# An efficient approach for performing Sequential Pattern mining in distributed frameworks using co-occurrence information

*Project Report Submitted in Partial Fulfillment of Requirements for the Degree Of*

**BACHELOR OF TECHNOLOGY**
**IN**
**COMPUTER SCIENCE AND ENGINEERING**

by

Jakshat Desai (16JE001854) (B.Tech 2020)

Under the guidance of

Dr. A.C.S. Rao

(Department of Computer Science and Engineering)

**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**

**INDIAN INSTITUTE OF TECHNOLOGY (INDIAN SCHOOL OF MINES), DHANBAD**

**INDIA**

**May 2020**

# Certificate

This is to certify that the project entitled “**An efficient approach for performing Sequential Pattern mining in distributed frameworks using co-occurrence information**” submitted by Jakshat Desai (16JE001854) (B.Tech 2020) is absolutely based upon his own work under the supervision of **Dr. A.C.S. Rao** in the Department of Computer Science and Engineering, Indian Institute Of Technology, Dhanbad and that neither the project report nor any part of it has been submitted for any degree/diploma or any other academic award anywhere before.

**Dr. A.C.S. Rao**
Assistant Professor
(Project Guide)
Department of CSE
IIT (ISM), Dhanbad

# Acknowledgements

I, Jakshat Desai(16JE001854), a final year student pursuing B.Tech in Computer Science and Engineering from Indian Institute of Technology (Indian School of Mines), Dhanbad am elated to present the project titled "**An efficient approach for performing Sequential Pattern mining in distributed frameworks using co-occurrence information**".

With all humility I wish to express a deep sense of gratitude to Dr A.C.S Rao, Assistant Professor, Department of Computer Science and Engineering , Indian Institute of Technology (Indian School of Mines), Dhanbad, for his vigilance, for sharing knowledge and experience, for guiding during the entire work, for making immortal comments and hence, acting as a beacon at each and every step in the accomplishment of the project work. His inclination towards research reflected in his speech at every meeting and it inspired me profoundly to try to find new ways to solve problems. This project would not have been possible without the zeal and motivation imparted to me by him.

I would also like to thank Mr. Ravi Kumar Singh, Research Scholar, Department of Computer Science and Engineering , Indian Institute of Technology (Indian School of Mines), Dhanbad, for his constant support and guidance in the study and experimental work that was required for the project. His patience and readiness to help further motivated me to overcome the challenges that I faced throughout the course of the project. It wouldn't have been possible for me to finish the work without him.

**Jakshat Desai(16JE001854)**
B.Tech Computer Science and Engineering

# Abstract

Sequences are found everywhere around us - in nature, in the market, in people's behaviour and several other places. Thus it is not surprising that sequential pattern mining is such a widely used and researched concept today. With the increasing popularity of distributed systems and parallel computing, the focus of research has been shifting from creating new serial sequential pattern mining approaches to finding efficient algorithms to mine frequent sequences parallely from distributed storages with the help of existing frameworks like MapReduce.

The focus of this project is one such efficient algorithm called Distributed Sequential Pattern mining using Co-occurrence information(DSPC) from the paper titled "**A novel mapreduce algorithm for distributed mining of sequential patterns using co-occurrence information**" by Sumalatha Saleti and R. B. V. Subramanyam. This project involves understanding the concepts, properties, advantages and disadvantages of previously used serial as well as parallel sequential pattern mining algorithms and then comparing them to DSPC. As a part of this project, DSPC is studied in detail and each of the functions involved in it is rewritten and broken down if needed in order to clarify the algorithm and to get a strong conceptual understanding of the same. The behaviour of the algorithm is also observed by implementing the algorithm and running it on popular sequential pattern mining datasets. The observations are then analyzed and inferences are drawn from them. Based on the in depth analysis of the algorithm and it's behaviour, improvements in the algorithm are then suggested in order to better it's performance. The improvements have been tested on several datasets and the results compared with that of the original algorithm using observation tables and line graphs.

# Table of Contents

# 1. Introduction

## 1.1 Project Motivation

Since the advent of the 21st century, the number of devices connected to the internet have been increasing exponentially. As people use more and more devices and perform more and more activities on them, the amount of data being generated from various sources has risen from Terabytes to Petabytes. In today's world, analysis of this data has become a very crucial step in almost all fields, be it ecommerce, medicine, defense, stock market, farming, etc. Knowledge of what is about to happen as well as the events that occur together is of great significance and in many cases crucial.

Since not all data contains knowledge, mining important information from given data is very crucial. This is where data mining comes into play. Data Mining is a field of study which deals with extracting useful information or knowledge from large amounts of data automatically. The knowledge that is to be mined can be of various different types. One such type of knowledge is the information about events that occur together frequently and thus lead to a formation of a pattern in data. For instance the knowledge about items that are frequently bought by customers together can be useful for companies and shops for designing selling strategies that can increase sales and profits. Finding such patterns in the given data comes under pattern mining. In many cases, not only the set of events which occur together frequently but also the order in which the events occur together matters. Enter sequential pattern mining. Sequential pattern mining is a topic of data mining concerned with finding statistically relevant patterns between data examples where the values are delivered in a sequence.

We have been taught several pattern mining algorithms for itemset mining as a part of our curriculum. Some of these are Apriori algorithm, FP-growth algorithm, etc. These algorithms help us find frequent itemsets from the given data. However, the main issue with these algorithms is their recurring nature. A number of iterations of the algorithm are needed in order to reach the answer. These algorithms are highly space and time intensive, thus for large amounts of data the time taken to reach the answer and memory consumed are very large. Thus scalability-wise, these algorithms pose a great issue.

As mentioned before, the data generated nowadays is in petabytes. Thus if slow algorithms are used to mine it, the knowledge mined might become insignificant within the time it would take to find the useful knowledge. In fact even if fast algorithms are used, the data mining process might still consume a lot of time if a single machine is used for processing. Moreover, storing all this data on a single system is also very impractical and would lead to very slow data access. The solution to these problems is using distributed storage and distributed computing. Thus the data mining algorithm used should impart itself easily to distributed computing. Moreover it should also be fast so that the knowledge mined remains relevant on being mined. One such algorithm is DSPC(Distributed mining of Sequential Patterns using Co-occurrence information).

Implementation of the DSPC algorithm posed a challenging and interesting task that involved the use of Hadoop MapReduce programming in Java. Before this, I had learned how to configure a Hadoop cluster and create very basic MapReduce programmes in Python. Thus I was very excited to learn about MapReduce programming properly and that too while applying it in such a relevant field like sequential pattern mining. Another challenge in the implementation of the algorithm was to simulate a hadoop cluster without actually having multiple systems. It was difficult as well as exciting to be able to apply my knowledge of docker containers in order to solve this problem by creating cluster nodes in containers. It was very satisfying to see a five node Hadoop cluster with cluster nodes created from docker containers work on my laptop and execute simple MapReduce programmes.

Thus the opportunity of learning MapReduce programming, container technology and data mining algorithms and being able to apply all of those to solve various real life problems provided more than enough motivation for the project.

## 1.2 Aims and Objectives

The core objectives which have been designated as fundamental to the project are:

- Analysing various serial and parallel algorithms for sequential pattern mining and their effectiveness.
- Understanding the DSPC algorithm for sequential pattern mining along with all the associated theory.
- Analysing the effectiveness of DSPC algorithm for sequential pattern mining and finding out how easily and effectively it imparts itself to distributed computing.
- Creating a means to run distributed programmes on a single machine easily while being able to simulate a multi-node Hadoop cluster on the same.
- Implementing the DSPC algorithm in Hadoop MapReduce and executing it on the simulated cluster.
- Finding outputs of the implemented DSPC algorithm for datasets and compiling them.
- Comparing DSPC with previously used algorithms.
- Suggesting improvements in the implementation of DSPC in order to improve performance.
- Compiling and presenting all observations and inferences.

# 2. Literature Survey

## 2.1 Background

Data Mining is a field of computer science that deals with the automatic extraction of useful information from large quantities of data. It consists of several sub-branches including but not limited to classification, clustering, etc. One of the applications of data mining that is not only widely used today but is also crucial for most organizations is pattern mining. Pattern mining was first introduced when Agarwal and Srikant came up with the Apriori algorithm for association rule mining. The problem that the algorithm aimed at solving was that of discovering patterns that frequently occurred within data. Discovery of such patterns proves to be very useful in several applications such as in the analysis of consumer behaviour, DNA sequences, web access data, etc. Pattern mining algorithms provide us with a way to interpret the data and gain useful conclusions from it by telling us about patterns that frequently occur within the data. For instance, items bought together frequently by customers, sequences of DNA that are frequently found, etc.

Pattern mining has several subtypes like association rule mining, frequent itemset mining, etc. However these types of pattern mining approaches do not provide us information about the sequential ordering of events in the frequently occurring patterns. Such information might prove to be very useful in several cases. For example people would always buy accessories for an appliance only after they have bought the appliance itself. The pattern mining approach that allows us to find frequent ordered sequences of events from the data is sequential pattern mining. This branch of pattern mining has numerous applications such as market basket analysis, text analysis, e-learning, etc and is especially useful with time series data.

Sequential Pattern mining can be applicable to sequential data in which the data is in the form of ordered sequences of symbols( like customer transactions in a store) and to time series data in which the data is in the form of numeric sequences( for example electricity consumption for several different households). Some vocabulary associated with sequential pattern mining is elaborated upon in the following lines:

- An **itemset** is a set of items belonging to a predefined universal set of items.
- A **k-itemset** is an itemset consisting of k items. The length of such an itemset is said to be k.
- A **sequence** consists of one or more itemsets. The ordering of the itemsets in a sequence is non-trivial. The example of a sequence can be a sequence of a customer's transactions at a store over a period of time. Eg <(bread,butter),(bread,butter,milk),(sugar,wheat)>. All items within one pair of parentheses are the part of the same transaction and were all bought together. All the itemsets within the same pair of angle braces belong to the same sequence, i.e in the above example, they are all transactions belonging to the same customer.
- A **k-sequence** is a sequence in which the sum of the lengths of all the itemsets belonging to that sequence is k.

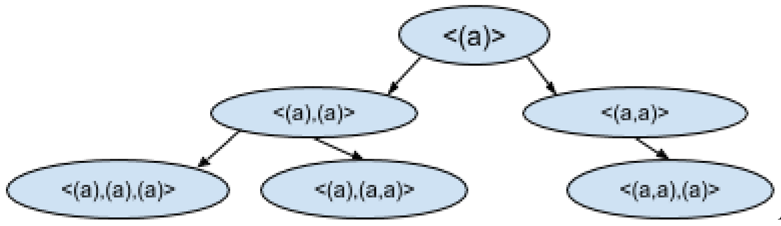
- A **sequence database** consists of a list of sequences belonging to the same dataset and each sequence has its own sequence identifier.
- A sequence A is said to be a **subsequence** of a sequence B or in other words, is said to be contained in a sequence B if and only if every itemset in A is a subset is a subset of some itemset in B. Note that we should be able to map each itemset in A to a different superset itemset in B. Sequential pattern mining aims at finding interesting subsequences in the given data.
- **Support of a sequence** A is represented by sup(A) and is equal to the number of sequences in the sequence database which contain A.
- **Relative support** of a sequence A is represented by relSup(A) and is equal to the ratio of the number of sequences in the sequence database which contain A to the total number of sequences in the sequence database.
- **Minimum Support** is the minimum support or the minimum relative support decided by the user for which a sequence is considered to be frequent.

Traditional sequential pattern mining algorithms are designed to run on a single node. Although these work well with small datasets, they face a lot of issues such as insufficient memory and computing power when dealing with big data. Nowadays the data to be mined is usually big data, hence using a single node to mine it is impractical. Thus the need for performing the mining task in a distributed fashion arises. Since traditional algorithms like Apriori are designed to run on a single node, they do not impart themselves for distributed computing efficiently on account of their serial and iterative nature. Thus new algorithms designed specially for distributed computation, that can run on distributed frameworks like MapReduce(by google) are required when performing sequential pattern mining on big data.

## 2.2 Introduction to Sequential Pattern Mining Algorithms

All approaches to sequential pattern mining perform sequence extension and itemset extension operations in order to generate various possible frequent sequences. An extension of a sequence is another sequence generated by either appending a 1-itemset to the end of the sequence (sequence extension or s-extension) or by adding another item to the last itemset of the sequence (itemset extension or i-extension). As per the Apriori property, every subsequence of a frequent sequence is also frequent. This property is employed by algorithms to reduce the size of the search space as extending an infrequent sequence will never yield a frequent sequence. A sequence database may have a very large number of frequent sequences in it. However many times, most of these frequent sequences may be redundant. For eg if a sequence <{a,b}{c}> is frequent then it goes without saying that <{a,b}> is also a frequent sequence and doesn't not need to be displayed separately since it is directly implied from the Apriori property. Thus the mining algorithms have to generate sequences which are not redundant.

Based on the manner in which algorithms search for frequent sequences, sequential pattern mining algorithms can be classified as either depth first search algorithms, breadth first search algorithms or pattern growth algorithms. Depth first search algorithms scan the sequence database and each time a frequent 1-sequence is found, it is recursively extended until it cannot be extended to a frequent sequence. Once this point is reached, the algorithm backtracks to explore other frequent sequences. Thus we don't move on from a 1-sequence to the next unless all frequent sequences that can be formed by recursively extending it haven't been explored. Examples of such algorithms include Spade, Spam, etc. Breadth first search algorithms on the other hand scan the sequence database and first find all the frequent 1-sequences in it. Once this has been done, these frequent 1-sequences are extended to obtain all possible 2-sequences in the sequence database. This process of exploration is continued until extension does not lead to any new frequent sequence. AprioriAll and GSP are examples of this class of algorithms. The main issue with the above two classes of algorithms is that their candidate generation process may generate a lot of patterns which are actually not present in the sequence database. This problem does not arise with pattern growth algorithms as they repeatedly scan the database for larger patterns and hence, all the patterns explored by the algorithm are always present in the sequence database. However repeated scans of the entire database is a costly operation and hence many pattern growth algorithms use some traits of the depth first search algorithms to solve this problem. PrefixScan algorithm falls in this category.



*Patterns are generated in depth first order of the illustrated tree in depth first search algorithms and in level order in case of breadth first search algorithms*

The time taken by the algorithm depends on the size of the search space and the cost of generating extensions and finding out if they are frequent. From this statement it may appear that pattern growth algorithms are faster than the rest, however the database scan overheads associated with them actually make them slower than many of the breadth first search and depth first search algorithms. The size of the search space, and hence indirectly the time taken also depends on the minimum support decided by the user. In general, for lower values of minimum support, the number of patterns that need to be generated and checked increases exponentially and hence it takes more time to find all the frequent patterns.

## 2.3 Some Serial Sequential Pattern Mining Algorithms

Some of the serial sequential pattern mining algorithms are explained in brief in the following lines:

- GSP stands for Generalized Sequential Patterns. This algorithm was proposed by Srikant and Agarwal as an improvement over AprioriAll, a sequential pattern mining algorithm proposed by Srikant and Agarwal previously. GSP overcomes several drawbacks of AprioriAll, the first being that GSP incorporates time gap constraints in which all events of the same transaction are only considered together in the same sequence if those events take place within a user specified minimum or maximum time interval. The second issue of AprioriAll that GSP solves is that of taking the definition of transactions very rigidly. In GSP, the elements of different transactions can be considered in the same sequence if the minimum and maximum times of the transactions are within a pre-specified time window. Apart from these two problems, AprioriAll also suffers from the issue of not being able to find patterns across different levels of hierarchy which is solved by GSP. GSP uses a breadth first search strategy to perform multiple passes over data to find frequent sequences. In the first pass, the algorithm scans the database to find all the frequent 1-sequences. In the kth pass, the algorithm generates a set of candidate k-sequences from all the frequent (k-1)-sequences found in the previous pass. This is called the joining phase. After this, the algorithm scans the database to get the support count of all the candidate k-sequences. The candidate sequences with support count less than the minimum support are then discarded in the pruning step and this concludes the kth pass of the algorithm. The remaining candidate sequences after the pruning step constitute the set of frequent k-sequences. The candidate sequences generated are stored in a hash tree data structure. The GSP algorithm takes 30% to 5 times less time than AprioriAll and hence is a major improvement over it. GSP also generates a lesser number of candidates than AprioriAll.

- PrefixSpan stands for Prefix-projected Sequential PatterN mining. This is a pattern growth algorithm for mining sequential patterns. AprioriAll based algorithms such as GSP suffer from several disadvantages. Firstly, they often generate large sets of candidate sequences. Secondly, AprioriAll based algorithms are breadth first search algorithms and hence in order to generate a sequence of length x, the database needs to be scanned x times. Thirdly generating long sequential patterns poses to be a big problem since an exponential number of long patterns can be generated from a combination of smaller patterns. The candidate generation method used in AprioriAll based algorithms leads to the above mentioned disadvantages. In order to solve the above issues, it was thought that candidate generation could be performed using an FP tree. This led to the development of a pattern growth algorithm called FreeSpan which performed much better than GSP. The main idea used in FreeSpan was that of projected databases in which patterns were grown only by extending local frequent sequences. The main disadvantage associated with FreeSpan was that projected databases often led to a high cost since the size of the projected database may be equal to that of the actual database when the sequence for which the projected database is being constructed is present in all the sequences of the sequence database. Also, since the current sequence could grow from any point in the sequence, checking for all possibilities became very expensive. These disadvantages were overcome by prefixSpan. PrefixSpan only considers frequent prefixes instead of considering all frequent subsequences. Thus in prefixSpan, no candidate sequences need to be generated and also the projected databases always keep shrinking. Just like in the case of FreeSpan, the main issue in PrefixSpan is also that of creating the projected databases. However, this problem can also be solved by using optimizations such as performing item pruning in the projected databases and using pseudo projection. PrefixSpan with optimizations is experimentally found to perform better than FreeSpan as well as GSP.

- SPAM stands for Sequential PAttern Mining. This was one of the first depth first search sequential pattern mining algorithms. Since it is a depth first search algorithm, it requires that the entire sequence database can be fit in the main memory. This method performs a depth first traversal of the lexicographic tree of sequences. In this tree, each node represents a sequence and the sequence represented by a node is either a sequence extension or an itemset extension of the sequence representing its respective parent. When performing a depth first traversal of the lexicographic sequence tree, if a node n has a support count not less than the minimum support, only then that node is stored and its children are visited. In other words, the subtrees rooted at nodes representing infrequent sequences in the lexicographic sequence tree are pruned during traversal. In order to avail fast support counting, the vertical bitmap data structure is created for each item in the dataset and each bitmap has a bit position for each transaction in the sequence database. If a data item x is present in a transaction y, then the yth bit position in the bitmap corresponding to the data item x is set to 1, else it is set to 0. Candidate generation is performed during traversal of the lexicographic sequence tree through the use of bitmaps. Although SPAM is faster than PrefixSpan for larger databases( on account of fast support counting), it is very memory intensive and hence is very space inefficient compared to other algorithms.

## 2.4 Some Parallel Sequential Pattern Mining Algorithms

Nowadays, the data to be mined is usually too huge to be able to be stored and processed by a single machine. Apart from this, most data is distributed over several machines. Hence, having algorithms which are able to perform data mining in a parallel and distributed manner have become a necessity in order to be able to perform the mining task faster and without having to move all the data from the distributed storage to a single system. Some of the algorithms which perform sequential pattern mining parallely are discussed as follows:

- (DPF)Data Parallel Formulation, (STPF)Static Task-Parallel Formulation and (DTPF)Dynamic Task-Parallel Formulation algorithm is similar to PrefixSpan and uses the concepts of projected databases and lexicographic tree of sequences for sequential pattern mining. There are two methods which can be used to partition the problem into smaller parts which can be sent to the mappers and both of these techniques distribute the problem amongst the processors in a balanced manner. This algorithm also dynamically assigns work to idle processors as per requirement. Though increasing the number of processors increases the speedup of the algorithm, it also leads to a load imbalancing problem. Apart from this, this algorithm suffers from other issues such as multiple database scans and high I/O overhead and the process of dynamic assignment of load to idle processors requires a very high amount of inter-process communication.

- DPSP stands for Distributed Progressive Sequential Patterns. This is an AprioriAll based algorithm developed for the MapReduce framework. DGSP or Distributed GSP was the predecessor of this algorithm. As the name suggests, DGSP was a direct distributed implementation of GSP. This algorithm used a combination of two MapReduce jobs repeatedly to generate candidate sequences and perform pruning respectively. The algorithm split the database into smaller parts and distributed them to the mappers which performed the sequence generation task. The reducers of the first task then consolidated the set of new candidate sequences and this set was split and sent to the mappers of the second MapReduce phase. After support counting was performed by the mappers, the reducers of the second phase pruned the candidate sequences whose support count was less than the minimum support. Thus as it is clearly visible, DGSP was a direct parallel implementation of GSP. On account of multiple database scans and multiple rounds of MapReduce jobs, this algorithm performed very poorly. Despite these disadvantages, a major advantage of DGSP was that it had very less inter-process communication. Thus, DPSP was created as an improvement over DGSP so that the advantage of using DGSP could be retained. DPSP's MapReduce jobs remove itemsets which are not of any use and also take care of some time gap constraints. One of the major disadvantages of DPSP is the same as that of DGSP and that is that of multiple rounds of MapReduce jobs which lead to load imbalancing. Apart from this, DPSP also does not perform well when it comes to mining long sequential patterns.

- SPAMC stands for Sequential PAttern Mining on the Cloud. As the name suggests, it is based on the SPAM algorithm. Just like the serial SPAM algorithm, SPAMC also constructs a lexicographic tree of sequences depthwise and uses it to find new sequences and prune them if required. SPAMC distributed the mining task among mappers very effectively in its iterative MapReduce approach. However, performing several rounds of MapReduce jobs implies that this method has to deal with high costs of repeatedly loading data. Apart from this, SPAMC also suffers from the load imbalance problem. Just like SPAM, SPAMC also uses a bitmap data structure which is unable to handle large databases. Thus SPAMC struggles with scalability when it comes to large databases. In order to overcome these drawbacks of SPAMC, SPAMC-UDLT was proposed. UDLT stands for Uniform Distributed Lexical sequence Tree. SPAMC-UDLT is highly scalable, performs the mining task without numerous rounds of MapReduce and also solves the load imbalance problem. In fact, SPAMC-UDLT provides better load balancing than several other existing algorithms for performing sequential pattern mining on the cloud. Moreover, SPAMC-UDLT is also much faster than SPAMC. The main issues with SPAMC-UDLT are that it doesn't deal with gap constraints and it also doesn't provide outputs in condensed representations.

- Apart from the above mentioned algorithms, there are many other algorithms in parallel sequential mining that cannot be classified as Apriori based, pattern growth or depth first search. These algorithms come under the category of hybrid algorithms. Several algorithms fall under this category. Many of them are designed keeping specific applications in mind such as ACME which was made for extracting repeating subsequences from very long sequences so that it could be applied to the field of bioinformatics( such as for the study of long DNA sequences). Other algorithms such as MG-FSM and MG-FSM+ are very scalable and incorporate constraints as well as condensed representations and are created to find frequent sequences fast while using memory efficiently and thus can be used for a wide range of applications.

## 2.5 Common drawbacks associated with Parallel Sequential Pattern Mining Algorithms

Some of the common drawbacks associated with parallel sequential mining algorithms are described as follows:

1. Many algorithms use an approach in which MapReduce jobs are run again and again in order to obtain the required answer. This is often time consuming and causes load imbalancing.
2. Some algorithms have a large communication overhead and/or a large I/O overhead.
3. The need to scan the sequence database is another disadvantage that algorithms often deal with. The obvious consequence is that the algorithm becomes time consuming and faces a lot of overhead to reload the database again and again. Also, in case of big data, scanning the database over and over again is not a feasible option.
4. Some algorithms are not able to scale well to larger databases on account of the time and memory consumed. The data structures used by an algorithm may be one of the reasons for this.
5. Many algorithms suffer with the issue of load imbalance. This means that the tasks distributed among the different nodes of the distributed system are of different sizes. While some nodes may finish their processing very fast, others may take a long time to finish processing. Load balancing is important for ensuring that the algorithm's execution takes place as fast as possible and also to ensure that particular machines do not deteriorate faster due to them performing more operations than the rest.
6. Some algorithms are unable to handle constraints.
7. GSP based algorithms are breadth first in nature and hence generate candidate sequences level wise. This often leads to a number of rounds of MapReduce and multiple database scans. These further lead to other problems like load imbalance.

## 2.6 Importance of good Parallel Sequential Mining Algorithms

Fast and memory efficient algorithms are increasingly in demand due to several reasons:

1. As mentioned previously, most databases are stored in distributed storages and thus bringing data to a single node in order to mine it is a tedious and memory inefficient operation.
2. Moreover it might not even be possible to bring all the data to a single machine due to memory constraints.
3. A single machine may not have enough processing power to mine all that data in time. Patterns may become irrelevant if not mined in the required time frame.

These are some of the reasons which call for the development of fast and memory efficient sequential pattern mining algorithms that can mine data distributed among different machines parallely. Such algorithms have very wide applications in several fields. Some of them are mentioned briefly in the following lines:

- Health Care: Patterns observed in the patients in terms of their activity and the symptoms displayed by them can help with diagnosis of diseases and with the development of systems to assist patients.
- Web Usage Mining: This application allows to find out the patterns in which consumers visit web sites which has several applications such as marketing.
- Bioinformatics: This application involves mining frequent patterns in DNA sequences which provides several important insights.
- Telecommunication: Sequential pattern mining in telecommunication can be used for mobile based e-commerce, for predicting the location of a mobile device, etc.
- Intrusion detection: In this application, sequential pattern mining proves to be useful in detecting network attacks.

These are just some of many applications of sequential pattern mining. From the above points we see that sequential pattern mining algorithms are crucial in several fields. On account of the wide application of sequential pattern mining, the increasing use of distributed storages and the exponentially rising sizes of databases, fast and efficient algorithms which can run on parallel platforms and are able to use data from distributed storages are greatly needed.

# 3. Distributed Sequential Pattern mining using Co-occurrence information(DSPC)

## 3.1 Introduction

As discussed in the previous chapter, sequential pattern mining plays a very crucial role in several fields and there is a need for a faster and more efficient sequential pattern mining algorithm that can mine frequent sequential patterns parallely from large sequence databases present in distributed storages. The preexisting algorithms for parallel sequential pattern mining suffer from several disadvantages, the major ones being iterative MapReduce jobs and high communication and scheduling overhead between the parallel processes. Thus an algorithm with the following features is required:

1. The algorithm should not use an iterative MapReduce approach so that communication and scheduling overhead is as less as possible.
2. The algorithm should be able to prune infrequent sequences as early as possible and should be able to execute properly even if the search space with which the algorithm is dealing is large.
3. Support counting often poses an issue in several algorithms. In some algorithms, although support counting doesn't take extra memory, it might take multiple database scans which is time consuming as well as inefficient as in the case of DGSP. Other algorithms such as SPAMC use a bitmap data structure for fast support counting, however this data structure is not scalable for large size of data. Thus the algorithm should use a data structure which allows fast support counting and also scales well as the size of the database increases.

DSPC or Distributed Sequential Pattern mining using Co-occurrence information is an algorithm that has all the above mentioned features. This algorithm is designed for the MapReduce distributed programming framework. Thus, before diving into the details of the algorithm, a brief discussion about MapReduce is in order. MapReduce is a distributed programming framework developed by Google for easily executing parallel programmes on a distributed computing system. There are four phases of execution in a MapReduce program:

- Splitting: In this phase the input provided to the MapReduce job is split into chunks.
- Mapping: In this phase, each split from the previous phase is passed onto a mapping function.
- Shuffling: This phase consolidates all the outputs from the mapping phase.
- Reducing: This phase combines values from shuffling phase and summarizes the dataset.

For implementation purposes Hadoop will be used for storing the sequence database in a distributed manner in the Hadoop HDFS and MapReduce programmes will be executed on the same. Hadoop divides jobs primarily into two tasks – Map tasks and Reduce tasks. Map tasks are a part of the splitting and mapping phase and reduce tasks consist of shuffling and reducing. Thus by the nature of MapReduce programmes, it is convenient to perform parallel computations on the task nodes for problems in which the order in which computations are performed on the tuples is insignificant. Thus preprocessed data can be fed to an HDFS after which a job can be run through a job tracker. The mapping task splits the data and gets the function output for each of the split chunks. The reducing task then combines the outputs of the mapping task and presents a single consolidated output. A diagrammatic representation of the entire process is shown below for an example. In the given example MapReduce programming has been used to calculate the number of instances of each distinct word in the input data.

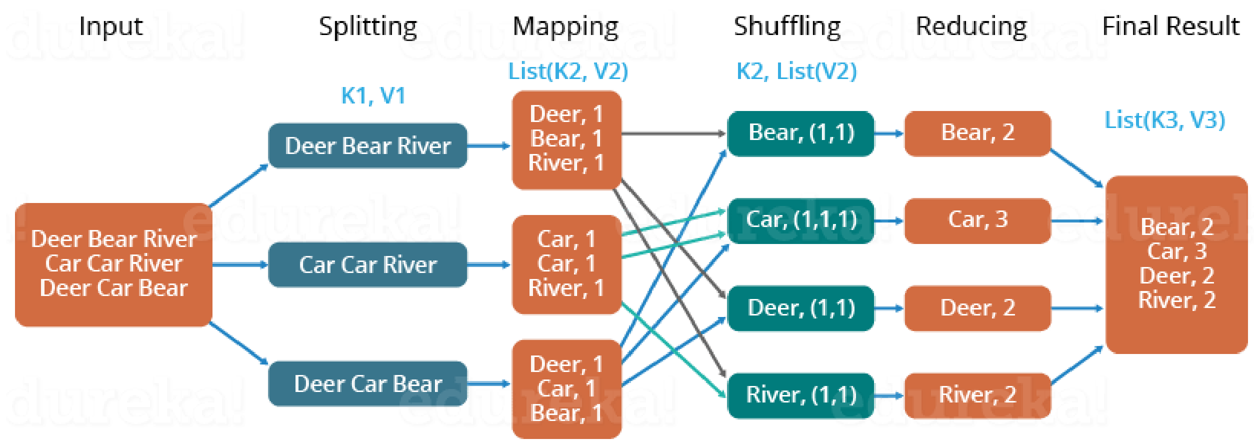


*Diagram 3.1 Diagrammatic representation of a word count programme in MapReduce*

Since MapReduce programming has been discussed, DSPC and how it solves the issues associated with previously used parallel sequential pattern mining algorithms can be elaborated upon now. The DSPC algorithm uses two data structures namely sequence index list(SIL) and Co-occurrence map(CMAP):

- The SIL data structure is used for fast support counting without repeatedly having to scan the input database. SIL for a particular sequence $s_1$ stores sequence id corresponding to only those sequences in the sequence database which contain the sequence $s_1$ along with the positions in the respective sequences where $s_1$ is located. Thus SIL doesn't waste space unnecessarily by not storing ids of sequences that don't contain $s_1$.
- CMAPs are used to discard infrequent sequences prior to support counting. This saves a lot of time that would have otherwise been wasted in counting the support for several infrequent sequences.

DSPC uses two MapReduce jobs to mine sequential pattern data. Thus, since the approach does not use iterative MapReduce jobs, communication and scheduling overheads as well as load imbalance issues are greatly reduced. In the first MapReduce job, the CMAP is constructed in a distributed manner and in the second MapReduce job, the CMAP created is used by all the nodes to discard all the infrequent sequences after which SILs are used for support counting and the sequences which do not satisfy the minimum support criteria are then pruned.

Thus, DSPC has all the features discussed in the beginning of this section. It does not use an iterative MapReduce approach and it also prunes infrequent sequences very early, prior to support counting using CMAPs. It also performs support counting fast using SILs which do not consume any more memory than is needed.

## 3.2 Algorithm

### 3.2.1 Important terms and concepts

| Sequence ID | Sequence |
|---|---|
| 1 | <(a,b,c)(c,d)(c,h)> |
| 2 | <(a,b)(a,d,f)(c,d,g)> |
| 3 | <(c,e,g)(h)> |
| 4 | <(c)(a,d,g)> |
| 5 | <(g)(h)> |

*Table 3.1 Sample sequence database*

- The Sequence Index List (SIL) of a sequence S is a mapping from sequence ids to lists of positions of S in sequences with those respective ids. For example, the SIL of the sequence <(a)> for the sequences given in table 3.1 is:

| Sequence ID | list of positions |
|---|---|
| 1 | {1} |
| 2 | {1,2} |
| 4 | {2} |

*Table 3.2 SIL of <(a)> for sequences in table 3.1*

- An item j is said to succeed an item k by itemset extension in a sequence S if j as well as k belong to the same itemset i which is a part of the sequence S and k comes after j in i. For example in the sequence <(1,2,3)(4,5)(6,7,8)>, 3 succeeds 1 and 2, 2 succeeds 1, 5 succeeds 4, 8 succeeds 6 and 7 and 7 succeeds 6 by itemset extension.
- An item j is said to succeed an item k by sequence extension in a sequence S if j belongs to the itemset $i_1$ in S and k belongs to the itemset $i_2$ in S such that $i_2$ appears after $i_1$ in S. For example in the sequence <(1,2,3)(4,5)(6,7,8)>, 4,5,6,7 and 8 succeed 1,2 and 3 and 6,7 and 8 succeed 4 and 5 by a sequence extension.
- The Co-occurrence List of an item j that is succeeded by an itemset extension for a given sequence database is a list of items which succeed j by an itemset extension in at least x number of distinct sequences in the sample database such that x is equal to the minimum support. It is denoted by $CL_i(j)$. For example, considering Table 3.1 to be the sequence database and minimum support as 2, $CL_i(a)$ = {b,d} since b succeeds a by itemset

extension in sequences with ids 1 and 2 and d succeeds a by itemset extension in sequences with ids 2 and 4.

- The Co-occurrence List of an item j that is succeeded by a sequence extension for a given sequence database is a list of items which succeed j by a sequence extension in at least x number of distinct sequences in the sample database such that x is equal to the minimum support. It is denoted by $CL_s(j)$. For example, considering Table 3.1 to be the sequence database and minimum support as 2, $CL_s(a) = \{c,d\}$ since c and d succeed a by sequence extension in sequences with ids 1 and 2.
- The Co-occurrence Map of a sequence database with respect to itemset extension($CMAP_i$) is a table which shows the $CL_i(j)$ for every item j in the sequence database for which $CL_i(j)$ is non empty. For example, the $CMAP_i$ for the sequence database in table 3.1 is given by:

| Item | $CL_i$ |
|---|---|
| a | {b,d} |
| c | {d,g} |
| d | {g} |

*Table 3.3 $CMAP_i$ for sequence database in table 3.1*

- The Co-occurrence Map of a sequence database with respect to sequence extension($CMAP_s$) is a table which shows the $CL_s(j)$ for every item j in the sequence database for which $CL_s(j)$ is non empty. For example, the $CMAP_s$ for the sequence database in table 3.1 is given by:

| Item | $CL_s$ |
|---|---|
| a | {c,d} |
| b | {c,d} |
| c | {d,h} |
| d | {c} |
| g | {h} |

*Table 3.4 $CMAP_s$ for sequence database in table 3.1*

- A sequence of the form <(x,y)> is considered to be frequent if and only if $CMAP_i(x)$ contains y. <(x,y)> is said to be an itemset extensible sequence of length 2. This can be easily proved using the definition of $CMAP_i$. If y is not present in $CL_i(x)$ then it implies that the number of sequences in which y succeeds x by itemset extension is less than minimum support. Since $CMAP_i(x)=CL_i(x)$, if $CMAP_i(x)$ does not contain y then <(x,y)> cannot be a frequent sequence.

- A sequence of the form <(x)(y)> is considered to be frequent if and only if $CMAP_s(x)$ contains y. <(x)(y)> is said to be a sequence extensible sequence of length 2. This can be easily proved using the definition of $CMAP_s$. If y is not present in $CL_s(x)$ then it implies that the number of sequences in which y succeeds x by sequence extension is less than minimum support. Since $CMAP_s(x)=CL_s(x)$, if $CMAP_s(x)$ does not contain y then <(x)(y)> cannot be a frequent sequence.

### 3.2.2 Overview

The algorithm, as illustrated in the diagram 3.2, consists of two MapReduce jobs. The first job involves the creation of $CMAP_i$ and $CMAP_s$. The mappers in this stage are each assigned a part of the input sequence database. Each mapper takes each sequence assigned to it one by one and computes two lists for each item j of every sequence s:

1. A list $L_i(j)$ which contains all items which succeed j by itemset extension in the sequence s.
2. A list $L_s(j)$ which contains all items which succeed j by sequence extension in the sequence s.

Each mapper outputs three types of key-value pairs - $(j,L_i(j))$,$(j,L_s(j))$ and (sequence id, sequence) for each sequence assigned to the mapper. The reducer uses the $L_i$ and $L_s$ generated by various mappers to find $CL_i$ and $CL_s$ respectively for the items present in the sequence database. For example if the number of $L_i(x)$ containing an item y is not less than minimum support then the item y is added to $CL_i(x)$. Similarly for $L_s$ and $CL_s$. The $(j,CL_i(j))$ and $(j,CL_s(j))$ key-value pairs in which CL is non-empty are sent to each of the nodes of the distributed system( also referred to as distributed cache) so that they can be used in the second MapReduce stage. The reducer of this stage also performs another task of emitting (sequence id, (item,index of itemset in sequence)) pairs for all instances of each item in each sequence present in the sequence database. The second job uses the (sequence id,(item,itemset index)) pairs emitted by the first phase to compute SILs for all 1-sequences present in the sequence database. Using these SILs along with the CMAPs, SILs for longer candidate sequences are further generated in this phase and many of the infrequent sequences are pruned prior to support counting. The next few sections describe the algorithm in detail.

### 3.2.3 The first MapReduce job

**Map Phase**

The input to the map phase of the first MapReduce job is a part of the sequence database(referred to as split) and the minimum support. The ith mapper receives the ith split of the sequence database. After processing the inputs, the map phase outputs three different types of key-value pairs, as discussed in the previous section.

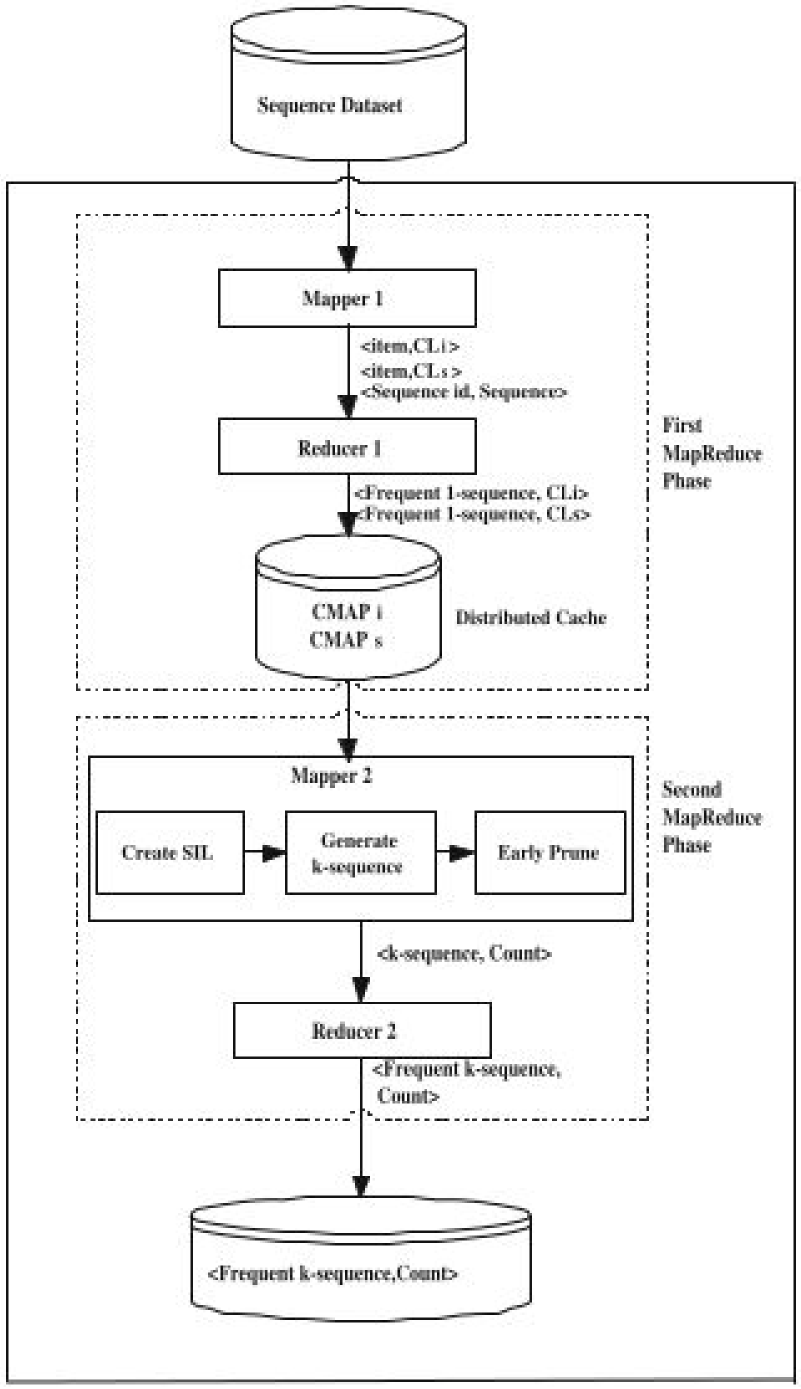


*Diagram 3.2 Overall flow of the DSPC algorithm*

The algorithms used in the map phase of the first MapReduce job of the DSPC algorithm are described as follows:

**Algorithm 3.1: Mapper function**

Input for ith mapper:

$SD_i$ - Sequence Dataset of ith split
min_sup - minimum support count

Output:

(item,$L_s$(item)) pairs
(item,$L_i$(item)) pairs
(sequence id, sequence) for all sequences in $SD_i$

Algorithm:

```
1.  function MAP1(sequence_id Id, sequence S)
2.  {
3.      Li := CREATE_Li(S)
4.      Ls := CREATE_Ls(S)
5.      for each key i in Li:
6.          output( i, Li(i) )
7.      end for
8.      for each key i in Ls:
9.          output( i, Ls(i) )
10.     end for
11.     output( Id, S )
12. }
```

The mapper function receives a sequence id and the respective sequence corresponding to it as input key-value pair. Two maps namely Li and Ls are constructed using the input sequence. Li is a mapping from an item to a list of items which co-occur with that item in the input sequence by itemset extension and Ls is a mapping from an item to a list of items which co-occur with that item in the input sequence by sequence extension. Li is created by the CREATE_Li() function and Ls is created by the CREATE_Ls() function. Once Li and Ls have been created, first all (i,Li(i)) pairs for each item i with a non empty Li(i) and then all (i,Ls(i)) pairs for each item i with a non empty Ls(i) are emitted by the mapper. After this, the mapper function also emits a (sequence id, sequence) pair corresponding to the input sequence.

**Algorithm 3.2: Create $L_i$ function**

Input:

A sequence S

Output:

A mapping $L_i$ from items to lists of items

Algorithm:

```
1.  function CREATE_Li(sequence S)
2.  {
3.      Li := new map which maps an item to a list of items
4.      for each itemset IS in S:
5.          for i=0 to size of IS:
6.              for j=i+1 to size of IS:
7.                  if Li[IS[i]] has not been initialized:
```

```
8.              initialize Li[IS[i]]
9.           end if
10.          Add IS[j] to Li[IS[i]]
11.       end for
12.    end for
13. end for
14. return Li
15. }
```

The CREATE_Li function takes a sequence as input and for each item in the sequence finds a list of items which succeed that item by itemset extension in that sequence. In order to do that, the function iterates over all the itemsets in the input sequence. For each itemset, the function iterates over all the items and adds all items that precede an item i in the itemset to the list Li[i]. Once all itemsets have been traversed, the list Li is returned.

**Algorithm 3.3: Create $L_s$ function**

Input:

A sequence S

Output:

A mapping $L_s$ from items to lists of items

Algorithm:

```
1.  function CREATE_Ls(sequence S)
2.  {
3.     Ls := new map which maps an item to a list of items
4.     for each itemset IS1 in S:
5.        for i=0 to size of IS1:
6.           for each itemset IS2 in S that comes before IS1:
7.              for j=0 to size of IS2:
8.                 if Ls[IS1[i]] has not been initialized:
9.                    initialize Ls[IS1[i]]
10.                end if
11.                Add IS2[j] to Ls[IS1[i]]
12.             end for
13.          end for
14.       end for
15.    end for
16.    return Ls
17. }
```

The CREATE_Ls function takes a sequence as input and for each item in the sequence finds a list of items which succeed that item by sequence extension in that sequence. In order to do that, the function iterates over all the itemsets in the input sequence. For each itemset IS1, the function iterates over all the itemsets that precede that IS1 in the sequence and adds the elements in these itemsets to the Ls of each item present in IS1. Once all itemsets IS1 have been iterated over, Ls is returned.

**Reduce Phase**

The outputs of all the mappers are the inputs to the reducer. The reducer receives three types of key value pairs as input - (item, list of all $L_i$(item) from all the mappers), (item, list of all $L_s$(item) from all the mappers) and (sequence id, sequence). After processing these inputs, the reducer produces three types of key value pairs as output - (item, $CL_i$(item)), (item,$CL_s$(item)) and (sequence id, pair(item,position) for all the items in the sequence). The $CL_i$s and $CL_s$s are used to construct $CMAP_i$ and $CMAP_s$ respectively which are stored in the distributed cache so that they can be used by all the nodes in the next MapReduce phase. The (sequence id, list of pairs(item,position)) key value pairs serve as the input for the next stage.

**Algorithm 3.4: Reduce function**

```
Input:
        (item, list of all L_s(item) from all the mappers) pairs
        (item, list of all L_i(item) from all the mappers) pairs
        (sequence id, sequence) for all sequences in the sequence database
Output:
        CMAP_i - Co-occurrence Map by itemset extension
        CMAP_s - Co-occurrence Map by sequence extension
        (sequence_id ID, pair(item,position) for all the items in the sequence with id ID) pairs
Algorithm:
    1. function REDUCE1(key k, values v)
    2. {
    3.    if k is a sequence id:
    4.       HANDLE_SEQUENCES(k,v)
    5.    else if k is an item:
    6.       HANDLE_CMAP(k,v)
    7. }
```

The reduce function receives three types of different key value pairs as input. Based on the type of key value pair, the key can either be a sequence id or an item. If it is a sequence id then the key and value are passed on to the HANDLE_SEQUENCES() function which outputs a (sequence id, list of pairs(item,position)) pair corresponding to the input (sequence id, sequence) pair. If the key is an item then the key and value are passed on to the HANDLE_CMAP() function which outputs (item,$CL_i$(item)) or (item,$CL_s$(item)) pair depending on whether the value is a list of $L_i$s or a list of $L_s$s respectively.

**Algorithm 3.5: Handle Sequences function**

```
Input:
        (sequence id, sequence) pair
Output:
        (sequence id, list of pairs(item,position))
```

Algorithm:

```
1.  function HANDLE_SEQUENCES(sequence_id ID, sequence S)
2.  {
3.      i := 1
4.      L := new list which stores pairs of items and their indices
5.      for each itemset IS in v:
6.          for each item j in IS:
7.              insert pair(j,i) in L
8.          end for
9.          i := i+1
10.     end for
11.     output(k,L)
12. }
```

The HANDLE_SEQUENCES() function takes a sequence and its sequence id as inputs and outputs a list of item-position pairs for each item in each itemset of the sequence along with the sequence id of the sequence. Position of an item refers to the index of the itemset in the sequence to which the item belongs. L is a list which can store (item,position) pairs. After L is initialized, the function iterates over all the itemsets present in the input sequence. For each itemset, the function iterates over each item and adds an (item,position) pair corresponding to that item to L. After the function has iterated over all the itemsets, the (sequence id,L) is output.

**Algorithm 3.6: Handle CMAP function**

Input:

(item, list of lists of items)

Output:

(item,Co-occurrence list of item succeeded by itemset extension)

(item,Co-occurrence list of item succeeded by sequence extension)

Algorithm:

```
1.  function HANDLE_CMAP(item x, list of lists values)
2.  {
3.      Initialize a map COUNT as a mapping from item to integer
4.      for each list L in values:
5.          for each item i in L:
6.              if i is a key present in COUNT:
7.                  COUNT[i] := COUNT[i]+1
8.              else
9.                  COUNT[i] := 1
10.             end if
11.         end for
12.     end for
13.     for each key k in COUNT:
14.         if COUNT[k]>=min_sup:
```

```
15.            if values is a list of Li:
16.               output(x,k) to CMAPi
17.            else if values is a list of Ls:
18.               output(x,k) to CMAPs
19.            end if
20.         end if
21.      end for
22.  }
```

The HANDLE_CMAP() function takes an item i and a list of lists of items as inputs. The second input consists of lists of items which co-occur with the item i in various different sequences. Thus using these lists, we can find out the number of times different items co-occur with item i in different sequences. For an item x, if this count comes out to be greater than minimum support, then x is added to $CMAP_i(i)$ if the input lists of items co-occur with i by itemset extension, otherwise if the input lists of items co-occur with i by sequence extension then x is added to $CMAP_s(i)$. In order to count the number of times each item occurs in the list of lists, a mapping from item to integer called COUNT is used. The function iterates over all the items in all the lists of items present in the input and whenever an item j is encountered, COUNT[j] is assigned the value 1 if j is not already present in COUNT, otherwise COUNT[j] is incremented. Once all the items in all the lists have been visited, the function iterates over all the keys in COUNT. If the count corresponding to an item is greater than minimum support, then the item is added to either $CMAP_i(i)$ or $CMAP_s(i)$ depending on whether the input consists of a list of $L_i$s or a list of $L_s$s respectively. Once $CMAP_i$ and $CMAP_s$ have been completely constructed, they are sent to the distributed cache.

### 3.2.4 The second MapReduce job

**Map Phase**

The map phase of the second MapReduce job takes the output of the first MapReduce job as input. It uses the (sequence ID, list of (item,position)) pairs and the $CMAP_i$ and $CMAP_s$ present in the distributed cache as input to prune out infrequent candidate sequences before beginning support counting and then to find the support counts of the remaining candidate sequences. The Map phase emits (sequence,support count) pairs as output.

**Algorithm 3.7: Mapper function**

Input:

ith split of (sequence_id ID, list of pair(item,position) for all the items in the sequence with id ID) pairs data from output of the reducer of the previous MapReduce phase

Output:

(sequence, support count of sequence for the current split)

Algorithm:

```
1.  function MAP2(sequence_id ID,list of pair(item,position) L)
2.  {
3.     Initialize ItemPosMapping as a mapping from item to list of positions
4.     for each pair p in L:
5.        Add p.position to ItemPosMapping[p.item]
6.     end for
7.     Initialize SIL as a mapping from sequence to a mapping from sequence ID to a list of positions
8.     SIL := CREATE_SIL(ID,ItemPosMapping)
9.     for each key S in SIL which is a 1-sequence:
10.       GENERATE_SEQUENCES(S,SIL)
11.    end for
12. }
```

The mapper function receives a split of the output of the first MapReduce job. Thus the input to a mapper function consists of a sequence ID and a list of (item,position) pairs corresponding to it. The function first creates a mapping called ItemPosMapping from items to list of positions. After this, the function iterates over each (item,position) pair in the input list and then adds each position to the list of positions which belongs to its corresponding item in the ItemPosMapping map. Using ItemPosMapping and the input sequence id, an SIL is created for all the 1-sequences present in the sequence with the input id using the CREATE_SIL() function. After this, the GENERATE_SEQUENCES() function is called using all the 1-sequences found, in order to generate candidate sequences and to populate the SIL. The function calls to GENERATE_SEQUENCES() also produce the output of the map phase.

**Algorithm 3.8: Function for creating SIL of 1-sequences**

Input:

Sequence id ID

ItemPosMapping, a mapping from item to list of positions of that item in the sequence with given ID

Output:

SIL of all 1-sequences contained in the sequence with the given ID

Algorithm:

```
1. function CREATE_SIL(sequence id ID, ItemPosMapping IPM)
2. {
3.     Initialize SIL as a mapping from a sequence to a mapping from a sequence ID to a list of
       positions
4.     for each item i which is a key in IPM:
5.        Sequence S := <(i)>
6.        initialize SIL[S]
7.        initialize SIL[S][ID]
8.        SIL[S][ID] := IPM[i]
9.     end for
10.    return SIL
11. }
```

The CREATE_SIL() function call takes a sequence ID and a mapping from items to list of positions called ItemPosMapping as input and returns the SIL of all 1-sequences present in the sequence with the given ID. The function first initializes SIL which is a mapping from a sequence to a mapping from a sequence ID to a list of positions. The ItemPosMapping IPM uses an item as key and the corresponding value is a list of positions at which that item is found in the sequence with the given ID. The function iterates over all the keys of ItemPosMapping and for each key k, it creates a sequence S = <(k)>. Each S is inserted in the SIL as a key and after that, the input ID is inserted as a key into SIL[S]. SIL[S][ID] is then assigned the list of positions in the sequence with the given ID which contain S. In other words, the value of IPM[k] is assigned to SIL[S][ID]. Once all the items in IPM have been iterated over, the SIL is returned.

**Algorithm 3.9: Function for candidate generation**

```
Input:
        CMAP_i and CMAP_s from distributed cache
        Sequence S
        SIL
Output:
        (sequence, support count) pairs for new candidate sequences
Algorithm:
  1. function GENERATE_SEQUENCES(Sequence S, SIL)
  2. {
  3.    lastitem := last item of sequence S
  4.
  5.    if lastitem is a key in CMAP_s:
  6.       for each item i in CMAP_s(lastitem):
  7.          if <(i)> is not a key in SIL:
  8.             continue
  9.          end if
 10.          newSeq := sequence obtained by adding <(lastitem)> at the end of S
 11.          if EARLY_PRUNE(newSeq) is false:
 12.             Initialize SIL[newSeq]
 13.             SIL[newSeq] := CREATE_SIL_SEQUENCE(false,SIL[S],SIL[<(i)>])
 14.             support_count := number of rows in SIL[newSeq]
 15.             if support_count>=1:
 16.                output(newSeq,support_count)
 17.                GENERATE_SEQUENCES(newSeq,SIL)
 18.             end if
 19.          end if
 20.       end for
 21.    end if
 22.
 23.    if lastitem is a key in CMAP_i:
 24.       for each item i in CMAP_i(lastitem):
```

```
            if <(i)> is not a key in SIL:
                continue
            end if
            newSeq := sequence obtained by adding lastitem at the end of the last itemset of S
            if EARLY_PRUNE(newSeq) is false:
                Initialize SIL[newSeq]
                SIL[newSeq] := CREATE_SIL_SEQUENCE(true,SIL[S],SIL[<(i)>])
                support_count := number of rows in SIL[newSeq]
                if support_count>=1:
                    output(newSeq,support_count)
                    GENERATE_SEQUENCES(newSeq,SIL)
                end if
            end if
        end for
    end if
}
```

The GENERATE_SEQUENCES() function takes a sequence S and the SIL as input and extends the sequence S to generate different candidate sequences from it. The function outputs those sequences which have a possibility of being frequent along with their support. The function first finds the last item in the input sequence and stores it in a variable names lastitem:

- First the function tries to extend the input sequence by sequence extension: If lastitem is a key in $CMAP_s$ then that means there are items which succeed lastitem by sequence extension in a number of sequences in the sequence database, which is greater than minimum support. Thus if lastitem is a key in $CMAP_s$ then the function iterates over the items in the list $CMAP_s$(lastitem). For each item i present in $CMAP_s$(lastitem), if <(i)> is not a key in the SIL then the function moves on to the next item in the list. This is because if an item is not present in the SIL, then none of the sequences explored so far contain that item. Hence any sequence containing that item cannot be frequent. Thus the function can only extend the sequence further if <(i)> is present in the SIL. The new sequence newSeq is generated from the input sequence S by adding the itemset (i) at the end of S. After this, newSeq is passed to the EARLY_PRUNE() function. If EARLY_PRUNE() returns true then this implies that newSeq is infrequent and thus it is discarded, after which the next item in $CMAP_s$(lastitem) is examined. If EARLY_PRUNE() returns false, the SIL of newSeq is computed. SIL computation is performed by passing the SIL of S, the SIL of <(i)> and a boolean value itemEx(which is passed as false when sequence extension is performed) to the CREATE_SIL_SEQUENCE() function. This function returns the SIL of newSeq which is added to the main SIL data structure. The number of rows in the SIL of newSeq which in other words means the number of sequences in which newSeq is present, gives the support of newSeq. If the support of newSeq is non zero, then newSeq is output along with its support count and GENERATE_SEQUENCES() is called for newSeq in order to further mine longer frequent sequences which are formed by further extending newSeq. Thus candidate generation takes place in a depth first manner. Once all the items in

$CMAP_s$(lastitem) have been iterated over, the function moves further on to extend the sequence by itemset extension.

- Extending the input sequence by itemset extension: If lastitem is a key in $CMAP_i$ then that means there are items which succeed lastitem by itemset extension in a number of sequences in the sequence database, which is greater than minimum support. Thus if lastitem is a key in $CMAP_i$ then the function iterates over the items in the list $CMAP_i$(lastitem). For each item i present in $CMAP_i$(lastitem), if <(i)> is not a key in the SIL then the function moves on to the next item in the list. The new sequence newSeq is generated from the input sequence S by adding i at the end in the last itemset of S. After this, newSeq is passed to the EARLY_PRUNE() function. If EARLY_PRUNE() returns true then this implies that newSeq is infrequent and thus it is discarded, after which the next item in $CMAP_i$(lastitem) is examined. If EARLY_PRUNE() returns false, the SIL of newSeq is computed. SIL computation is performed by passing the SIL of S, the SIL of <(i)> and a boolean value itemEx(which is passed as true when itemset extension is performed) to the CREATE_SIL_SEQUENCE() function. This function returns the SIL of newSeq which is added to the main SIL data structure. The number of rows in the SIL of newSeq gives the support of newSeq. If the support of newSeq is non zero, then newSeq is output along with its support count and GENERATE_SEQUENCES() is called for newSeq in order to mine longer frequent sequences which are formed by further extending newSeq. Once all the items in $CMAP_i$(lastitem) have been iterated over, the execution of the function gets completed.

**Algorithm 3.10: Function for checking if a sequence is infrequent**

Input:

$CMAP_i$ and $CMAP_s$ from distributed cache
Sequence S to be checked

Output:

true if S is to be pruned, false otherwise

Algorithm:

```
1.  function EARLY_PRUNE(Sequence S)
2.  {
3.     lastitem := last item in S
4.     $IS_n$ := last itemset in S
5.     for every item i in $IS_n$ except the last item:
6.        if i is not a key in $CMAP_i$ or $CMAP_i$(i) doesnt contain lastitem:
7.           return true
8.        end if
9.     end for
10.    if S has more than one itemsets:
11.       $IS_{n-1}$ := second last itemset in S
12.       for every item i in $IS_{n-1}$:
13.          if i is not a key in $CMAP_s$ or $CMAP_s$(i) doesnt contain lastitem:
14.             return true
15.          end if
16.         end for
17.     end if
18.     return false
19. }
```

The EARLY_PRUNE() function takes a sequence S as input and returns true if it is definitely infrequent and false otherwise. This function is called right after a sequence X has been extended by either itemset or sequence extension. X has already returned false for EARLY_PRUNE(). Thus we know that X is frequent and the only difference between X and S is that S has an extra item in the end. Thus the function stores the last item of S in a variable called lastitem. Now S can be frequent only if the following two conditions are satisfied:

1. lastitem is present inside the $CMAP_i$ of all the items preceding lastitem in the last itemset of S: If this condition is not satisfied for some item i, then this implies that the number of sequences in the sequence database in which lastitem succeeds i by itemset extension is less than minimum support. Thus, any sequence in which lastitem succeeds i by itemset extension cannot be frequent.
2. lastitem is present inside the $CMAP_s$ of all the items in the second last itemset of S: If this condition is not satisfied for some item i, then this implies that the number of sequences in the sequence database in which lastitem succeeds i by sequence extension is less than minimum support. Thus, any sequence in which lastitem succeeds i by sequence extension cannot be frequent. Although this check can be performed for all the itemsets in S other than the last itemset, it is just performed for the items in the second last itemset. This is done to reduce the number of operations since multiple itemsets may have repeated items which can lead to redundant checks.

In order to check for the first condition, the algorithm first iterates over all the items present in the last itemset of S, excluding the last item. For each item i, a check is made to see if lastitem is present in $CMAP_i$(i). In case it is not present for any i, S is deemed infrequent and the function returns true. If lastitem is present in the $CMAP_i$ for all i, then the first condition is satisfied and thus the function now checks for the second condition. In order to do so, the algorithm iterates over all the items of the second last itemset of S. For each item i, a check is made to see if lastitem is present in $CMAP_s$(i). In case it is not present for any i, S is deemed infrequent and the function returns true. If lastitem is present in the $CMAP_s$ for all i, then the second condition is also satisfied and so S cannot be early pruned. Thus the function returns false.

**Algorithm 3.11: Function for creating SIL of new sequences**

Input:

itemEx: if true then sequence is extended by itemset extension otherwise by sequence extension

SIL[S]: SIL of sequence S which is to be extended

SIL[item]: SIL of the sequence <(item)>. item is uses to extend S by sequence or itemset extension depending on the value of itemEx to obtain the new sequence

Output:

SIL of the new extended sequence

Algorithm:

```
1. function CREATE_SIL_SEQUENCE(boolean itemEx, SIL[S], SIL[<(item)>])
```

```
{
    Initialize SILnewSeq which is a mapping from sequence ID to list of positions
    for every sequence id ID1 in SIL[S]:
        for every sequence id ID2 in SIL[item]:
            if ID1 is equal to ID2:
                Initialize posList as a list of positions
                if itemEx is true:
                    for each position x in SIL[S][ID1]:
                        for each position y in SIL[item][ID1]:
                            if x is equal to y:
                                insert x in posList
                                break
                            end if
                        end for
                    end for
                else
                    for each position x in SIL[S][ID1]:
                        for each position y in SIL[item][ID1]:
                            if y>x:
                                insert y in posList
                                break
                            end if
                        end for
                    end for
                end if
                if posList is not empty:
                    Initialize SILnewSeq[ID1]
                    SILnewSeq[ID1] := posList
                end if
                break
            end if
        end for
    end for
    return SILnewSeq
}
```

The CREATE_SIL_SEQUENCE() function takes the SIL of a sequence S, the SIL of a sequence <(i)> (where i is an item) and a boolean value itemEx as inputs and returns the SIL of the extended sequence. If itemEx is true then the extended sequence newSeq is formed by the itemset extension of S, otherwise it is formed by the sequence extension of S. S is extended by adding the item i at the end of S. The function finds out all the sequence ids that are keys in both SIL[S] and SIL[<(i)>]. For each of these ids ID, the function initializes a list of positions called posList. There can be one of the two cases from here on:

1. itemEx is true: In this case the function finds out all the common positions in SIL[S][ID] and SIL[<(i)>][ID] and inserts them in posList. This is done because if a position x is present in both SIL[S][ID] as well as in SIL[<(i)>][ID] then it implies that both S as well as i are present in xth itemset of the sequence with sequence id equal to ID. In other words, newSeq is present in the xth itemset of the sequence with sequence id equal to ID. Thus x is to be included in SIL[newSeq][ID]. Once all common positions are found and inserted in posList, posList is assigned to SILnewSeq[ID].
2. itemEx is false: In this case the function finds out all the positions in SIL[<(i)>][ID] which precede any position in SIL[S][ID] and inserts them in posList. This is done because if a position x present in SIL[<(i)>][ID] is smaller than a position y in SIL[S][ID], then it implies that i succeeds S by sequence extension in the sequence with id equal to ID. What this means is that newSeq is a subsequence of that sequence and it is positioned at y( that's where the last item of newSeq is located in the sequence with the given ID). In other words, newSeq has the index y in the sequence with sequence id equal to ID. Thus y is to be included in SIL[newSeq][ID]. Once all y have been found and included in posList, poslist is assigned to SILnewSeq[ID].

Once all the common IDs in SIL[S] and SIL[<(i)>] have been visited, SILnewSeq is returned.

## Reduce Phase

The reduce phase of the second MapReduce job receives the outputs of all the mappers as input. The reducer receives (sequence, list of support counts of the sequence) pairs as input and it emits the frequent sequences along with their support as output.

**Algorithm 3.12: Reducer function**

Input:
    (Sequence,list of support counts from different mappers) for different sequences
    min_sup: minimum support
Output:
    Frequent sequences and their support count
Algorithm:

```
1. function REDUCE2(Sequence S, list L of support counts)
2. {
3.    total_count := 0
4.    for each x in L:
5.       total_count := total_count + x
6.    end for
7.    if total_count>=min_sup:
8.       output(S,total_count)
9.    end if
10. }
```

The reduce function receives (Sequence,list of support counts) pairs as input. For each sequence, the function adds all the supports in the corresponding list of supports to find the total support count of the sequence. If this total support count is not less than the minimum support count, then the sequence is frequent and hence it is output along with its total support count. This concludes the mining task.

## 3.3 Advantages and Disadvantages

Some of the advantages of the DSPC algorithm over other parallel sequential pattern mining algorithms such as SPAM and DPSP are as follows:

1. Since the number of rows in the sequence index list of a sequence S gives the support count of S, once SIL of S has been obtained, support counting is very fast and easy. Finding SILs of new sequences from SIL's from old sequences can be done through simple operations and hence the overall process of support counting is fast. Moreover, as the length of the sequence increases, the size of SIL decreases.
2. CMAP computation is a fast process even when minimum support is low and the early pruning strategy which uses CMAP, significantly reduces the time taken to mine sequential patterns by removing the infrequent sequences before their SILs are computed.
3. DSPC runs exactly two MapReduce jobs unlike most algorithms which use an iterative MapReduce approach. This leads to lesser execution time, scheduling and communication overhead.
4. Since DSPC uses a non-iterative MapReduce approach, it scales very well when the size of the input sequence database is increased. Another reason for good scalability is that DSPC only reads the input sequence database once after which further computations are performed only using CMAP and SILs.
5. As DSPC does not suffer from much communication and scheduling overhead, increasing the number of nodes increases the performance more as compared to other iterative MapReduce approaches. In other words, DSPC has a higher speedup.
6. DSPC is less memory intensive compared to other algorithms since CMAPs don't consume a lot of memory. Moreover as the minimum support is increased, the size of CMAP decreases. On account of early pruning in DSPC, several infrequent sequences are removed prior to creating their SILs with the help of CMAP. This also saves a lot of memory, hence leading to lower consumption.

Some of the disadvantages of DSPC are as follows:

1. Since DSPC is a depth first algorithm, if the minimum support is low and or or the computing nodes do not have sufficient memory to hold the sequences, a memory overflow might take place.
2. Computation of SIL for a new sequence takes O(mn) time where m and n are the number of rows in the SILs of the sequences which are being combined to create the new sequence. If the minimum support is small then mn can be a large quantity.
3. If the frequent sequences are very long then a memory overflow might take place.
4. For smaller values of minimum support, the size of CMAP is much larger. In an extreme case if nodes have less storage then sending the CMAP to each node might not be feasible.

However these disadvantages are faced less with DSPC than with other algorithms. Hence it is still more favourable to use DSPC compared to the rest of the approaches.

# 4. Implementation

The implementation of the DSPC algorithm has been done using Java 1.8.0_221 and can be used with Hadoop version 3.2.1. The programme was developed in Intellij Idea IDE with the use of Maven and executed on a virtual 6 node cluster created using docker containers, on a single Machine with 8 GB RAM and operating system Ubuntu 18.04.

The code takes the sequence database input in the form of a file uploaded on the HDFS cluster. Each line of the file consists of a sequence. Two adjacent itemsets in a sequence are separated by the delimiter "-1" and two adjacent items in an itemset are separated by a single space. The items can be any string not containing spaces,"-1" and "-2". Each sequence ends with "-2". The frequent sequences output by the algorithm are also represented in the same format. The code creates two folders - first_phase_output and second_phase_output for storing the outputs of the first and second MapReduce jobs respectively. The first_phase_output file consists of two files" CMAP-r-00000 which stores CMAP$_i$ and CMAP$_s$ and freqItemPos which consists of the (sequence id, (item,index of itemset in sequence)) pairs for all the sequences in the sequence database. The second_phase_output directory consists of a file called part-r-00000 which stored the frequent sequences mined by the algorithm.

## 4.1 Maven dependencies used

```
<dependencies>
 <dependency>
   <groupId>junit</groupId>
   <artifactId>junit</artifactId>
   <version>4.11</version>
   <scope>test</scope>
 </dependency>
 <dependency>
   <groupId>org.apache.hadoop</groupId>
   <artifactId>hadoop-client</artifactId>
   <version>3.2.1</version>
 </dependency>
</dependencies>
```

## 4.2 Code for the MapReduce algorithm

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.URI;
import java.util.*;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.conf.*;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapreduce.*;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.mapreduce.lib.output.MultipleOutputs;
import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;

public class DSPC {

  //minimum support count
  private final static int min_sup = 100;

  //First MapReduce Phase begins here

  //First Map Phase
  public static class MapPhase1 extends Mapper<LongWritable, Text, Text, Text> {

    //First Map phase helper functions

    //Algorithm 3.2: function to create Li
    private HashMap<String,Set<String>> createLi(String value)
    {
      HashMap<String,Set<String>> Li = new HashMap<>();
      String[] tokenizer = value.split("-1");
      String itemsetTokenizer;
      int tokenNum=0;
      while (tokenNum<tokenizer.length) {
        itemsetTokenizer = tokenizer[tokenNum];
        String[] itemset = itemsetTokenizer.split(" ");
        for(int i=0;i<itemset.length;i++)
        {
          String itemi = itemset[i];
          if(itemi.isEmpty())
          {
            continue;
          }
          if(itemi.equals("-2"))
          {
            break;
          }
          for(int j=i+1;j<itemset.length;j++)
          {
            String itemj = itemset[j];
            if(itemj.isEmpty())
            {
              continue;
            }
            if(itemj.equals("-2"))
            {
              break;
            }
```

```
                Set<String> CLitemi;
                if(Li.containsKey(itemi))
                {
                    CLitemi = Li.get(itemi);
                    CLitemi.add(itemj);
                    Li.replace(itemi,CLitemi);
                }
                else
                {
                    CLitemi = new HashSet<String>();
                    CLitemi.add(itemj);
                    Li.put(itemi,CLitemi);
                }
            }
        }
        tokenNum++;
    }
    return Li;
}

//Algorithm 3.3: function to create Ls
private HashMap<String,Set<String>> createLs(String value)
{
    HashMap<String,Set<String>> Ls = new HashMap<>();
    String[] tokenizer = value.split("-1");
    String itemsetTokenizer;
    List<String[]> prevItemSets = new ArrayList<>();
    int tokenNum=0;
    while (tokenNum<tokenizer.length) {
        itemsetTokenizer = tokenizer[tokenNum];
        String[] itemset = itemsetTokenizer.split(" ");
        for(int k=0;k<itemset.length;k++)
        {
            String itemk = itemset[k];
            if(itemk.isEmpty())
            {
                continue;
            }
            if(itemk.equals("-2"))
            {
                break;
            }
            for(String[] prevItemset: prevItemSets)
            {
                for (int j = 0; j < prevItemset.length; j++)
                {
                    String itemj = prevItemset[j];
                    if(itemj.isEmpty())
                    {
                        continue;
                    }
                    if(itemj.equals("-2"))
                    {
                        break;
                    }
                    Set<String> CLitemj;
                    if (Ls.containsKey(itemj))
                    {
                        CLitemj = Ls.get(itemj);
                        CLitemj.add(itemk);
                        Ls.replace(itemj, CLitemj);
                    }
                    else
                    {
                        CLitemj = new HashSet<>();
                        CLitemj.add(itemk);
                        Ls.put(itemj, CLitemj);
                    }
                }
            }
        }
        prevItemSets.add(itemset);
        tokenNum++;
    }
    return Ls;
}

//Li and Ls context write helper
private void mapContextWrite(int k, HashMap<String,Set<String>> CMAP,Context context) throws IOException,
InterruptedException {
    for(HashMap.Entry<String,Set<String>> entry: CMAP.entrySet())
    {
        String v = "";
        for(String i: entry.getValue())
        {
            v = v+" "+i;
        }
        String kval = entry.getKey();
        String keyString = ((k==-1)?"s":"i")+kval;
        Text val = new Text(v);
        Text key = new Text(keyString);
        context.write(key,val);
    }
}

//Algorithm 3.1: Mapping function for first phase
public void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {
    String line = value.toString();
    HashMap<String,Set<String>> Li = createLi(line);
    HashMap<String,Set<String>> Ls = createLs(line);
```

```
            Text k = new Text("q"+key.toString());
            context.write(k,value);
            mapContextWrite(1,Li,context);
            mapContextWrite(-1,Ls,context);
        }
    }

    //First Reduce phase
    public static class ReducePhase1 extends Reducer<Text, Text, Text, Text> {

        //First Reduce phase helper functions
        private MultipleOutputs output;

        @Override
        public void setup(Context context)
        {
            output = new MultipleOutputs(context);
        }

        @Override
        public void cleanup(Context context) throws IOException, InterruptedException
        {
            output.close();
        }

        //Algorithm 3.6: function to process and output CMAP data
        private void handleCMAP(char type, Text key, Iterable<Text> values, Context context) throws IOException,
InterruptedException {
            HashMap<String,Integer> CMAP = new HashMap<>();
            for (Text val : values) {
                String[] temp = val.toString().split(" ");
                for (int c=0;c<temp.length;c++)
                {
                    if(temp[c].isEmpty())
                    {
                        continue;
                    }
                    String i = temp[c];
                    if(CMAP.containsKey(i))
                    {
                        CMAP.replace(i,CMAP.get(i)+1);
                    }
                    else
                    {
                        CMAP.put(i,1);
                    }
                }
            }
            Text val;
            String fileName = "CMAP";
            key.set(type+key.toString());
            for(HashMap.Entry<String,Integer> entry: CMAP.entrySet())
            {
                String t = entry.getKey();
                val = new Text(t);
                if(entry.getValue()>=min_sup)
                {
                    output.write(fileName,key,val);
                }
            }
        }

        //Algorithm 3.5: function to process and output sequence data
        private void handleSequences(Text key, Iterable<Text> values, Context context) throws IOException, InterruptedException
{
            for(Text val: values)
            {
                Text v;
                String value = "";
                Integer i=1;
                String[] tokenizer = val.toString().split("-1");
                int tokenNum=0;
                while(tokenNum<tokenizer.length)
                {
                    String[] temp = tokenizer[tokenNum].split(" ");
                    for (int j = 0; j < temp.length; j++)
                    {
                        if(temp[j].isEmpty())
                        {
                            continue;
                        }
                        if(temp[j].equals("-2"))
                        {
                            break;
                        }
                        //item+" "+position+" "
                        value+=temp[j]+" "+i+" ";
                    }
                    i++;
                    tokenNum++;
                }
                v = new Text(value);
                output.write("freqItemPos",key,v);
            }
        }

        //Algorithm 3.4: Reduce function for first phase
        public void reduce(Text key, Iterable<Text> values, Context context)
                throws IOException, InterruptedException {
            char type = (char) key.charAt(0);
            String k = key.toString().substring(1);
            switch (type)
            {
              case 's':
              case 'i':
                handleCMAP(type,new Text(k),values,context);
                break;
              case 'q':
                handleSequences(new Text(k),values,context);
            }
        }

}

//Second MapReduce Phase Begins here

//Second Map phase
public static class MapPhase2 extends Mapper<LongWritable, Text, Text, IntWritable>
{
        private HashMap<String,Set<String>> CMAPi;
        private HashMap<String,Set<String>> CMAPs;

        //function to read CMAPi and CMAPs from distributed cache
        @Override
        public void setup(Context context) throws IOException, InterruptedException {
            CMAPi = new HashMap<>();
            CMAPs = new HashMap<>();
            URI[] cacheFiles = context.getCacheFiles();

            for (int i = 0; cacheFiles != null && i < cacheFiles.length; i++)
            {
              try
              {
                String line = "";
                FileSystem fs = FileSystem.get(context.getConfiguration());
                String path = cacheFiles.toString();
                Path getFilePath = new Path(cacheFiles[i].toString());
                String temp[] = path.split("/");
                char type;
                BufferedReader reader = new BufferedReader(new InputStreamReader(fs.open(getFilePath)));
                while ((line = reader.readLine()) != null) {
                  type = line.charAt(0);
                  line = line.substring(1);
                  StringTokenizer tokenizer = new StringTokenizer(line);
                  String item = tokenizer.nextToken();
                  String val = tokenizer.nextToken();
                  Set<String> CL;
                  if (type == 'i')
                  {
                    if (CMAPi.containsKey(item))
```

```
                    {
                        CL = CMAPi.get(item);
                        CL.add(val);
                        CMAPi.replace(item, CL);
                    }
                    else
                    {
                        CL = new HashSet<>();
                        CL.add(val);
                        CMAPi.put(item, CL);
                    }
                }
                else
                {
                    if (CMAPs.containsKey(item))
                    {
                        CL = CMAPs.get(item);
                        CL.add(val);
                        CMAPs.replace(item, CL);
                    }
                    else
                    {
                        CL = new HashSet<>();
                        CL.add(val);
                        CMAPs.put(item, CL);
                    }
                }
            }
        } catch (Exception e) {
            System.out.println("Unable to read the File");
            System.exit(1);
        }
    }
}

//helper functions for second map phase

//Algorithm 3.8: Function to create SIL of 1-sequences
private HashMap<String,HashMap<Integer,Set<Integer>>> createSIL(Integer Sid, HashMap<String,Set<Integer>>
itemPosMapping)
{
    HashMap<String,HashMap<Integer,Set<Integer>>> SIL = new HashMap<>();
    for(HashMap.Entry<String,Set<Integer>> entry: itemPosMapping.entrySet())
    {
        Set<Integer> itemPosList = entry.getValue();
        String item = entry.getKey()+" -1 -2";
        HashMap<Integer,Set<Integer>> data = new HashMap<>();
        data.put(Sid,itemPosList);
        SIL.put(item,data);
    }
```

```
        return SIL;
    }

    //Algorithm 3.10: Function to detect infrequent sequences prior to support counting
    private boolean earlyPrune(String Seq, String lastitem)
    {
        String[] itemsets = Seq.split("-1");
        String[] ik = itemsets[itemsets.length-2].split(" ");
        String[] ik_1 = ((itemsets.length>=3)?itemsets[itemsets.length-3]:"").split(" ");
        for(int i=0;i<ik.length;i++)
        {
            String x = ik[i];
            if(x.isEmpty() || x.equals(lastitem))
            {
                continue;
            }
            if(!CMAPi.containsKey(x) || !CMAPi.get(x).contains(lastitem) )
            {
                return true;
            }
        }
        for(int i=0;i<ik_1.length;i++)
        {
            String y = ik_1[i];
            if(y.isEmpty())
            {
                continue;
            }
            if(!CMAPs.containsKey(y) || !CMAPs.get(y).contains(lastitem))
            {
                return true;
            }
        }
        return false;
    }

    //Algorithm 3.11: Function to create SIL of newly generated sequences
    private HashMap<Integer,Set<Integer>> createSILsequence(boolean itemEx, HashMap<Integer,Set<Integer>>
X,HashMap<Integer,Set<Integer>> Y)
    {
        HashMap<Integer,Set<Integer>> Z = new HashMap<>();
        for(HashMap.Entry<Integer,Set<Integer>> entry: X.entrySet())
        {
            Set<Integer> polistx = entry.getValue();
            Set<Integer> polistz = new HashSet<>();
            for(HashMap.Entry<Integer,Set<Integer>> entry1: Y.entrySet())
            {
                if(entry.getKey()!=entry1.getKey())
                {
                    continue;
                }
                Set<Integer> polisty = entry1.getValue();
                if(itemEx)
                {
                    for(Integer posx: polistx)
                    {
                        for(Integer posy: polisty)
                        {
                            if(posx==posy)
                            {
                                polistz.add(posx);
                                break;
                            }
                        }
                    }
                }
                else
                {
                    for(Integer posx: polistx)
                    {
                        for(Integer posy: polisty)
                        {
                            if(posy>posx)
                            {
                                polistz.add(posy);
                            }
                        }
                    }
                }
            }
            if(polistz.size()>0)
            {
                Z.put(entry.getKey(), polistz);
            }
        }
        return Z;
    }

    //Algorithm 3.9: Function for candidate generation
    private void generateSequences(String Seq, HashMap<String,HashMap<Integer,Set<Integer>>> SIL, Context context)
throws IOException, InterruptedException
    {
        if(Seq.length()<=6)
        {
            return;
        }
        String lastitem = "";
        for(int i=Seq.length()-7;i>=0 && Seq.charAt(i)!=' ';i--)
        {
            lastitem=Seq.charAt(i)+lastitem;
        }
        String newSeq;
        if(CMAPs.containsKey(lastitem))
        {
            for (String itemc : CMAPs.get(lastitem))
            {
                if(!SIL.containsKey(itemc+" -1 -2"))
                {
                    continue;
                }
                newSeq=Seq.substring(0,Seq.length()-2)+itemc+" -1 -2";
                if(!earlyPrune(newSeq,itemc))
                {
                    HashMap<Integer,Set<Integer>> SILnewSeq = createSILsequence(false,SIL.get(Seq),SIL.get(itemc+" -1 -2"));
                    SIL.put(newSeq,SILnewSeq);
                    Integer count = SILnewSeq.size();
                    if(count>=1)
                    {
                        context.write(new Text(newSeq+"\t"),new IntWritable(count));
                        generateSequences(newSeq,SIL,context);
                    }
                }
            }
        }
        if(CMAPi.containsKey(lastitem))
        {
            for(String itemc: CMAPi.get(lastitem))
            {
                if(!SIL.containsKey(itemc+" -1 -2"))
                {
                    continue;
                }
                newSeq=Seq.substring(0,Seq.length()-5)+itemc+" -1 -2";
                if(!earlyPrune(newSeq,itemc))
                {
                    HashMap<Integer,Set<Integer>> SILnewSeq = createSILsequence(true,SIL.get(Seq),SIL.get(itemc+" -1 -2"));
                    SIL.put(newSeq,SILnewSeq);
                    Integer count = SILnewSeq.size();
                    if(count>=1)
                    {
                        context.write(new Text(newSeq+"\t"),new IntWritable(count));
                        generateSequences(newSeq,SIL,context);
                    }
                }
            }
        }
    }

    //Algorithm 3.7: Mapping function for second phase
    public void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException
```

```
    {

        String line = value.toString();
        StringTokenizer tokenizer = new StringTokenizer(line);
        Integer Sid = Integer.parseInt(tokenizer.nextToken());
        HashMap<String,Set<Integer>> itemPosMapping = new HashMap<>();
        while (tokenizer.hasMoreTokens())
        {
            String item = tokenizer.nextToken();
            Integer position = Integer.parseInt(tokenizer.nextToken());
            if(itemPosMapping.containsKey(item))
            {
                Set<Integer> itemsetPosList = itemPosMapping.get(item);
                itemsetPosList.add(position);
                itemPosMapping.replace(item,itemsetPosList);
            }
            else
            {
                Set<Integer> itemsetPosList = new HashSet<>();
                itemsetPosList.add(position);
                itemPosMapping.put(item,itemsetPosList);
            }
        }
        HashMap<String,HashMap<Integer,Set<Integer>>> SIL = createSIL(Sid,itemPosMapping);
        HashMap<String,HashMap<Integer,Set<Integer>>> SILcpy = new HashMap<>();
        for(HashMap.Entry<String,HashMap<Integer,Set<Integer>>> entry: SIL.entrySet())
        {
            SILcpy.put(entry.getKey(),entry.getValue());
        }
        for(HashMap.Entry<String,HashMap<Integer,Set<Integer>>> entry: SILcpy.entrySet())
        {
            String k = entry.getKey();
            generateSequences(k, SIL, context);
        }
    }
}

//Second Reduce phase
public static class ReducePhase2 extends Reducer<Text, IntWritable, Text, IntWritable>
{
    //Algorithm 3.12: Reduce function for second phase
    public void reduce(Text key, Iterable<IntWritable> values, Context context) throws IOException, InterruptedException
    {
        int tot_count = 0;
        for(IntWritable value : values)
        {
            tot_count += value.get();
        }
        if(tot_count>=min_sup)
        {
            context.write(key, new IntWritable(tot_count));
        }
      }
    }

    public static void main(String[] args) throws Exception {
      Configuration conf = new Configuration();

      Job job1 = new Job(conf, "Phase1");

      job1.setJarByClass(DSPC.class);
      job1.setOutputKeyClass(Text.class);
      job1.setOutputValueClass(Text.class);
      job1.setMapperClass(MapPhase1.class);
      job1.setReducerClass(ReducePhase1.class);
      job1.setInputFormatClass(TextInputFormat.class);
      job1.setOutputFormatClass(TextOutputFormat.class);

      FileInputFormat.addInputPath(job1, new Path(args[0]));
      MultipleOutputs.addNamedOutput(job1, "CMAP", TextOutputFormat.class, Text.class, Text.class );
      MultipleOutputs.addNamedOutput(job1, "freqItemPos", TextOutputFormat.class, Text.class, Text.class );
      FileOutputFormat.setOutputPath(job1, new Path(args[1]+"/first_phase_output"));

      job1.waitForCompletion(true);

      Job job2 = new Job(conf,"Phase2");

      //Adding CMAPi and CMAPs to distributed cache
      try {
        job2.addCacheFile(new URI("hdfs://namenode:9820"+args[1]+"/first_phase_output/CMAP-r-00000"));
      }
      catch (Exception e) {
        System.out.println("CMAP File Not Added");
        System.exit(1);
      }

      job2.setJarByClass(DSPC.class);
      job2.setOutputKeyClass(Text.class);
      job2.setOutputValueClass(IntWritable.class);
      job2.setMapperClass(MapPhase2.class);
      job2.setReducerClass(ReducePhase2.class);
      job2.setInputFormatClass(TextInputFormat.class);
      job2.setOutputFormatClass(TextOutputFormat.class);

      FileInputFormat.addInputPath(job2, new Path(args[1]+"/first_phase_output/freqItemPos-r-00000"));
      FileOutputFormat.setOutputPath(job2, new Path(args[1]+"/second_phase_output"));

      job2.waitForCompletion(true);
    }
  }
```

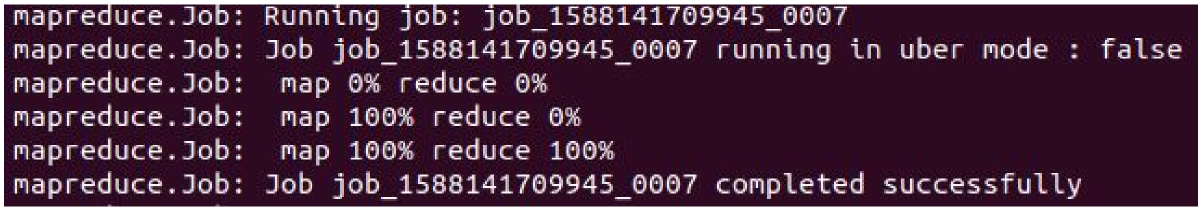
# 5. Observations, Analysis and Proposed Improvements

The DSPC algorithm was tested on a 6 node cluster created using docker containers. The namenode was created on the base system which has a RAM of 8GB and Ubuntu 18.04. The datanodes were implemented using docker containers running Ubuntu operating systems. Since all the containers were running on the same machine, the total memory shared by the cluster was 8GB. On account of limitations on memory and computation capacity, the algorithm couldnt be run completely on large datasets for several smaller values of minimum support.

## 5.1 Observations

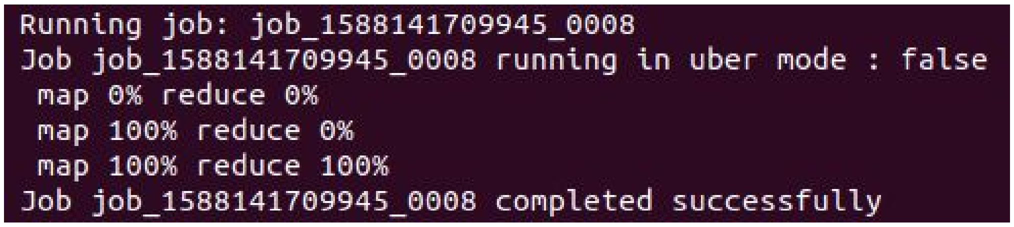
The outputs and observations of the algorithm for some well known data sets are displayed as follows:
(Dataset source: http://www.philippe-fournier-viger.com/spmf/index.php?link=datasets.php)

**BMSWebView1 (Gazelle) (KDD CUP 2000)**
This is an e-commerce dataset containing about 60000 sequences with nearly 350 containing more than 20 items in them. This dataset has an item count count of nearly 500 with an average sequence length of about 3. Both the phases of the algorithm were successfully able to run on the dataset for several values for minimum support values greater than 100:



```
mapreduce.Job: Running job: job_1588141709945_0007
mapreduce.Job: Job job_1588141709945_0007 running in uber mode : false
mapreduce.Job:  map 0% reduce 0%
mapreduce.Job:  map 100% reduce 0%
mapreduce.Job:  map 100% reduce 100%
mapreduce.Job: Job job_1588141709945_0007 completed successfully
```

*First phase*



```
Running job: job_1588141709945_0008
Job job_1588141709945_0008 running in uber mode : false
 map 0% reduce 0%
 map 100% reduce 0%
 map 100% reduce 100%
Job job_1588141709945_0008 completed successfully
```

*Second phase*

For a minimum support of 500, the following frequent patterns were mined:

| | |
|---|---|
| 10295 -1 10307 -1 -2 | 916 |
| 10295 -1 10311 -1 -2 | 738 |
| 10295 -1 10315 -1 -2 | 722 |
| 10307 -1 10311 -1 -2 | 621 |
| 10311 -1 10315 -1 -2 | 771 |

```
10311 -1 12487 -1 -2     615
10311 -1 12703 -1 -2     576
12483 -1 12487 -1 -2     877
12487 -1 12703 -1 -2     631
12487 -1 32213 -1 -2     506
12695 -1 12703 -1 -2     615
12703 -1 32213 -1 -2     571
12815 -1 12895 -1 -2     552
12827 -1 12895 -1 -2     590
33433 -1 33469 -1 -2     509
33449 -1 33469 -1 -2     1204
```

For a minimum support of 300, some sequences of length 3 were also mined along with several sequences of length 2:

```
10295 -1 10307 -1 10311 -1 -2    417
10295 -1 10307 -1 10315 -1 -2    335
10295 -1 10311 -1 10315 -1 -2    351
10311 -1 12483 -1 12487 -1 -2    310
10311 -1 12487 -1 12703 -1 -2    322
12487 -1 12703 -1 32213 -1 -2    315
```

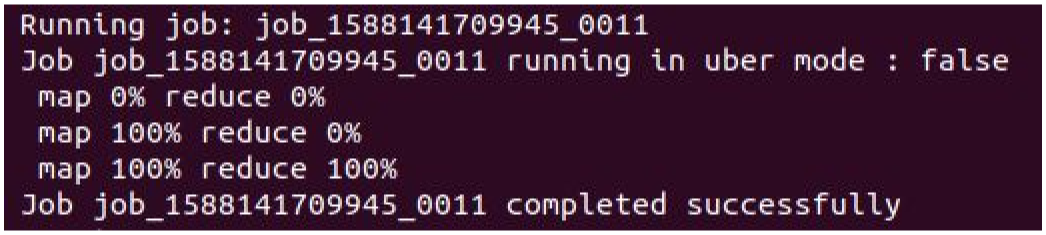
For a minimum support of 200, some sequences of length 3 and 4 were also mined along with several sequences of length 2 and 3:

```
10295 -1 10307 -1 10311 -1 10315 -1 -2            205
10311 -1 12487 -1 12703 -1 32213 -1 -2            200
```

It was observed that the first MapReduce job took almost the same time for all the above values of minimum support, however the time taken was slightly larger for the smaller values. The time taken by the second MapReduce phase increased with the decrease in minimum support. When a very high support value(20000) was used, the CMAPs couldn't be created and hence the second MapReduce phase did not get executed since there were no sequences with that much support count.

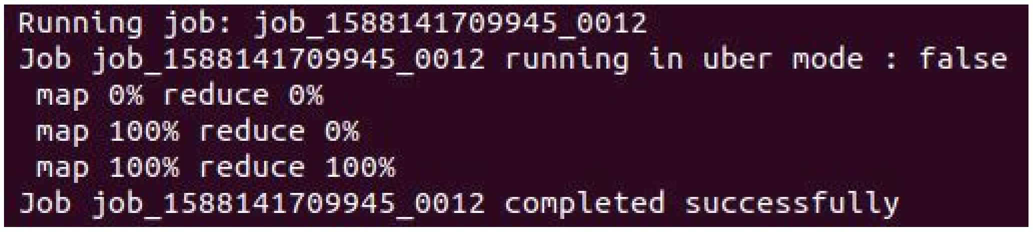
**MSNBC dataset**

This dataset consists of the activities performed by thousands of users on a single day. Each sequence consists of the various sections of msnbc.com visited by a user during the course of the entire day. The MSNBC dataset is available on the UCI ML Repository, however the dataset used here is a subset of the same and consists of 32000 sequences. The dataset has 17 distinct items and the average sequence length is 13. For the given dataset, both phases of the algorithm were successfully able to run for several values of minimum support values greater than 14000:



```
Running job: job_1588141709945_0011
Job job_1588141709945_0011 running in uber mode : false
 map 0% reduce 0%
 map 100% reduce 0%
 map 100% reduce 100%
Job job_1588141709945_0011 completed successfully
```

*First phase*



```
Running job: job_1588141709945_0012
Job job_1588141709945_0012 running in uber mode : false
 map 0% reduce 0%
 map 100% reduce 0%
 map 100% reduce 100%
Job job_1588141709945_0012 completed successfully
```

*Second phase*

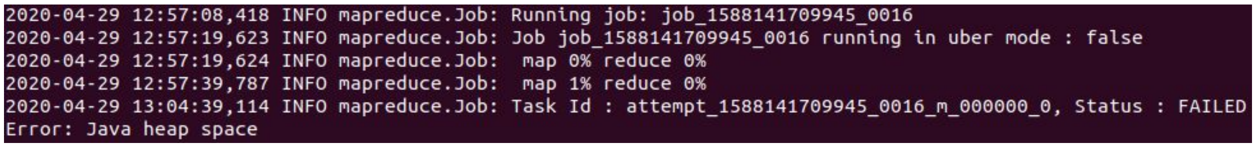
For a minimum support of 20000 only the following pattern was mined:

| | |
|---|---|
| 1 -1 1 -1 -2 | 22556 |

For a minimum support of 15000 the following patterns were mined:

| | |
|---|---|
| 1 -1 1 -1 -2 | 22556 |
| 1 -1 1 -1 1 -1 -2 | 19213 |
| 1 -1 1 -1 1 -1 1 -1 -2 | 15743 |
| 1 -1 2 -1 -2 | 16785 |

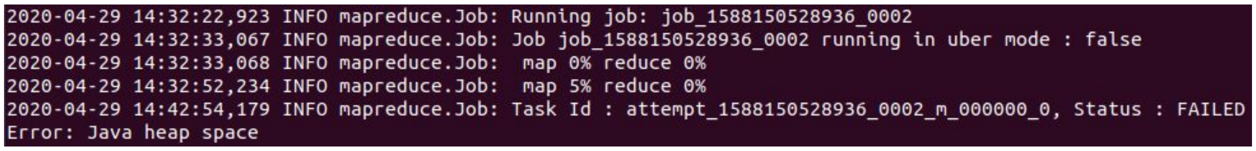
For a minimum support of 13000 the first MapReduce job executed successfully. However the second MapReduce job failed with an error of  overflowed Java heap space:



```
2020-04-29 12:57:08,418 INFO mapreduce.Job: Running job: job_1588141709945_0016
2020-04-29 12:57:19,623 INFO mapreduce.Job: Job job_1588141709945_0016 running in uber mode : false
2020-04-29 12:57:19,624 INFO mapreduce.Job:  map 0% reduce 0%
2020-04-29 12:57:39,787 INFO mapreduce.Job:  map 1% reduce 0%
2020-04-29 13:04:39,114 INFO mapreduce.Job: Task Id : attempt_1588141709945_0016_m_000000_0, Status : FAILED
Error: Java heap space
```

*Second phase for minimum support of 13000*

For a minimum support of 14000 the first MapReduce job executed successfully. However the second MapReduce job failed with an error of  overflowed Java heap space:

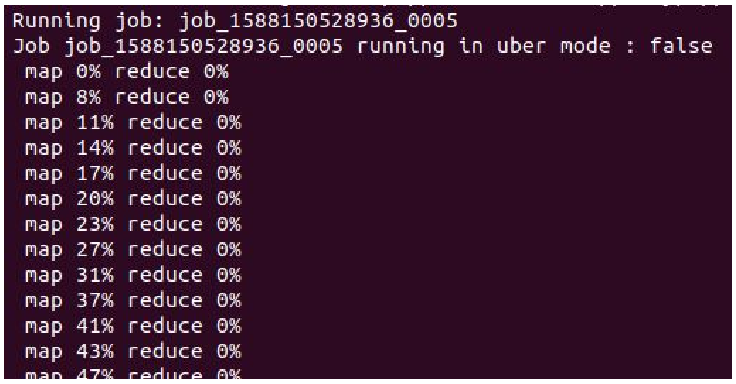


```
2020-04-29 14:32:22,923 INFO mapreduce.Job: Running job: job_1588150528936_0002
2020-04-29 14:32:33,067 INFO mapreduce.Job: Job job_1588150528936_0002 running in uber mode : false
2020-04-29 14:32:33,068 INFO mapreduce.Job:  map 0% reduce 0%
2020-04-29 14:32:52,234 INFO mapreduce.Job:  map 5% reduce 0%
2020-04-29 14:42:54,179 INFO mapreduce.Job: Task Id : attempt_1588150528936_0002_m_000000_0, Status : FAILED
Error: Java heap space
```

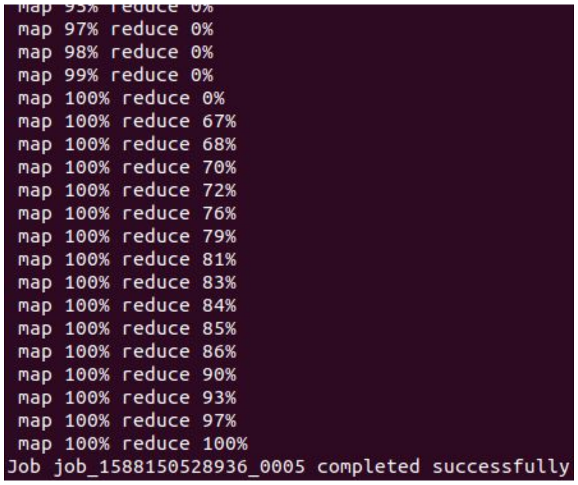
*Second phase for minimum support of 14000*

It was observed that the map phase of the second MapReduce job failed due to insufficient memory for minimum support of 14000 and less. However, the first MapReduce job executed successfully and almost took the same time to execute for all values of minimum support. The time taken as well as memory consumed by the second MapReduce job increased quite a bit with the decrease in minimum support.

**Kosarak dataset**
This is the subset of a dataset of browsing sessions for various users for a Hungarian news portal. This is a much larger dataset than the rest of the datasets used. It consists of 990,000 sequences with 41,270 distinct items and an average sequence length of about 8. Both the phases of the algorithm were successfully able to execute on the dataset for minimum support values greater than 14000:

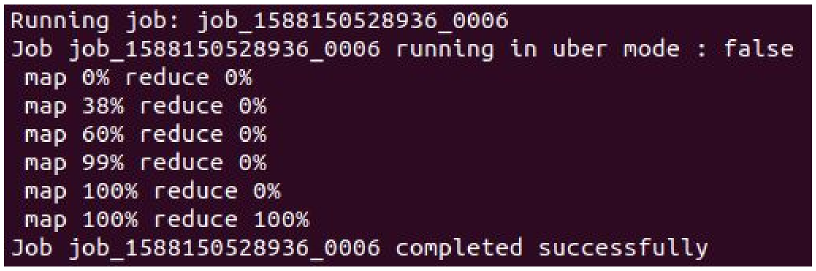


```
Running job: job_1588150528936_0005
Job job_1588150528936_0005 running in uber mode : false
 map 0% reduce 0%
 map 8% reduce 0%
 map 11% reduce 0%
 map 14% reduce 0%
 map 17% reduce 0%
 map 20% reduce 0%
 map 23% reduce 0%
 map 27% reduce 0%
 map 31% reduce 0%
 map 37% reduce 0%
 map 41% reduce 0%
 map 43% reduce 0%
 map 47% reduce 0%
```



```
 map 95% reduce 0%
 map 97% reduce 0%
 map 98% reduce 0%
 map 99% reduce 0%
 map 100% reduce 0%
 map 100% reduce 67%
 map 100% reduce 68%
 map 100% reduce 70%
 map 100% reduce 72%
 map 100% reduce 76%
 map 100% reduce 79%
 map 100% reduce 81%
 map 100% reduce 83%
 map 100% reduce 84%
 map 100% reduce 85%
 map 100% reduce 86%
 map 100% reduce 90%
 map 100% reduce 93%
 map 100% reduce 97%
 map 100% reduce 100%
Job job_1588150528936_0005 completed successfully
```

*First phase output*



```
Running job: job_1588150528936_0006
Job job_1588150528936_0006 running in uber mode : false
 map 0% reduce 0%
 map 38% reduce 0%
 map 60% reduce 0%
 map 99% reduce 0%
 map 100% reduce 0%
 map 100% reduce 100%
Job job_1588150528936_0006 completed successfully
```

*Second phase output*

For a minimum support of 40000 the following patterns were mined:

```
1 -1 3 -1 -2        84660
1 -1 6 -1 -2        132113
1 -1 6 -1 3 -1 -2           57802
11 -1 1 -1 -2       91882
11 -1 1 -1 3 -1 -2          40268
11 -1 1 -1 6 -1 -2          86092
11 -1 148 -1 -2             55759
11 -1 218 -1 -2             61656
11 -1 218 -1 148 -1 -2      50098
11 -1 218 -1 6 -1 -2        60630
11 -1 218 -1 6 -1 148 -1 -2         49866
11 -1 27 -1 -2      46103
11 -1 27 -1 6 -1 -2         44923
11 -1 3 -1 -2       161286
11 -1 6 -1 -2       324013
11 -1 6 -1 148 -1 -2        55230
11 -1 6 -1 3 -1 -2          143682
11 -1 6 -1 7 -1 -2          55835
11 -1 7 -1 -2       57074
218 -1 148 -1 -2            58823
218 -1 6 -1 -2      77675
218 -1 6 -1 148 -1 -2       56838
27 -1 6 -1 -2       59418
27 -1 7 -1 -2       40235
4 -1 6 -1 -2        45377
6 -1 148 -1 -2      64750
6 -1 3 -1 -2        265180
6 -1 7 -1 -2        73610
```

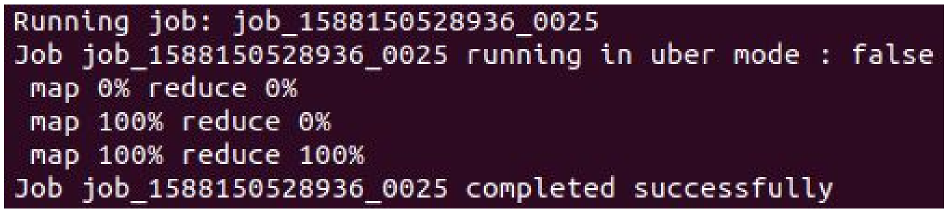
For a minimum support of 14000 the algorithm mined 163 patterns and for a minimum support of 20000 it mined 94 frequent sequences.

It was observed that both the MapReduce jobs took a lot of time to complete execution, however the execution didn't fail due to memory constraints even for smaller values of minimum support. The first MapReduce job took more time to execute compared to the second in this case, which was unlike what was observed for the MSNBC and BMSW datasets. However, just like in the case of those datasets, the first MapReduce job took almost the same time irrespective of the minimum support count whereas the time taken by the second MapReduce phase increased with decrease in minimum support.

**FIFA**

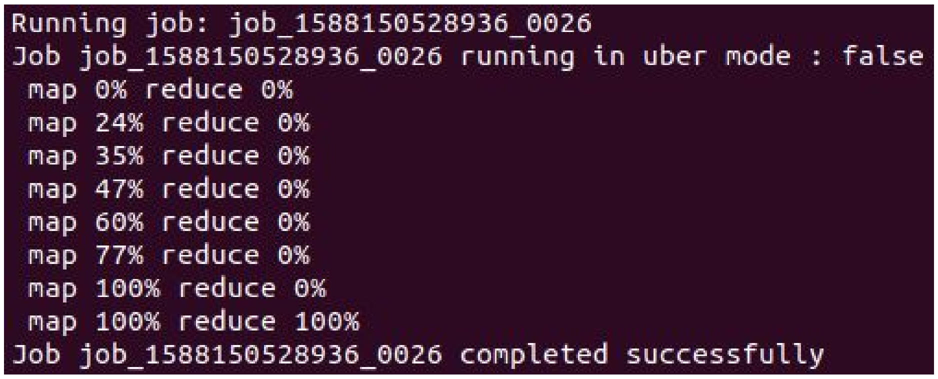
This is a user clicks dataset from FIFA 1998's website. It consists of nearly 20,500 sequences containing around 3000 distinct items. The dataset has an average sequence length of almost 35.

Both the phases of the algorithm were successfully able to execute on the FIFA dataset for minimum support greater than 4500:



```
Running job: job_1588150528936_0025
Job job_1588150528936_0025 running in uber mode : false
 map 0% reduce 0%
 map 100% reduce 0%
 map 100% reduce 100%
Job job_1588150528936_0025 completed successfully
```

*First phase output*



```
Running job: job_1588150528936_0026
Job job_1588150528936_0026 running in uber mode : false
 map 0% reduce 0%
 map 24% reduce 0%
 map 35% reduce 0%
 map 47% reduce 0%
 map 60% reduce 0%
 map 77% reduce 0%
 map 100% reduce 0%
 map 100% reduce 100%
Job job_1588150528936_0026 completed successfully
```

*Second phase output*

For a minimum support of 6000 the algorithm mined 37 patterns, for a minimum support of 5500 it mined 172 frequent sequences, for minimum support of 5000 it mined 256 patterns and for a minimum support of 4500 it mined 517 patterns. For a minimum support of 4000, the first MapReduce phase executed successfully. However, the memory limit exceeded for the second MapReduce job. The time taken for the execution of the first job was almost the same for all values of minimum supports. However the extra time taken to execute the second MapReduce job increased drastically for lower values of minimum support for the same decrease in its value - the difference in the time taken for execution of the second job between the cases when minimum support count was 5000 and when it was 4500 was much higher than that between when the minimum support count was 6000 and when it was 5500.

## 5.2 Analysis

From the observations, the following inferences can be made about the DSPC algorithm:

1. The time and memory consumed for the execution of the first MapReduce job depends very less on the minimum support. Although both the quantities slightly increased for a decrease in the value of minimum support count, it was observed for all the datasets that this increase was very less. The first MapReduce job mainly depended on the number of sequences in the dataset and the number of distinct items. Time taken for execution of the job increased with the increase in the size of the dataset as is seen in the case of the Kosarak dataset. The memory consumed by the first job increased with the number of distinct items in the dataset. This is because $CMAP_s$ and $CMAP_i$ would have a higher

number of keys if the dataset has more number of items. This was observed in the case of the Kosarak dataset for which the first MapReduce job consumed quite a lot of memory whereas in the case of the MSNBC dataset, it consumed very less memory.

2. The time taken to execute the second MapReduce job mainly depended on the minimum support. It was observed in all the datasets that decreasing the minimum support count led to an increase in the time taken for the execution of the job. Moreover the increase in time taken on decreasing the minimum support is much higher for smaller values of minimum support count. This increase is also higher if the dataset consists of long sequences. This is clearly observed for the FIFA dataset which had the highest increase in time consumed for decrease in minimum support, followed by the MSNBC dataset, followed by the Kosarak dataset. The lowest increase was observed for the BMSW dataset. FIFA has the highest average sequence length, followed by MSNBC, followed by Kosarak which is finally followed by BMSW.
3. For smaller values of minimum support, the memory limit often exceeds during the execution of the second MapReduce job. This is especially true for datasets with longer sequences. DSPC follows a depth-first approach for candidate generation. Thus if the dataset consists of long sequences, the number of possible candidate sequences that can arise from each sequence becomes exponentially higher than that in case of a dataset with small sequences for the same minimum support. On account of this, storing the SILs for all the candidate sequences arising from a long sequence becomes a very memory consuming task which may also lead to exceeding the memory limit. Due to this, the memory limit exceeded problem is encountered especially in the case of FIFA and MSNBC for relatively high values of minimum support. This inference is further reinforced by the fact that MSNBC gives memory limit exceeded for values of minimum support less than 10000 whereas that number is just 100 for BMSW, despite the fact that MNSW has 60000 sequences whereas MSNBC has 32000 which is nearly half of that quantity.
4. The DSPC algorithm is highly time and memory efficient. The first MapReduce job of the DSPC algorithm calculated $CMAP_s$ and $CMAP_i$ for large datasets such as Kosarak within the duration of a minute and that too on a virtual cluster hosted on a system with just 8 GB RAM. For all the observations taken, the first MapReduce job executed flawlessly without any memory or time issues. The second MapReduce job worked for all datasets for decent values of the support count and only gave memory limit exceeded error for very small minimum support counts and datasets with long average sequence lengths. Thus if a more powerful cluster is used, the algorithm can perform the sequential pattern mining task in very less time and using very less memory.

## 5.3 Proposed Improvements

Although the DSPC algorithm is already very efficient, some optimizations can make it work much more optimally:

- SIL data structure: One of the improvements would improve performance significantly is in the implementation of the SIL data structure. In order to conceptually explain SIL of a sequence S, sequence ID needs to be incorporated. Conceptually, SIL[S][ID] gives the list of positions at which S appears in the sequence with id equal to ID. However, in the implementation of the map function of the second MapReduce job, it is seen that the inputs to the map function are a sequence id and a list of (item,position) pairs corresponding to the items present in the sequence with the input id. Thus one map function call only deals with one sequence. SILs are created only through calls to a map function. Thus all the SILs created through a map function can only be created using the sequence with the sequence id taken input by the function. A map function emits (sequence,support count) pairs to the reducer. Thus no information about sequence ID reaches the reducer. Hence it is safe to say that incorporating sequence ID in a SIL generated by a map function is redundant.

In order to solve this problem, SILs should be represented as Map<Sequence_ID,Map<Sequence,position_list>> instead of Map<Sequence,Map<Sequence_ID,position_list>>. Now SIL[ID][S] would give the list of positions at which S appears in the sequence with id equal to ID. Now each map function call would only have to deal with populating SIL[ID] for the sequence id of the sequence associated with the function call. This is illustrated in the following example:

*Original SILs:*

for a sequence S1:

| Sequence ID | position list |
|---|---|
| 1 | {3,4} |
| 2 | {1,3} |

for a sequence S2:

| Sequence ID | position list |
|---|---|
| 1 | {1,2} |
| 3 | {3,7} |

*Modified SILs:*

for sequence ID = 1

| Sequence | position list |
|---|---|
| S1 | {3,4} |
| S2 | {1,2} |

for sequence ID = 2

| Sequence | position list |
|---|---|
| S1 | {1,3} |

for sequence ID = 3

| Sequence | position list |
|---|---|
| S2 | {3,7} |

Thus from the above illustration it can be seen that the new implementation of the SIL data structure deals with only one sequence at a time. Thus using the new implementation of SIL, each map function call would be solely responsible for creating the SIL for the respective sequence associated with it. Since sequence id information associated with a map function is not used anywhere, each map function only has to deal with a map of the form Map<Sequence,position_list> which would function as the SIL for that map function. This would reduce one map reference when trying to access values in the SIL which would in turn save time. Memory which would have otherwise been wasted in unnecessarily storing information about sequence id would also be saved.

- CREATE_SIL_SEQUENCE() function: Considering the improved implementation of SIL as discussed in the above point, one important thing to notice is that all position lists in the SILs are always sorted in ascending order. Thus, using this information the CREATE_SIL_SEQUENCE() function can be improved by using a concept similar to the merge algorithm from merge sort in order to create the SIL of the new sequence more efficiently. The existing algorithm for the function has a time complexity of O(mn) where m is the size of SIL[S] and n is the size of SIL[<(item)>] for the given sequence ID. The improved algorithm that utilizes the new implementation of SIL is as follows:
(Node: SIL[S](x) represents x$^{th}$ element in the SIL of S)

```
function CREATE_SIL_SEQUENCE(boolean itemEx, SIL[S], SIL[<(item)>])
{
  Initialize SILnewSeq as a list of positions
  if itemEx is true:
    x := 0, y := 0
    while x<size of SIL[S] and y<size of SIL[<(item)>]:
      if SIL[S](x) = SIL[<(item)>](y):
        insert SIL[S](x) in SILnewSeq
        x := x+1
        y := y+1
      else if SIL[S](x)<SIL[<(item)>](y):
        x := x+1
      else
        y := y+1
      end if
    end while
  else
    y := size of SIL[<(item)>] - 1
    while size of SIL[S]>0 and y>=0:
```

```
20.         if SIL[<(item)>](y)>SIL[S](0):
21.            insert SIL[<(item)>](y) in SILnewSeq
22.         else
23.            break
24.         end if
25.            end while
26.    end if
27.    return SILnewSeq
28. }
```

The while loop from line 6 to 16 uses the concept of merge algorithm to find out the common positions in SIL[S] and SIL[<(item)>]. If the current index x in SIL[S] points to a position smaller than the position in SIL[<(item)>] pointed by y, then x is incremented to find a position that is closer to SIL[<(item)>](y). Otherwise if SIL[S](x) is greater than SIL[<(item)>](y) then y is incremented in order to bring SIL[<(item)>](y) closer to SIL[S](x). If SIL[S](x) is equal to SIL[S](y) then the position represented by SIL[S](x) is also a position at which the sequence formed by performing an itemset extension of S by adding *item* to the last itemset of S occurs. Let this sequence be represented by $S_i$. Thus in other words, if SIL[S](x) is equal to SIL[<(item)>](y) then SIL[S](x) is added to SIL[$S_i$]. After SIL[S](x) is added to SIL[$S_i$], x and y are both incremented. The overall time complexity of this while loop is O(m+n).

Let $S_s$ represent the sequence formed by adding the itemset (item) to the end of S. Now a position i in SIL[<(item)>] can be added to SIL[$S_s$] if i comes after any position in SIL[S]. Since SIL[S] is a sorted list, every position i in SIL[<(item)>] greater than the first position in SIL[S] should appear in SIL[$S_s$]. This is exactly what the while loop from lines 19 to 25 does. The overall time complexity of this while loop is O(n).

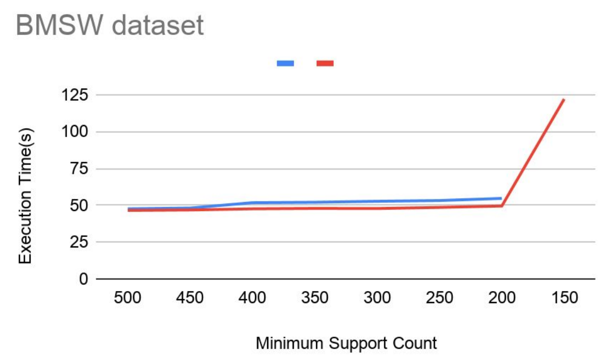
The above implementation of the CREATE_SIL_SEQUENCE() function has a time complexity of O(m+n) as compared to the time complexity of the original implementation which is O(mn). Since this function is called for every candidate sequence, using the improved implementation of the function substantially reduces the amount of time taken to mine frequent sequential patterns, especially for large datasets and datasets with longer sequences. This can be seen from the experimental data:(Each observation mentioned has been recorded five times and then averaged)

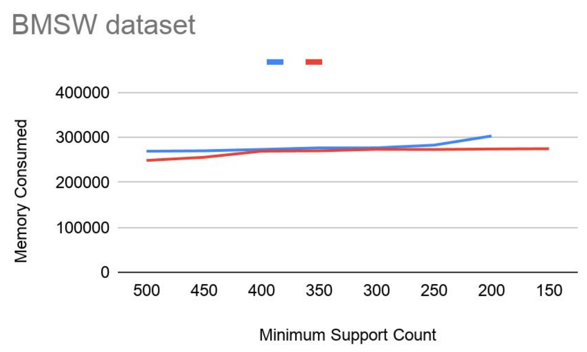
**Observation table:**

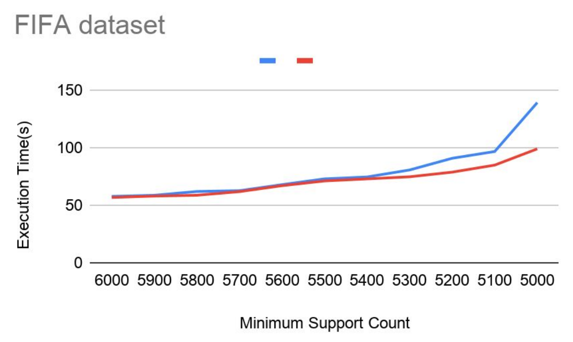
| **Dataset** | **Minimum Support** | **Original Algorithm** | | **Modified SIL and CREATE_SIL()** | |
|---|---|---|---|---|---|
| | | **Time(s)** | **Memory(kb)** | **Time(s)** | **Memory(kb)** |
| BMSW | 500 | 47.67 | 269568 | 46.55 | 249116 |
| | 450 | 48.18 | 270472 | 46.87 | 256120 |
| | 400 | 51.81 | 273564 | 47.65 | 269680 |
| | 350 | 52.1 | 276996 | 47.92 | 270492 |

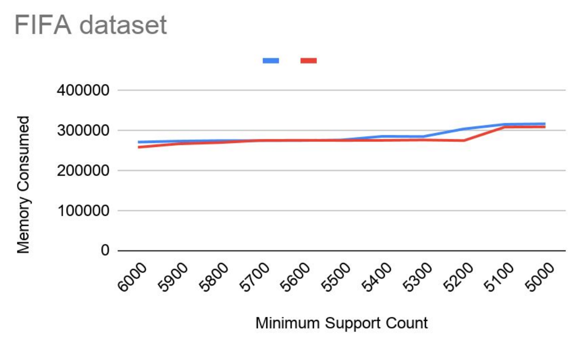
| | | | | | |
|---|---|---|---|---|---|
| | 300 | 52.77 | 277136 | 47.84 | 273864 |
| | 250 | 53.26 | 283120 | 48.62 | 273432 |
| | 200 | 54.72 | 303516 | 49.56 | 274524 |
| | 150 | Error | Error | 122.1 | 275184 |
| FIFA | 6000 | 57.8 | 270280 | 56.82 | 257384 |
| | 5900 | 58.73 | 272664 | 58.03 | 265988 |
| | 5800 | 62.01 | 273916 | 58.75 | 269040 |
| | 5700 | 62.71 | 273824 | 61.87 | 274392 |
| | 5600 | 67.94 | 274192 | 67.05 | 274824 |
| | 5500 | 72.98 | 275528 | 71.16 | 274364 |
| | 5400 | 74.61 | 284652 | 72.95 | 274620 |
| | 5300 | 80.68 | 284020 | 74.76 | 275608 |
| | 5200 | 90.82 | 303140 | 78.79 | 274028 |
| | 5100 | 96.7 | 314563 | 84.9 | 307708 |
| | 5000 | 139.15 | 315778 | 98.99 | 308168 |
| MSNBC | 20000 | 51.33 | 268276 | 46.89 | 255520 |
| | 19000 | 52.37 | 269636 | 47.8 | 269596 |
| | 18000 | 52.68 | 270448 | 48.17 | 269776 |
| | 17000 | 52.45 | 271648 | 48.34 | 272684 |
| | 16000 | 52.98 | 272864 | 47.85 | 272588 |
| | 15000 | 53.52 | 277376 | 49.61 | 273604 |
| | 13500 | Error | Error | 49.84 | 273032 |
| | 13000 | Error | Error | 50.68 | 273116 |
| | 12500 | Error | Error | 50.83 | 273616 |
| | 11500 | Error | Error | 51.55 | 274732 |
| | 10500 | Error | Error | 51.66 | 274668 |
| | 9500 | Error | Error | 52.82 | 275556 |
| | 9000 | Error | Error | 53.74 | 282036 |
| | 8500 | Error | Error | 70.69 | 272240 |
| | 8000 | Error | Error | 78.63 | 282300 |
| | 7500 | Error | Error | 107.68 | 283522 |
| Kosarak | 40000 | 468.21 | 293848 | 463.76 | 273645 |
| | 20000 | 471.84 | 301837 | 469.9 | 279475 |
| | 14000 | 499.35 | 313820 | 479.76 | 293746 |

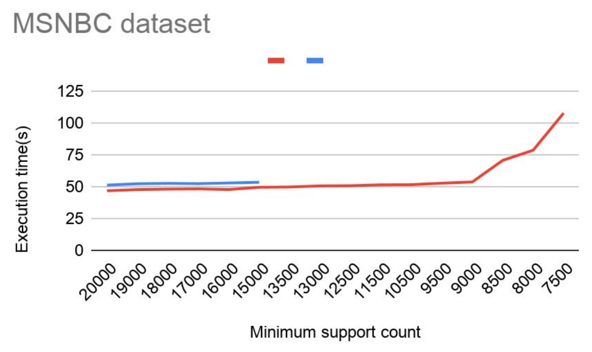
**Graphical plots:**
Blue line represents the original algorithm and the red line represents modified SIL and CREATE_SIL().

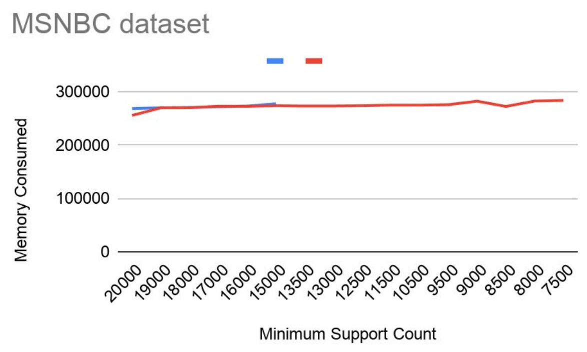


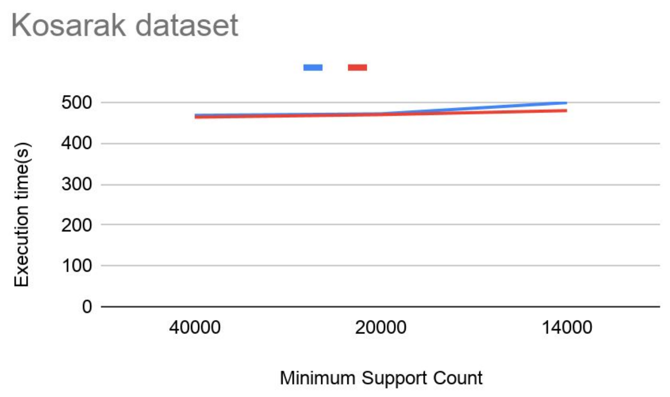


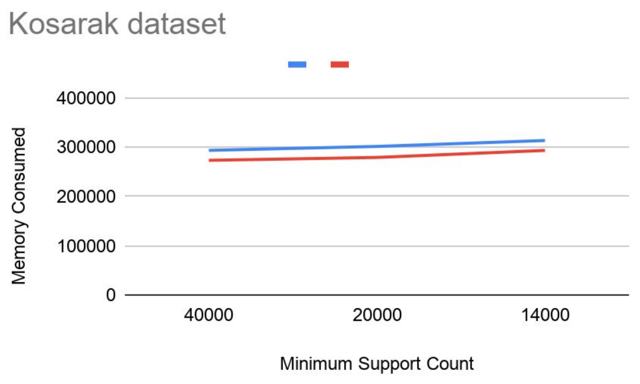












From the observations it is clearly visible that the algorithm performs much better with the proposed modifications. From all the execution time vs minimum support curves, it is observed that with the decrease in minimum support, the difference between execution times of the original algorithm and the improved version keeps increasing. This is expected since the formation of SIL for new sequences in the former takes O(mn) time whereas in the latter it takes O(m+n). This is especially observed in the case of the FIFA dataset which has longer sequences and hence larger values of m and n. Although it is difficult to comment on memory consumption, from the above graphs it is observed that as a general trend, the improved version of the algorithm consumes less memory. The superiority of the improved version in terms of memory consumption can be clearly seen from the observation table wherein the improved version gives an output even for those values of minimum support for which the original algorithm produces an error(memory limit exceeded).

# 6. Challenges Faced

I was faced with several challenges during the course of the entire project. Overcoming them involved learning a lot of new things which was a very educational experience for me. Although I couldn't overcome some of the challenges on account of non-availability of required infrastructure, working within the resources I had and finishing the project proved to be very challenging and fulfilling. Some of the major challenges that I faced are mentioned as follows:

- **Unavailability of a cluster**: Since I did not have a Hadoop cluster available with me, one of the first and also the biggest challenges that I had to overcome was that of getting a Hadoop cluster on which I could execute MapReduce programmes. I just had my laptop and no other machine, so one of the options I had was to use a single node cluster in order to run a programme. Although this would have allowed me to execute the programme, I wouldn't have been able to witness the parallel and distributed nature of the algorithm at work. In order to solve this issue, virtual machines could be used to simulate cluster nodes. However the problem with virtual machines is that they consume a lot of memory for performing a lot of processes that I didn't require. My laptop has a RAM of 8 GB. Thus, using virtual machines I wouldn't have been able to simulate a cluster of more than three nodes. Thus, finally I decided to use docker containers to simulate the cluster. Docker containers consume minimal memory and start and stop quickly. They are clean and make it very easy to create a new cluster and destroy an old one without any mess. However, the only issue with using docker containers was that they didn't have anything preinstalled in them. Thus I needed to create a docker image which had hadoop installed and configured to work as a datanode and tasktracker. Apart from this the image also had to have an ssh server installed and running so that the datanodes can be started remotely from the namenode. The docker images also had to have its hosts file configured so that the namenode and other datanodes can be identified by name. Containers initialized using this docker image had to be able to communicate with each other and the namenode. Achieving all this although challenging, was a very enjoyable and immersive task.

- **Unavailability of implementations of existing algorithms**: I looked up several research papers and several algorithms for performing sequential pattern mining in serial as well as distributed manners, however implementations for the same were not available in the most cases. This especially stands true for the DSPC algorithm. I had to implement the algorithm all on my own, hence the implementation might not be the most efficient one.

- **Mistakes in algorithms written in the original paper**: The paper for the DSPC algorithm has several mistakes in the algorithms such as missing loops, breaks written instead of continue, etc. I got to know about a lot of those mistakes after I implemented the algorithm and got incorrect outputs. Understanding those mistakes and making changes to the code was a very gruelling task. However, it also gave me a very strong conceptual understanding of the DSPC algorithm.

- **Limitations of a single machine**: Although I was able to simulate a 6 node cluster on my laptop, the limitations of my cluster were the same as that of my machine which had a

RAM of 8GB. Execution of data mining tasks on a distributed framework is a computation and memory intensive task. Thus, my cluster was unable to run the DSPC algorithm for large datasets and datasets containing long sequences. My computer stopped working every time I tried to execute the algorithm for large datasets. Thus I had to execute my code for small datasets sampled from the large ones. Moreover it wasn't possible for me to test the speedup of the algorithm since I couldn't change the number of nodes in the cluster as per requirement.

- **Unavailability of a proper source to study MapReduce programming**: All the sources which teach MapReduce programming always use the example of the word count problem to explain things. Moreover the flow of the programme is explained rather than the syntax. Also, the syntax is different for different versions of the same library. So in order to understand MapReduce programming and associated concepts such as distributed cache, I had to understand the word count programme properly and go through documentations which was a time consuming task as well as a great learning experience.

- **Recording time the time taken for the execution of the algorithm**: My system has several processes running on it and it is not possible for me to stop them. On account of this and several other factors, the exact time taken for the execution of the algorithm on a dataset cannot be recorded properly and hence I had to draw conclusions on the behaviour of the algorithm based on approximations and averages.

# 7. Appendix

## 7.1 Setting up Hadoop in your system

The instructions to set up Hadoop in Ubuntu 18.04 can be found at:
https://linuxconfig.org/how-to-install-hadoop-on-ubuntu-18-04-bionic-beaver-linux
Configuration file settings for setting up a namenode or a datanode in a system with Hadoop installed can be found at:
https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-common/ClusterSetup.html

## 7.2 Implementation of test environment

The test environment was implemented using docker containers. Thus a prerequisite of setting up the test environment is that the system should have docker installed in it. The instructions for the installation of docker on Ubuntu can be found at: https://docs.docker.com/engine/install/ubuntu/. Once docker has been installed, a docker image for Ubuntu needs to be downloaded from dockerHub. The docker image along with the command to pull it can be found at: https://hub.docker.com/_/ubuntu/. After this, a docker container has to be created using this image using the docker run command. Once a container is created, some softwares such as vim, java 1.0.8_221, openssh server, etc need to be installed in it. After this, the instructions mentioned in section 7.1 are to be followed in order to install Hadoop in the container and configure it as a datanode. The container is then to be committed so as to create an image for a datanode. After this, whenever a new datanode is required, a new container can be initiated using this datanode image using the docker run command. In order to make sure that the containers in the cluster are able to communicate with each other, when the containers are instantiated using the docker run command, the --dns attribute should be used. For the sake of convenience, startup and shutdown shell scripts can be used for setting up and destroying the cluster in a clean manner:

clusterStartup script:
```
docker run --dns 8.8.8.8 --hostname datanode1 -itd datanode4
docker run --dns 8.8.8.8 --hostname datanode2 -itd datanode4
docker run --dns 8.8.8.8 --hostname datanode3 -itd datanode4
docker run --dns 8.8.8.8 --hostname datanode4 -itd datanode4
docker run --dns 8.8.8.8 --hostname datanode5 -itd datanode4
sudo -u hduser_ echo 'Y' | sudo -u hduser_ hadoop namenode -format
sudo -u hduser_ /hadoop/sbin/start-all.sh
```

clusterShutdown script:
```
sudo -u hduser_ /home/jake/hadoop/sbin/stop-all.sh
docker stop $(docker ps)
```

## 7.3 Executing the job

The instructions for executing a MapReduce job on a Hadoop cluster can be found at:
https://hadoop.apache.org/docs/r1.2.1/mapred_tutorial.html.

# 8. References

- A novel mapreduce algorithm for distributed mining of sequential   patterns using co-occurrence information - Sumalatha Saleti, R. B. V. Subramanyam
- A Survey of Sequential Pattern Mining - Philippe Fournier-Viger, Jerry Chun-Wei Lin, Rage Uday Kiran, Yun Sing Koh, Rincy Thomas
- A Survey of Parallel Sequential Pattern Mining - Wensheng Gan, Jerry Chun-Wei Lin, Philippe Fournier-Viger, Han-Chieh Chao, Philip S. Yu
- Mining Sequential Patterns: Generalizations and Performance Improvements - Ramakrishnan Srikant, Rakesh Agrawal
- Sequential PAttern Mining using A Bitmap Representation - Jay Ayres, Johannes Gehrke, Tomi Yiu, Jason Flannick
- PrefixSpan: Mining Sequential Patterns Efficiently by Prefix-Projected Pattern Growth - Jian Pei, Jiawei Han, Behzad Mortazavi-Asl, Helen Pinto, Qiming Chen, Umeshwar Dayal, Mei-Chun Hsu
- Applications of Pattern Discovery Using Sequential Data Mining - Manish Gupta, Jiawei Han