameter.

As a result, the whole process can be divided into two parts as;

1. Candidate Generation: The aim of candidate generation process is to minimize the number of candidates on each level. Therefore the work for database pass can be reduced. Generally combining multiple iterations for candidate generation to a single database pass is helpful for reducing the count of expensive database passes. In order to do this, firstly candidates for the next level $l + 1$ are computed and then with the assumption that all candidates of level $l + 1$ are frequent candidates for the following level $l + 2$ are computed. And the process goes on. In this approach no frequent episode is missed, however candidate collections can be larger. When we look at the time complexity of Algorithm 3, it is independent of the length of the event sequence and it is polynomial in the size of the collection of frequent episodes. The candidate generation operations are very similar for parallel and serial episodes. Small changes in the algorithm for parallel episodes about candidate generation is enough to adapt this method for serial episodes [1].
2. Recognition: The process for this part is remarkably different for parallel and serial episodes. For the implementation of the database pass, the algorithms recognizing the episodes in sequences in an incremental method are given and for two windows $w = (w, t_s, t_s + win)$ and $w' = (w', t_s + 1, t_s + win + 1)$, the sequences w and w' of events are similar to each other. After the episodes in w are recognized, updates are done incrementally in data structures to shift the window w to get w'. Algorithms firstly checks the empty window at the

beginning of the sequence and finally they check the empty window at the end of the sequence. While computing the frequency of episodes, the windows on the input sequence are considered.

2.1. For Parallel Episodes: Here, for each candidate parallel episode α we have a counter α.event_count that holds how many events of α are present in the window. If α.event_count becomes equal to the size of episode α : |α|, this means that the current window contains α and we save the starting time of this window in α.inwindow. If α.event_count decreases again, this means that α is no longer contained in the current window and we increase the value of α.freq_count by the number of windows where α remained entirely in the window. Finally the value of α.freq_count gives us the total count of windows where α occurs. In order to get candidate episodes efficiently, they are indexed by the number of events of each type that they contain. Here all episodes that contain exactly *a* events of type A are in the list *contains (A , a)*. If we shift the window, the events contained by the window will change and the episodes affected by this change will be updated. For example, there is one event of type C in the window and a second one comes in, all episodes in the list *contains (C , 2)* are updated with the information that both events of type C they are expecting are now present [1].

2.2. For Serial Episodes: A data structure such as state automata that accept the candidate episodes is used in processing the event sequence. We have an automaton for each serial episode denoted by α. There can be several instances of each automaton at the same time, so that the active states reflect the (disjoint) prefixes of α occurring in the window. When the first event of the episode comes into the window, a new instance of the automaton is initialized and this automaton is removed when the same event leaves the window. If the episode is entirely contained in the window, this means that automaton of the episode reached its accepting state.

Here, if there are no other automata for this episode in the accepting state the starting time of the window is saved in α.inwindow. If an automata in the accepting state is removed and if there are no other automata for α in the accepting state the field α.freq_count is increased by the number of windows where α remained entirely in the window. The automatas in the same state make the same transitions and they produce the same results so it is not useful to have multiple automata in the same state. The automata reaching the common state last will be remove last, too. So that it will be enough to process this automata among the others in the same state. As a result, the maximum number of the automatas for an episode will be equal to the number of events in this episode. When to remove should be known for each automata. Thus, all the automata for α can be represented with one array of size |α| and the value of α.initialized[i] is the latest initialization time of an automaton that has reached its ith state. For each event type A ϵ E, the automata that accept A are linked together to a list waits (A). The list contains entries of the form (α, x) meaning that episode α is waiting for its xth event.

When an event (A, t) enters the window during a shift, the list waits (A) is traversed. If an automaton reaches a common state i with another automaton, the earlier entry α.initialized[i] is simply overwritten. The transitions made during one shift of the window are stored in a list transitions. They are represented in the form (α, x, t) meaning that episode α got its xth event and the latest initialization time of the prefix of length x is t. Updates regarding the old states of the automata are done immediately but updates for the new states are done only after all transitions have been identified, in order to not overwrite any useful information. For easy removal of automata when they go out of the window, the automata initialized at time t are stored in a list beginsat (t) [1].

If an episode is not serial or parallel, the recognition of this arbitrary episode can be operated by the recognition of a hierarchical combination of serial and parallel episodes. For example, episode γ in Figure 2 [18] is a serial combination of two episodes: a parallel episode δ' consisting of A and B, and an episode δ'' consisting of C alone. The occurrence of an episode in a window can be tested using such

hierarchical structure: to see whether episode γ occurs in a window one checks (using a method for serial episodes) whether the subepisodes δ and δ'' occur in this order; to check the occurrence of δ' one uses a method for parallel episodes to verify whether A and B occur. On the other hand there are some complications to be taken into account. Sometimes it may be necessary to duplicate an event node to obtain a decomposition to serial and parallel episodes. This is helpful for injective episodes but for non-injective episodes we need more complex methods. Here, composite events have duration values where the elementary events do not. An alternative to the recognition of general episodes is to handle all episodes basically like parallel episodes and to check the correct partial ordering only when all events are in the window. Parallel episodes can be located efficiently; after they have been found, checking the correct partial ordering is relatively fast [1].

The operations in algorithms for candidate generation and recognition for parallel and serial episodes can be summarized as;

- Algorithm 1 is the base of the operations in episode mining for this technique. It takes the set of event types, the event sequence, the set of episodes, a window width win, a frequency threshold as min_fr, and a confidence threshold as min_conf as parameters and it describes how rules and their confidences can be computed from the frequencies of episodes.
- Algorithm 2 computes the collection F (s , win, min_fr) of frequent episodes from a class ε of episodes. The algorithm performs a levelwise (breadth-first) search in the class of episodes following the subepisode relation. The search starts from the most general episodes, i.e., episodes with only one event. On each level the algorithm first computes a collection of candidate episodes, and then checks their frequencies from the event sequence.
- Algorithm 3 computes candidates for parallel episodes and serial episodes. The method in this algorithm can be easily adapted to deal with the classes of parallel episodes, serial episodes, and injective parallel and serial episodes. Potential candidates can be identified by creating all combinations of two episodes in the same block. For the efficient identification of blocks, we store

in F_l .*block_start*[j] for each episode F_l [j] the i such that F_l [i] is the first episode in the block.

- Algorithm 4 executes the recognition phase for parallel episodes and calculates the frequency counts of the episodes. Then, if the frequency count is above the given threshold value, it is considered as a frequent episode.
- Algorithm 5 does the same job as Algorithm for but this time it calculates the frequency counts of the serial episodes and finds the frequent ones.

We can consider the following example to understand the execution of the steps while recognizing the episodes and calculating the frequency values for WINEPI approach.

Example 3.2.2.1: Assume that we have a small event sequence as;

T (time of occurrence)	Event
1	1
2	3
3	2

Here, assume that we have a frequency threshold as 0.25 and our window width is 3. For serial episodes, the process for the recognition of 1-length episodes can be illustrated as in the following way;

C:{{1},{2},{3}}	win=3
s= < (1,1),(2,3),(3,2) >	

In the following table, the data structures and their contents are given after the initialization phase of the recognition algorithm.

Initialization	
t=0	
{1}.initialized[1]	0
{2}.initialized[1]	0
{3}.initialized[1]	0
waits(1)	{(1,1)}
waits(2)	{(2,1)}
waits(3)	{(3,1)}
{1}.frequency_count	0
{2}.frequency_count	0
{3}.frequency_count	0
beginsat(-2)	{ }
beginsat(-1)	{ }
beginsat(0)	{ }
beginsat(1)	{ }
beginsat(2)	{ }
beginsat(3)	{ }
beginsat(4)	{ }
beginsat(5)	{ }
beginsat(6)	{ }
transitions	{ }
{1}.inwindow	
{2}.inwindow	
{3}.inwindow	

At first iteration, time (t) = 1 and the algorithm gets the event 1 occurs at this time.

t=1 start=-1	
{1}.initialized[1]	1
{2}.initialized[1]	0
{3}.initialized[1]	0
waits(1)	{(1,1)}
waits(2)	{(2,1)}
waits(3)	{(3,1)}

{1}.frequency_count	0
{2}.frequency_count	0
{3}.frequency_count	0
beginsat(-2)	{ }
beginsat(-1)	{ }
beginsat(0)	{ }
beginsat(1)	{ (1,1) }
beginsat(2)	{ }
beginsat(3)	{ }
beginsat(4)	{ }
beginsat(5)	{ }
beginsat(6)	{ }
transitions	{ { { 1 },1,1 } }
{1}.inwindow	-1
{2}.inwindow	
{3}.inwindow	

At time(t) = 2, the type of event occurs is 3.

t=2 start=0	
{1}.initialized[1]	1
{2}.initialized[1]	0
{3}.initialized[1]	2
waits(1)	{(1,1)}
waits(2)	{(2,1)}
waits(3)	{(3,1)}
{1}.frequency_count	0
{2}.frequency_count	0
{3}.frequency_count	0
beginsat(-2)	{ }
beginsat(-1)	{ }
beginsat(0)	{ }
beginsat(1)	{ (1,1) }

beginsat(2)	{ (3,1) }
beginsat(3)	{ }
beginsat(4)	{ }
beginsat(5)	{ }
beginsat(6)	{ }
transitions	{ { { 3 },1,2 } }
{1}.inwindow	-1
{2}.inwindow	
{3}.inwindow	0

Event with type 2 occurs at time(t)=3.

t=3 start=1	
{1}.initialized[1]	1
{2}.initialized[1]	3
{3}.initialized[1]	2
waits(1)	{(1,1)}
waits(2)	{(2,1)}
waits(3)	{(3,1)}
{1}.frequency_count	0
{2}.frequency_count	0
{3}.frequency_count	0
beginsat(-2)	{ }
beginsat(-1)	{ }
beginsat(0)	{ }
beginsat(1)	{ (1,1) }
beginsat(2)	{ (3,1) }
beginsat(3)	{ (2,1) }
beginsat(4)	{ }
beginsat(5)	{ }
beginsat(6)	{ }
transitions	{ { { 2 },1,3 } }
{1}.inwindow	-1

{2}.inwindow	1
{3}.inwindow	0

At time(t)=4 no event occurs and the sequence comes to end. Here, the event with type 1 comes out of window and its frequency count is calculated as 3.

t=4 start=2	
{1}.initialized[1]	0
{2}.initialized[1]	3
{3}.initialized[1]	2
waits(1)	{(1,1)}
waits(2)	{(2,1)}
waits(3)	{(3,1)}
{1}.frequency_count	0-(-1)+2 = 3
{2}.frequency_count	0
{3}.frequency_count	0
beginsat(-2)	{ }
beginsat(-1)	{ }
beginsat(0)	{ }
beginsat(1)	{ (1,1) }
beginsat(2)	{ (3,1) }
beginsat(3)	{ (2,1) }
beginsat(4)	{ }
beginsat(5)	{ }
beginsat(6)	{ }
transitions	{ }
{1}.inwindow	-1
{2}.inwindow	1
{3}.inwindow	0

At time(t)=5 no event occurs and the sequence comes to end. Here, the event with type 3 comes out of window and its frequency count is calculated as 3.

t=5 start=3	
{1}.initialized[1]	0
{2}.initialized[1]	3
{3}.initialized[1]	0
waits(1)	{(1,1)}
waits(2)	{(2,1)}
waits(3)	{(3,1)}
{1}.frequency_count	3
{2}.frequency_count	0
{3}.frequency_count	0-0+3 = 3
beginsat(-2)	{ }
beginsat(-1)	{ }
beginsat(0)	{ }
beginsat(1)	{ (1,1) }
beginsat(2)	{ (3,1) }
beginsat(3)	{ (2,1) }
beginsat(4)	{ }
beginsat(5)	{ }
beginsat(6)	{ }
transitions	{ }
{1}.inwindow	-1
{2}.inwindow	1
{3}.inwindow	0

At time(t)=6 no event occurs and the sequence comes to end. Here, the event with type 2 comes out of window and its frequency count is calculated as 3.

t=6 start=4	
{1}.initialized[1]	0
{2}.initialized[1]	0
{3}.initialized[1]	0
waits(1)	{(1,1)}

waits(2)	{(2,1)}
waits(3)	{(3,1)}
{1}.frequency_count	3
{2}.frequency_count	0-1+4=3
{3}.frequency_count	3
beginsat(-2)	{ }
beginsat(-1)	{ }
beginsat(0)	{ }
beginsat(1)	{ (1,1) }
beginsat(2)	{ (3,1) }
beginsat(3)	{ (2,1) }
beginsat(4)	{ }
beginsat(5)	{ }
beginsat(6)	{ }
transitions	{ }
{1}.inwindow	-1
{2}.inwindow	1
{3}.inwindow	0

As a result frequencies are counted and the frequency values are calculated with the formula;

$$frequency = \alpha.freq\ count / (T_e - T_s + win - 1)$$

Episode	Frequency count	Frequency
{1}	3	3/7=0.42
{2}	3	3/7=0.42
{3}	3	3/7=0.42

3.2.3 A Fast Algorithm For Finding Frequent Episodes In Event Streams (Non-overlapping Approach)

While mining temporal patterns from event streams, the method of discovering frequent episodes can be used. Here, the episodes denote the patterns and these patterns are ordered collections of event types. For instance, (A → B → C) denotes a temporal pattern where an event type A , is followed some time later by a B and a C, in that order. In a data sequence if all events of an episode are included in their order, this indicates that this episode occurs in this sequence. The count of the occurrences of the episode determines whether the episode is interesting or not. If the episode occurs frequently enough in the sequence, this episode is considered an interesting one. As a result the task about data mining here is to find all interesting episodes. This means that these episodes' frequency counts will be greater than the threshold given by the user.

According to the method of Mannila [1], the frequency of an episode is the number of windows in the event sequence in each of which the episode occurs. Here, the algorithm for counting these windows uses finite state automata as a data structure. For the worst case, time complexity of the algorithm is linear in the total time spanned by the event stream, the size of episodes and the number of candidates. The space needed by the algorithm is also linear in the size of episodes and the number of candidates. Some extensions to this windows-based frequency have also been proposed. There have also been some theoretical studies into this framework whereby one can estimate or bound the expected frequency of an episode in a data stream of a given length if we have a prior Markov or Bernoulli model for the data generation process. Thus, if sufficient training data is available, we can first estimate a model for the data source and then, on new data from the same source, can assess the significance of discovered episodes by comparing the actual frequencies with the expected frequency. As an alternative approach, a new notion for episode frequency based on the non-overlapped occurrences of an episode in the given data sequence is proposed. Here the algorithm has the same order of worst case time and space

complexities as the windows-based counting algorithm of. On the other hand, this non-overlapped occurrences based algorithm is seen more efficient and it is considered to run faster than the window-based algorithm after empirical investigations. Another important advantage of the non-overlapped occurrences count is that it facilitates a formal connection between discovery of frequent episodes and learning of generative models for the data sequence in terms of some specialized family of Hidden Markov Models. This formal connection allows us to assess statistical significance of episodes discovered without needing any prior model estimation step. As a result, when all these advantages are considered the non-overlapped occurrences count becomes an attractive method for applications involving frequent episode discovery from event streams [2].

The definition of non-overlapped occurrences of an episode can be given as:

Two occurrences of an episode are said to be non-overlapped if no event corresponding to one occurrence appears in between events corresponding to the other. If we assume that we have a collection of non-overlapping occurrences, here every pair of occurrences in it must be also non-overlapped [2].

Example 3.2.3.1: <(A, 1), (A, 2), (B, 3), (A, 7), (C, 8), (B, 9), (B, 10), (D, 11), (C, 12), (C, 13)>.

While recognizing the occurrence of a serial episode this can be done by using a finite state automaton. For instance, for the episode (A → B → C), we would have an automaton that transits to state 1 on seeing an event of type A and then waits for an event of type B to transit to its next state and so on until it transits to its final state, when an occurrence of the episode is regarded as complete. Counting all occurrences might be very inefficient because different instances of the automaton of an episode are needed to keep track of all its state transition possibilities. For instance, there are a total of eighteen occurrences of the episode (A → B → C) in the event sequence in the example. Four of them can be listed as :

1. {(A, 1), (B, 3), (C, 8)}

2. 2. {(A, 1), (B, 3), (C, 12)}
3. {(A, 1), (B, 3), (C, 13)}
4. {(A, 1), (B, 9), (C, 12)}

On seeing the event (A, 1) in the example event sequence 3.2.3.1, we can transit an automaton of this episode into state 1.However, at the event (B, 3), we cannot simply let this automaton transit to state 2. That way, we would miss an occurrence which uses the event (A, 1) but some other occurrence of the event type B later in the sequence. Hence, at the event (B, 3), we need to keep one instance of this automaton in state 1 and transit another new instance of the automaton for this episode into state 2. As is easy to see, we may need spawning of arbitrary number of new instances of automata if no occurrence is to be missed for an episode. Moreover, counting all occurrences renders candidate generation inefficient as well. This is because, when using total number of occurrences as the frequency definition, subepisodes may be less frequent than corresponding episodes. For example, in example sequence 3.2.3.1, while there are eighteen occurrences of (A → B → C), there are only eight occurrences of the subepisode (A → B). So, under such a frequency definition, level-wise procedures cannot be used for candidate generation. Each occurrence: h of episode α is associated with a set of events $\{(E_{h(vi)}, t_{h(vi)}) : v_i \in V_\alpha\}$ in the data stream. Two occurrences, h1 and h2, of an episode are considered as distinct occurrences if they do not share any events in the event sequence. In the event sequence in the example there can be at most three distinct occurrences of (A → B → C) and these are:

1. {(A, 1), (B, 3), (C, 8)}
2. {(A, 2), (B, 9), (C, 12)}
3. {(A, 7), (B, 10), (C, 13)}

Example 3.2.3.2: Consider the sequence

<(A, 1), (B, 2), (A, 3), (B, 4), (A, 7), (B, 8), . . .>

Two occurrences of an episode in an event sequence are non-overlapped if no event corresponding to one occurrence appears in between events corresponding to the other occurrence. In the sequence given in the Example 3.2.3.1 there can be at most one non-overlapped occurrence of (A → B → C), e.g., {(A, 2), (B, 3), (C, 8)} (since every other occurrence of (A → B → C) overlaps with this one). Similarly, in example 3.2.3.2, we need to keep track of only one of the pairs of event types A and B, since any other occurrence of (A → B → C) will have to overlap with this occurrence. In general, there can be many sets of non-overlapped occurrences of an episode in an event sequence. In order to count the non-overlapped occurrences of an episode, we need only one automaton. Until the automaton reaches its final state, we do not need a new instance of this automaton after the initialization [2].

3.2.3.1 Algorithm For Serial Episodes

This algorithm counts the non-overlapped counts for serial episodes. It looks at each event in the input sequence and makes necessary changes to the automata in waits(). When processing the ith event in the data stream, namely, (Ei, ti), the automata in waits(Ei) are considered. Every automaton (α, j) waiting for Ei is transited to its next state. This involves removing (α, j) from waits(Ei) and adding, either (α, j+1) or (α, 1) to the appropriate waits(.) list. This means that if the automaton has not yet reached its final state, it waits next for α[j + 1] and α(j+1) is added to waits(α[j+1]). If instead, an automaton has reached its final state, then a new automaton for the episode is initialized by adding (α, 1) to waits (α[1]). Note that since this process of adding to the waits(.) list is performed inside the loop over all elements in waits(E_i), it is in appropriate to add to this list from within the loop. Hence, as was mentioned earlier, we use a temporary storage called bag. Whenever we want to add an element to waits(E_i) it is stored first in bag which is later emptied

into waits(E_i) after exiting from the loop. Finally, the episode frequency is incremented every time its automaton reaches the final state. Since a new automaton for the episode is initialized only after an earlier one reached its final state, the algorithm counts non-overlapped occurrences of episodes [2].

3.2.3.2 Algorithm For Parallel Episodes

If all of the events in a parallel episode are included without considering the order of them in the event sequence, this indicates an occurrence of the window. The only difference between recognizing the occurrences of the parallel and serial episodes is that we do need to take the order of the events into consideration.

This algorithm obtains the non-overlapped occurrences-based frequencies for a set of candidate parallel episodes. The inputs of the algorithm are the set of candidates, the data stream and the frequency threshold and the output is the set of frequent episodes. The main data structure here is waits(.) list as in the serial episodes but it is slightly different. Each entry in the list waits(A), is an ordered pair like, (α, j), which indicates that there is a partial occurrence of which still needs j events of type A before it can become a complete occurrence. The initialization process involves adding the relevant ordered pairs for each episode into appropriate waits(.) lists. For example, episode α= (AABCCC) will initially figure in three lists, namely, waits(A), waits(B) and waits(C) and they will have entries (α, 2), (α, 1) and (α, 3) respectively. There are two quantities associated with each episode, α, namely α.freq, which stores the frequency of α and α.counter, which indicates the number of events in the sequence that constitute the current partial occurrence of α. As we go down the event sequence, for each event (E_i, t_i), the partial occurrences waiting for an E_i are considered for update. If $(\alpha, j) \in$ waits (E_i) , then having seen an E_i now (α, j) is replaced by (α, j − 1) in waits(E_i) if sufficient number of events of type E_i for are not yet accounted for in the current partial occurrence. Note that this needs to be done through the temporary storage bag since we cannot make changes to waits (E_i) from within the loop. Also, α.counter is incremented, indicating that the partial occurrence for has progressed by one more node. When α .counter = $|\alpha|$ = N, it means that the

N events necessary for completing an occurrence have appeared in the event sequence. We increment the frequency by one and start waiting for a fresh occurrence of by once again adding appropriate elements to the waits(.) lists [2].

3.2.4 Minimal Occurrences Approach (MINEPI)

Instead of looking at the windows and only considering whether an episode occurs in a window or not, we now look at the exact occurrences of episodes and the relationships between those occurrences. One of the advantages of this approach is that focusing on the occurrences of episodes allows us to more easily find rules with two window widths, one for the left-hand side and one for the whole rule, such as "if A and B occur within 15 seconds, then C follows within 30 seconds". The approach is based on minimal occurrences of episodes. Besides the new rule formulation, the use of minimal occurrences gives raise to the following new method, called MINEPI, for the recognition of episodes in the input sequence. For each frequent episode we store information about the locations of its minimal occurrences. In the recognition phase we can then compute the locations of minimal occurrences of a candidate episode α as a temporal join of the minimal occurrences of two subepisodes of α. This is simple and efficient, and the confidences and frequencies of rules with a large number of different window widths can be obtained quickly, i.e., there is no need to rerun the analysis if one only wants to modify the window widths. In the case of complicated episodes, the time needed for recognizing the occurrence of an episode can be significant; the use of stored minimal occurrences of episodes eliminates unnecessary repetition of the recognition effort. We identify minimal occurrences with their time intervals in the following way.

Given an episode α and an event sequence s, we say that the interval [ts , te) is a minimal occurrence of α in s,

- If (1) α occurs in the window w = (w , ts , te) on s
- If (2) α does not occur in any proper subwindow on w, i.e., α does not occur

in any window w' = (w' , ts' , te') on s such that ts ≤ t's ≤ t'e and width (w') < width (w)

The set of intervals of minimal occurrences of an episode α in a given event sequence is denoted by mo (α) = { [ts , te) | [ts , te) is a minimal occurrence of α }. [18]

Example 3.2.4.1: Here, we consider the Figure 1 and Figure 2. The parallel episode β consisting of event types A and B has four minimal occurrences in s: mo(β) = { [35, 38) , [46, 48) , [47, 58) , [57 , 60) }. The partially ordered episode γ has the following three minimal occurrences: [35 , 39) , [46 , 51) , [57 , 62).

An episode rule (with two time bounds) is an expression $\beta[win_1] \Rightarrow \alpha[win_2]$, where β and α are episodes such that $\beta \preceq \alpha$, and win_1 and win_2 are integers. The informal interpretation of the rule is that if episode β has a minimal occurrence at interval $[t_s,t_e)$ with $t_e - t_s \leq win_1$, then episode α occurs at interval $[t_s , t'_e)$ for some t'_e such that $t'_e - t_s \leq win2$. Formally this can be expressed in the following way. Given win_1 and β, denote mo $(win_1(\beta)) = \{ [t_s , t_e) \in mo(\beta) \mid te - ts \leq win_1\}$. Further, given α and an interval $[u_s ,u_e)$, define $occ(\alpha; [u_s , u_e)$ = true if and only if there exists a minimal occurrence $[u'_s \; u'_e) \in mo(\alpha)$ such that $u_s \leq u'_s$ and $u'_e \leq u_e$.

The confidence of an episode rule $\beta\, [win_1] \Rightarrow \alpha\, [win_2]$ is now

$$\frac{|\{[t_s, t_e) \in mo_{win_1}(\beta) \mid occ(\alpha, [t_s, t_s + win_2))\}|}{|mo_{win_1}(\beta)|}$$

Example 3.2.4.2: Continuing the previous example, we have, e.g., the following rules and confidences. For the rule $\beta[3] \Rightarrow \gamma[4]$ we have |{[35 , 38) , [46 , 48) , [57 , 60) }| in the denominator and |{[35 , 38)}| in the numerator, so the confidence is 1/3. For the rule $\beta[3] \Rightarrow \gamma\,[5]$ the confidence is 1.

There exists a variety of possibilities for the temporal relationships in episode rules with two time bounds. For example, the partial order of events can be such that the left-hand side events follow or surround the unseen events in the right-hand side. Such relationships are specified in the rules since the rule right-hand side α is a superepisode of the left-hand side β, and thus α contains the partial order of each event in the rule. Alternatively, rules that point backwards in time can be defined by specifying that the rule $\beta[win_1] \Rightarrow \alpha[win_2]$ describes the case where episode β has a minimal occurrence at an interval $[t_s , t_e)$ with $t_e - t_s \leq win_1$, and episode α occurs at interval $[t'_s , t_e)$ for some t'_s such that $t_e - t'_s \leq win_2$. For brevity, we do not consider any alternative definitions. While frequency has a nice interpretation as the probability that a randomly chosen window contains the episode, the concept is not very useful with minimal occurrences: (1) there is no fixed window size, and (2) a window may contain several minimal occurrences of an episode. Instead of frequency, we use the concept of support, the number of minimal occurrences of an episode: the support of an episode α in a given event sequence s is $|mo(\alpha)|$. Similarly to a frequency threshold, we now use a threshold for the support: given a support threshold min sup, an episode α is frequent if $|mo(\alpha)| \geq$ min sup. The current episode rule discovery task can be stated as follows. Given an event sequence s, a class E of episodes, and a set W of time bounds, find all frequent episode rules of the form $\beta[win_1] \Rightarrow \alpha[win_2]$, where $\beta , \alpha \in \varepsilon$, $\beta \preceq \alpha$, and $win_1, win_2 \in W$ [1].

3.2.4.1 Finding Minimal Occurrences of Episodes

We know that the subepisodes of a frequent episode are frequent. Thus, we can use the main candidate generation algorithms used for WINEPI also for MINEPI. The minimal occurrences of a candidate episode α are located in the following way. In the first iteration of the main algorithm, mo(α) is computed from the input sequence for all episodes α of size 1. In the rest of the iterations, the minimal occurrences of a candidate α are located by first selecting two suitable subepisodes α_1 and α_2 of α, and then computing a temporal join between the minimal occurrences of α_1 and α_2. To be more specific, for serial episodes the two subepisodes are selected so that α_1 contains

all events except the last one and α_2 in turn contains all except the first one. The minimal occurrences of α are then found with the following specification:

$$mo(\alpha) = \{[t_s, u_e) \mid \text{there are } [t_s, t_e) \in mo(\alpha_1) \text{ and } [u_s, u_e) \in mo(\alpha_2) \text{ such that } t_s < u_s, t_e < u_e, \text{ and } [t_s, u_e) \text{ is minimal}\}.$$

For parallel episodes, the subepisodes α_1 and α_2 contain all events except one; the omitted events must be different. The minimal occurrences of a candidate episode α can be found in a linear pass over the minimal occurrences of the selected subepisodes α_1 and α_2. The time required for one candidate is thus $O|mo(\alpha_1)| + |mo(\alpha_2)| + |mo(\alpha)|$, which is O(n), where n is the length of the event sequence. To optimize the running time, α_1 and α_2 can be selected so that $|mo(\alpha_1)| + |mo(\alpha_2)|$ is minimized. While minimal occurrences of episodes can be located quite efficiently, the size of the data structures can be even larger than the original database, especially in the first couple of iterations. Finally, note that MINEPI can be used to solve the task of WINEPI. Namely, a window contains an occurrence of an episode exactly when it contains a minimal occurrence. The frequency of an episode α can thus be computed from mo(α) [1].

3.2.4.2 Finding Confidences of Rules

An episode rule with two time bounds is defined as an expression $\beta[win_1] \Rightarrow \alpha[win_2]$; where β and α are episodes such that $\beta \preceq \alpha$, and win_1 and win_2 are integers. To find such rules, first note that for the rule to be frequent, the episode α has to be frequent. Rules of the above form can thus be enumerated by looking at all frequent episodes α, and then looking at all subepisodes β of α. The evaluation of the confidence of the rule $\beta[win_1] \Rightarrow \alpha[win_2]$ can be done in one pass through the structures mo(β) and mo(α), as follows. For each $[ts , te) \in mo(\beta)$ with $te - ts \leq win1$, locate the minimal

occurrence [us , ue) of α such that ts ≤ us and [us , ue) is the first interval in mo(α) with this property. Then check whether u_e - $t_s \leq win_2$.

3.3 Microarchitecture and Branch Prediction

3.3.1 Microarchitecture

When you specify the computational, communication and storage elements of a computer system and how these hardware components interact and how they are controlled, this can be called a computer microarchitecture. You can determine which computations can be performed most efficiently and which forms of program and data organization will perform optimally by considering the architecture of a machine [18].

The term architecture as applied to computer design, was first used in 1964 by Gene Amdahl, G. Anne Blaauw, and Frederick Brooks, Jr., the designers of the IBM System/360. The System/360 marked the introduction of families of computers, that is, a range of hardware systems all executing essentially the same basic machine instructions. The System/360 also precipitated a shift from the preoccupation of computer designers with computer arithmetic, which had been the main focus since the early 1950s. In the 1970s and 1980s, computer architects focused increasingly on the instruction set. In the current decade, however, designers' main challenges have been to implement processors efficiently, to design communicating memory hierarchies, and to integrate multiple processors in a single design [19].

3.3.2 Basic Microcomputer Design

Figure 3 [20] shows the basic design of a hypothetical microcomputer. CPU (The Central Processor Unit), where calculations and logic operations take place, contains a limited number of storage locations named registers, a high-frequency clock, a control unit, and an arithmetic logic unit.

- The clock synchronizes the internal operations of the CPU with other system components.
- CU (Control Unit) coordinates the sequencing of steps involved in executing machine instructions.
- ALU (The Arithmetic Logic Unit) performs arithmetic operations such as addition and subtraction and logical operations such as AND, OR, and NOT.

The CPU is attached to the rest of the computer via pins attached to the CPU socket in the computer’s motherboard. Most pins connect to the data bus, the control bus, and the address bus. The memory storage unit is where instructions and data are held while a computer program is running. The storage unit receives requests for data from the CPU, transfers data from RAM (Random Access Memory) to the CPU, and transfers data from the CPU into memory. A bus is a group of parallel wires that transfer data from one part of the computer to another. A computer’s system bus usually consists of three separate buses: the data bus, the control bus, and the address bus. The data bus transfers instructions and data between the CPU and memory. The control bus uses binary signals to synchronize actions of all devices attached to the system bus. The address bus holds the addresses of instructions and data when the currently executing instruction transfers data between the CPU and memory. Many personal computers use the PCI (Peripheral Component Interconnect) bus developed by Intel Corporation. In addition, many computers have a PCI Express graphics slot, which is significantly faster than the older AGP graphics slot [20].

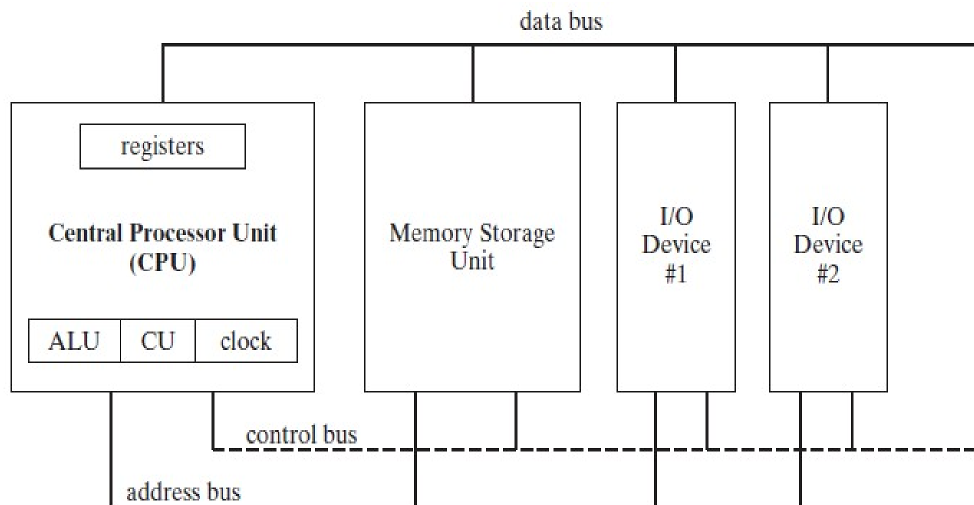


Figure 3: Block Diagram of a Microcomputer

A microprocessor incorporates most or all of the functions of a CPU on a single IC (Integrated Circuit). The first microprocessors emerged in the early 1970s and were used for electronic calculators, using BCD (Binary-Coded Decimal) arithmetic on 4-bit words. Other embedded uses of 4- and 8-bit microprocessors, such as terminals, printers and various kinds of automation followed rather quickly. Affordable 8-bit microprocessors with 16-bit addressing also led to the first general purpose microcomputers in the mid-1970s [18].

In computer engineering, microarchitecture (sometimes abbreviated to µarch or uarch) is the way a given ISA (Instruction Set Architecture) is implemented on a processor. A given ISA may be implemented with different microarchitectures. Implementations might vary due to different goals of a given design or due to shifts in technology. Computer architecture is the combination of microarchitecture and instruction set design [21].

3.3.3 Clock and Instruction Execution Cycle

An internal clock pulsing at a constant rate synchronizes the system bus and CPU for each operation. Clock cycle or in other words a machine cycle is the basic unit of time for machine instructions. The time required for one complete clock pulse gives the length of a clock cycle. In the Figure 4, a clock cycle is depicted as the time between one falling edge and the next:

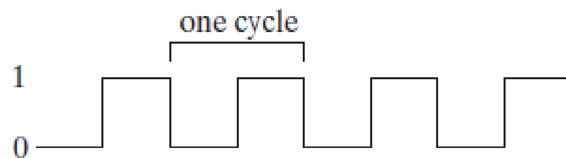


Figure 4: One clock cycle

The duration of a clock cycle is the reciprocal of the clock's speed, measured in oscillations per second. A clock that oscillates 1 billion times per second (1 GHz), for example, produces a clock cycle with a duration of one billionth of a second (1 nanosecond). A machine instruction requires at least one clock cycle to execute, and a few require in excess of 50 clocks (the multiply instruction on the 8088 processor, for example). Instructions requiring memory access often have empty clock cycles called wait states because of the differences in the speeds of the CPU, the system bus, and memory circuits [21].

Instruction execution cycle is an individual operation of the execution of a single machine instruction. A program is firstly loaded into memory before the execution. Here, the instruction queue contains the instructions waiting to be executed. Executing a machine instruction requires three basic steps: fetch, decode and execute. Two more steps are required when the instruction uses a memory operand: fetch operand and store output operand.

Each of the steps [22] is described as follows:

1. Fetch the instruction from main memory: The CPU uses the value of the PC (Program Counter) on the address bus. The CPU then fetches the instruction from main memory via the data bus into the MDR (Memory Data Register). The value from the MDR is then placed into the CIR (Current Instruction Register), a circuit that holds the instruction temporarily so that it can be decoded and executed.

2. Decode the instruction: The instruction decoder interprets and implements the instruction. The IR (Instruction Register) holds the current instruction, while the PC (Program Counter) holds the address in memory of the next instruction to be executed.

3. Fetch data from main memory: Read the effective address from main memory if the instruction has an indirect address. Fetch required data from main memory to be processed and place it into data registers.

4. Execute the instruction: From the instruction register, the data forming the instruction is decoded by the control unit. It then passes the decoded information as a sequence of control signals to the relevant function units of the CPU to perform the actions required by the instruction such as reading values from registers, passing them to the ALU to add them together and writing the result back to a register. A condition signal is sent back to the control unit by the ALU if it is involved.

5. Store results: Also called write back to memory. The result generated by the operation is stored in the main memory, or sent to an output device. Based on the condition feedback from the ALU, the PC is either incremented to address the next instruction or updated to a different address where the next instruction will be fetched. The cycle is then repeated.

3.3.4 Branch Prediction

For the performance of a pipelined processor, branch prediction is one of the most important techniques. The main point about branch prediction is that it enables the processor to begin executing instructions before the branch outcome is certain. If the prediction is not correct, then the penalty will be called the branch delay. A two-way branching is usually implemented with a conditional jump instruction. A conditional jump can either be "not taken" and continue execution with the first branch of code which follows immediately after the conditional jump - or it can be "taken" and jump to a different place in program memory where the second branch of code is stored. It is not known for certain whether a conditional jump will be taken or not taken until the condition has been calculated and the conditional jump has passed the execution stage in the instruction pipeline [23].

If branch prediction is not used, then the conditional jump instruction should be waited to pass the execute stage before the next instruction can enter the fetch stage in the pipeline. The mission of the branch predictor is to avoid this waste of time by trying to guess whether the conditional jump is most likely to be taken or not taken. After guessing the branch to be the most likely, this branch is fetched and executed speculatively. If after the execution of the conditional jump, it is decided that the branch taken is wrong, then the speculatively executed or partially executed instructions are discarded and the pipeline starts over with the correct branch [24].

If a branch misprediction occurs, the time wasted here is equal to the number of stages in the pipeline from the fetch stage to the execute stage. Modern microprocessors tend to have quite long pipelines so that the misprediction delay is between 10 and 20 clock cycles. The longer the pipeline the higher the need for a good branch predictor. You can not make a prediction about a conditional jump when it is executed for the first time. Here,the branch predictor keeps records of whether branches are taken or not taken. When it encounters a conditional jump that has been seen several times before then it can base the prediction on the past history. The branch predictor may, for example, recognize that the conditional jump is taken

more often than not, or that it is taken every second time [23].

We can talk about three main types of branch predictions. These are static branch prediction, dynamic branch prediction and branch profiling. Static branch prediction always predicts a branch is taken or not taken. We meet two kinds of branch instruction in most programs, conditional branch instructions and unconditional branch instructions. An unconditional branch effectively has a condition that is always true. In addition, a large majority of conditional branches are loop back edges taken and only terminate when the conditions are not satisfied for one time. So it is reasonable to predict all branches are taken, and a lot of tests show that it can achieve very high accuracy. Dynamic prediction uses the history of outcome of branch instructions to make it more accurate. Most branch predictors use a 2-bit prediction scheme [25].

Branch profiling can be considered as a type of branch prediction which can be located between static and dynamic is branch profiling. This method uses branch history information gathered during one run of a program to improve the branch prediction accuracy during following runs. Improving the accuracy may be done statically by creating a compiler that uses the collected profile information to modify its static prediction for individual branches [26].

A branch target predictor is the part of a processor that predicts the target of a taken conditional branch or an unconditional branch instruction before the target of the branch instruction is computed by the execution unit of the processor [27].

Branch target prediction is not the same as branch prediction. Branch prediction attempts to guess whether a conditional branch will be taken or not-taken. In more parallel processor designs, as the instruction cache latency grows longer and the fetch width grows wider, branch target extraction becomes a bottleneck.

The recurrence is:

- Instruction cache fetches block of instructions
- Instructions in block are scanned to identify branches
- First predicted taken branch is identified
- Target of that branch is computed
- Instruction fetch restarts at branch target

3.3.5 IPC , ILP and Performance

High performance processors are increasing the amount of speculative work they perform in order to improve instruction-level parallelism they discover. Branch prediction is the main method of providing speculative opportunities for a processor, therefore the accuracy of branch prediction is becoming very important [26].

3.3.5.1 How Programs Run - Load and Execute Process

The order of the events which occurred while running a program from the command prompt can be given as in the following way [28]:

- The operating system (OS) searches for the program's filename in the current disk directory. If it cannot find the name there, it searches a predetermined list of directories for the filename. If the OS fails to find the program filename, it issues an error message.
- If the program file is found, the OS retrieves basic information about the program's file from the disk directory, including the file size and its physical location on the disk drive.
- The OS determines the next available location in memory and loads the program file into memory. It allocates a block of memory to the program and

enters information about the program’s size and location into a table which is sometimes called descriptor table. Additionally, the OS may adjust the values of pointers within the program so they contain addresses of program data.

- The OS executes a branching instruction that causes the CPU to begin execution of the program’s first machine instruction. As soon as the program begins running, it is called a process. The OS assigns the process an identification number which is called the process ID, and this is used to keep track of it while running.
- The process runs by itself. It is the OS’s job to track the execution of the process and to respond to
- requests for system resources. Examples of resources are memory, disk files, and input-output devices.
- When the process ends,its handle is removed and the memory it used is released so it can be used by other programs.

3.3.5.2 IPC (Instruction Per Cycle)

IPC can be considered as an aspect of a processor's performance. It can be defined as the average number of instructions executed at each clock cycle and it is the multiplicative inverse of cycles per instruction (CPI). The number of instructions per second for a processor can be derived by multiplying the instructions per cycle and the clock speed which can be measured in cycles per second or Hertz [Hz] of a processor. The number of instructions per second is an approximate indicator of the likely performance of the processor [29].

The number of instructions executed per clock is not a constant for a given processor; it depends on how the particular software being run interacts with the processor, and indeed the entire machine, particularly the memory hierarchy. When comparing different instruction sets, a simpler instruction set may lead to a higher IPC figure than an implementation of a more complex instruction set using the same

chip technology; however, the more complex instruction set may be able to achieve more useful work with fewer instructions [28].

The useful work that can be done with any computer depends on many factors besides the processor speed. These factors include the processor architecture, the internal layout of the machine, the speed of the disk storage system, the speed of other attached devices, the efficiency of the operating system, and most importantly the high level design of the application software in use. For users and purchasers of a computer system, instructions per clock is not a particularly useful indication of the performance of their system. For an accurate measure of performance relevant to them, application benchmarks are much more useful. Awareness of its existence is useful, in that it provides an easy-to-grasp example of why clock speed is not the only factor relevant to computer performance [29].

3.3.5.3 ILP (Instruction-level Parallelism)

"Instruction-level Parallelism (ILP) is a family of processor and compiler design techniques that speed up execution by causing individual machine operations, such as memory loads and stores, integer additions and floating point multiplications, to execute in parallel. It is a measure of how many of the operations in a computer program can be performed simultaneously" [30].

Pipelining can overlap the execution of instructions when they are independent of one another. This potential overlap among instructions is called instruction-level parallelism (ILP) since the instructions can be evaluated in parallel. The amount of parallelism available within a basic block is quite small. The average dynamic branch frequency in integer programs was measured to be about 15%, meaning that about 7 instructions execute between a pair of branches. Since the instructions are likely to depend upon one another, the amount of overlap we can exploit within a basic block is likely to be much less than 7. To obtain substantial performance enhancements, we must exploit ILP across multiple basic blocks [26].

One of the goals in compiler and processor design is to identify and take advantage of as much ILP as possible. Generally, programs are written in a sequential form and instructions execute one after the other in an order specified by the programmer. ILP allows the compiler and the processor to overlap the execution of multiple instructions or even to change the order in which instructions are executed. ILP existence level in a program is changeable for programs and it is really application specific. In some fields, such as graphics and scientific computing the amount can be very large. On the other hand, workloads such as cryptography exhibit much less parallelism [31].

The simplest and most common way to increase the amount of parallelism available among instructions is to exploit parallelism among iterations of a loop. This type of parallelism is often called loop-level parallelism.

Example 3.3.5.3.1 :

```
for (i=1; i<=1000; i= i+1)
x[i] = x[i] + y[i];
```

This is a parallel loop. Every iteration of the loop can overlap with any other iteration, although within each loop iteration there is little opportunity for overlap [25].

Example 3.3.5.3.2 :

```
for (i=1; i<=100; i= i+1){
   a[i] = a[i] + b[i]; //s1
   b[i+1] = c[i] + d[i];//s2
}
```

Statement s1 uses the value assigned in the previous iteration by statement s2, so there is a loop-carried dependency between s1 and s2. Despite this dependency, this loop can be made parallel because the dependency is not circular:

- Neither statement depends on itself;
- While s1 depends on s2, s2 does not depend on s1.

A loop is parallel unless there is a cycle in the dependencies, since the absence of a cycle means that the dependencies give a partial ordering on the statements. To expose the parallelism, the loop must be transformed to conform to the partial order.

Two observations are critical to this transformation:

- There is no dependency from s1 to s2. Then, interchanging the two statements will not affect the execution of s2.
- On the first iteration of the loop, statement s1 depends on the value of b[1] computed prior to initiating the loop.

This allows us to replace the loop above with the following code sequence, which makes possible overlapping of the iterations of the loop [26]:

```
a[1] = a[1] + b[1];
 for (i=1; i<=99; i= i+1){
   b[i+1] = c[i] + d[i];
   a[i+1] = a[i+1] + b[i+1];
 }
 b[101] = c[100] + d[100];
```

Example 3.3.5.3.3 :

```
for (i=1; i<=100; i= i+1){
 a[i+1] = a[i] + c[i]; //S1
 b[i+1] = b[i] + a[i+1]; //S2
}
```

This loop is not parallel because it has cycles in the dependencies, namely the statements S1 and S2 depend on themselves [26].

Micro-architectural techniques that are used to exploit ILP include [31] :

- Instruction pipelining where the execution of multiple instructions can be partially overlapped.
- Superscalar execution in which multiple execution units are used to execute multiple instructions in parallel. In typical superscalar processors, the instructions executing simultaneously are adjacent in the original program order.
- Out-of-order execution where instructions execute in any order that does not violate data dependencies. Note that this technique is independent of both pipelining and superscalar.
- Register renaming which refers to a technique used to avoid unnecessary serialization of program operations imposed by the reuse of registers by those operations, used to enable out-of-order execution.
- Speculative execution which allow the execution of complete instructions or parts of instructions before being certain whether this execution should take place. A commonly used form of speculative execution is control flow speculation where instructions past a control flow instruction (e.g., a branch) are executed before the target of the control flow instruction is determined. Several other forms of speculative execution have been proposed and are in use including speculative execution driven by value prediction, memory dependence prediction and cache latency prediction.

- Branch prediction which is used to avoid stalling for control dependencies to be resolved. Branch prediction is used with speculative execution.

CHAPTER 4

EPISODE MINING ON ARCHITECTURE SIMULATION DATA

4.1 General Overview

In this thesis work, we have pursued two main goals;

- Operating data mining techniques about episode mining on architecture simulation data and showing the previously known relationships between the events occurred during the execution of the programs
- Searching for the interesting relationships between the events and finding out rules correlated with the IPC values obtained during the execution. Then, with the help of these relationships, to make predictions about the performance about a given program's execution in an architecture.

In order to accomplish this, three types of techniques have been implemented and these implementations have lead to a data mining tool specifically about episode mining on event sequences which we call "EMT (Episode Mining Tool)". EMT has three components, namely "Data Pre-processor, Episode Miner and Output Analyser". EMT supports both a command line and as well as a GUI interface. In command line version the parameters are taken from command line for pre-processing operations or the related method of episode mining operation and the results are printed to an output file. In GUI version, the parameters are taken from the GUI components and the results are printed to the output file. And also, the results can be analysed with the help of output analyser component by visualizing the

outputs about the episodes or rules generated in various types of charts. In addition to visualizing, analysing multiple output files in a single process is enabled here.

We have worked on event sequence data, which is modelled as a sequence of events and each event has an associated time of occurrence. Therefore, EMT can be used to analyse any dataset which is an event sequence. The only condition is the format of the data. The input structure must follow this simple format in each line:

<Time of occurrence> <Event type>

4.2 Implementation Details of The Algorithms

The three types of algorithms implemented during this research are;

- Mannila's window episode mining algorithms (WINEPI) for parallel and serial episodes [1]
- Non-overlapping occurrence counting algorithms for parallel an serial episodes [2]
- Mannila's minimal occurrence based algorithms (MINEPI) for serial and parallel episodes [1]

In general, we directly implemented the original format of these algorithms but in some parts we had to make changes due to the differences between the structure of our dataset and the reference dataset of these algorithms.

1-WINEPI implementation:WINEPI is a window based approach and there are four parameters such as “window width, frequency threshold, confidence threshold and maximum length of episodes to be generated”.

- Window width: As mentioned in the background information chapter for WINEPI algorithms the event sequence is sliced into windows that has a starting time point (t_s) and an ending time point (t_e). While calculating the frequency counts, the count of the windows that the episode occurs are considered and this window width shows the size of these windows.
- Frequency Threshold: This is the threshold value for determining the frequent episodes. If the episodes frequency value is greater than this threshold, then it is considered as a frequent episode, it is infrequent otherwise.
- Confidence Threshold: After generating the frequent episodes, the rules are generated based on these frequent episodes and this parameter is used to determine if a rule is interesting or not.
- Maximum Length: Other parameters except this one occur in the original specification of the algorithm. This value is used to generate the rules of the episodes having a specified maximum length. Originally the algorithm does not terminate until the generated set of frequent episodes is an empty set. However, for instance, we may want to use a small threshold value and in our event sequence there may be frequent episodes with size "10" or "15" although we want to investigate only episodes with length "3" or smaller. Assuming that we have a huge data set in terms of gigabytes, the execution time will be out of acceptable borders in this situation. So, when we give this maximum length parameter to our implementation we can get and analyse the outputs about our dataset in a less amount of time.

We also have a different approach about processing the input dataset in our implementations. Here, the input datasets may be huge sized files. Thus, reading the input file for once and taking it into memory and processing it may seem to reduce the time cost as reducing the I/O operations. But this time we have to face with the memory constraints because for instance trying to read a 10 GB input into memory and producing the necessary data structures about the algorithm for this data will not be feasible and if you are not running a server type computer having enough memory for this operation, the program can not be executed and will lead an out of memory

error. Therefore, to avoid such a situation, we used an iterative technique for processing the input data such as holding only a window of the event sequence in memory and getting a new time point (it may include an event or not) and removing the first time point of the window (again it may include an event or not) in each iteration. As a result, we developed a sliding window mechanism while processing the input values. For example, assume that our window width is 5 and in an iteration we have a window containing events as in Figure 5.

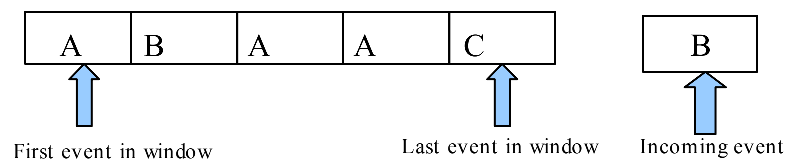


Figure 5: Before reading a new event from sequence

After reading an event, for instance with type "B" from the input value, our window will be as in Figure 6;

Figure 6: After reading a new event from sequence

The candidate generation algorithm for both parallel and serial episodes have been implemented as given in the original versions for WINEPI. Also, the rule generation for parallel and serial episodes and the recognition algorithm for calculating the frequencies of parallel episodes have been implemented as in the original forms of the algorithms.

On the other hand, we had to make some changes in recognizing the occurrences of serial episodes in WINEPI approach. Mannila refers to a telecommunication network alarm dataset while presenting algorithms and here also it is stated that:

"It is useless to have multiple automata in the same state, as they would only make the same transitions and produce the same information. It suffices to maintain the one that reached the common state last since it will be also removed last. There are thus at most |α| automata for an episode α. For each automaton we need to know when it should be removed. We can thus represent all the automata for α with one array of size |α|: the value of α.initialized[i] is the latest initialization time of an automaton that has reached its ith state. Recall that α itself is represented by an array containing its events; this array can be used to label the state transitions" [1].

As a result of this approach there are some problems about calculating the occurrences of the serial episodes according to the original form of the algorithm. Especially, the problem can be seen if there are multiple occurrences of the same event in a single window. To indicate this, we can review an example sequence such as;

s={<1,1>,<2,2>,<3,3>,<1,4>,<2,5>,<3,6>}

For this example, if we trace the Algorithm 5 given to calculate the occurrences of serial episodes in [18] for the WINEPI approach with window width = 4 for candidate episode {1}, we obtain the transitions;

s={1,2,3,1,2,3}	Ts=1 Te=7
WinWidth=4	
Candidates: {1}	

t=1

Start=-2 t=1	
Initialized	[1]:1
transitions	({1},1,1)
beginsat	[1]:({1},1)
waits	({1},1)
freq_count	0
inwindow	-2

…

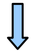
t=4

Start=1 t=4	
initialized	[1]:4
transitions	({1},1,4)
beginsat	[1]:({1},1) [4]:({1},1)
waits	({1},1)
freq_count	0
inwindow	-2

t=5

Start=2 t=5	
initialized	[1]:0
transitions	
beginsat	[1]:({1},1) [4]:({1},1)
waits	({1},1)
freq_count	4
inwindow	-2

...



t=8

Start=5 t=8	
initialized	[1]:0
transitions	
beginsat	[1]:({1},1) [4]:({1},1)
waits	({1},1)
freq_count	11
inwindow	-2

…

At the end of execution, the frequency count of episode {1} is found as 11. But in fact the accurate value must be 7. The basic reason of this wrong result is that the values for "initialized" and "inwindow" does not change where they should do. As

stated before, this occurs when there are multiple occurrences of an event in a single window.

We guess that the input dataset have time spaces between the alarm events and the same type of alarm does not repeatedly occur in a single window in experiments given in the reference work. But in our dataset, we use cycle values as time of occurrences of events and this situation may occur in our event sequences. Therefore, we needed to design a straight-forward algorithm for recognizing the occurrences of the serial episodes. About this recognition operation, it is given that:

"A practical alternative to the recognition of general episodes is to handle all episodes basically like parallel episodes, and to check the correct partial ordering only when all events are in the window. Parallel episodes can be located efficiently; after they have been found, checking the correct partial ordering is relatively fast " [18].

Then, simply by using the concepts of sliding window mechanism, we developed a straight-forward algorithm. Here, we just get the events into the window and check whether each candidate episode occurs in this window or not. There is an important point to be checked here because there may be multiple events in a cycle, means in a time point as given in Figure 7;

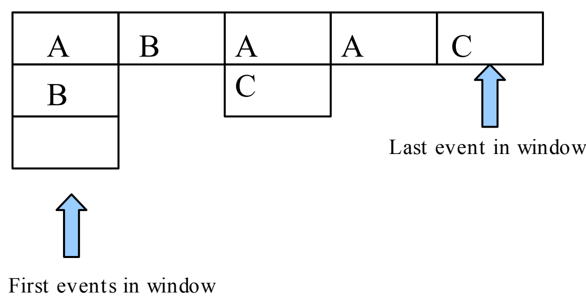


Figure 7: Processing multiple events in a single cycle

In such a situation our algorithm should handle all of the combinations of this window content. Here, to check whether a candidate episode such as "{C,C,A,C}" we should check these combinations of the window;

- {A,B,A,A,C}:No occurrence ⇒ X
- {B,B,A,A,C}:No occurrence ⇒ X
- {C,B,A,A,C}:No occurrence ⇒ X
- {A,B,C,A,C}:No occurrence ⇒ X
- {B,B,C,A,C}:No occurrence ⇒ X
- {C,B,C,A,C}:Episode occurs ⇒ √

The process of our algorithm can be given as;

1 Read a cycle from the input sequence.

2 If there is an occurrence of event type 'A' in the incoming cycle add this event to the window,else add a null event to the window.

3 Remove the first cycle instance in the window.

4 For all candidate episodes ;

4.1 For all combinations of cycles containing multiple events, generate a window;

4.2 Check if the candidate episode occurs in the generated window,

4.3 If there is an occurrence

4.3.1 Increment the frequency count for this episode,

4.3.2 Go to the step 4, to process a new candidate episode.

4.4 Else, go on for the next generated window.

5 Repeat this process until the end of the event sequence.

2-Non-overlapping occurrence based implementation: In this part, non-overlapping occurrences of the episodes are calculated and with the given threshold values the rules are generated from the frequent episodes. For this approach, the original forms

of the algorithms given in [2] for parallel and serial episodes have been implemented. In order to generate candidate episodes in each iteration and rule generation process the same method in WINEPI implementation has been used.

3-MINEPI implementation: Here, the minimal occurrences of the candidate episodes are calculated and there is one more parameter as a difference from the previous approaches. We use two window width values as given in the algorithms for MINEPI and the implementation for this part is done with respect to the original algorithms given in [18]. Again the same method in WINEPI for generating candidates in each iteration and rule generation process are applied here.

4.3 Experiments and Results

During this thesis work we worked with different datasets for different analysis. The datasets contain events and their cycle values corresponding to the time of occurrence values in our algorithms. The events in datasets used in experiments are instance based events and they do not have duration values. Events occur in a time value and there is not a time interval for the occurrences of the events. Here, firstly we considered all types of events and worked with a huge dataset. However, we saw that to extract useful information from such a large domain would not be feasible. Therefore, we decided to concentrate on only some specific types of events and ignore the other types. For further analysis, we added event types about IPC values to our architectural events. But here we realised that in the output of programs' execution in an architecture, some sequences appeared repeatedly and the execution sequences consisted of these unique sequences.

Here, these sequences corresponds to the program blocks such as conditional branches or loop structures. Therefore, finally we decided to investigate these unique sequences and find out relationships between the rules generated from these sequences and the IPC values during the execution of them. The investigated datasets, their structures and the results are described in the following experimental details:

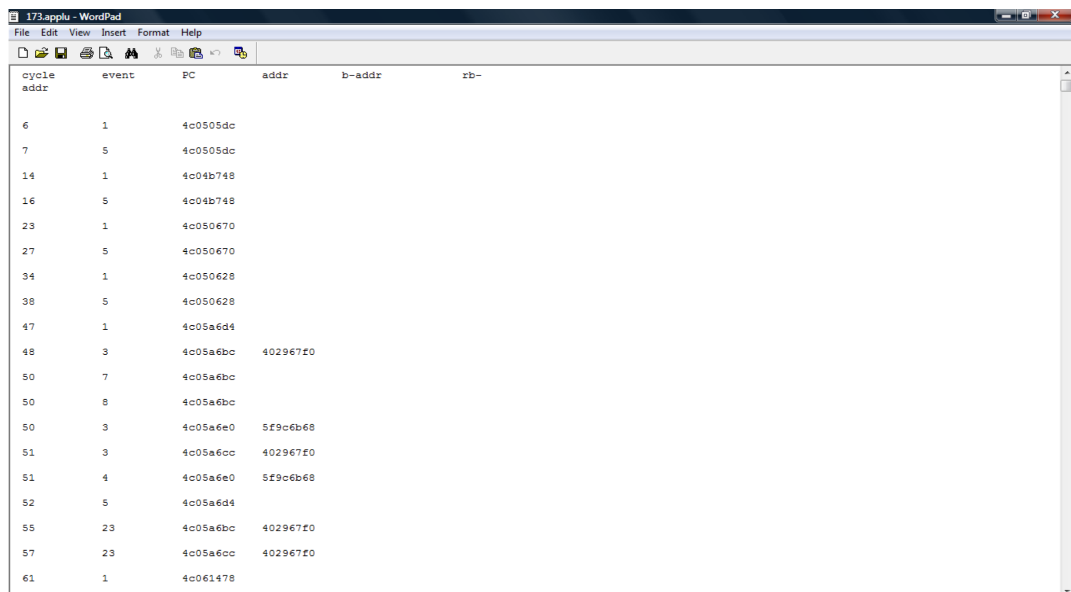
4.3.1 Experiment 1

Input Data Structure: The dataset used in this step consists all event types of data obtained from the architecture simulation. Thus, the benchmarks have large sizes and the format of the input files are as in the following:

```
<cycle> <event> <inst-PC> <mem-addr> <block-addr> <replacement-block-addr>
```

Here, we are interested in the first two columns:the cycle values corresponding to the time of occurrences in our algorithms and the event types.

The input data format is shown in Figure 8:


173.applu - WordPad

cycle	event	PC	addr	b-addr	rb-addr
6	1	4c0505dc			
7	5	4c0505dc			
14	1	4c04b748			
16	5	4c04b748			
23	1	4c050670			
27	5	4c050670			
34	1	4c050628			
38	5	4c050628			
47	1	4c05a6d4			
48	3	4c05a6bc	402967f0		
50	7	4c05a6bc			
50	8	4c05a6bc			
50	3	4c05a6e0	5f9c6b68		
51	3	4c05a6cc	402967f0		
51	4	4c05a6e0	5f9c6b68		
52	5	4c05a6d4			
55	23	4c05a6bc	402967f0		
57	23	4c05a6cc	402967f0		
61	1	4c061478			

Figure 8: Input file for experiment-1

The simulation data is obtained from an experimental architecture that implements speculation in a 2-threaded multi-threaded architecture.

Here, the event types existing in this dataset and their meanings are:

1: Branch misprediction.

2: L1 data cache miss

3: L2 data cache miss

4: Load misspeculation

5: Roll back due to Branch misprediction

6: Roll back due to Load misspeculation

21: Branch misprediction.

22: L1 data cache miss

23: L2 data cache miss

24: Load misspeculation

25: Roll back due to Branch misprediction

26: Roll back due to Load misspeculation

7: Main-thread enters the runahead mode

8: Recovery-thread is forked

9: Recovery-thread rolls back due a miss-branch, the main-thread is killed.

10: Recovery-thread rolls back due a wrong-value detection, the main-thread is killed.

11: Main-thread enters the blocking mode.

12: Main-thread is running the runahead mode, it is killed.

13: Main-thread is running the blocking mode. Recovery process is done, the recovery-thread stops.

100: L1 data cache: a block(block-addr) is fetched from L2, a victim(replacement-block-addr) block is kicked out if conflicted.

200: L2 data cache: a block(block-addr) is fetched from memory, a victim(replacement-block-addr) block is kicked out if conflicted.

Results: We obtained different result sets with different parameters from EMT about this dataset. As an example the result output for the parallel episodes and the generated rules with the WINEPI approach can are shown in Figure 9 and Figure 10.

```
output_input_ilk_26_08_09 - WordPad
File Edit View Insert Format Help

Date: 2009-08-26 09:42:54
window Width: 4
minimum frequency:0.0
maximum episode size:4
maximum cycle of the input file: 12377038
*********Mannila Approach Parallel episodes results***********
episode: 1 freq_count: 221544 frequency: 0.017899600518250087
episode: 5 freq_count: 212048 frequency: 0.017132373211162996
episode: 3 freq_count: 11237 frequency: 9.078910330389279E-4
episode: 7 freq_count: 1644 frequency: 1.3282663151339302E-4
episode: 8 freq_count: 1644 frequency: 1.3282663151339302E-4
episode: 4 freq_count: 8838 frequency: 7.140643365665253E-4
episode: 23 freq_count: 7323 frequency: 5.916602327083803E-4
episode: 6 freq_count: 2812 frequency: 2.271949439268012E-4
episode: 200 freq_count: 13848 frequency: 0.0011188462245726684
episode: 100 freq_count: 479319 frequency: 0.03872647700144041
episode: 24 freq_count: 444 frequency: 3.587288588317914E-5
episode: 26 freq_count: 268 frequency: 2.165300319074777E-5
episode: 21 freq_count: 560 frequency: 4.5245081294099813E-5
episode: 25 freq_count: 452 frequency: 3.651924418738056E-5
episode: 9 freq_count: 452 frequency: 3.651924418738056E-5
episode: 11 freq_count: 24 frequency: 1.939074912604278E-6
episode: 10 freq_count: 204 frequency: 1.6482136757136363E-5
episode: 13 freq_count: 4 frequency: 3.2317915210071297E-7
episode: 12 freq_count: 984 frequency: 7.950207141677539E-5
episode: 22 freq_count: 22 frequency: 1.777485336553921 3E-6
episode: 2 freq_count: 381519 frequency: 0.030824746732577978
episode: 1 , 1 freq_count: 11165 frequency: 9.02073808301115E-4
episode: 1 , 5 freq_count: 68456 frequency: 0.005530888009051602
episode: 1 , 3 freq_count: 470 frequency: 3.797355037183377E-5
episode: 1 , 7 freq_count: 129 frequency: 1.0422527655247992E-5
```

Figure 9: Outputs for frequency of the episodes in experiment-1

```
output_input_ilk_26_08_09 - WordPad
File Edit View Insert Format Help

*******Rules Generated************
1 -> 1 , 1 ( confidence: 0.05039630953670603 )( Rule of Parallel
Episodes )
1 -> 1 , 5 ( confidence: 0.3089950529014553 )( Rule of Parallel
Episodes )
5 -> 1 , 5 ( confidence: 0.322832566211 4238 )( Rule of Parallel
Episodes )
1 -> 1 , 3 ( confidence: 0.002121474740909255 )( Rule of Parallel
Episodes )
3 -> 1 , 3 ( confidence: 0.04182611017175402 )( Rule of Parallel
Episodes )
1 -> 1 , 7 ( confidence: 5.822771097389232E-4 )( Rule of Parallel
Episodes )
7 -> 1 , 7 ( confidence: 0.07846715328467153 )( Rule of Parallel
Episodes )
1 -> 1 , 8 ( confidence: 5.822771097389232E-4 )( Rule of Parallel
Episodes )
8 -> 1 , 8 ( confidence: 0.07846715328467153 )( Rule of Parallel
Episodes )
1 -> 1 , 4 ( confidence: 0.010918824251615934 )( Rule of Parallel
Episodes )
4 -> 1 , 4 ( confidence: 0.27370445802217697 )( Rule of Parallel
Episodes )
1 -> 1 , 23 ( confidence: 7.763694796518976E-4 )( Rule of Parallel
Episodes )
23 -> 1 , 23 ( confidence: 0.023487641676908 37 )( Rule of Parallel
Episodes )
1 -> 1 , 6 ( confidence: 0.0010607373704546274 )( Rule of Parallel
Episodes )
6 -> 1 , 6 ( confidence: 0.08357041251778094 )( Rule of Parallel
Episodes )
1 -> 1 , 200 ( confidence: 0.001516628750947893 )( Rule of Parallel
Episodes )
200 -> 1 , 200 ( confidence: 0.024263431542461 )( Rule of Parallel
Episodes )
1 -> 1 , 100 ( confidence: 0.022573393998483373 )( Rule of Parallel
Episodes )
100 -> 1 , 100 ( confidence: 0.010433552602755159 )( Rule of Parallel
Episodes )
1 -> 1 , 24 ( confidence: 3.159643231141444E-4 )( Rule of Parallel
Episodes )
24 -> 1 , 24 ( confidence: 0.15765765765765763 )( Rule of Parallel
```

Figure 10: Outputs for confidence of the rules in experiment-1

Analysis: During program execution in a speculative processor, multiple branch mis-predictions may be observed in rapid succession. One of the main reasons behind this phenomenon is the exploitation of instruction-level parallelism. The processor issues multiple instructions at each cycle and never waits for the resolution of branches as long as pending branch instructions continue to resolve correctly, i.e., predictions continue to be correct. As a result, at any given time there are many branch instructions waiting for resolution. When one of these branch instructions is mis-predicted, several others preceding this branch might have been mis-predicted as well. The first rule shown in Figure 10 clearly shows this behaviour: a branch mis-prediction leads to multiple branch mis-predictions.

The second and the third rules also indicate expected behaviour. Once a mis-prediction is detected, a roll-back is initiated, i.e., 1 -> 1, 5.

On a few occasions, branch mis-predictions may lead to additional cache misses. The rule 1 -> 1, 3 shows such expected clustering of events. However, rules 1 -> 1, 7, 7 -> 1, 7; 1-> 1, 8; 8 -> 1, 8 all appear to be coincidental bearing no significance from an architectural perspective.

Rule 4 -> 1,4 indicates that an incorrect load value obtained from a load speculation may trigger branch mis-speculations, obviously unexpectedly. This rule yields information that is not common knowledge in computer architecture. Although a deeper analysis of the processor behaviour is needed to assess the frequency and the importance of the phenomenon, it clearly is a case which indicates that there is merit in investigating architectural simulation data using data mining techniques.

In order to understand this particular case better, let us review the process of load speculation in an ILP processor. Load speculation is the process of executing load instructions out of program order, before preceding store instructions complete.

Consider the following code:

```
I1:    SW  $8, a1
I2:    LW  $4, a2
```

If I2 is executed before I1 and a1 != a2, this will lead to improved performance because the instructions waiting for the value of register 4 can proceed sooner. If a1 = a2, the load instruction will obtain the stale value from the memory and a load mis-speculation will result.

We reason that the observed case arises because of the interaction of load mis-speculation with branch prediction and validation.

Consider the sequence:

```
I1:    SW  $8, a1
I2:    LW  $4, a2
I3:    Beq $4, $8, L1
```

and assume that I3 has been correctly predicted. However, if the load has been speculatively executed and the speculation is not successful the load will obtain the wrong value. The branch instruction, although correctly predicted, may be considered an incorrect prediction because the processor upon verifying the values of $4 and $8 does not find them to be equal. Note that the processor would correctly conclude that the branch was correctly predicted had the memory operations been executed in program order. As a result, we observe 4 -> 1,4, i.e., a load misspeculation leads to a branch misprediction as well as an additional load misspeculation.

Rules 1-> 1, 23;   23 -> 1, 23 may represent coincidental occurrences, but they also might represent scenarios where because of a branch mis-prediction the processor's touching of additional cache misses.

Clearly, such a case is possible when the program incorrectly takes a path that should not have been taken; since the data references in this unvisited region will not likely to be in the cache.

Rules 1-> 1,6; 6-> 1,6; 1 -> 1, 200 and 200 -> 1, 200 appear to be coincidences, albeit being frequent. Similarly, 1 -> 1, 24, and 24 -> 1,24 do not appear to be significant from an architectural perspective.

For the next step we decided to consider only specific types of events and search for the relationships between these event types.

4.3.2 Experiment 2

Input Data Structure: The dataset used in this step consists only four types of events about the function calls and branches in program executions in architecture simulation. Here, the dataset provided by the architecture was in binary format to reduce the size of the input file. We used a simple C code to transform this binary data into a text input and also for the uniformness of input files processed in these experimental steps to indicate the event types with numbers instead of characters as given in the binary file. The structure of the binary data and this code piece to pre-process these data can be given as;

```
// ----------------------------------------------------------------------
//  CFEVENT
//  Data        : cfCode    : Control flow's code
//                               'C' = Function call
//                               'R' = Function return
//                               'F' = Forward branch
//                               'B' = Backward branch
//                 : cycles    : Instruction cycles
//                 : pc        : Program counter
//                 : bt        : Branch target
//                 : ilpCount    : Number of committed stages
//                 : ilpTotal    : Number of total issues
//
//  The data structure uses to store single control flow's event
//  ----------------------------------------------------------------------
```

```
typedef struct {
  char cfCode;
  unsigned int cycles;
  unsigned int pc;
  unsigned int bt;
  short ilpCount;
  short ilpTotal;
} CFEVENT;


CFEVENT curCFEvent;
double curILP;
while(srcFile.read((char*)&curCFEvent, sizeof(CFEVENT))) {
          // Avoid division by zero
          if(curCFEvent.ilpCount == 0) curILP = 0.0;
          else curILP = curCFEvent.ilpTotal / ((double)curCFEvent.ilpCount);

          // Write to destination file
          destFile << curCFEvent.cycles << " ";

          if(curCFEvent.cfCode == 'C') destFile << "4 ";            // Function call
          else if(curCFEvent.cfCode == 'R') destFile << "-4 ";     // Function return
          else if(curCFEvent.cfCode == 'F') destFile << "2 ";      // Forward branch
          else if(curCFEvent.cfCode == 'B') destFile << "-2 ";   // Backward branch
          else destFile << "999999 ";

          destFile << curCFEvent.pc << " "; // program counter
          destFile << curCFEvent.bt << " "; // branch target
          destFile << curCFEvent.ilpCount << " ";
          destFile << curCFEvent.ilpTotal << " ";
          destFile << curILP << "\n";    }
```

After this pre-processing operation the input file for EMT was obtained. The structure of the input file can be given as in Figure 11:

sample - WordPad

File Edit View Insert Format Help

```
8 4 2147483647 2147483647 1 1 1
17 4 2147483647 2147483647 1 3 3
32 4 2147483647 2147483647 3 10 3.33333
45 4 2147483647 2147483647 2 9 4.5
160 4 2147483647 2147483647 7 23 3.28571
177 2 2147483647 2147483647 2 2 1
184 4 2147483647 2147483647 1 6 6
197 2 2147483647 2147483647 6 12 2
206 2 2147483647 2147483647 1 8 8
214 2 2147483647 2147483647 1 1 1
223 2 2147483647 2147483647 7 11 1.57143
228 2 2147483647 2147483647 0 0 0
247 2 2147483647 2147483647 12 26 2.16667
261 2 2147483647 2147483647 0 0 0
272 2 2147483647 2147483647 2 5 2.5
285 2 2147483647 2147483647 3 3 1
299 2 2147483647 2147483647 1 1 1
312 2 2147483647 2147483647 3 5 1.66667
314 -4 2147483647 2147483647 2 4 2
315 2 2147483647 2147483647 1 5 5
322 -2 2147483647 2147483647 1 3 3
323 4 2147483647 2147483647 1 4 4
336 2 2147483647 2147483647 4 10 2.5
338 2 2147483647 2147483647 2 8 4
349 2 2147483647 2147483647 2 2 1
360 -2 2147483647 2147483647 1 1 1
367 2 2147483647 2147483647 3 5 1.66667
372 2 2147483647 2147483647 1 2 2
372 2 2147483647 2147483647 2 4 2
383 2 2147483647 2147483647 2 7 3.5
383 2 2147483647 2147483647 1 4 4
```

Figure 11: Input file for experiment-2

Here, mainly we have four types of events and these events are :

4: Function Call

-4 : Function Return

2 : Forward Branch

-2: Backward Branch

We marked the events in the other types with the number "999999" and while processing the input file we did not took the episodes and rules including the event "999999" into account. We only considered these four types of events.

Results: We processed the input data with our three approaches and we did this for both parallel and serial episodes. While we examined the outputs, we noticed that we should consider the changes in the IPC values during the execution of the programs to make predictions about the performance of running the program in an architecture. So, for the next step we decided to prepare a new dataset containing the IPC values.

As an example, the results obtained for the frequencies of the serial episodes and the results for the confidence of the generated rules with the non-overlapping occurrences based algorithm can be given as in Figure 12 and Figure 13;

```
out_non_overlap_serial - WordPad
File  Edit  View  Insert  Format  Help

Date: 2009-10-01 08:36:16
window Width: 50
minimum frequency:0.0010
maximum episode size:4
maximum cycle of the input file: 2147483647
*********Non overlapping Serial episodes results***********
episode: 4 :freq_count: 0 :frequency: 7460206.0
episode: 2 :freq_count: 0 :frequency: 3.0451012E7
episode: -4 :freq_count: 0 :frequency: 7461418.0
episode: -2 :freq_count: 0 :frequency: 1.02446336E8
episode: 4 , 4 :freq_count: 0 :frequency: 3730103.0
episode: 4 , 2 :freq_count: 0 :frequency: 4415128.0
episode: 4 , -4 :freq_count: 0 :frequency: 4367016.0
episode: 4 , -2 :freq_count: 0 :frequency: 2694262.0
episode: 2 , 4 :freq_count: 0 :frequency: 4415128.0
episode: 2 , 2 :freq_count: 0 :frequency: 1.5225506E7
episode: 2 , -4 :freq_count: 0 :frequency: 5080184.0
episode: 2 , -2 :freq_count: 0 :frequency: 5171533.0
episode: -4 , 4 :freq_count: 0 :frequency: 4367015.0
episode: -4 , 2 :freq_count: 0 :frequency: 5080183.0
episode: -4 , -4 :freq_count: 0 :frequency: 3730709.0
episode: -4 , -2 :freq_count: 0 :frequency: 2675923.0
episode: -2 , 4 :freq_count: 0 :frequency: 2694262.0
episode: -2 , 2 :freq_count: 0 :frequency: 5171533.0
episode: -2 , -4 :freq_count: 0 :frequency: 2675923.0
episode: -2 , -2 :freq_count: 0 :frequency: 5.1223168E7
episode: 4 , 4 , 4 :freq_count: 0 :frequency: 2486735.0
episode: 4 , 4 , 2 :freq_count: 0 :frequency: 2536969.0
episode: 4 , 4 , -4 :freq_count: 0 :frequency: 2518044.0
episode: 4 , 4 , -2 :freq_count: 0 :frequency: 1786340.0
episode: 4 , 2 , 4 :freq_count: 0 :frequency: 2536969.0
episode: 4 , 2 , 2 :freq_count: 0 :frequency: 4031406.0
episode: 4 , 2 , -4 :freq_count: 0 :frequency: 4353801.0
episode: 4 , 2 , -2 :freq_count: 0 :frequency: 2688802.0
```

Figure 12: Outputs for frequency of the episodes in experiment-2

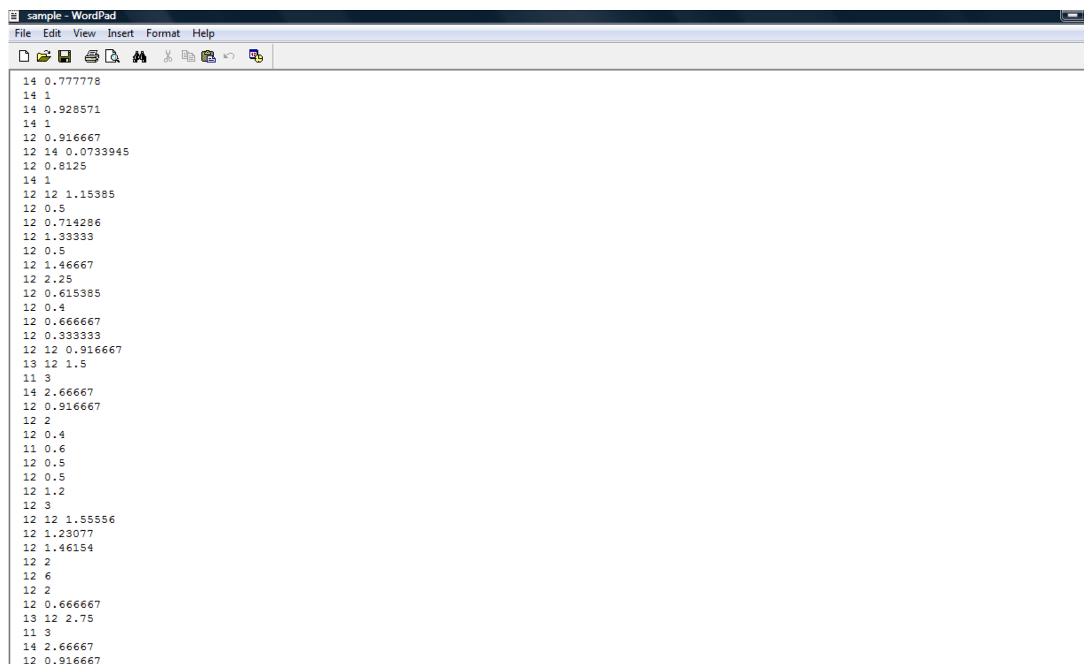

```
out_non_overlap_serial - WordPad
File  Edit  View  Insert  Format  Help

Date: 2009-10-01 08:36:16
window Width: 50
minimum frequency:0.0010
maximum episode size:4
maximum cycle of the input file: 2147483647
*********Non overlapping Serial episodes results***********
episode: 4 :freq_count: 0 :frequency: 7460206.0
episode: 2 :freq_count: 0 :frequency: 3.0451012E7
episode: -4 :freq_count: 0 :frequency: 7461418.0
episode: -2 :freq_count: 0 :frequency: 1.02446336E8
episode: 4 , 4 :freq_count: 0 :frequency: 3730103.0
episode: 4 , 2 :freq_count: 0 :frequency: 4415128.0
episode: 4 , -4 :freq_count: 0 :frequency: 4367016.0
episode: 4 , -2 :freq_count: 0 :frequency: 2694262.0
episode: 2 , 4 :freq_count: 0 :frequency: 4415128.0
episode: 2 , 2 :freq_count: 0 :frequency: 1.5225506E7
episode: 2 , -4 :freq_count: 0 :frequency: 5080184.0
episode: 2 , -2 :freq_count: 0 :frequency: 5171533.0
episode: -4 , 4 :freq_count: 0 :frequency: 4367015.0
episode: -4 , 2 :freq_count: 0 :frequency: 5080183.0
episode: -4 , -4 :freq_count: 0 :frequency: 3730709.0
episode: -4 , -2 :freq_count: 0 :frequency: 2675923.0
episode: -2 , 4 :freq_count: 0 :frequency: 2694262.0
episode: -2 , 2 :freq_count: 0 :frequency: 5171533.0
episode: -2 , -4 :freq_count: 0 :frequency: 2675923.0
episode: -2 , -2 :freq_count: 0 :frequency: 5.1223168E7
episode: 4 , 4 , 4 :freq_count: 0 :frequency: 2486735.0
episode: 4 , 4 , 2 :freq_count: 0 :frequency: 2536969.0
episode: 4 , 4 , -4 :freq_count: 0 :frequency: 2518044.0
episode: 4 , 4 , -2 :freq_count: 0 :frequency: 1786340.0
episode: 4 , 2 , 4 :freq_count: 0 :frequency: 2536969.0
episode: 4 , 2 , 2 :freq_count: 0 :frequency: 4031406.0
episode: 4 , 2 , -4 :freq_count: 0 :frequency: 4353801.0
episode: 4 , 2 , -2 :freq_count: 0 :frequency: 2688802.0
```


Figure 13: Outputs for confidence of the rules in experiment-2

4.3.3 Experiment 3

Input Data Structure: In this step, we not only took the events occurred during the execution of a program in an architecture,but also the mean of the IPC values and their changes. As in the previous step, again we were interested in four types of events about the branches and functions. The dataset used in this step can be given as in Figure 14:




```
14 0.777778
14 1
14 0.928571
14 1
12 0.916667
12 14 0.0733945
12 0.8125
14 1
12 12 1.15385
12 0.5
12 0.714286
12 1.33333
12 0.5
12 1.46667
12 2.25
12 0.615385
12 0.4
12 0.666667
12 0.333333
12 12 0.916667
13 12 1.5
11 3
14 2.66667
12 0.916667
12 2
12 0.4
11 0.6
12 0.5
12 0.5
12 1.2
12 3
12 12 1.55556
12 1.23077
12 1.46154
12 2
12 6
12 2
12 0.666667
13 12 2.75
11 3
14 2.66667
12 0.916667
```

Figure 14: Input file for experiment-3

Each line in this input file is a single machine cycle as in the previous experimental steps. Each column indicates an event with its type. Number of events in a single cycle is not fixed but it should have at least two events per cycle which are branch and IPC. This means that the last token in each line shows an IPC value and the previous tokens are the events occurred during this cycle.

Example 4.3.3.1: 14 0.777

14 12 0.5

14 14 12 8

- The first line means first cycle contains a function return and IPC of value 0.777
- The second line means second cycle contains a function return, a forward branch and IPC of value 0.5
- The third line means third cycle contains two function return, a forward branch and IPC of value 8.

As mentioned before, we have four types of events and these events are :

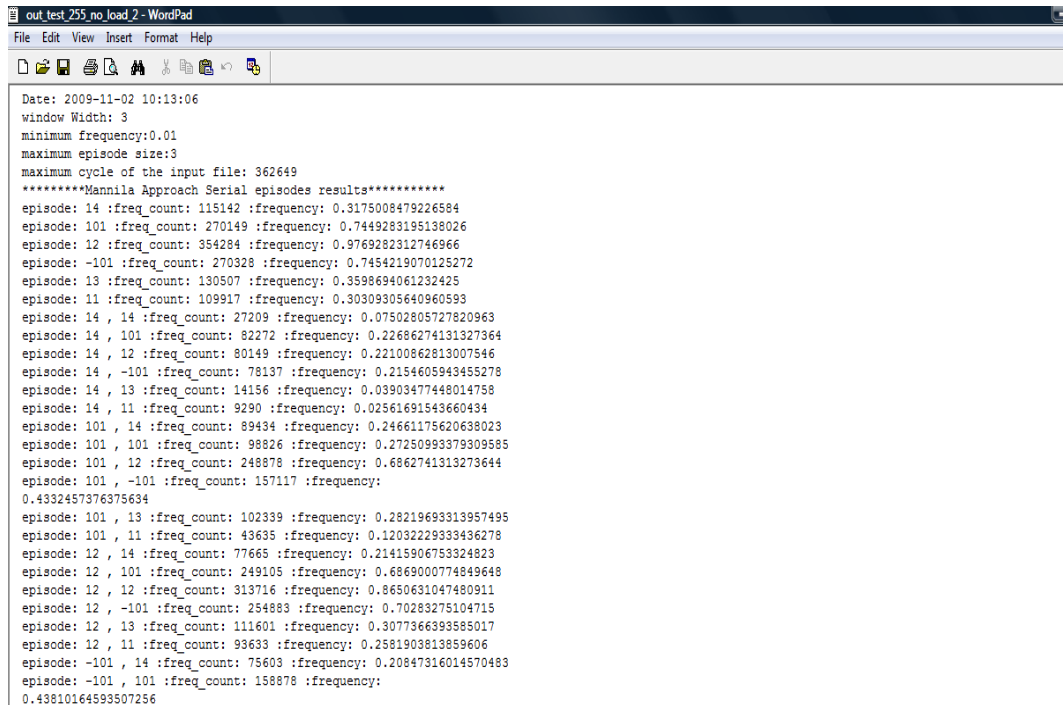
11: Function Call
12: Function Return
13: Forward Branch
14: Backward Branch

Here, we did some pre-processing before running EMT with this input file. There are events in each line with IPC values and we generated a new input by starting a counter from 1 to indicate the cycles and print the events in the same line to the same cycle. Also, after each cycle we checked the IPC values. We classified the IPC values in 8 classes from 0 to 8. Then, in each cycle if the IPC's class had changed and had decreased, then we added a new event marked as "-101" which meant "IPC decreases". If the IPC's class had changed and had increased, then we added a new event marked as "-101" which meant "IPC increases". In this way, we generated a new input file containing 6 types of events about branches, functions and IPC changes.

Results: We used 6 different dataset in this step and applied our three episode mining techniques with various different parameters. We looked for relationships between IPC values and the four event types used in this step. As a result, we realised that some blocks in program executions occurred repeatedly in our input datasets and instead of reviewing the whole sequence, it would be better to consider these unique sequences indicating these program blocks. Then, it would be easier to see the relationships between the rules generated about these event types and IPC values.

Therefore, it would be possible to make predictions about the performance of the programs in an architecture.

As an example the output for the frequencies of episodes and the output for the rules generated by EMT with the frequent episodes with the WINEPI approach for serial episodes can be given as in Figure 15 and Figure 16.



out_test_255_no_load_2 - WordPad

File Edit View Insert Format Help

```
Date: 2009-11-02 10:13:06
window Width: 3
minimum frequency:0.01
maximum episode size:3
maximum cycle of the input file: 362649
*********Mannila Approach Serial episodes results***********
episode: 14 :freq_count: 115142 :frequency: 0.3175008479226584
episode: 101 :freq_count: 270149 :frequency: 0.7449283195138026
episode: 12 :freq_count: 354284 :frequency: 0.9769282312746966
episode: -101 :freq_count: 270328 :frequency: 0.7454219070125272
episode: 13 :freq_count: 130507 :frequency: 0.3598694061232425
episode: 11 :freq_count: 109917 :frequency: 0.30309305640960593
episode: 14 , 14 :freq_count: 27209 :frequency: 0.07502805727820963
episode: 14 , 101 :freq_count: 82272 :frequency: 0.22686274131327364
episode: 14 , 12 :freq_count: 80149 :frequency: 0.22100862813007546
episode: 14 , -101 :freq_count: 78137 :frequency: 0.2154605943455278
episode: 14 , 13 :freq_count: 14156 :frequency: 0.03903477448014758
episode: 14 , 11 :freq_count: 9290 :frequency: 0.02561691543660434
episode: 101 , 14 :freq_count: 89434 :frequency: 0.24661175620638023
episode: 101 , 101 :freq_count: 98826 :frequency: 0.27250993379309585
episode: 101 , 12 :freq_count: 248878 :frequency: 0.6862741313273644
episode: 101 , -101 :freq_count: 157117 :frequency:
0.4332457376375634
episode: 101 , 13 :freq_count: 102339 :frequency: 0.28219693313957495
episode: 101 , 11 :freq_count: 43635 :frequency: 0.12032229333436278
episode: 12 , 14 :freq_count: 77665 :frequency: 0.21415906753324823
episode: 12 , 101 :freq_count: 249105 :frequency: 0.6869000774849648
episode: 12 , 12 :freq_count: 313716 :frequency: 0.8650631047480911
episode: 12 , -101 :freq_count: 254883 :frequency: 0.70283275104715
episode: 12 , 13 :freq_count: 111601 :frequency: 0.3077366393585017
episode: 12 , 11 :freq_count: 93633 :frequency: 0.2581903813859606
episode: -101 , 14 :freq_count: 75603 :frequency: 0.20847316014570483
episode: -101 , 101 :freq_count: 158878 :frequency:
0.43810164593507256
```

Figure 15: Outputs for frequency of the episodes in experiment-3

out_test_255_no_load_2 - WordPad

File Edit View Insert Format Help

```
*******Rules Generated************
14 -> 14 , 14 ( confidence: 0.23630821073109726 )( Rule of Serial
Episodes )
14 -> 14 , 101 ( confidence: 0.7145264108665821 )( Rule of Serial
Episodes )
14 -> 14 , 12 ( confidence: 0.6960883083496899 )( Rule of Serial
Episodes )
14 -> 14 , -101 ( confidence: 0.6786142328602943 )( Rule of Serial
Episodes )
14 -> 14 , 13 ( confidence: 0.12294384325441629 )( Rule of Serial
Episodes )
14 -> 14 , 11 ( confidence: 0.08068298275173265 )( Rule of Serial
Episodes )
101 -> 101 , 14 ( confidence: 0.33105434408419054 )( Rule of Serial
Episodes )
101 -> 101 , 101 ( confidence: 0.365820343588168 )( Rule of Serial
Episodes )
101 -> 101 , 12 ( confidence: 0.9212619702460494 )( Rule of Serial
Episodes )
101 -> 101 , -101 ( confidence: 0.5815938611655049 )( Rule of Serial
Episodes )
101 -> 101 , 13 ( confidence: 0.37882427845374217 )( Rule of Serial
Episodes )
101 -> 101 , 11 ( confidence: 0.16152197491014217 )( Rule of Serial
Episodes )
12 -> 12 , 14 ( confidence: 0.21921678653283808 )( Rule of Serial
Episodes )
12 -> 12 , 101 ( confidence: 0.703122353817841 )( Rule of Serial
Episodes )
12 -> 12 , 12 ( confidence: 0.8854929943209402 )( Rule of Serial
Episodes )
12 -> 12 , -101 ( confidence: 0.7194313036998566 )( Rule of Serial
Episodes )
12 -> 12 , 13 ( confidence: 0.31500434679522643 )( Rule of Serial
```

Figure 16: Outputs for confidence of the rules in experiment-3

4.3.4 Experiment 4

Input Data Structure: As mentioned in the results of the previous step, we used a different approach this time. We decided to consider the unique event sequences in program executions and given ID value to each sequence that represents a block. We had two different datasets in this step and one of them contains 47 unique sequences and the other contained 316 unique sequences. The contents of the input files can be given as in Figure 17.

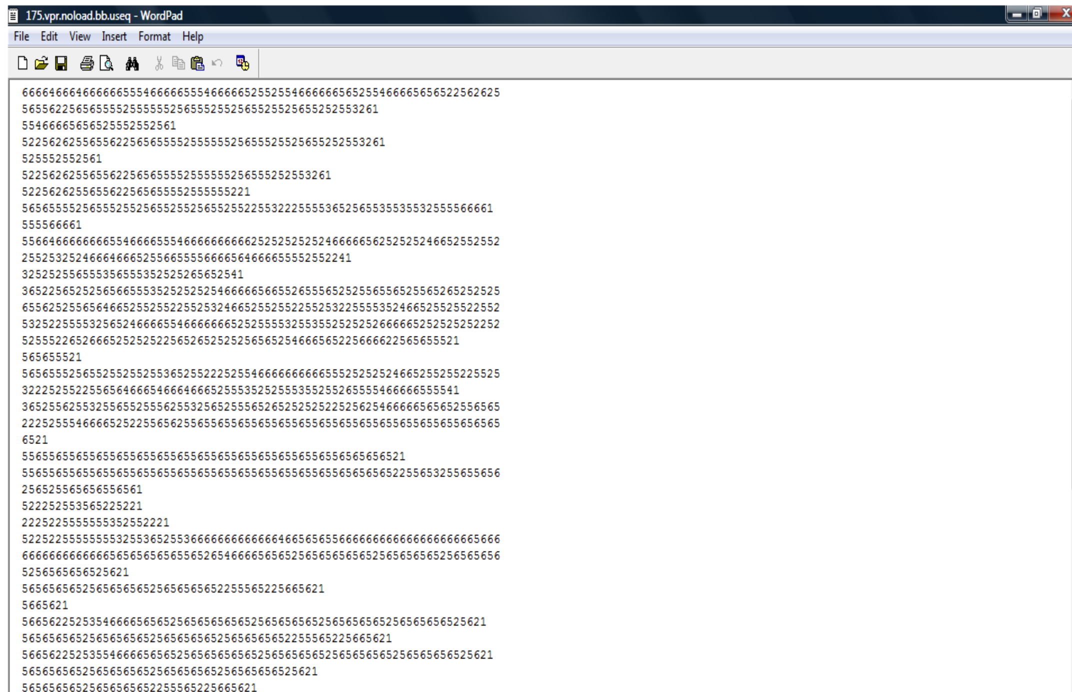


Figure 17: Input file for experiment-4

The file format is as follows:

1. Each line represents a unique sequence for the program.
2. Each line contains a string.
3. Each character in a string represents an event.
   3.1 Character "1" = backward branch
   3.2 Character "2" = forward branch
   3.3 Character "3" = function return
   3.4 Character "4" = function call
   3.5 Character "5" = load
   3.6 Character "6" = store

Example 4.3.4.1 : 556622
21

In this example line 1 represents two load events followed by two store events then two forward branch events. Line 2 represents a forward branch event follow by a backward branch event.

Here, we did some pre-processing in these input datasets. We took each line as a sequence of events and with a counter starting from 1 and increasing incrementally we marked each event in a cycle. Finally, we generated 47 inputs for the first dataset and 316 input files for the second dataset. Then, we applied our 3 episode mining methods for parallel and serial episodes with various different parameters.

Results: We searched for the relationships between the events in these datasets in this step. We processed each block indicated with an id and an input sequence, and generated rules for each of these blocks. Then, we saw that to make predictions and comments about performance of the programs we should combine these generated rules representing these sequences with IPC changes. But here, the changes in IPC would be calculated between the values in two different architectures in the next experimental step.

As an example, the results obtained for one of the input sequences of the first dataset for the frequencies of the episodes and the rules generated with the frequent episodes according to the WINEPI approach for serial episodes can be given as in Figure 18 and Figure 19:

1_sequence_1258033416176.input.winepi_serial_output - WordPad

File Edit View Insert Format Help

```
Date: 2009-11-13 11:12:35
window Width: 3
minimum frequency:0.01
maximum episode size:3
maximum cycle of the input file: 112
*********Mannila Approach Serial episodes results***********
episode: 6 :freq_count: 77 :frequency: 0.6754385964912281
episode: 4 :freq_count: 18 :frequency: 0.15789473684210525
episode: 5 :freq_count: 72 :frequency: 0.631578947368421
episode: 2 :freq_count: 36 :frequency: 0.3157894736842105
episode: 3 :freq_count: 3 :frequency: 0.02631578947368421
episode: 1 :freq_count: 3 :frequency: 0.02631578947368421
episode: 6 , 6 :freq_count: 44 :frequency: 0.38596491228070173
episode: 6 , 4 :freq_count: 4 :frequency: 0.03508771929824561
episode: 6 , 5 :freq_count: 25 :frequency: 0.21929824561403508
episode: 6 , 2 :freq_count: 9 :frequency: 0.07894736842105263
episode: 6 , 3 :freq_count: 0 :frequency: 0.0
episode: 6 , 1 :freq_count: 2 :frequency: 0.017543859649122806
episode: 4 , 6 :freq_count: 12 :frequency: 0.10526315789473684
episode: 4 , 4 :freq_count: 0 :frequency: 0.0
episode: 4 , 5 :freq_count: 0 :frequency: 0.0
episode: 4 , 2 :freq_count: 0 :frequency: 0.0
episode: 4 , 3 :freq_count: 0 :frequency: 0.0
episode: 4 , 1 :freq_count: 0 :frequency: 0.0
episode: 5 , 6 :freq_count: 22 :frequency: 0.19298245614035087
episode: 5 , 4 :freq_count: 8 :frequency: 0.07017543859649122
episode: 5 , 5 :freq_count: 45 :frequency: 0.39473684210526316
episode: 5 , 2 :freq_count: 19 :frequency: 0.16666666666666666
episode: 5 , 3 :freq_count: 2 :frequency: 0.017543859649122806
episode: 5 , 1 :freq_count: 0 :frequency: 0.0
episode: 2 , 6 :freq_count: 7 :frequency: 0.06140350877192982
episode: 2 , 4 :freq_count: 0 :frequency: 0.0
episode: 2 , 5 :freq_count: 20 :frequency: 0.17543859649122806
episode: 2 , 2 :freq_count: 6 :frequency: 0.05263157894736842
```

Figure 18: Outputs for frequency of the episodes in experiment-4

1_sequence_1258033416176.input.winepi_serial_output - WordPad

File Edit View Insert Format Help

```
*******Rules Generated************
6 -> 6 , 6 ( confidence: 0.5714285714285714 )( Rule of Serial Episodes
)
6 -> 6 , 4 ( confidence: 0.05194805194805195 )( Rule of Serial
Episodes )
6 -> 6 , 5 ( confidence: 0.3246753246753247 )( Rule of Serial Episodes
)
6 -> 6 , 2 ( confidence: 0.11688311688311688 )( Rule of Serial Episodes
)
6 -> 6 , 1 ( confidence: 0.025974025974025972 )( Rule of Serial
Episodes )
4 -> 4 , 6 ( confidence: 0.6666666666666666 )( Rule of Serial Episodes
)
5 -> 5 , 6 ( confidence: 0.3055555555555556 )( Rule of Serial Episodes
)
5 -> 5 , 4 ( confidence: 0.1111111111111111 )( Rule of Serial Episodes
)
5 -> 5 , 5 ( confidence: 0.625 )( Rule of Serial Episodes )
5 -> 5 , 2 ( confidence: 0.2638888888888889 )( Rule of Serial Episodes
)
5 -> 5 , 3 ( confidence: 0.027777777777777776 )( Rule of Serial
Episodes )
2 -> 2 , 6 ( confidence: 0.19444444444444445 )( Rule of Serial Episodes
)
2 -> 2 , 5 ( confidence: 0.5555555555555556 )( Rule of Serial Episodes
)
2 -> 2 , 2 ( confidence: 0.16666666666666666 )( Rule of Serial Episodes
)
3 -> 3 , 2 ( confidence: 0.6666666666666666 )( Rule of Serial Episodes
)
6 -> 6 , 6 , 6 ( confidence: 0.2987012987012987 )( Rule of Serial
Episodes )
6 , 6 -> 6 , 6 , 6 ( confidence: 0.5227272727272727 )( Rule of Serial
Episodes )
```

Figure 19: Outputs for confidence of the rules in experiment-4

Also, in this step we analysed the results of this input file given in the example in the visual component of EMT and for the frequencies of the episodes, we got a result as shown in Figure 20:

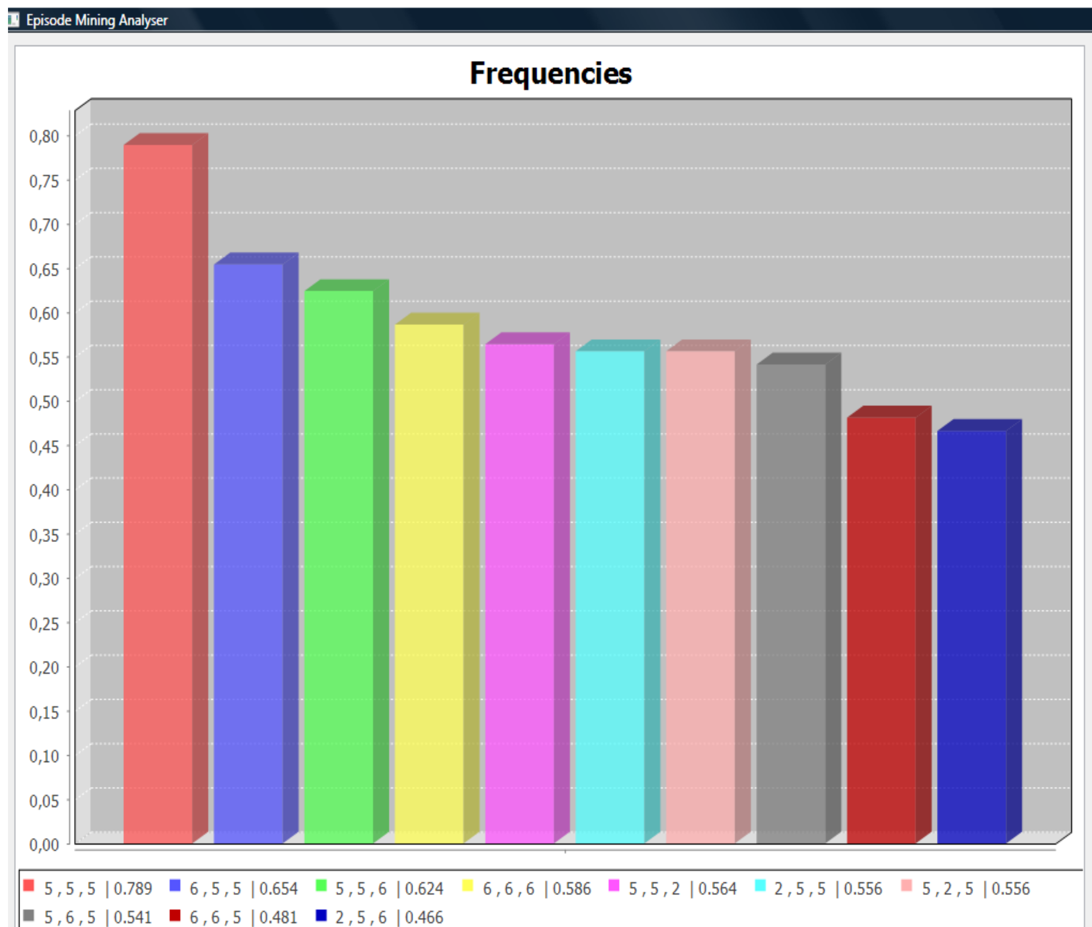


Figure 20: The first 10 episodes with length 3 according to their frequencies in the experiment-4

And for the rules and their confidences we got the results from the same input file as shown in Figure 21:

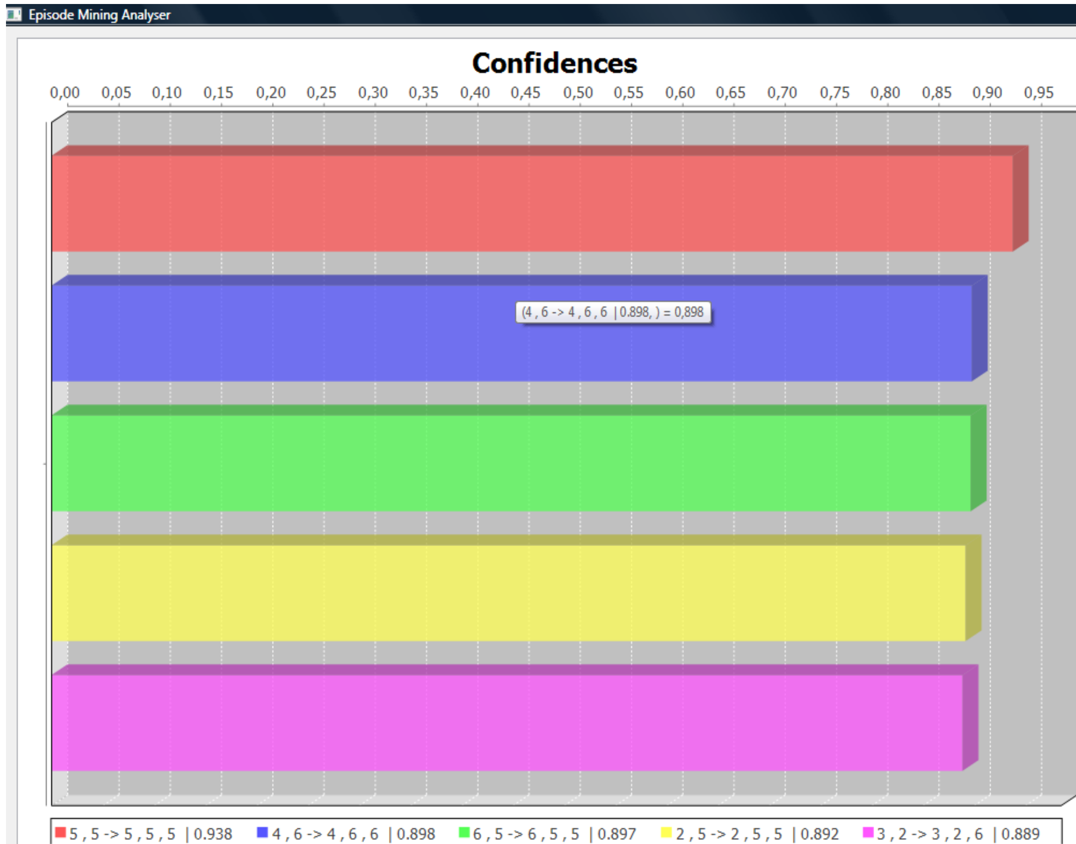


Figure 21: The first 5 rules with length 3 according to their confidences in the experiment-4

4.3.5 Experiment 5

Input Data Structure: In this step, we had a different approach and we used four different datasets for two different architectures. These input files contains the execution sequence of the blocks specified in the previous step and indicated by the unique event sequences, and the IPC values during their executions.

The format of the input files can be given as:

Each line contains two columns. The first column is block number and second column is IPC.

Example 4.3.5.1 : 1 0.67
3 1.5
3 1.5

Line 1 means block 1 executed at 0.67 IPC, lines 2 and 3 mean block 3 executed at 1.5 IPC.

Results:In this situation, we had the rules generated from the unique sequences meaning the blocks from the previous step and also the IPC values of these blocks. Here, we generated a relation table for these rules and blocks as in the following:

```
        Rule        BlockID      Confidence
----------------------------------------------
           66.665565         12      1.0000
            66.66555         12      1.0000
            6.665565         12      1.0000
             6.66555         12      1.0000
         665.665565         208      1.0000
           665.66556        208      1.0000
           665.66555        208      1.0000
           6.6565565        208      1.0000
            6.656556        208      1.0000
            6.656555        208      1.0000
        66556.665565        208      1.0000
         6655.665565        208      1.0000
          6655.66556        208      1.0000
....
```

Here, the first column indicates the rule in a slightly different format. Here, "66.665565" shows the rule "66 -> 665565" and the second column indicates for which block this rule is generated. And the last column gives a confidence value which is the ration of in how many blocks the rule is generated. This confidence value can be calculated as "1 / count of the blocks where the rule is generated".
After that we generated another relation table containing the changes in IPC values in different architectures. This table can be given as:

Rule	Delta IPC	Confidence
6.646	0	1.0000
64.646	0	1.0000
46.464	0	1.0000
4.464	0	1.0000
4.44	0	1.0000
3.32	0	1.0000
5.51	0	0.9959
5.54	0	0.9785
4.43	0	0.9785
2.21	0	0.9766
5555.55555	0	0.9638
555.55555	0	0.9638
55.55555	0	0.9638
5.55555	0	0.9638
2.22	0	0.9362
5.535	0	0.6667
53.535	0	0.6667
5.53	0	0.6363
5.52	0	0.6102
2.25	0	0.5679
55.552	0	0.5475
5.552	0	0.5475
6.61	0	0.5259
55.553	0	0.5217
5.553	0	0.5217
666.6666	-178	0.5000
666.6666	-169	0.5000
66.6666	-178	0.5000
66.6666	-169	0.5000

…

Here, according to these results for instance there is 0 change in IPC values for all blocks where the rule "6->646" is generated in two different architectures. And there is -1.78 change in IPC values in 0.5 of the blocks and -1.69 change in IPC values in

0.5 of the blocks where the rule “666->6666” is generated in two different architectures.

We tried to see which rules can represent which of these blocks in this analysis, but as seen in the example the changes in IPC values for a block and the rule to represent this block is not constant for all. Therefore, we think that by using the IPC values for these program blocks obtained in two different architecture, and the rules generated for these blocks, one more episode mining step can say something about their relationships to make comments about the performance of a program in a different architecture before the execution.

4.3.6 Experiment 6

Input Data Structure: In the previous experiment, we generated the rules from the unique sequences that we can call program blocks and also we had IPC values obtained during the execution of a program in an architecture. In this step, we used these IPC values for two different architectures and tried to find a relationship between the change in IPC values and the rules generated from the program blocks.

Here, first of all by using the pre-processing component of EMT, we generated an input dataset for the episode mining component. We had the sequence of program blocks such as :

```
1
2
3
3
4
5
2
3
4
6
6
6
6
...
```

Also, we had the IPC values corresponding to the execution of these blocks with the given id numbers in two different architectures such as;

```
0.6206
0.7174
1.3571
2.3750
1.1429
1.2778
1.7368
2.3750
1.1429
1.0000
1.7368
2.3750
1.1429
1.0000
1.0000
0.2482
1.0000
...
```

Then, by using the pre-processing component we generated an input file for episode mining containing the rules for the given block in the execution sequence and the IPC change value (delta IPC) for two architectures. For each of the executed blocks we repeated this data and encapsulated the rules of block and delta IPC value in a single time value.

Example 4.3.6.1: For the block no:1, in the input sequence there are lines such as;

```
1 6661
1 5551
1 66661
1 666662
1 55551
1 555552
10
...
```

These lines mean that;

`1 6661` : Time (t) = 1 and rule 6->66 ; the number:1 at the end of 6661 gives the separation point for the left part of the rule.

1 666662 : Time (t) = 1 and rule 66->666 ; the number:2 at the end of 666662 gives the separation point for the left part of the rule.

10: At the end of each time value, this value indicates the change in IPC value for two architectures. If there is an extra 0 number, this means that the value is a negative change in IPC. Here, 10 means -1 for delta IPC value.

Also, the value here is not the exact IPC change value. EMT takes a parameter for delta IPC level and this value should be multiplied with the IPC level to get the correct value in IPC change. This conversion is done in order to classify the changes in IPC values. For instance if the value in dataset is “-2” and the IPC level is “0.25”, then this means that change in IPC value is between -0.50 and -0.75.

Results: After generating the input file for the episode mining operation on rule strings and IPC changes, we assigned the parameter window_width=1 and generate episodes in the format “<rule_string, delta_IPC_value>” and also rules in the format “<rule_string> -> <rule_string, delta_IPC_value>”.

We used a coefficient as 0.9 to classify the changes in IPC values and got results for episodes and rules. For episodes containing a rule and a delta IPC value an example output can be given as;

Example 4.3.6.2:
-1 5565 -> 556565 freq_count 147 frequency 0.002202445163610212

This line means that the rule “5, 5, 6, 5 -> 5, 5, 6, 5, 6, 5” and delta IPC value :-1 x 0.9=-0.9 has a frequency of 0.002.Since the number of rules are not small and the frequency of them being seen with a certain delta IPC level value is too low, we decided to consider the rules consisting of a rule and a delta IPC level value.

The results for episodes and frequency values can be given as in Figure 22:

```
F:\emaden\experiment_6\out_vortex.txt_processed - Notepad++
out_vortex.txt_processed
1618 5 56 -> 5655655 freq_count  4482  frequency  0.06715210355987054
1619 5 5655 -> 5655655 freq_count  4482  frequency  0.06715210355987054
1620 5 565565 -> 5655655 freq_count  4482  frequency  0.06715210355987054
1621 5 56 -> 5655656 freq_count  4482  frequency  0.06715210355987054
1622 5 5655 -> 5655656 freq_count  4482  frequency  0.06715210355987054
1623 5 565565 -> 5655656 freq_count  4482  frequency  0.06715210355987054
1624 5 56 -> 5656556 freq_count  4482  frequency  0.06715210355987054
1625 5 5656 -> 5656556 freq_count  4482  frequency  0.06715210355987054
1626 5 565655 -> 5656556 freq_count  4482  frequency  0.06715210355987054
1627 5 65 -> 6556565 freq_count  4482  frequency  0.06715210355987054
1628 5 6556 -> 6556565 freq_count  4482  frequency  0.06715210355987054
1629 5 655656 -> 6556565 freq_count  4482  frequency  0.06715210355987054
1630 5 65 -> 6565565 freq_count  4482  frequency  0.06715210355987054
1631 5 6565 -> 6565565 freq_count  4482  frequency  0.06715210355987054
1632 5 656556 -> 6565565 freq_count  4482  frequency  0.06715210355987054
1633 -1 5 -> 55556 freq_count  675  frequency  0.010113268608414239
1634 -1 555 -> 55556 freq_count  675  frequency  0.010113268608414239
1635 -1 5 -> 56565 freq_count  647  frequency  0.009693755243917057
1636 -1 565 -> 56565 freq_count  647  frequency  0.009693755243917057
1637 -1 5 -> 5566 freq_count  674  frequency  0.010098285988253625
1638 -1 556 -> 5566 freq_count  674  frequency  0.010098285988253625
1639 -1 65 -> 6566 freq_count  647  frequency  0.009693755243917057
1640 -1 6 -> 6655 freq_count  674  frequency  0.010098285988253625
1641 -1 665 -> 6655 freq_count  674  frequency  0.010098285988253625
1642 -1 66 -> 6656 freq_count  647  frequency  0.009693755243917057
1643 -1 5 -> 5666 freq_count  647  frequency  0.009693755243917057
1644 -1 566 -> 5666 freq_count  647  frequency  0.009693755243917057
1645 -1 66 -> 6665 freq_count  674  frequency  0.010098285988253625
1646 -1 5 -> 55566 freq_count  645  frequency  0.009663790003595829
1647 -1 555 -> 55566 freq_count  645  frequency  0.009663790003595829
1648 -1 5 -> 55656 freq_count  646  frequency  0.009678772623756443
1649 -1 556 -> 55656 freq_count  646  frequency  0.009678772623756443
1650 -1 5 -> 55665 freq_count  646  frequency  0.009678772623756443
1651 -1 556 -> 55665 freq_count  646  frequency  0.009678772623756443
1652 -1 5 -> 56566 freq_count  646  frequency  0.009678772623756443
1653 -1 565 -> 56566 freq_count  646  frequency  0.009678772623756443
1654 -1 5 -> 56655 freq_count  646  frequency  0.009678772623756443
1655 -1 566 -> 56655 freq_count  646  frequency  0.009678772623756443
1656 -1 5 -> 56656 freq_count  646  frequency  0.009678772623756443
1657 -1 566 -> 56656 freq_count  646  frequency  0.009678772623756443
1658 -1 6 -> 65555 freq_count  646  frequency  0.009678772623756443
1659 -1 655 -> 65555 freq_count  646  frequency  0.009678772623756443
```

Figure 22: Episode mining results for episodes and frequency values

The lines for rules generated give the relationships between the rules and delta IPC values and the confidence of these relationships. An example can be given as;

Example 4.3.6.3:

(5 -> 55) -> -7 ,(5 -> 55) confidence: 0.7499999999999999

The line in this example means that the rule “5 -> 5, 5” causes -7 x 0.9 = 6.3 change in IPC value. And the confidence of this relationship is %74 according to the episode mining operations on datasets.

```
F:\emaden\experiment_6\out_vortex.txt_processed - Notepad++
File Edit Search View Format Language Settings Macro Run TextFX Plugins Window ?
out_vortex.txt_processed
1875 Rules generated:
1876 (6 -> 66 ) -> 0 ,(6 -> 66 ) confidence: 0.058338617628408376 ) Rule of Parallel Episodes )
1877 (6 -> 66 ) -> -1 ,(6 -> 66 ) confidence: 0.05887939221272555 ) Rule of Parallel Episodes )
1878 (6 -> 66 ) -> 1 ,(6 -> 66 ) confidence: 0.1471147748890298 ) Rule of Parallel Episodes )
1879 (6 -> 66 ) -> 2 ,(6 -> 66 ) confidence: 0.19297133130861396 ) Rule of Parallel Episodes )
1880 (6 -> 66 ) -> 3 ,(6 -> 66 ) confidence: 0.3075459733671528 ) Rule of Parallel Episodes )
1881 (6 -> 66 ) -> 4 ,(6 -> 66 ) confidence: 0.08840903781330307 ) Rule of Parallel Episodes )
1882 (6 -> 66 ) -> -3 ,(6 -> 66 ) confidence: 0.013888888888888888 ) Rule of Parallel Episodes )
1883 (6 -> 66 ) -> 5 ,(6 -> 66 ) confidence: 0.16513700230284017 ) Rule of Parallel Episodes )
1884 (6 -> 66 ) -> -1 ,(6 -> 66 ) confidence: 0.025698361312285152 ) Rule of Parallel Episodes )
1885 (6 -> 66 ) -> -1 ,(6 -> 66 ) confidence: 0.8182784272051009 ) Rule of Parallel Episodes )
1886 (6 -> 66 ) -> -1 ,(6 -> 66 ) confidence: 0.33333333333333337 ) Rule of Parallel Episodes )
1887 (6 -> 66 ) -> -1 ,(6 -> 66 ) confidence: 1.0 ) Rule of Parallel Episodes )
1888 (5 -> 55 ) -> 0 ,(5 -> 55 ) confidence: 0.11191637315993859 ) Rule of Parallel Episodes )
1889 (5 -> 55 ) -> -1 ,(5 -> 55 ) confidence: 0.019845570306150095 ) Rule of Parallel Episodes )
1890 (5 -> 55 ) -> -1 ,(5 -> 55 ) confidence: 0.27825261158594494 ) Rule of Parallel Episodes )
1891 (5 -> 55 ) -> 1 ,(5 -> 55 ) confidence: 0.1557617628465637 ) Rule of Parallel Episodes )
1892 (5 -> 55 ) -> -2 ,(5 -> 55 ) confidence: 0.34845360824742266 ) Rule of Parallel Episodes )
1893 (5 -> 55 ) -> 2 ,(5 -> 55 ) confidence: 0.19748487311478372 ) Rule of Parallel Episodes )
1894 (5 -> 55 ) -> 3 ,(5 -> 55 ) confidence: 0.2627336765104308 ) Rule of Parallel Episodes )
1895 (5 -> 55 ) -> 4 ,(5 -> 55 ) confidence: 0.07816309943104849 ) Rule of Parallel Episodes )
1896 (5 -> 55 ) -> -3 ,(5 -> 55 ) confidence: 0.21759259259259256 ) Rule of Parallel Episodes )
1897 (5 -> 55 ) -> -6 ,(5 -> 55 ) confidence: 0.8999999999999999 ) Rule of Parallel Episodes )
1898 (5 -> 55 ) -> -5 ,(5 -> 55 ) confidence: 0.7692307692307692 ) Rule of Parallel Episodes )
1899 (5 -> 55 ) -> -4 ,(5 -> 55 ) confidence: 0.5925925925925926 ) Rule of Parallel Episodes )
1900 (5 -> 55 ) -> 5 ,(5 -> 55 ) confidence: 0.13268761853156327 ) Rule of Parallel Episodes )
1901 (5 -> 55 ) -> -7 ,(5 -> 55 ) confidence: 0.7499999999999999 ) Rule of Parallel Episodes )
1902 (5 -> 55 ) -> -1 ,(5 -> 55 ) confidence: 0.020771245371624673 ) Rule of Parallel Episodes )
1903 (5 -> 55 ) -> -1 ,(5 -> 55 ) confidence: 0.9776833156216791 ) Rule of Parallel Episodes )
1904 (5 -> 55 ) -> -1 ,(5 -> 55 ) confidence: 1.0 ) Rule of Parallel Episodes )
1905 (5 -> 55 ) -> -1 ,(5 -> 55 ) confidence: 1.0 ) Rule of Parallel Episodes )
1906 (5 -> 55 ) -> -1 ,(5 -> 55 ) confidence: 1.0 ) Rule of Parallel Episodes )
1907 (6 -> 666 ) -> 0 ,(6 -> 666 ) confidence: 0.039973246252508164 ) Rule of Parallel Episodes )
1908 (6 -> 666 ) -> -1 ,(6 -> 666 ) confidence: 0.04685026907249129 ) Rule of Parallel Episodes )
1909 (6 -> 666 ) -> 1 ,(6 -> 666 ) confidence: 0.15218161073297398 ) Rule of Parallel Episodes )
1910 (6 -> 666 ) -> 2 ,(6 -> 666 ) confidence: 0.17531573356414998 ) Rule of Parallel Episodes )
1911 (6 -> 666 ) -> 3 ,(6 -> 666 ) confidence: 0.32580556320572845 ) Rule of Parallel Episodes )
1912 (6 -> 666 ) -> 4 ,(6 -> 666 ) confidence: 0.08529724200338355 ) Rule of Parallel Episodes )
1913 (6 -> 666 ) -> 5 ,(6 -> 666 ) confidence: 0.18102057677932093 ) Rule of Parallel Episodes )
1914 (6 -> 666 ) -> -1 ,(6 -> 666 ) confidence: 0.0266357162529016 ) Rule of Parallel Episodes )
1915 (6 -> 666 ) -> -1 ,(6 -> 666 ) confidence: 0.7194473963868225 ) Rule of Parallel Episodes )
1916 (6 -> 666 ) -> -1 ,(6 -> 666 ) confidence: 0.33333333333333337 ) Rule of Parallel Episodes )
```

Figure 23: Episode mining results for rules and confidence values

As a result, the output in Figure 23 shows that we have found a relationship between the rules generated from the unique sequences obtained from program executions and IPC changes in two different architectures.

Analysis: In these results, it can be seen that there are different levels of relationships between the IPC changes and the generated rules. For the input datasets used in this experiment, the relationships between the events "load" and "store" are not modelled and this is the main reason of this situation. We can say that for the generated results, an increase in IPC level can be expected for the rules starting with the event "store". On the other hand, if a rule starts with the event "load", there may be an increase in IPC level or not. As an alternative approach, instead of classifying the IPC changes values into levels such as ..,-2,-1,0,1,2..., three categories such as "IPC increases", "IPC decreases" and "IPC does not change" can be used to analyse the relationships between the generated rules and IPC values.

CHAPTER 5

EPISODE MINING TOOL (EMT)

5.1 General Properties

EMT is designed to investigate the relationships between events in a given event sequence. Here three types of techniques in episode mining are implemented. These are window based WINEPI approach for parallel and serial episodes, minimal occurrence based approach for parallel and serial episodes and non-overlapping occurrence based approach for parallel and serial episodes. This tool has some specific features about mining architectural events but it can be used to for episode mining operations in any dataset containing time of occurrence values and event types.

Main features of EMT can be given as :

- Before processing the input data, the event types that will not be considered during the mining operations can be specified
- The pre-processing operations can be done on input dataset. Here, it is important that the data should be in appropriate format to perform the related pre-processing operations.
- While performing episode mining operations on architecture simulation, the dataset may be viewed as a collection of unique sequences representing program blocks. Here, each sequence can be given as a single line and new input files containing event sequences for these program blocks can be produced for each unique sequence.

- Episode mining with parameters as window width, minimum frequency threshold and minimum confidence threshold can be operated for our three types of episode mining techniques.
- The outputs generated after episode mining can be visually analysed according to the episodes' frequency values and rules' confidence values with respect to the length parameter.
- Multiple output files can be analysed in a single step and they can be grouped according to the common rules or episodes.

5.2 Components of EMT

There are three main components in EMF as ;

- Data pre-processor: This component does the pre-processing operation on dataset and generates a new input file with a postfix "_processed". Here it takes the types of events to be ignored during the episode mining process and if there are IPC values in the input sequence it can produce an input file containing the changes in IPC values. Also it can process unique sequences in a file where each sequence is represented with a line.
- Episode miner: This is the core of the components in EMT. After pre-processing the input dataset if necessary, all of the episode mining operations with our three techniques are done with using this component. It takes the necessary parameters in addition to the input file according to the chosen algorithms. Then, it produces an output file containing the episodes' frequencies and generated rules with confidence values.
- Output analyser: This component is the visual part of EMT. Here, after the results are generated in the previous component, this output file can be analysed with different types of graphs with different selections with the help of output analyser. Also, multiple output files can be analysed and grouped according to their episodes' frequencies or rules' confidences in a single step.

During the episode mining operations, if MINEPI approach is chosen, EMT generates the minimal occurrences of the episodes in separate files with the extension ".minepi" and the user can look at the minimal occurrences of episodes in these files named with the episodes.

5.3 Usage

EMT has two usage modes. One of them is the command line mode and the other is the GUI mode. For command line mode, pre-processor and episode miner components can be used which are two separate executable files. Since the output analyser contains GUI components and gives the results as types of charts it can only be used from the GUI mode.

CHAPTER 6

CONCLUSION AND FUTURE WORK

Data mining has been gaining more importance and its application areas are getting wide with the increase in volumes of data. We can see the usage of data mining techniques in several types of areas such as retail, financial, communication and marketing organizations and science and engineering. One of the branches in data mining is mining sequence data where data can be viewed as a sequence of events each having a time of occurrence. Sequence and episode mining techniques and algorithms have been applied various kinds of data such as occurrences of recurrent illnesses and alarms in a telecommunication network.

The motivation in this thesis work is to apply episode mining algorithms to a new kind of data, architecture simulation data, and generating facilities and preparing an environment to make predictions about the performance of programs in an architecture. Here the events in datasets are instance based events and they do not have duration values. For this thesis work, analysing event logs of an architecture by using episode mining techniques and using the results of this analysis to make predictions about the performance of a program in this architecture before executing the codes can be considered as a new approach for computer architecture and data mining.

An episode mining tool (EMT) has been developed to analyse the benchmark datasets obtained from running programs in an architecture. Three approaches have been implemented as window based occurrences, minimal occurrences and non-overlapping occurrences. Implementing these techniques in such a data mining tool

is a non-existing application and this tool can be used to analyse any type of dataset containing instance based event sequences. Almost all of the facilities of this episode mining tool can be used with different types of datasets consisting events in a time sequence.

In this thesis, as the first analysis task, we generated some patterns containing architecture event types. Here, we have found patterns that are accurate for architecture simulation. Therefore, we have accomplished our goal for finding patterns supporting the expected behaviours and some general rules for the computer architecture by using data mining techniques.

For the main purpose of thesis work, we analysed the program executions and we saw that the executions can be given as a set of unique sequences. Therefore, we analysed these unique sequences, program blocks in other words, and generated rules containing architecture event types. Also, we took the IPC changes between the execution of blocks in different architectures and generated a list containing relationships between IPC change values and the rules.

As a result, after analysing the program blocks and generate rules related with these blocks, the relationships obtained from EMT will help to make predictions about the performance of a program in an architecture before running.

The analysis of this thesis work and the data mining tool developed here can be considered as a base step for analysing computer architecture datasets by using data mining techniques and this approach can be improved by using other different aspects of data mining as a future work. For example, instance based events are considered here and there is not a duration for the occurrences of the events. To extend this approach, continuous events having duration values may be evaluated by using temporal data mining techniques. Therefore, more interesting and hidden relations between sequences and subsequences of events might be discovered.

Another point here may be about visualization and the EMT may be developed by adding visual extensions. Especially, the output analyser component can be considered as a starting point and more visual features such as time series charts containing frequent episodes or confident rules generated from program blocks can be added to facilitate the analysis of results. Finally, by using the analysis methods and results in this thesis work and the features of EMT, a stable method may be developed to make predictions about the performance of programs in an architecture. This method may contain some statistical and mathematical techniques and generate some numerical results. After that, the confidence of the developed method and its results may be checked by making some experiments on different architectures.

REFERENCES

[1] Heikki Mannila, Hannu Toivonnen, A. Inkeri Verkamo, Discovery of Frequent Episodes in Event Sequences, Data Mining and Knowledge Discovery, vol:1 issue:3, pages:259 - 289, January 1997

[2] Srivatsan Laxman, P. S. Sastry, K. P. Unnikrishnan, A Fast Algorithm For Finding Frequent Episodes In Event Streams, International Conference on Knowledge Discovery and Data Mining, Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining, pages:410 – 419, August 2007

[3] Bill Palace, Data Mining Technology Note prepared for Management 274A Anderson Graduate School of Management at UCLA, Spring 1996

[4] Hand D., Mannila H., Smyth P., Principles of Data Mining, Massachusetts Institute of Technology, 2001, ISBN 0-262-08290-X

[5] Jeffrey W. Seifert, Analyst in Information Science and Technology Policy Resources, CRS Report RL31798, Data Mining:An Overview, December 16, 2004

[6] Osmar R. Zaïane, CMPUT690 Principles of Knowledge Discovery in Databases, University of Alberta, Chapter-1 Introduction to Data Mining, 1999

[7] Fayyad Usama, Gregory Piatetsky-Shapiro and Padhraic Smyth, From Data Mining to Knowledge Discovery in Databases, AI Magazine, vol:17 pages:37-54, 1996

[8] Sotiris Kotsiantis, Dimitris Kanellopoulos, Association Rules Mining: A Recent Overview, International Transactions on Computer Science and Engineering, vol:32 pages:71-82, 2006

[9] M.H.Margahny and A.A.Mitwaly, Fast Algorithm for Mining Association Rules, International Journal of computer and software, vol:2 No:1, 2007

[10] Ruoming Jin, Gagan Agrawal, An Algorithm for In-Core Frequent Itemset Mining on Streaming Data, Fifth IEEE International Conference on Data Mining (ICDM'05) pages:210-217, 2005

[11] Wenbin Fang, Mian Lu, Xiangye Xiao, Bingsheng He1, Qiong Luo, Frequent Itemset Mining on Graphics Processors, Data Management On New Hardware, Proceedings of the Fifth International Workshop on Data Management on New Hardware, session: Exploiting parallel hardware, pages:34 – 42, 2009

[12] Mohammed J. Zaki and Ching-Jui Hsiao, CHARM: An Efficient Algorithm for Closed Itemset Mining, IEEE Transactions on Knowledge and Data Engineering, vol:17 issue:4, pages: 462-278, April 2005

[13] Unil Yun and John J. Leggett, WFIM: Weighted Frequent Itemset Mining with a weight range and a minimum weight, SIAM International Data Mining Conference 2005

[14] Agrawal R, Imielinski T, Swami AN. Mining Association Rules between Sets of Items in Large Databases, SIGMOD. June 1993, pages:207-216

[15] Marek Wojciechowski, Maciej Zakrzewicz, Data Mining Query Scheduling for Apriori Common Counting, 6th Int'l Baltic Conf. on Databases and Information Systems, 2004

[16] Qiankun Zhao Nanyang, Sequential Pattern Mining: A Survey, Technical Report 2003118, Nanyang Technological University, Singapore, 2003

[17] Jianyong Wang, Jiawei Han, BIDE: Efficient Mining of Frequent Closed Sequences, ICDE, Proceedings of the 20th International Conference on Data Engineering, page:79, ISBN:0-7695-2065-0, 2004

[18] D. Patterson and J. Hennessy, Computer Organization and Design: The Hardware/Software Interface., Morgan Kaufmann Publishers Inc., 2004, ISBN 1558606041

[19] Ronald A. Thisted, Computer Architecture, Encyclopedia of Biostatistics, Wiley: New York, 1998

[20] Kip Irvine, Assembly Language for Intel-Based Computers, 5th Edition, ISBN: 0-13-238310-1

[21] Miles Murdocca and Vincent Heuring, Computer Architecture and Organization, An Integrated Approach, Wiley ISBN-13: 978-0471733881, 2007

[22] Osborne, Adam, An Introduction to Microcomputers., Volume 1: Basic Concepts (2nd ed.), 1980

[23] Jon Paul Shen, Mikko Lipasti, Modern processor design: Fundamentals of Superscalar Processors, ISBN 0-07-057064-7, 2005
[24] Daniel A. Jimenez, Reconsidering Complex Branch Predictors, HPCA, Proceedings of the 9th International Symposium on High-Performance Computer Architecture, page:43, 2003
[25] Laplante, Phillip A., Dictionary of Computer Science, Engineering and Technology, CRC Press, 2001, ISBN 0849326915.
[26] John D. Johnson, Branch Prediction Using Large Self History , Stanford University, Technical Report No. CSL-TR-92-553, December 1992
[27] Tse-Yu Yeh, Yale N. Patt, Two-level adaptive training branch prediction, International Symposium on Microarchitecture, pages: 51 - 61, 1991
[28] Bosky Agarwal, Instruction Fetch Execute Cycle, CS 518 Montana State University, Fall 2004
[29] Dean M.Tullsen, Susan J.Eggers, Henry M.Levy, Simultaneous multithreading:maximizing on-chip parallelism, International Symposium on Computer Architecture, pages: 392-403, 1995
[30] B.Ramakrishna Rau, Joseph A. Fisher, Instruction-Level Parallel Processing: History, Overview and Perspective, The Journal of Supercomputing, vol:7 issue:1-2, pages: 9 – 50, 1993
[31] Sally A. McKee, Reflections on the memory wall, Conference On Computing Frontiers, session: Special session on memory wall, page: 162, 2004

APPENDIX A

USAGE OF EMT IN GUI MODE

As shown in Figure 24, in GUI mode, the user firstly runs the executable EMT file and chooses the "Data Pre-processing" operation from the menu.

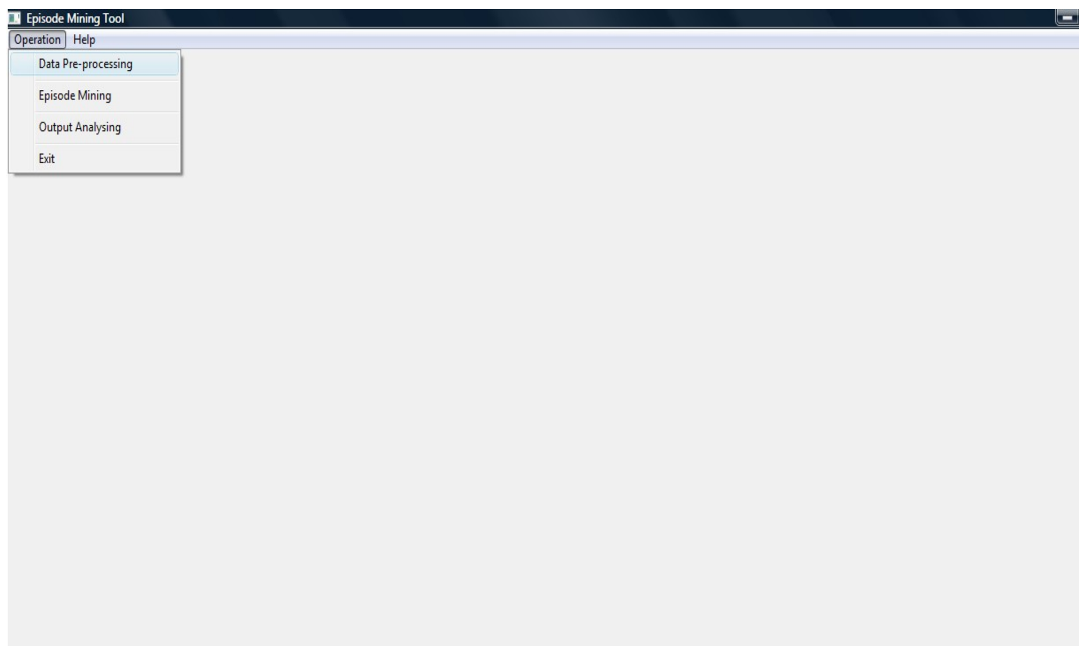


Figure 24: Pre-processing operation in EMT

After that the user should select the "Open" item to be processed from the menu of pre-processing component as shown in Figure 25.

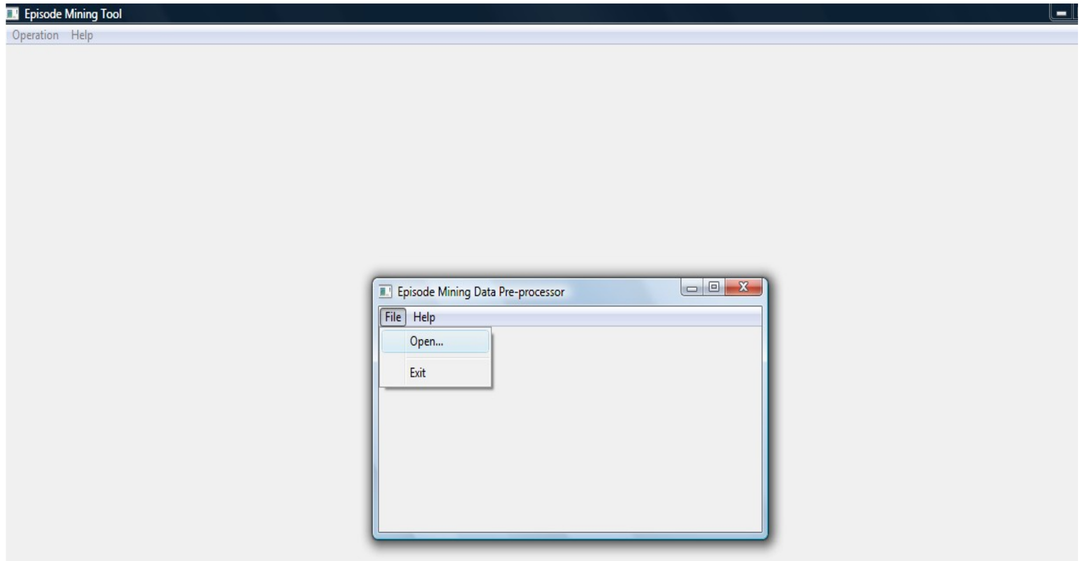


Figure 25: Opening input file for pre-processing

Then a browser is opened and the user should select the file from the browser as in the Figure 26.

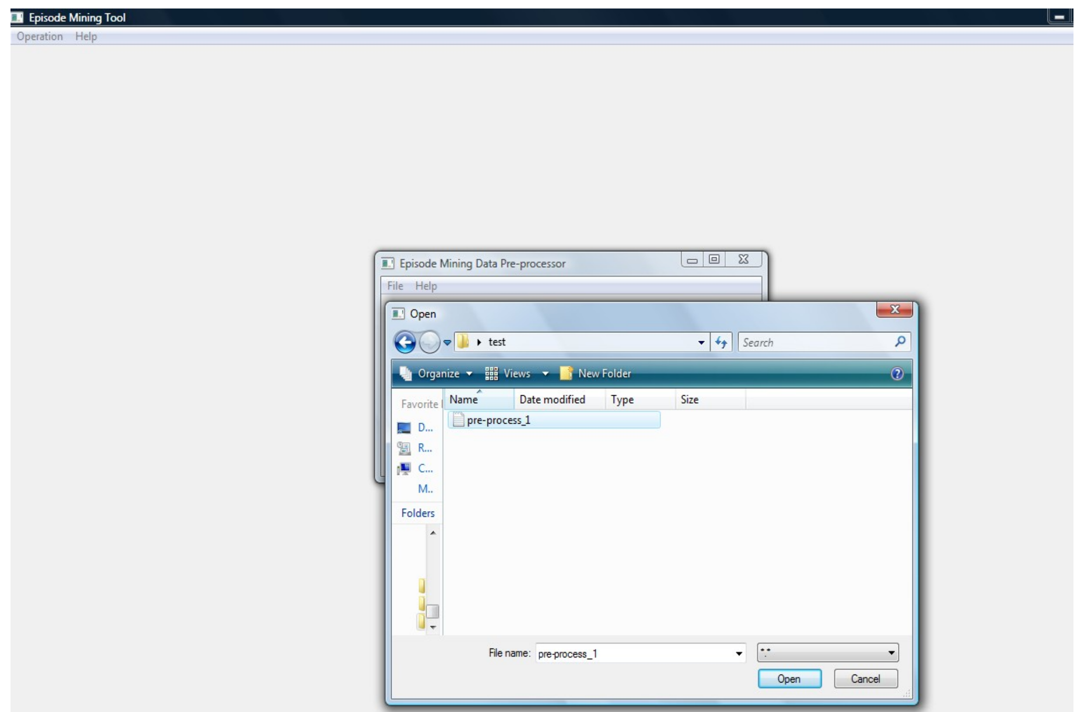


Figure 26: Selecting file from the browser for pre-processing

If any type of episode is desired to be ignored and deleted from the dataset, they can be given as in Figure 27;

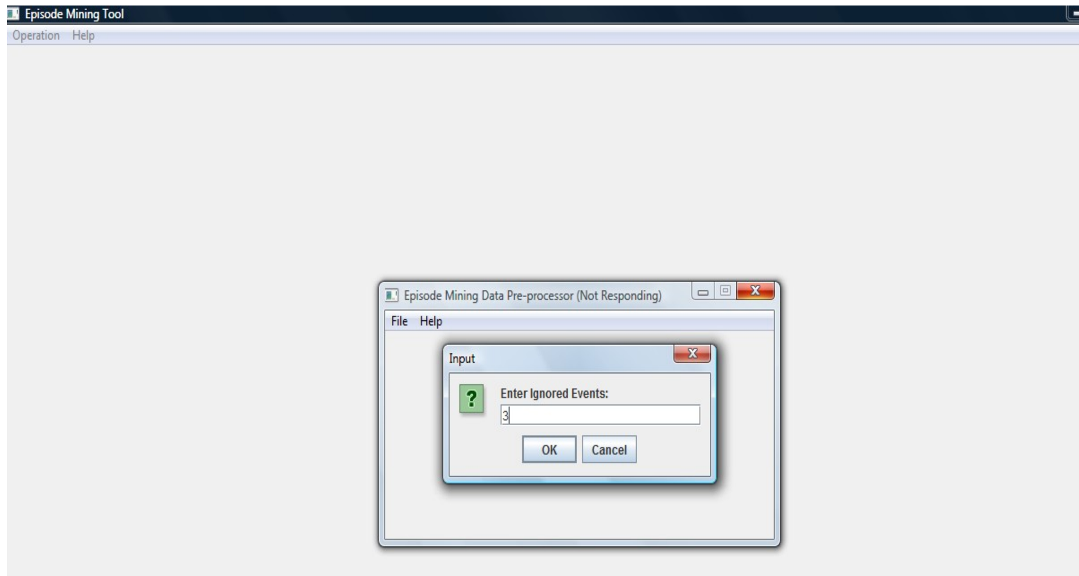


Figure 27: Ignore events in the given dataset

Then, according to the structure of the input file, the type of pre-processing operation is selected as shown in Figure 28. If the dataset contains IPC values, a new input is generated according to the changes in IPC values. If the input dataset contains unique sequences, new input files can be generated for each of the line in dataset.

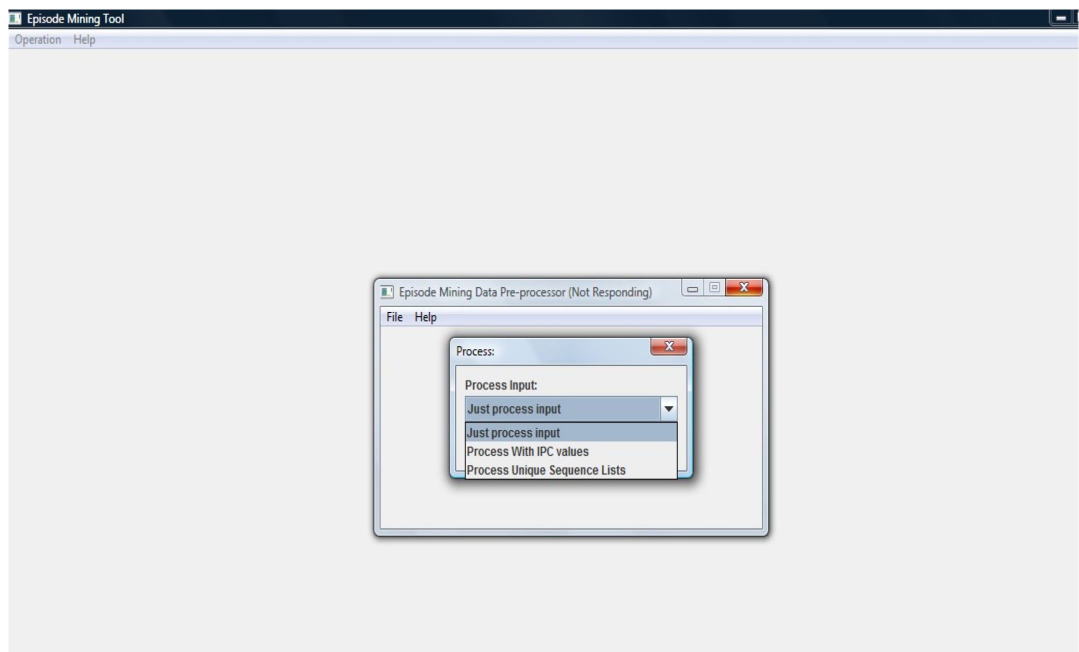


Figure 28: Selecting the pre-processing method

After selecting the method, new input file is generated and a message is given to user as shown in Figure 29.

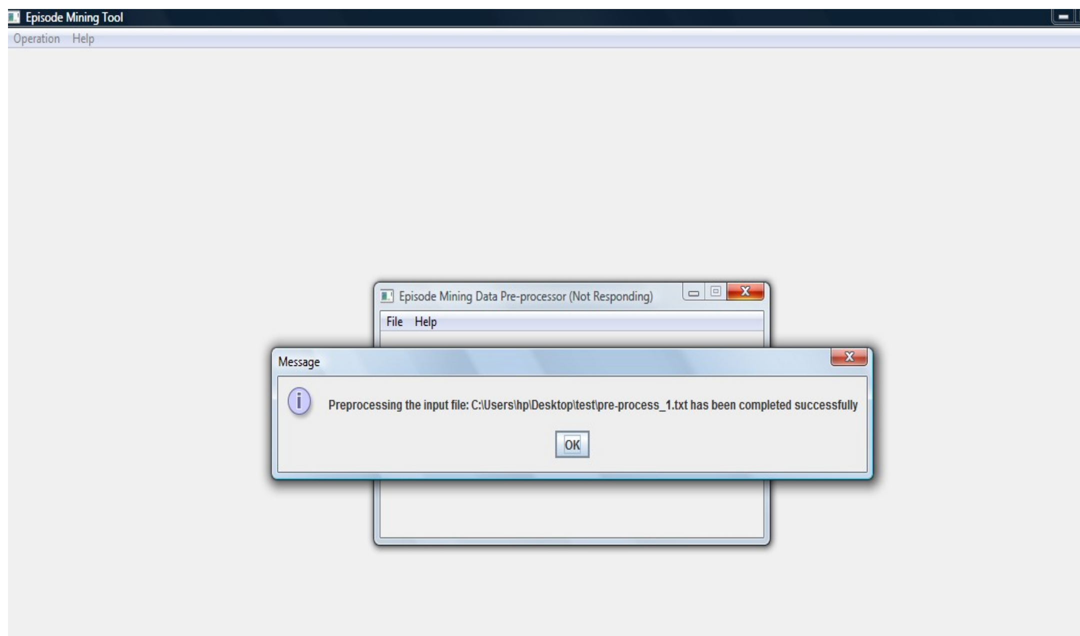


Figure 29: Message given to the user after pre-processing

For investigating the relationships between the rules of unique sequences and IPC changes, firstly we should provide an input file in an appropriate form. To do this, the user selects the "Process Unique Sequence Lists with IPC changes" from the menu of pre-processing component of EMT as in Figure 30.

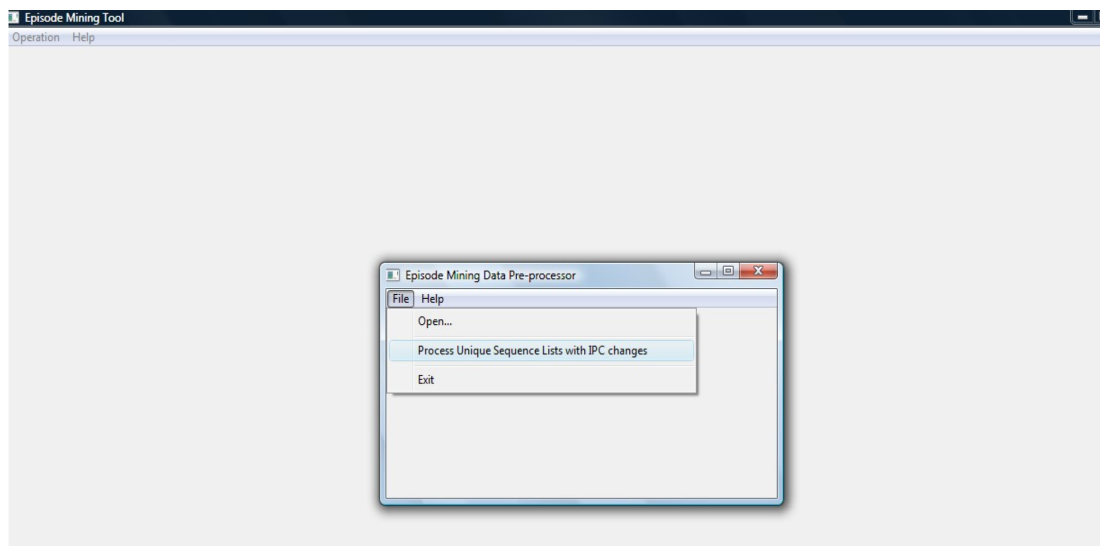


Figure 30: Selecting processing unique sequences with IPC changes

Then, a message is given to the user to select the input files containing the block execution sequence and IPC values for two architectures as shown in Figure 31.

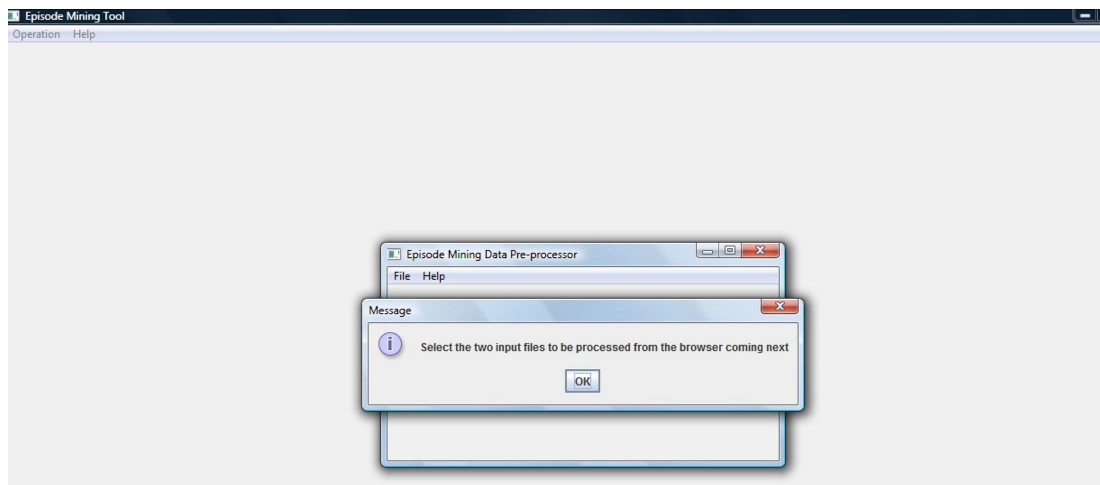


Figure 31: Message given to the user to select the input datasets

After that, the user selects the two datasets from the browser in the next screen as in Figure 32.

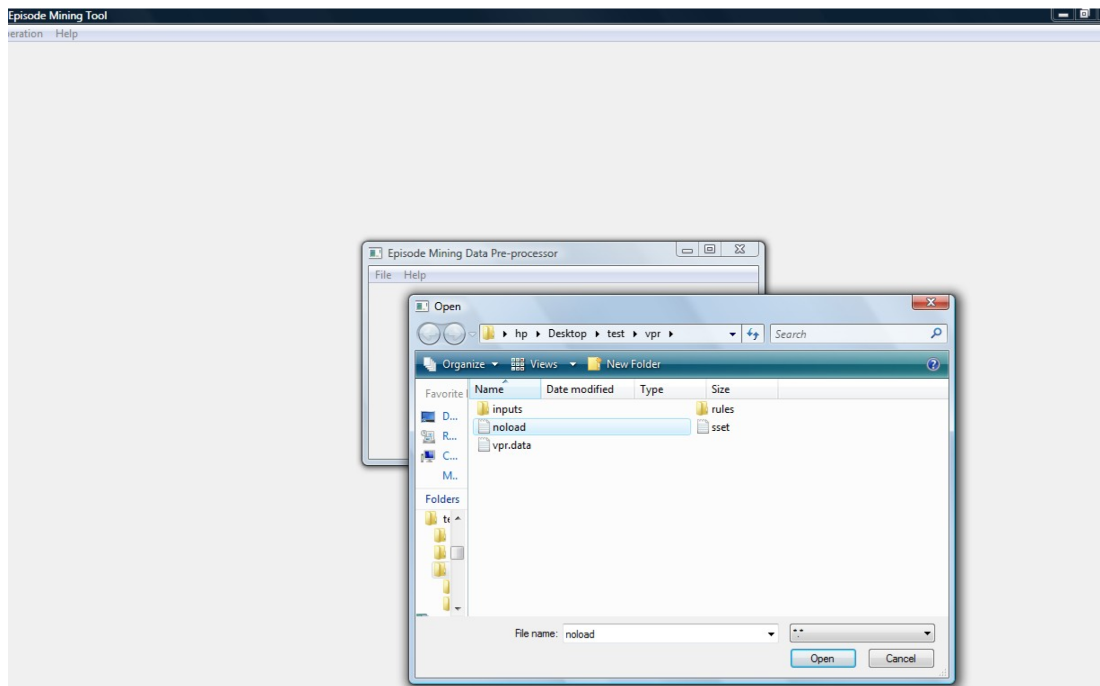


Figure 32: Selecting the datasets for two architectures

The user specifies the directory which includes the rules for the unique sequences from the next coming browser as shown in Figure 33.

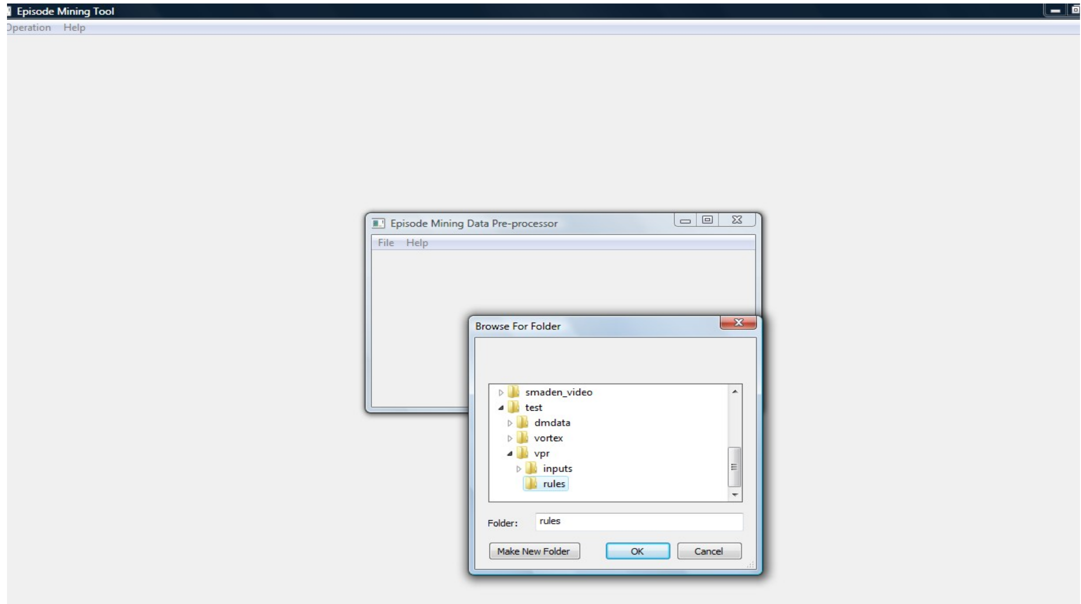


Figure 33: Selecting the directory including the rules for unique sequences

After selecting the directory, the user gives a name for the output file containing an event sequence of rule strings and changes in IPC values as in Figure 34.

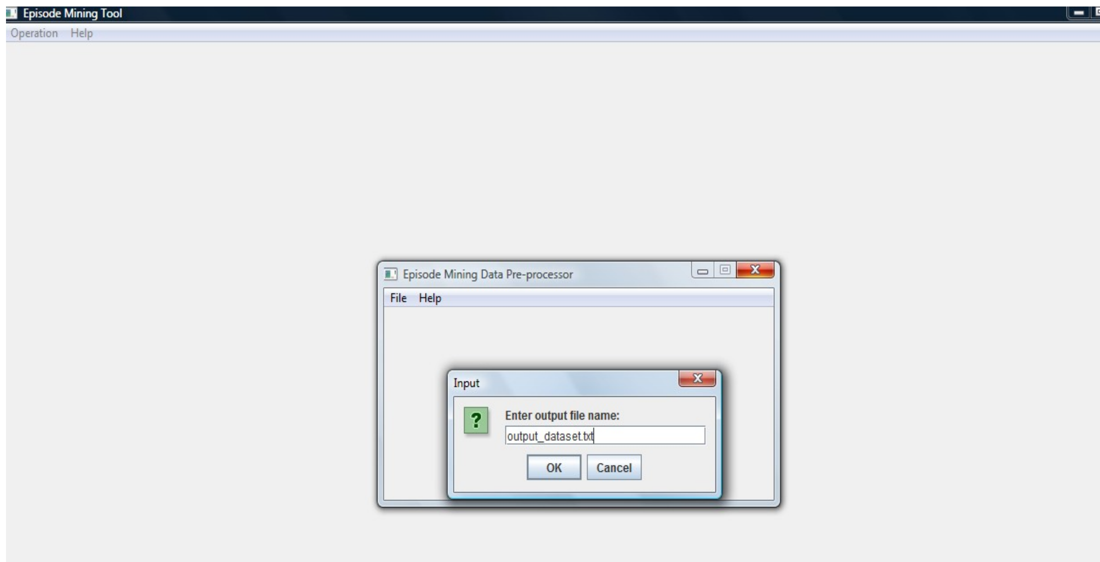


Figure 34: Getting the name of the output event sequence

The user enters a level for expressing the changes in IPC values as in Figure 35.

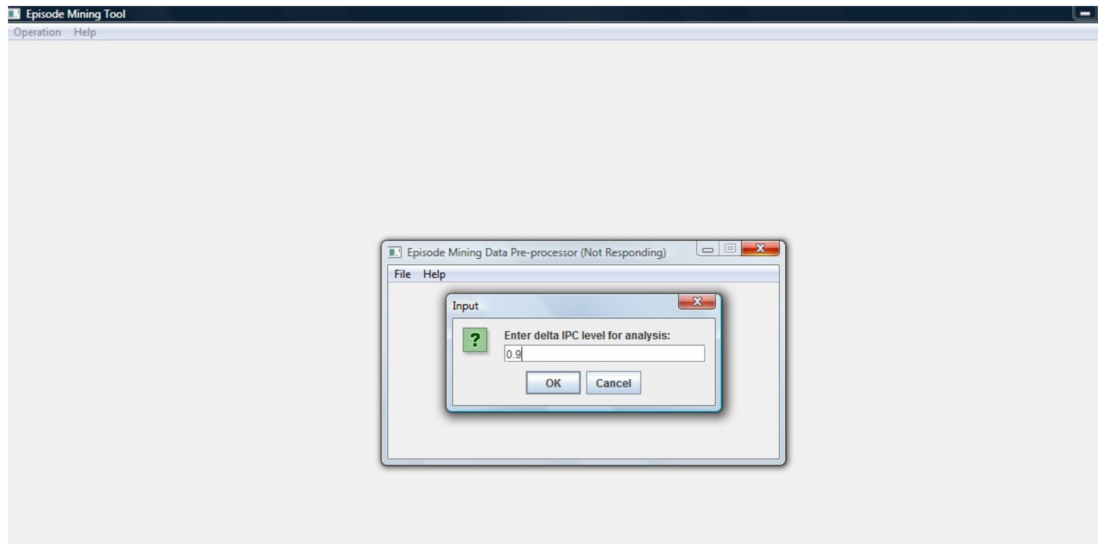


Figure 35: Getting an IPC level to express the changes

After pre-processing is completed, an information message is given to the user as shown in Figure 36.

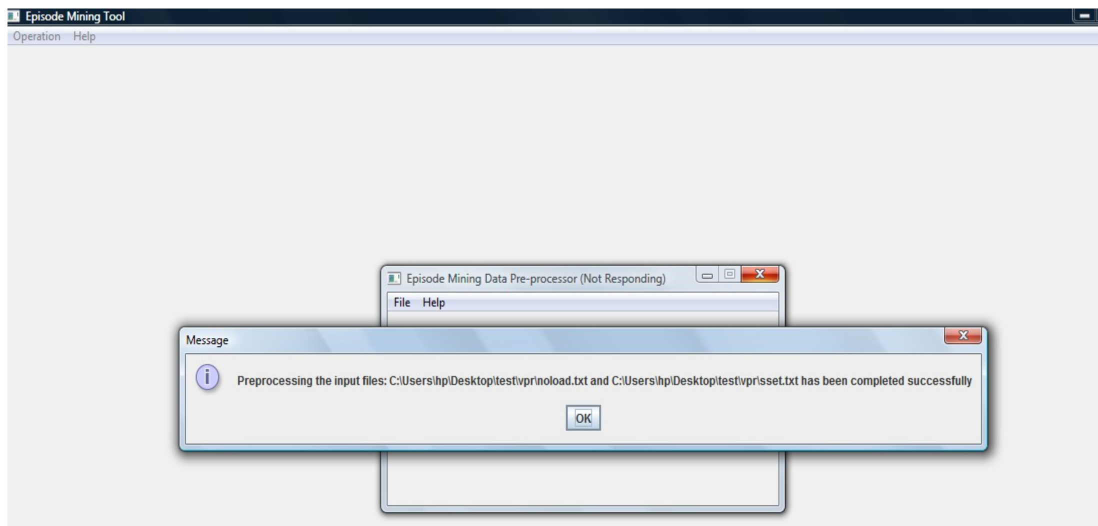


Figure 36: Message given to the user after pre-processing of two datasets is completed

Another operation that the users can do with the pre-processing component is to filter the results obtained from episode mining operation as shown in Figure 37. Here, the user specifies the input file that contains episodes and rules and the pre-processor filters the episodes and rules consisting of only a rule and a delta-IPC level value.

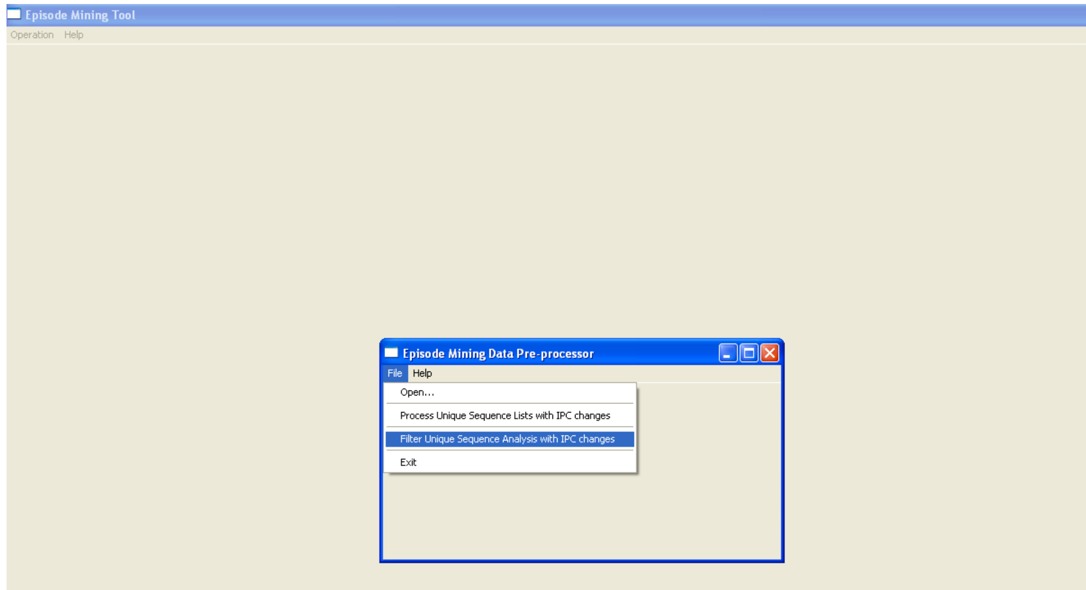


Figure 37: Selecting filter operation for pre-processing

Firstly the user gives the input file containing rules and episodes as in Figure 38;

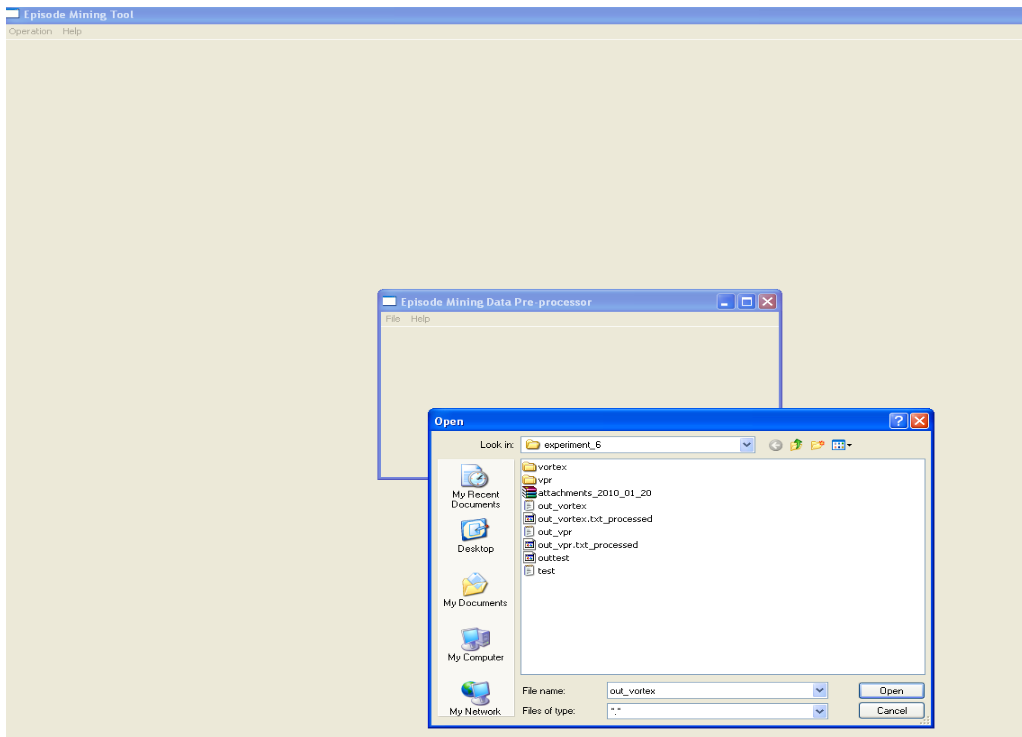


Figure 38: Specifying input file for filtering

To use this component the user selects the “Episode Mining” item from the “Operation” in main menu of EMT as shown in Figure 39.

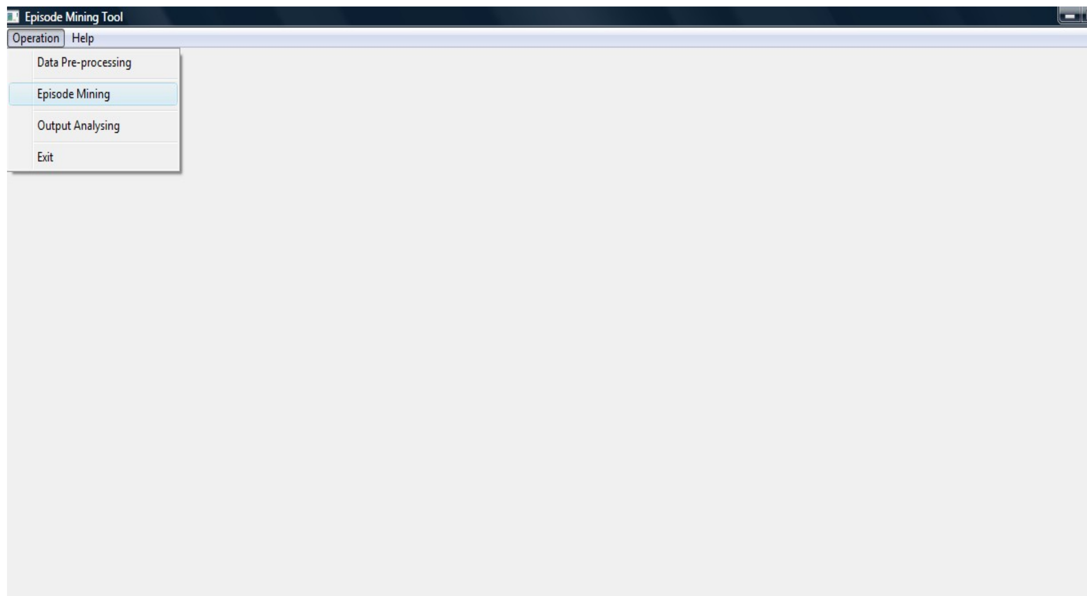


Figure 39: Episode mining operation in EMT

Then the user selects which technique to be applied for what type of episodes from the algorithm menu of the episode miner component as in Figure 40.

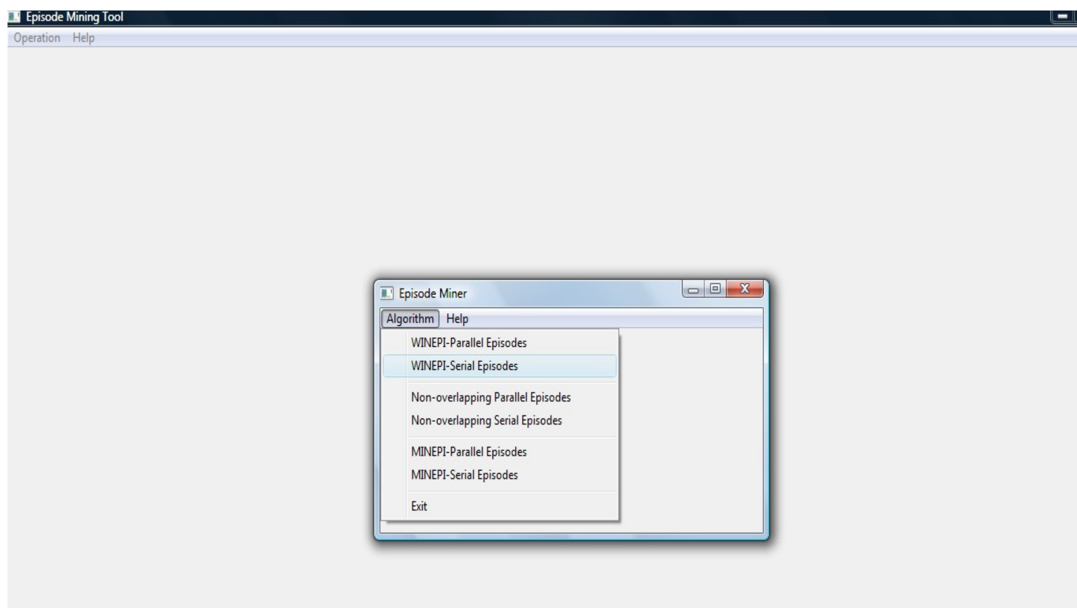


Figure 40: Algorithm selection in episode miner

Then the user gives the window width as shown in Figure 41, minimum frequency threshold as shown in Figure 42, minimum confidence threshold, as shown in

Figure 43, maximum episode length as shown in Figure 44 and input types of events as shown in Figure 45 to be ignored parameters from the dialog boxes opened in EMT.

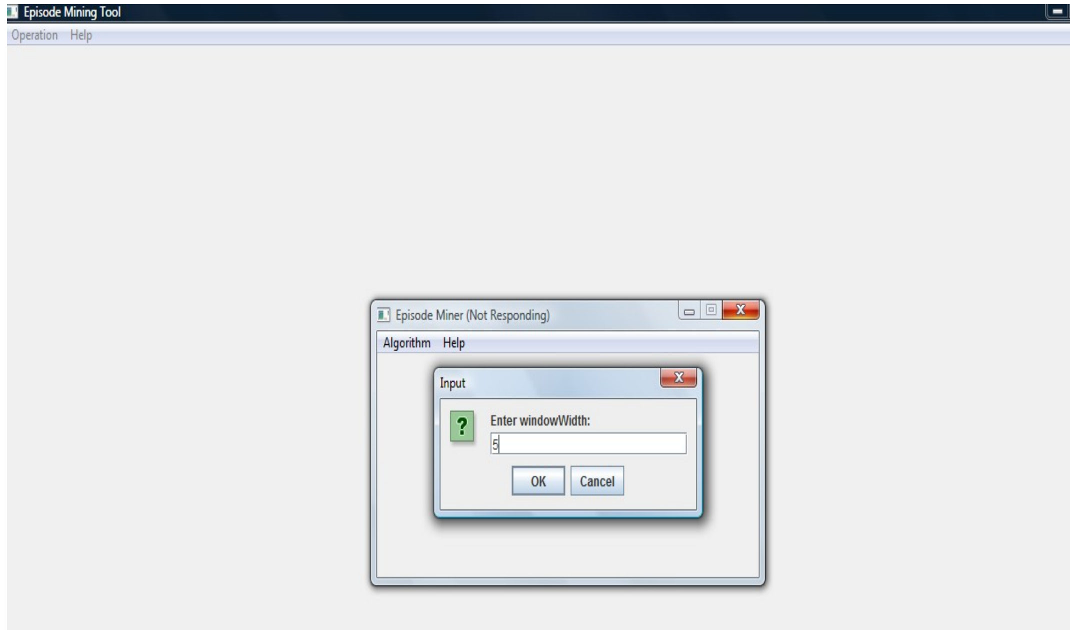


Figure 41: Giving window width parameter in episode miner

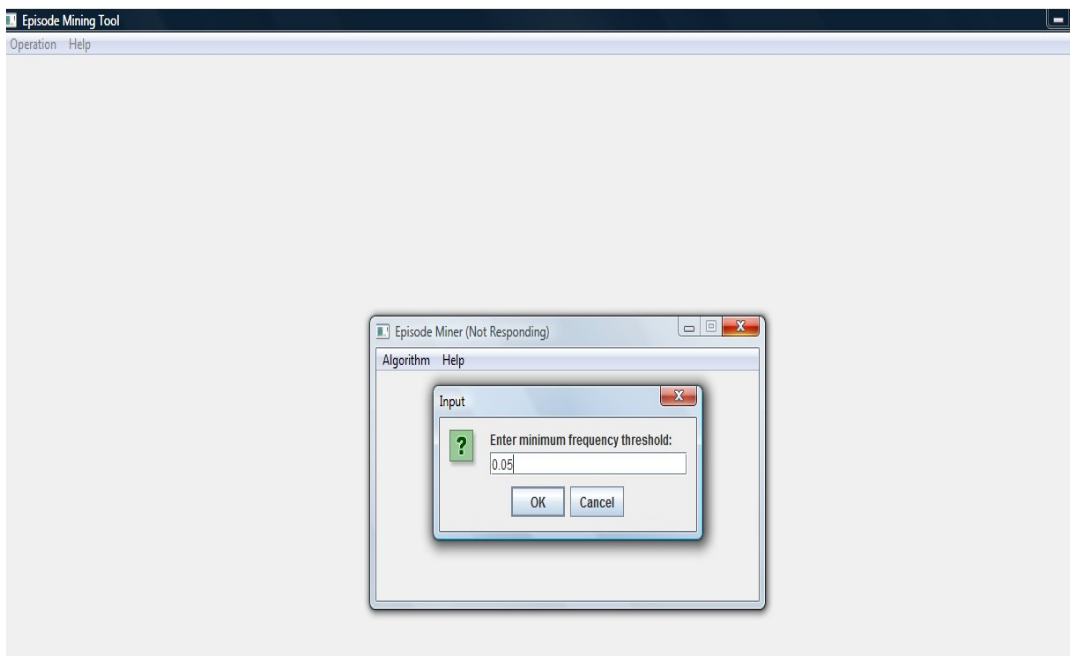


Figure 42: Giving minimum frequency parameter in episode miner

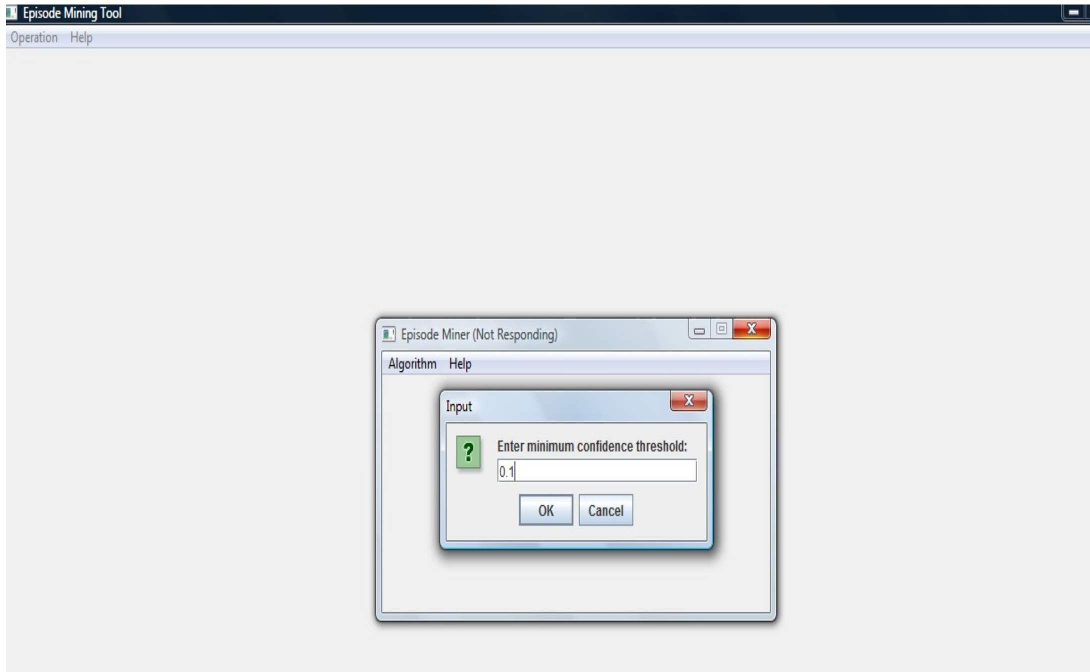


Figure 43: Giving minimum confidence parameter in episode miner

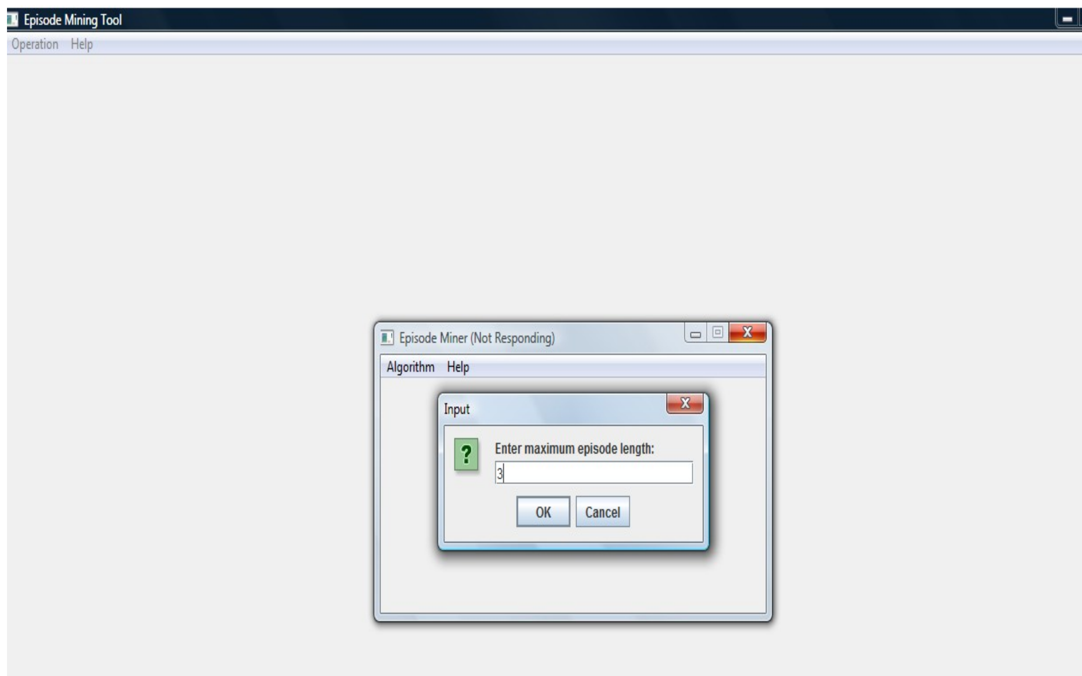


Figure 44: Giving maximum episode length parameter in episode miner

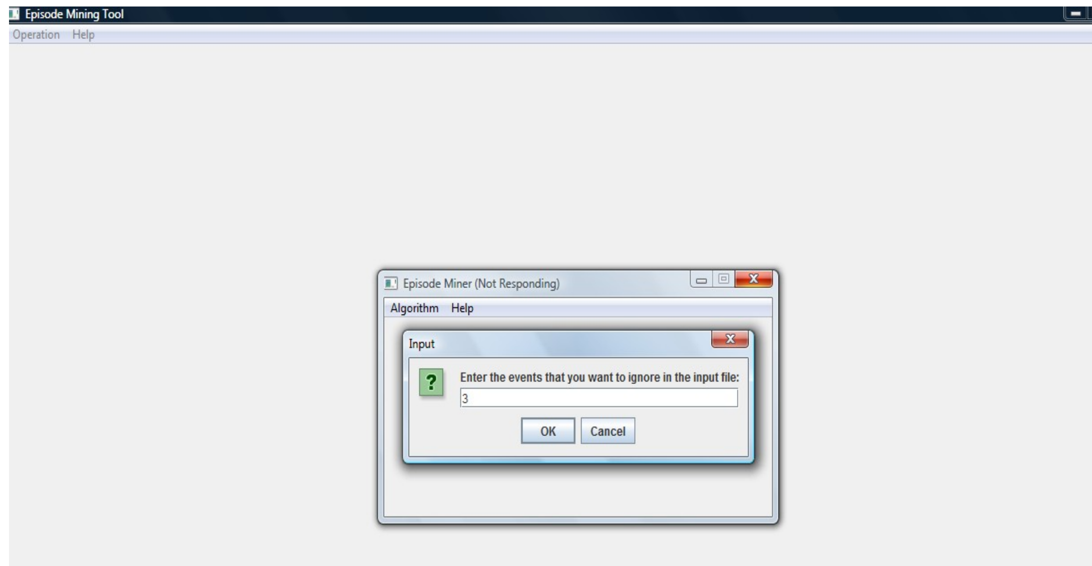


Figure 45: Giving ignored event types parameter in episode miner

After that the user selects the input file containing the dataset to be processed according to the chosen method as shown in Figure 46.

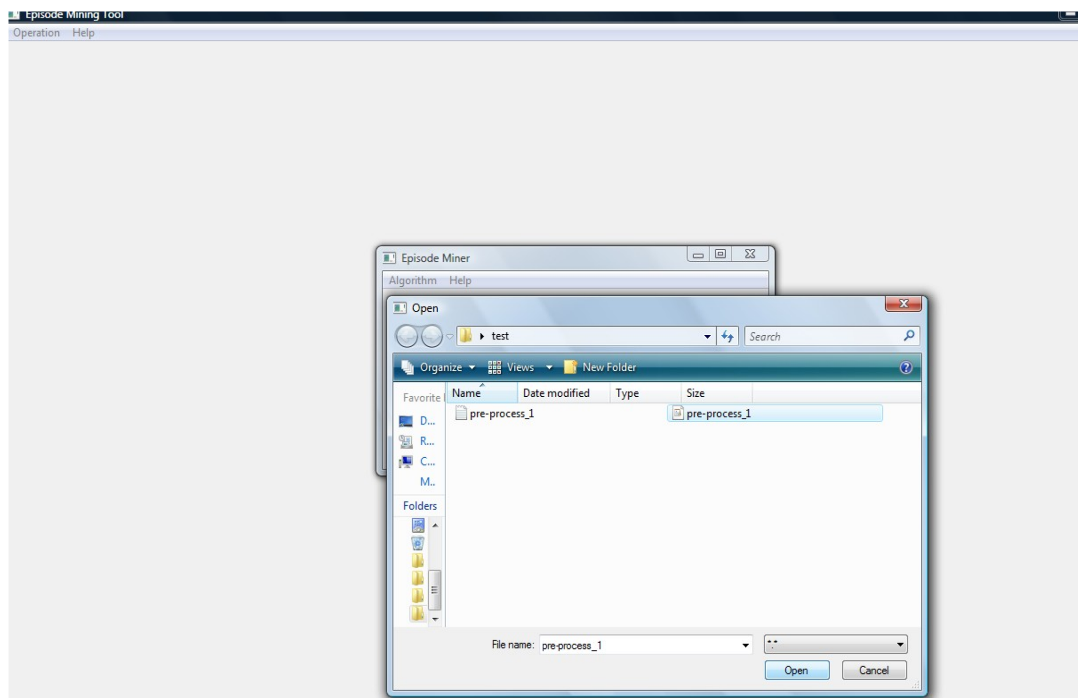


Figure 46: Selecting file from the browser for episode mining

Finally the user gives the name of the output file where the results to be written as shown in Figure 47 and a message is given to the user after the execution is completed as in Figure 48.

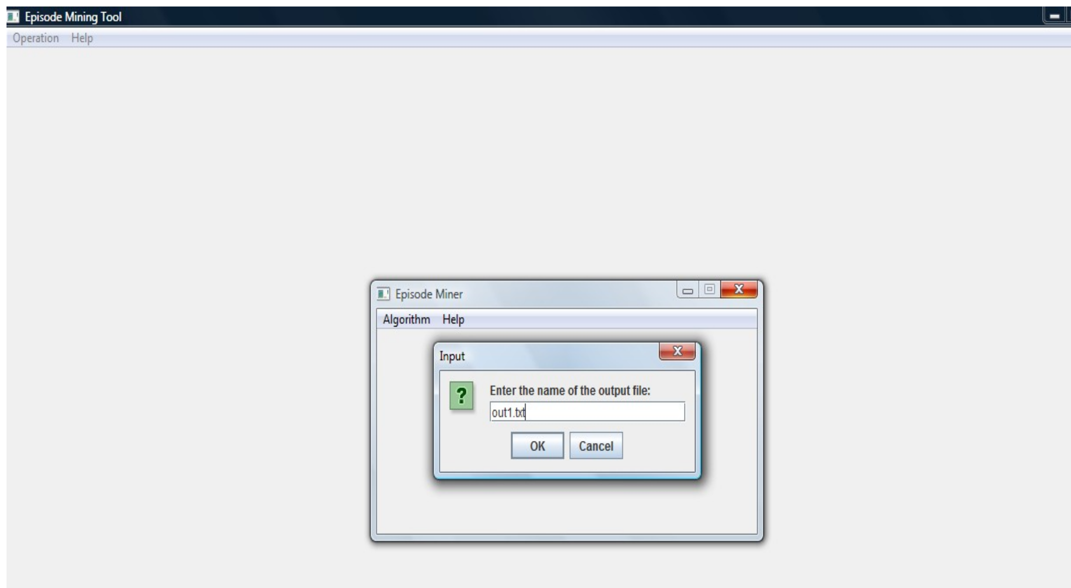


Figure 47: Specifying output file for episode mining

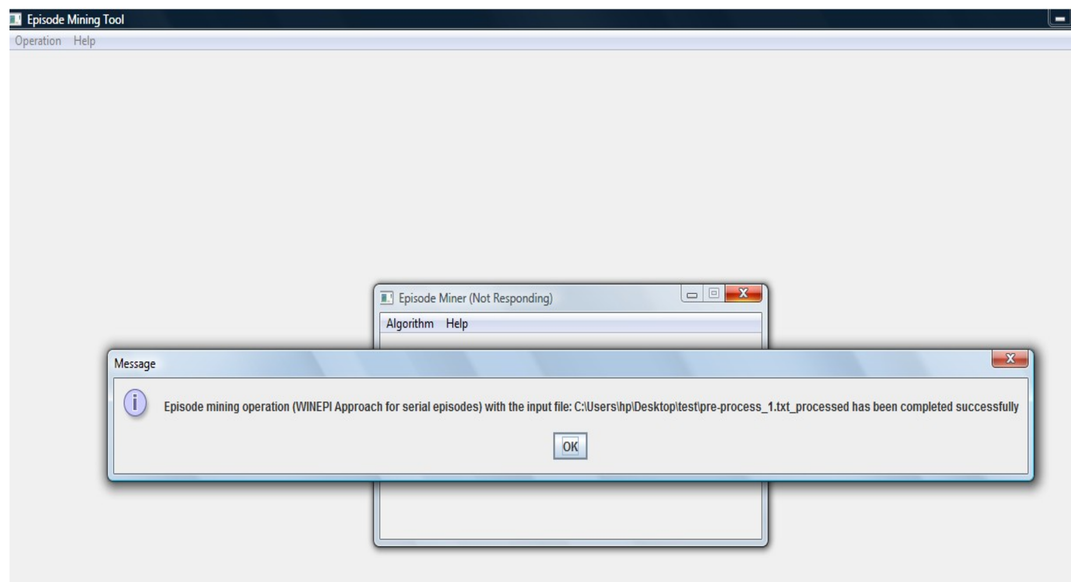


Figure 48: Message given to the user after episode mining

By using this component of EMT, the user can analyse the output files obtained from the previous components and see the results of episode mining operations in different types of charts. The user should firstly choose the “Output Analysing” item from the “Operation” menu of EMT as shown in Figure 49.

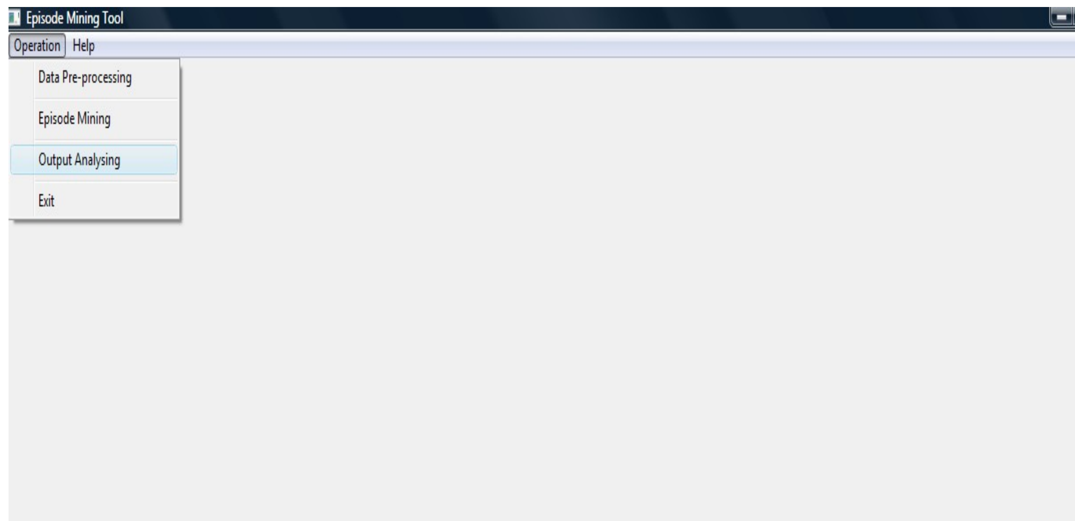


Figure 49: Output analysing operation in EMT

Then to analyse a single output file, the user chooses “Open single file” item from the “File” menu of the output analyser as shown in Figure 50.

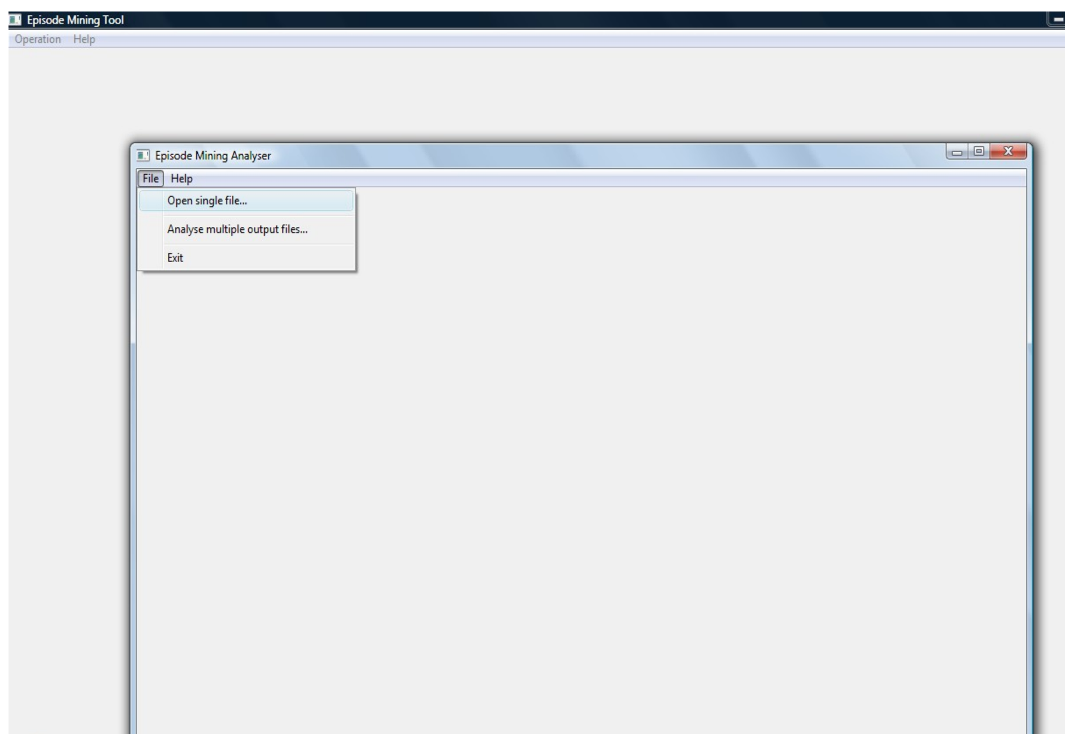


Figure 50: Open single file in output analyser

After that the user selects the input file to be analysed as shown in Figure 51.

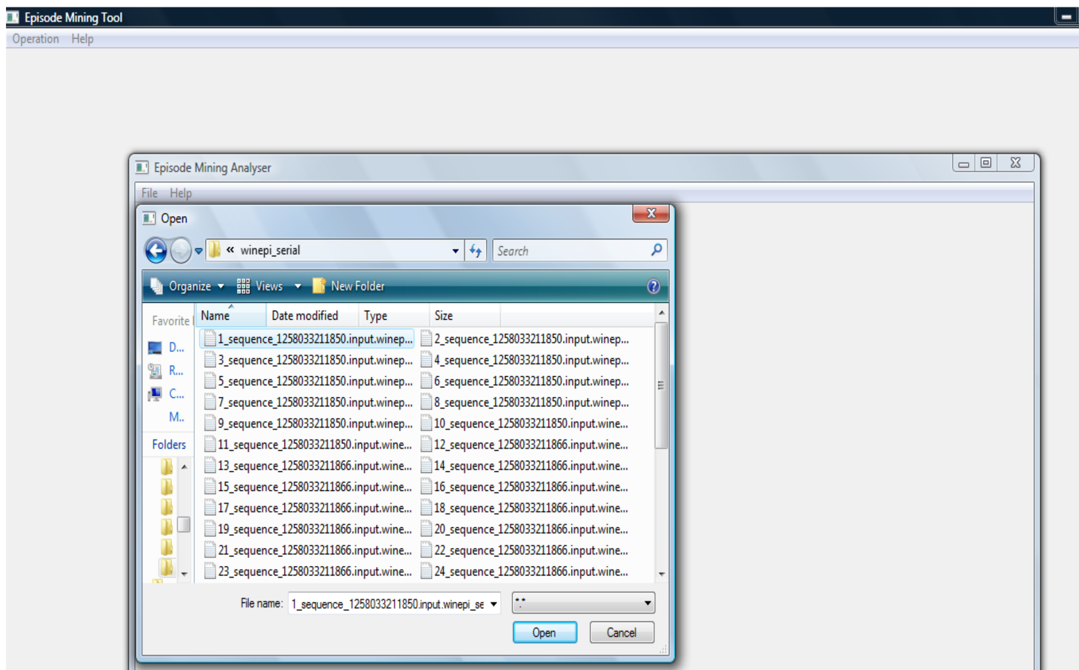


Figure 51: Selecting file from the browser for output analysing

For the next step, the user decides whether he wants to analyse according to the frequencies of the episodes or the confidence values of the rules generated as in Figure 52.

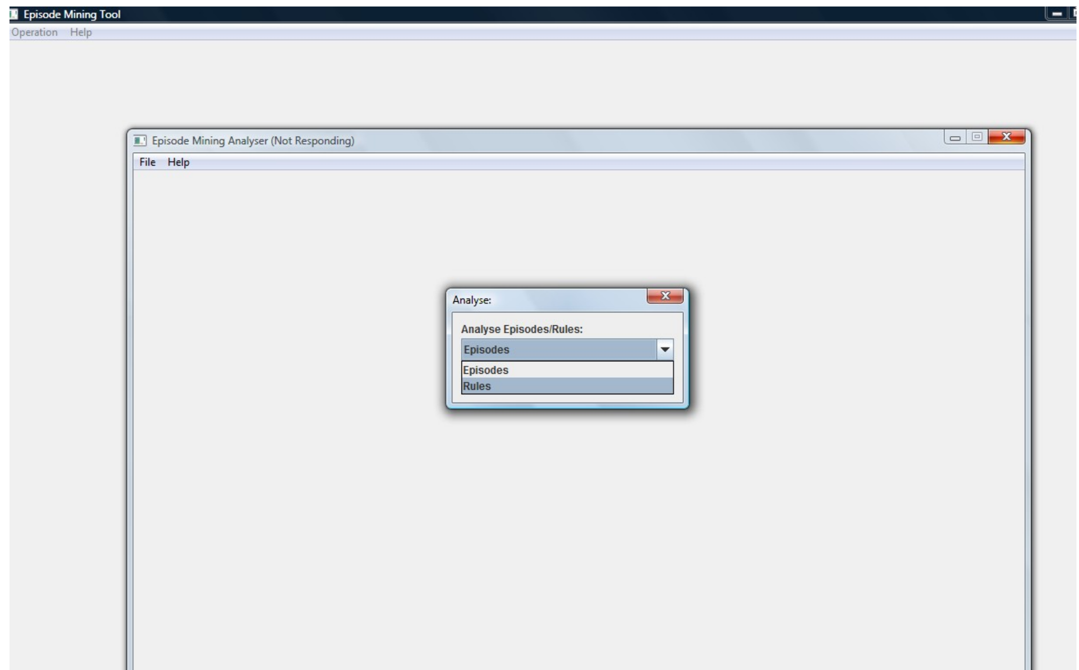


Figure 52: Analyse type selection for a single output

Then, the user gives the length of the rules or episodes as shown in Figure 53.

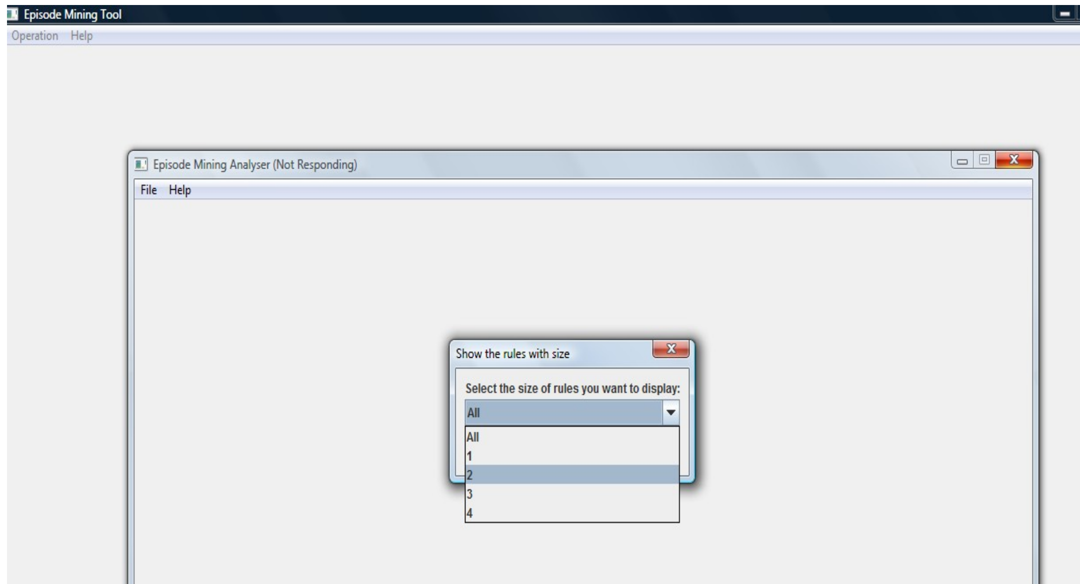


Figure 53: Rule length selection for output analysing

And the user selects the type of chart for the results of the analysis shown in Figure 54.

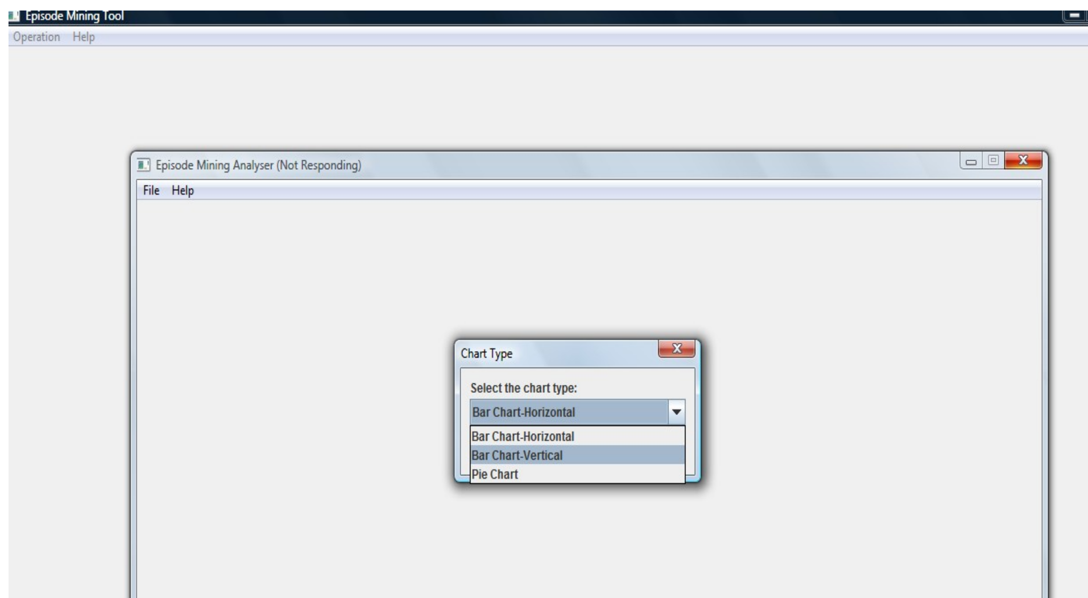


Figure 54: Chart type selection for output analysing

Then how many rules and which rules to be shown is selected by the user as shown in Figure 55.

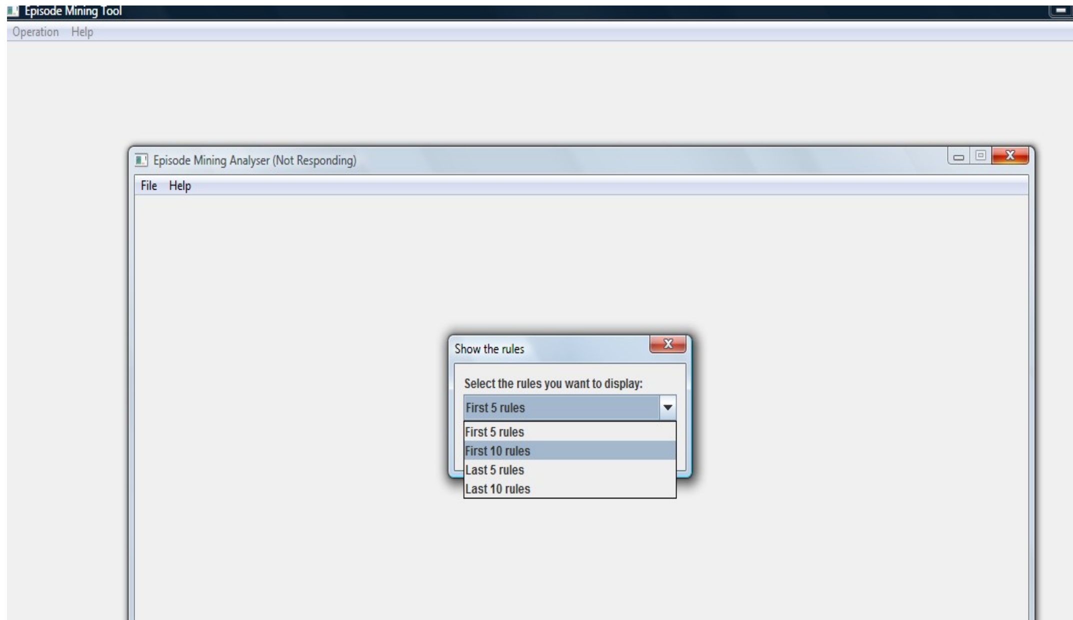


Figure 55: Rule count and type selection for output analysing

Finally, the result of analysis is given as a chart as shown in Figure 56.

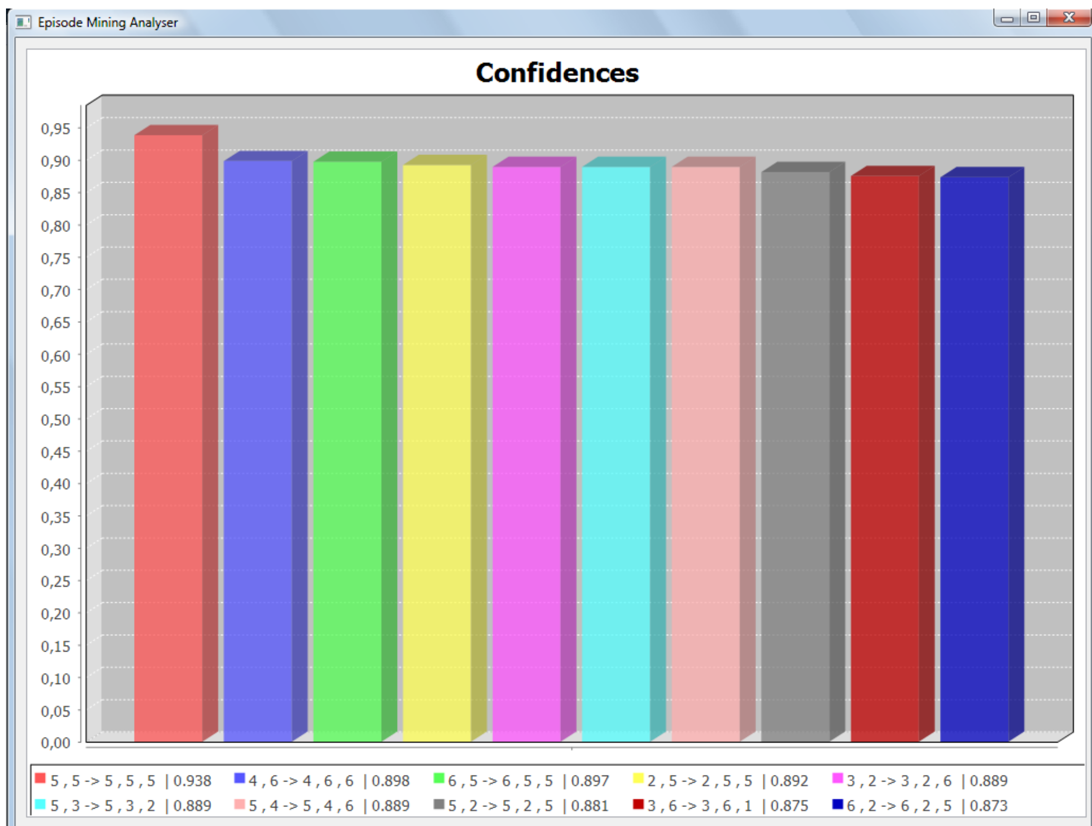


Figure 56: Result for single output analysis

If the user wants to analyse multiple output files in a single step, firstly he chooses the "Analyse multiple files" option and selects the directory containing the output files from the browser as shown in Figure 57.

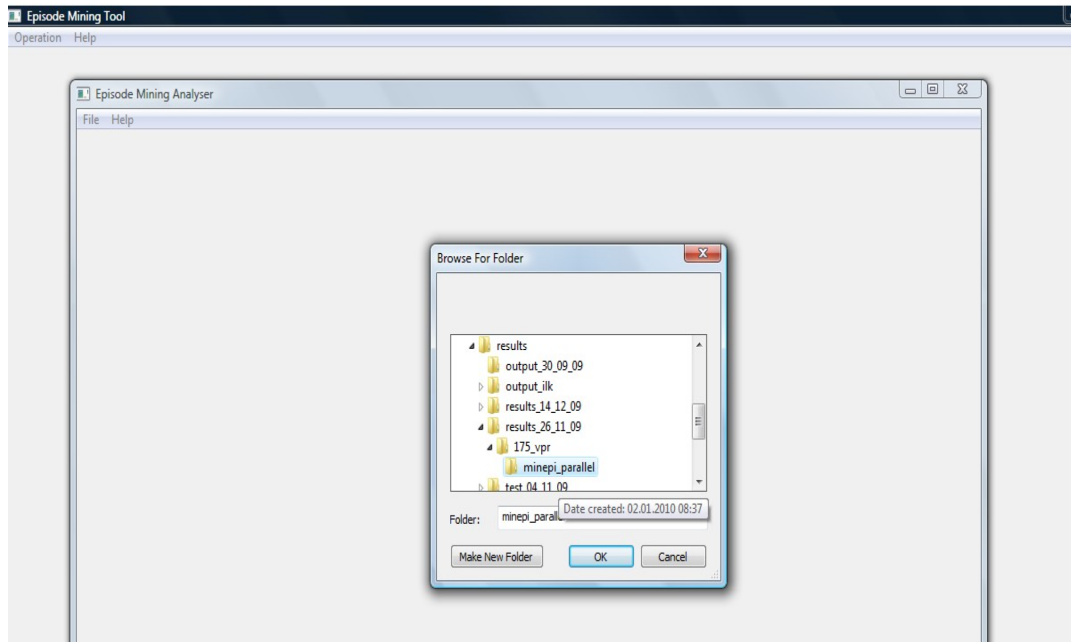


Figure 57: Selecting directory for multiple output file analysis

Then the user selects the way to analyse as in Figure 58 and the analyse type as "according to maximum/minimum confidence/frequency" as shown in Figure 59.

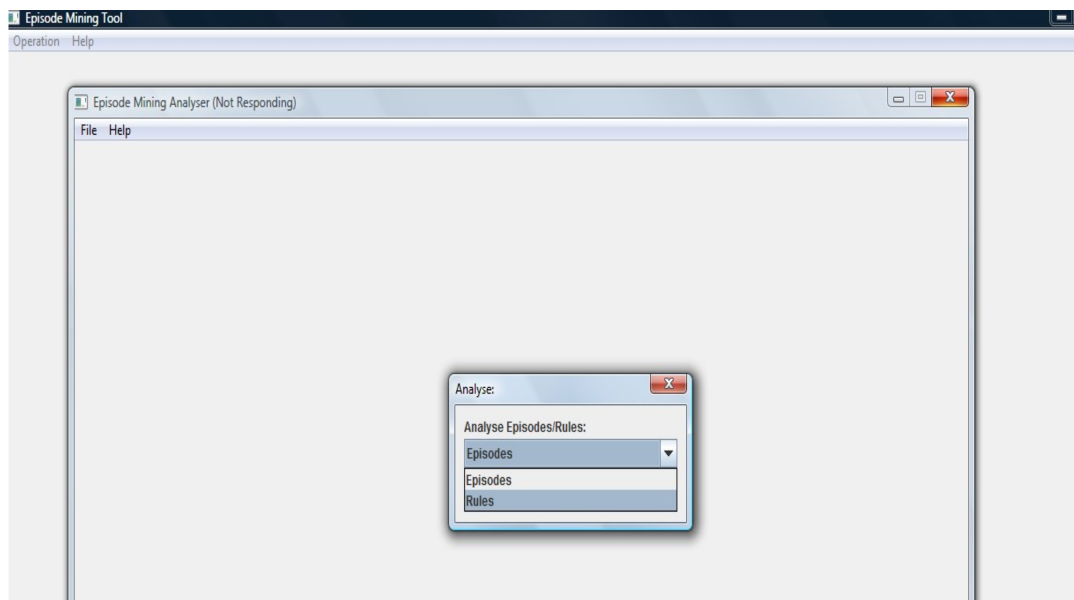


Figure 58: Analyse episodes/rules

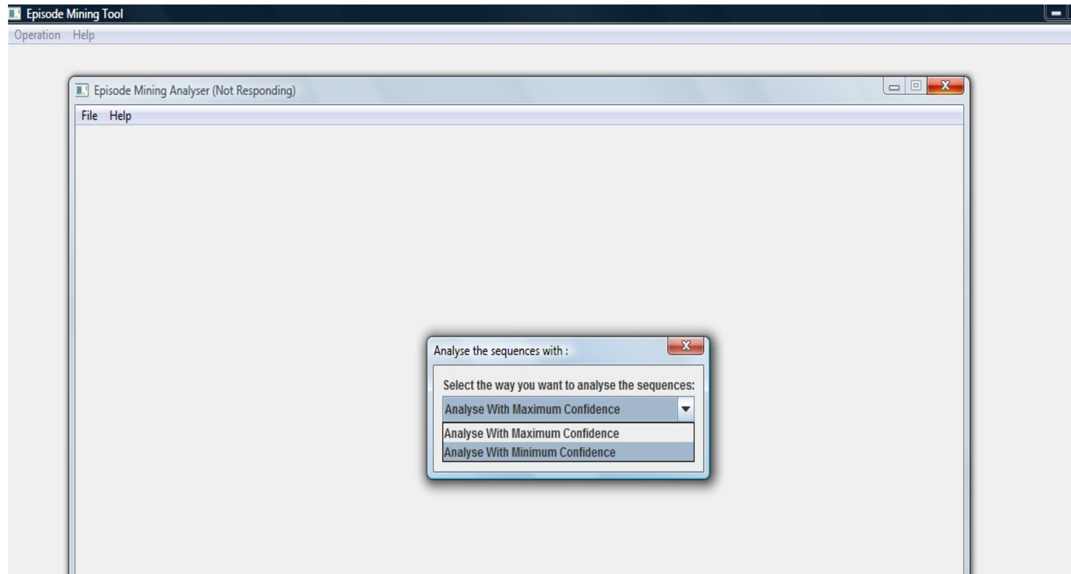


Figure 59: Selecting analyse type

After that EMT gets the length of episodes/rules and asks whether to group the output files or not as shown in Figure 60.

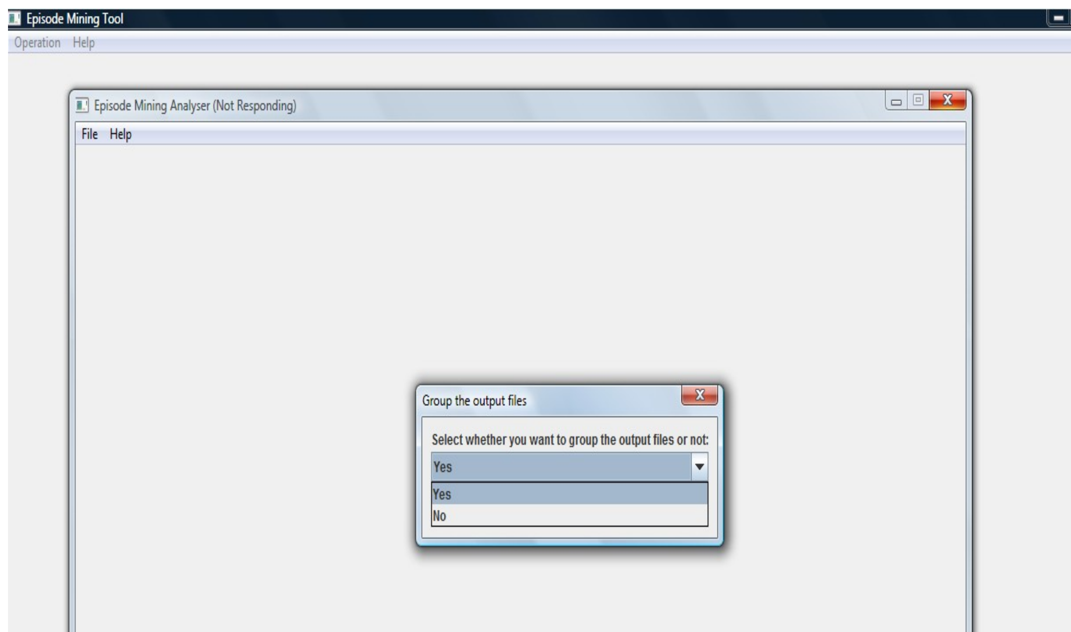


Figure 60: Grouping the output files

Then, the user selects the chart type and the result chart is given as shown in Figure 61.

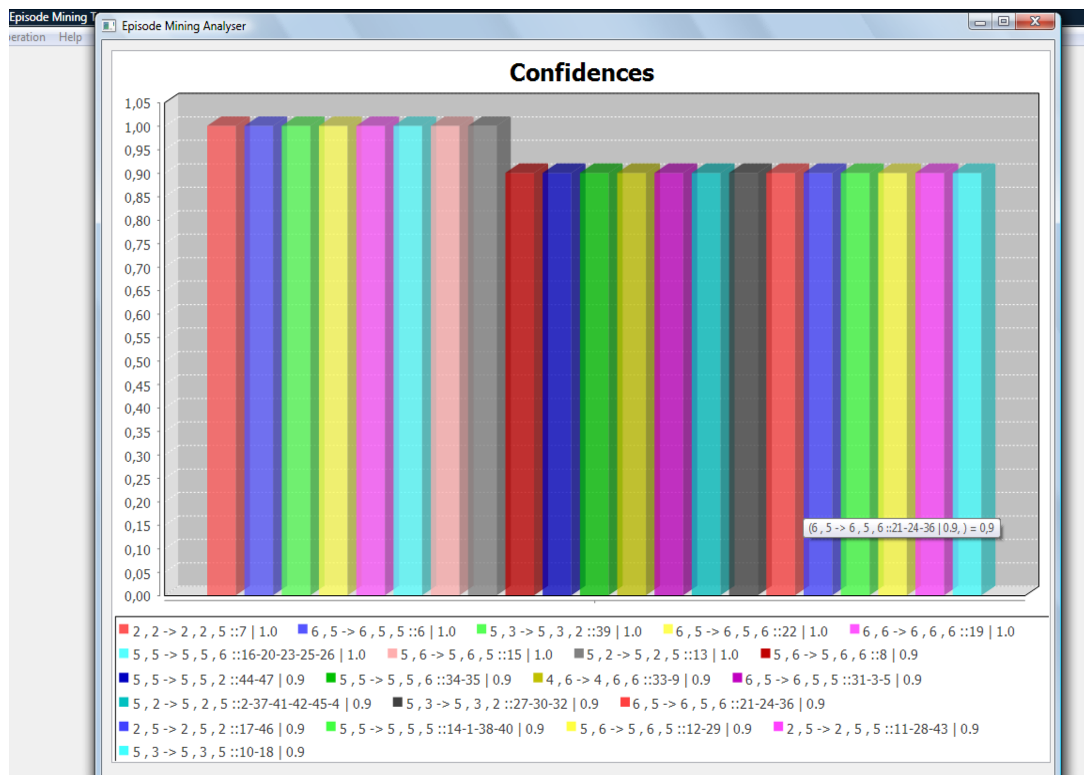


Figure 61: Resulting chart after grouping the output files

APPENDIX B

USAGE OF EMT IN COMMAND LINE MODE

In command line the user gives the necessary parameters to run pre-processor and the results are printed to the given output file as shown in Figure 62:

Figure 62: Runing pre-processor from command line

Here, the parameters should be given as:

DataPreProcessor.jar <input file> <outputFile> <option>

The option can be "--IPC" to process the input containing IPC values , "--unique-sequence" to process the input file containing unique sequences in its each line, "--AnalyseRulesWithIPC" to generate rules obtained from unique sequences and IPC changes in given files and "--filterAnalysis" to filter the rules containing only a rule and an IPC level value.

For episode mining, the user gives the parameters to episode miner as in the following way:

EpisodeMiner.jar <input file> <window width> <min frequency> <max episode length> <min confidence> <algorithm string> <output file> <window width-2>

Here, the parameter "window width-2" is used only for MINEPI algorithms and the parameter algorithm string is an option string containing 6 characters where each of these characters represent an implemented algorithm and must be either 0 or 1. The order of the algorithms can be given as:

- WINEPI for parallel episodes
- WINEPI for serial episodes
- Non-overlapping counts for parallel episodes
- Non-overlapping counts for serial episodes
- MINEPI for parallel episodes
- MINEPI for serial episodes

For instance, a string such as "000001" means that apply only "MINEPI for serial episodes" as shown in Figure 63.

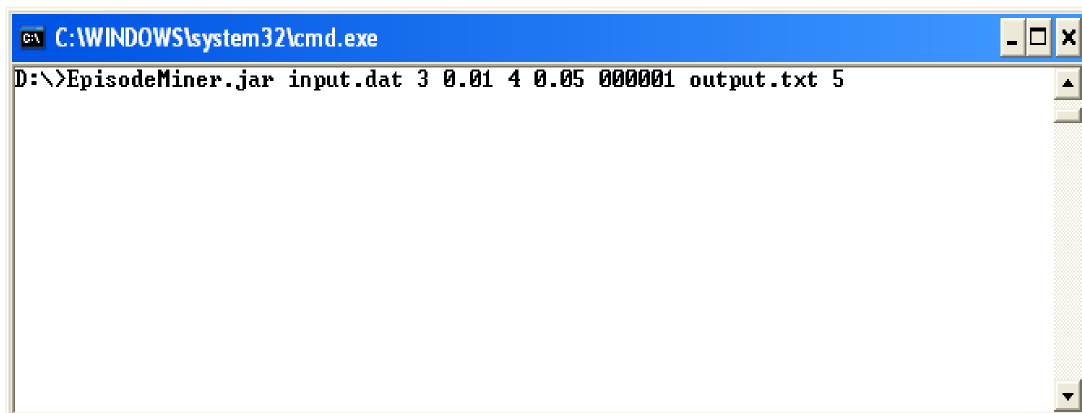


Figure 63: Runing episode miner from command line