

FREQUENT SUBGRAPH MINING OVER DYNAMIC GRAPHS

**A Thesis Submitted to
the Graduate School of Engineering and Sciences of
İzmir Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of**

**DOCTOR OF PHILOSOPHY
in Computer Engineering**

**by
Nourhan N. I. ABUZAYED**

**July 2022
İZMİR**

ACKNOWLEDGMENTS

I would like to thank my thesis supervisor, Assoc. Prof. Dr. Belgin ERGENÇ BOSTANOĞLU, for her supervision, and for giving me the opportunity to work with her. I appreciate her sincerity, patience. Her endless support and confidence were the motivation for my courage and enthusiasm. I have learned many things from her.

My gratitude and appreciation to the members of my thesis committee Assoc. Prof. Mustafa ÖZUYSAL and Assist. Prof. Dr. Mutlu BEYAZIT for their constructive feedback, time and effort spent on my thesis.

I am also grateful to IYTE staff, especially Assist. Prof. Dr. Serap ŞAHİN, for her support during my master and PhD study.

I especially thank my lovely husband Dr. Mazen ABUZAYED, for his love, wise, patience, encouragement, and support. Completing this thesis would be very difficult without him.

I would also like to thank my close friend Dr. Arzum KARATAŞ, for her kindness and encouragement to complete this thesis. I am very lucky to have a friend like her.

I would also like to thank my friends Leyla TEKİN and Dr. Rowanda AHMED for their kindness, encouragement and being good friends during my study.

Finally, I would also like to thank the Islamic Development Bank (IDB) for supporting me by Doctoral Scholarship.

ABSTRACT

FREQUENT SUBGRAPH MINING OVER DYNAMIC GRAPHS

Frequent subgraph mining (FSM) is an essential and challenging graph mining task used in several applications. Modern applications employ evolving graphs, so FSM is more challenging with evolving graphs due to the streaming nature of the input, and the exponential time complexity of the algorithms. Sampling schemes are used if approximate results serve the purpose. This thesis introduces three approximate frequent subgraph mining algorithms in evolving graphs. Those algorithms use novel controlled reservoir sampling. A sample reservoir of the evolving graph and an auxiliary heap reservoir data structure are kept together in a fixed sized reservoir. When the whole reservoir is full, and space has required the edges of lower degree or higher nodes are deleted. This selection is done by utilizing the heap data structure as a heap reservoir, which keeps the node degrees. By keeping the edges of higher degree nodes in the sample reservoir, accuracy is maximized without sacrificing time and space, in contrast, keeping the edges of lower degree nodes in the sample reservoir, accuracy is minimized with higher time and space. The first algorithm is Controlled Reservoir Sampling with Unlimited heap size (UCRS), where the used heap reservoir size is unlimited. The second algorithm is Controlled Reservoir Sampling with Limited heap size (LCRS). It is a modified version of UCRS, but the heap reservoir size is limited, as a result; sample reservoir size in the whole reservoir increases since the total number of nodes dedicated for the whole reservoir includes the nodes of the heap reservoir also. The third algorithm is Maximum Controlled Reservoir Sampling (MCRS). It is a modified version of UCRS, but the candidate edge for deletion is an edge with maximum node degrees. Experimental evaluations to measure scalability and recall performances of the three algorithms in comparison to state of art algorithms are performed on dense and sparse evolving graphs. Findings show that UCRS and LCRS algorithms are scalable and achieve better recall than edge based reservoir algorithms. LCRS achieves the best recall in comparison to edge or subgraph based reservoir algorithms. MCRS has the worst speed-up and recall among the other proposed and competitor algorithms.

ÖZET

DEĞİŞKEN VERİ ÜZERİNDE SIK ALT ÇİZGE MADENCİLİĞİ

Sık alt çizgeler madenciliği bir çok veri madenciliği uygulaması için temel ve zorlu bir iştir. Modern uygulamalar devingen çizgelerle çalışmakta olup, girdilerindeki veri akışı, sık alt çizge madenciliği algoritmalarının karmaşıklığını arttırmaktadır. Yaklaşık sonuçların yeterli olduğu durumlarda örnekleme dayalı yaklaşımlar kullanılmaktadır. Bu tez kapsamında üç adet yaklaşık sık alt çizge madenciliği algoritması önerilmektedir. Önerilen algoritmalarda yenilikçi olarak, kontrollü depolama ile örneklem oluşturma yaklaşımı kullanılmıştır. Devingen çizgeye ilişkin örneklem sabit boyutlu depoda tutulmakta ve yardımcı bir yığın veri yapısı kullanılmaktadır. Bu yığın veri yapısında depodaki çizge düğümlerinin bağlantı dereceleri tutulmaktadır. Sabit boyutlu depo dolduğunda ve yeni alan gereksinimi ortaya çıktığında bu depodan çizgenin en düşük dereceli düğümleri çıkarılmaktadır. Devingen çizge örneklemini tutan sabit depoda yüksek dereceli düğümlerin kalması sağlanarak sonuçlardaki doğruluk, yer ve zaman maliyetini artırmadan yükseltebilmektedir. İlk olarak limitsiz boyutlu kontrollü depo örneklemesine dayalı “Controlled Reservoir Sampling with Unlimited heap size (UCRS)” algoritması önerilmiştir; adından da anlaşılacağı üzere kullanılan yardımcı yığının boyutu kısıtlanmamıştır. İkinci algoritma “Controlled Reservoir Sampling with Limited heap size (LCRS)” da büyüyen depo ile büyüyen yığının boyutuna sınır getirilmektedir. Üçüncü algoritma “Maximum Controlled Reservoir Sampling (MCRS)” ilk algoritmaya benzemektedir; yığın boyutu sınırlandırılmamıştır ancak depodan düğüm silmek gerektiğinde en düşük dereceli düğüm yerine en yüksek dereceli düğüm . çıkarılmaktadır. Her üç algoritmanın başarımlarını değerlendirmeleri zaman, ölçeklenebilirlik ve doğruluk ölçütleri ile yapılmıştır. Başarımlarını değerlendirmelerinde önerilen algoritmalar iki güncel rakip algoritma ile yoğun ve seyrek veri setleri üzerinde karşılaştırılmıştır. Bulgular UCRS ve LCRS algoritmalarının ölçeklenebilir olduğunu, rakip kenar tabanlı algoritmadan daha doğru sonuçlar verdiğini göstermiştir. LCRS tüm rakip algoritmalarından daha iyi başarımlar elde etmiştir. MCRS algoritmasının sonuçları tüm algoritmalar arasında en kötüdür.

To my father and the soul of my mother

To my beloved husband

To my sweet children Ali, Mayar, Mohammed, and Mustafa

To the soul of the martyr journalist Sherine Abu Aqleh

TABLE OF CONTENTS

LIST OF FIGURES	viii
LIST OF TABLES.....	xi
LIST OF TERMS AND ABBREVIATIONS.....	xii
CHAPTER 1. INTRODUCTION	1
1.1 Contributions of the Thesis	2
1.2. Organization of the Thesis	4
CHAPTER 2. BACKGROUND AND PROBLEM FORMULATION	5
2.1. Graph Basics	6
2.2. Frequent Subgraph Mining	10
2.3. Graph Sampling	11
2.4. Problem Definition.....	18
CHAPTER 3. RELATED WORK.....	19
3.1. Graph Mining Algorithms for Static Environment.....	20
3.1.1. Exact Algorithms.....	23
3.1.2. Approximate Algorithms.....	26
3.2. Frequent Subgraph Mining Algorithms for Dynamic Environment....	30
3.2.1. Exact Algorithms	34
3.2.2. Approximate Algorithms	37
CHAPTER 4. CONTROLLED RESERVOIR SAMPLING	41

4.1. Controlled Reservoir Sampling Algorithm with Unlimited Heap Size (UCRS)	42
4.2. Controlled Reservoir Sampling Algorithm with Limited Heap Size (LCRS).....	48
4.3. Maximum Controlled Reservoir Sampling (MCRS) Algorithm.....	52
4.4. Deleting an Edge in UCRS, LCRS, MCRS and Random algorithms..	57
CHAPTER 5. PERFORMANCE EVALUATION.....	60
5.1. Scalability.....	61
5.2. Recall	72
5.3. Heap size	78
5.4. Discussion on Experiments	82
CHAPTER 6. CONCLUSION	87
REFERENCES	90

LIST OF FIGURES

<u>Figure</u>	<u>Page</u>
Figure 2.1. Example of Graph G.....	6
Figure 2.2. Subgraphs of the Graph G	6
Figure 2.3. Dynamic graph G at different points of time	7
Figure 2.4. Example of (graph inputs as a series of small graphs) Transformation of a time series	8
Figure 2.5. (a) Dynamic graph G at different points of time (b) Subgraph g1 (c) Subgraph g2	9
Figure 2.6. Finding Frequent Subgraphs (Input and Output)	10
Figure 2.7. Graph sampling	11
Figure 2.8. Random node sampling from a graph	13
Figure 2.9. Random edge sampling from a graph.....	14
Figure 2.10. Random walk sampling from a graph	16
Figure 2.11. Breadth-first sampling of a large graph.....	17
Figure 3.1. Frequent Subgraph Mining Algorithms	19
Figure 3.2. Block diagram of proposed solution of Approximate GRAMI.....	27
Figure 3.3. The pseudo code of the first sampling technique	28
Figure 3.4. The pseudo code of the second sampling technique	28
Figure 3.5. The pseudo code of the third sampling technique	29
Figure 3.6. Example of Triest algorithm after adding a new coming edge (FG), when the sample reservoir is full.....	39
Figure 3.7. Example of SR or OSR algorithm after adding a new coming edge (FG), when the sample reservoir is full	40
Figure 4.1. Pseudo code of the UCRS algorithm.....	43
Figure 4.2. Deleting edge by random edge deletion and minimum controlled edge deletion.....	45
Figure 4.3. Example of the UCRS algorithm when whole reservoir (sample and heap reservoirs) is not full	46
Figure 4.4. Example of UCRS after adding a new coming edge (FG), while the whole reservoir (sample and heap reservoirs) is full	47
Figure 4.5. Pseudo code of the LCRS algorithms.....	49

<u>Figure</u>	<u>Page</u>
Figure 4.6. Example of the LCRS algorithm when whole reservoir (sample and heap reservoirs) is not full.....	51
Figure 4.7. Example of the LCRS algorithms, adding a coming node (FG) while the whole reservoir (sample reservoir and heap reservoir) is full.....	52
Figure 4.8. Pseudo code of the MCRS algorithm.....	54
Figure 4.9. Example of MCRS after adding a new coming edge (XE), while the whole reservoir (sample and heap reservoirs) is not full	55
Figure 4.10. Example of MCRS after adding a new coming edge (FG), while the whole reservoir is full	56
Figure 4.11. Deleting edge with minimum node degrees, maximum node degrees and random edge	58
Figure 5.1.A. Scalability performance of the algorithms while changing the dataset size on Datasets D1 for M= 600.....	62
Figure 5.1.B. Scalability performance of the algorithms while changing the dataset size on Datasets D1 for M= 600.....	63
Figure 5.2.A. Scalability performance of the algorithms while changing the dataset size on Datasets D1 for M= 1200.....	64
Figure 5.2.B. Scalability performance of the algorithms while changing the dataset size on Datasets D1 for M= 1200.....	64
Figure 5.3.B. Scalability performance of the algorithms while changing the dataset size on Datasets D1 for M= 1800.....	66
Figure 5.4.A. Scalability performance of the algorithms while changing the dataset size on Datasets D2 for M= 600.....	67
Figure 5.4.B. Scalability performance of the algorithms while changing the dataset size on Datasets D2 for M= 600.....	68
Figure 5.5.A. Scalability performance of the algorithms while changing the dataset size on Datasets D2 for M= 1200.....	69
Figure 5.5.B. Scalability performance of the algorithms while changing the dataset size on Datasets D2 for M= 1200.....	69
Figure 5.6.A. Scalability performance of the algorithms while changing the dataset size on Datasets D2 for M= 1800.....	70
Figure 5.6.B. Scalability performance of the algorithms while changing the dataset size on Datasets D2 for M= 1800.....	71

<u>Figure</u>	<u>Page</u>
Figure 5.7. Recall of the algorithms while changing the dataset size on Dataset D1 (M=600 nodes).....	73
Figure 5.8. Recall of the algorithms while changing the dataset size on Dataset D1 (M=1200 nodes).....	74
Figure 5.9. Recall of the algorithms while changing the dataset size on Dataset D1 (M=1800 nodes).....	74
Figure 5.10. Recall of the algorithms while changing the dataset size on Dataset D2 (M=600 nodes).....	75
Figure 5.11. Recall of the algorithms while changing the dataset size on Dataset D2 (M=1200 nodes).....	76
Figure 5.12. Recall of the algorithms while changing the dataset size on Dataset D2 (M=1800 nodes).....	76
Figure 5.13. Number of nodes in heap while changing the dataset size on Dataset D1 (M = 600 nodes).....	78
Figure 5.14. Number of nodes in heap while changing the dataset size on Dataset D1 (M = 1200 nodes).....	79
Figure 5.15. Number of nodes in heap while changing the dataset size on Dataset D1 (M = 1800 nodes).....	79
Figure 5.16. Number of nodes in heap while changing the dataset size on Dataset D2 (M = 600 nodes).....	80
Figure 5.17. Number of nodes in heap while changing the dataset size on Dataset D2 (M = 1200 nodes).....	80
Figure 5.18. Number of nodes in heap while changing the dataset size on Dataset D2 (M = 1800 nodes).....	81
Figure 5.19. Summary of Recall and Speed-up on Datasets D1 and D2.....	86

LIST OF TABLES

<u>Table</u>	<u>Page</u>
Table 3.1. Algorithms for static frequent subgraph mining.....	21
Table 3.2. Algorithms for dynamic frequent subgraph mining	32
Table 5.1. Properties of the datasets	61
Table 5.2. Scalability speed-up of the algorithms while varying the dataset size	72
Table 5.3. Recall of the algorithms while changing the datasets sizes	77
Table 5.4. Correlation between heap size and reservoir sizes for UCRS, LCRS and MCRS	82
Table 5.5. Summary of Execution Time Speed-ups and Recall Results	84

LIST OF TERMS AND ABBREVIATIONS

AGM	Apriori Graph Based Mining
BFS	Breadth First Strategy
CAM	Canonical Adjacency Matrix
DFS	Depth First Strategy
FFSM algorithm	Fast Frequent Subgraph Mining
FRISS	Frequent Relevant, and Induced Subgraph Subsequences
FS	First Sampling
FSM	Frequent Subgraph Mining
GERM	Graph Evolution Rule Miner
LCRS	Controlled Reservoir Sampling with Limited Heap Size
MCRS	Maximum Controlled Reservoir Sampling
OSR	Optimized Subgraph Reservoir
RE	Random Edge sampling
RN	Random Node sampling
RS	Reservoir Sampling
SR	Subgraph Reservoir
SRS	Simple Random Sampling
SSIGRAM	Spark based Single Graph Mining
UCRS	Sampling with Unlimited Heap Size

CHAPTER 1

INTRODUCTION

Graphs represent the complex and arbitrary relations among attributes of real-world data, such as users (nodes) and the relationships between them (edges) in social networks, atoms (nodes) and bonds (edges) in chemical structures, proteins (nodes) and protein interactions (edges) in biological networks, computers (nodes) and links between them (edges) in computer networks (Chakrabarti and Faloutsos, 2006) (Fournier-Viger *et al.*, 2020). Due to the increase in structured and semi-structured data represented in graphs, there has been rising interest in the mining graph data. Graph mining has several sub-categories such as graph classification, graph clustering and frequent subgraph mining, etc. (Jiang, Coenen and Zito, 2004) (Jiang, Coenen and Zito, 2013).

Frequent subgraph mining is defined as finding all the subgraphs in a given graph that appear more than a given support threshold. It is a widely studied problem as it results in the discovery of recurrent structures, themes or ideas in the given graph database which can be used further for performing other graph mining applications such as graph classification, graph partitioning, graph clustering, graph correlations etc. (Cuzzocrea *et al.*, 2015). Nowadays dynamic graph-based applications which deal with the dynamic data emerged i.e., social networks where friendships (i.e., edges of graph) are linked and dissolved over time, protein-to-protein interaction networks where knowledge is frequently updated. Because of these applications, the need for incremental frequent subgraph mining approaches has become essential. The increments can be represented in two different ways in a graph: (a) by a series of small graphs, and (b) as a stream of node and edge updates to the graph (Ray, Holder and Choudhury, 2014). These increments can be done in three different ways; first edges or/and nodes are being added to the network over time. Second, attributes of existing edges or/and nodes are modified over time. Third, edges or/and nodes previously present are being removed from the network.

Although frequent subgraph mining has been widely studied (Yan and Jiawei, 2002) (Huan, Wang and Prins, 2003) (Inokuchi, Washio and Motoda, 2000) (Ranu and Singh, 2009) (Fournier-Viger *et al.*, 2019), few works exist for dynamic frequent subgraph

mining (Kuramochi and Karypis, 2004) (Abdelhamid *et al.*, 2017). Static algorithms assume that graphs do not change over time and try to find all frequent subgraphs in the data. On the other hand, dynamic frequent subgraph mining algorithms deal with change in the data, however most of them concentrate on exact output similar to static algorithms. Exact algorithms search for all the frequent patterns; this requires high execution time and memory consumption. Therefore, for faster results users are willing to trade-off accuracy in cases where approximate results can serve the purpose. There are two recent works that are designed for approximate outputs (frequent subgraphs); Triest (De Stefani *et al.*, 2016), SR and OSR (Aslay *et al.*, 2018). They provide simple approximate approaches with trade-off between time and accuracy. Both solutions in (De Stefani *et al.*, 2016) and (Aslay *et al.*, 2018) use sampling technique based on the method which is proposed in (Vitter, 1985) where a randomized sampling schema that uses fixed sized reservoir is presented. The algorithm in (De Stefani *et al.*, 2016) relies on sampling edges, while the algorithms in (Aslay *et al.*, 2018) do sampling subgraphs to gain more accuracy. However, Triest (De Stefani *et al.*, 2016), SR and OSR (Aslay *et al.*, 2018) algorithms have some limitations; these limitations are trade-off between time and accuracy. SR and OSR are more accurate than Triest, while the recent one is faster than SR and OSR as appear in empirical results in (Aslay *et al.*, 2018). The main purpose of approximation is having high number of retrieved patterns (high recall) with a minimized execution time, so faster solutions with higher recall are still needed.

1.1. Contributions of the Thesis

The main contribution of this thesis is to introduce frequent subgraph mining algorithms in dynamic environment. We propose three approximate frequent subgraph mining algorithms that work on evolving graph data. Sampling is done by selecting a representative subset of the original graph by facilitating fixed size reservoir similar to recent competitors (De Stefani *et al.*, 2016) (Aslay *et al.*, 2018). The difference is in the management of the sample reservoir in the whole reservoir. Competitors use randomized sampling where the degrees of the nodes are not considered. When the reservoir is full any node even with highest degree can be deleted. This results in low recall.

The main contributions of this thesis are as follows.

- **Controlled Reservoir Sampling Algorithm with Unlimited heap size (UCRS):** this proposed algorithm use novel controlled edge-based sampling strategy with again fixed sized reservoir. Reservoir keeps the edges of the sample reservoir together with the nodes of the heap reservoir; the heap reservoir is a heap data structure keeps the degrees of the nodes in the sample. Management of the sample in the reservoir is done with the help of a heap data structure. Whenever an edge deletion is required, nodes of the edges that have lowest degree are chosen. In other words, in reservoir management, instead of random edge deletion, the edges that have nodes with low connectivity are potential targets to be removed from the sample reservoir, and if the node degrees are 1, they should be removed from heap reservoir. By this way, accuracy is maximized.

- **Controlled Reservoir Sampling Algorithm with Limited heap size (LCRS),** this algorithm is a modified version of UCRS. In LCRS, the heap reservoir size is minimized, as a result; sample reservoir size increases since the total number of nodes dedicated for the reservoir includes the nodes of the heap reservoir together with the nodes of the edges of the sample reservoir.

- **Maximum Controlled Reservoir Sampling (MCRS):** in this proposed algorithm, when the whole reservoir is full, an edge should be deleted from the sample reservoir to be replaced by a new edge, this candidate edge is an edge with maximum node degrees of its source and destination, while in UCRS the candidate edge for deletion is an edge with minimum node degrees of its source and destination. It is very similar to UCRS, but instead of deleting edges with minimum node degrees, it deletes edges with maximum node degrees. This algorithm is proposed as a heuristic and to check the validity of UCRS and LCRS.

The performance of UCRS, LCRS and MCRS algorithms are evaluated together with comparison to Triest (De Stefani *et al.*, 2016) and SR, OSR (Aslay *et al.*, 2018) using sparse and dense datasets. In the experiments the scalability, recall and heap size are measured. The findings are as follows; LCRS and UCRS have noticeable speed-up over (SR and OSR). UCRS and LCRS achieve high scalability, they can be as good as the fastest competitor algorithm. For the recall measurements, both LCRS and UCRS algorithms are better than Triest. LCRS achieves the highest recall on sparse datasets, while on dense dataset; LCRS can

achieve the highest recalls with large sizes of whole reservoir. MCRS has the worst speed-up and recall among the other proposed and competitor algorithms.

1.2. Organization of the Thesis

This thesis is organized as follows.

In Chapter 2 the fundamentals of graph terminology, frequent subgraph mining in dynamic and static graphs are given. then some graph sampling methods and techniques are explained. in addition, the problem and problem formulation are given.

In Chapter 3 a detailed discussion the state of art in frequent subgraph mining algorithms is reviewed. The main methodologies of existing frequent subgraph mining algorithms for static and dynamic environments are discussed. In both environments, some exact and approximate algorithms are explained and discussed.

In Chapter 4 three approximate frequent subgraph mining algorithms are introduced. They are designed to work in dynamic environment. Proposed algorithms use controlled reservoir sampling. Two of the methods namely UCRS and LCRS are designed for deleting edges with minimum node degrees when whole reservoir is full, while one is namely MCRS is designed for deleting edges with maximum node degrees when whole reservoir is full. each algorithm is explained with a motivating example.

In Chapter 5 the proposed algorithms are evaluated by using a set of experiments. These experiments are for measuring the scalability, the recall, and the heap size, the experiments are done in comparison to recent competitors, and then the experiments are discussed.

In Chapter 6 conclusion of this thesis is given with a summary, and possible future research directions are pointed out.

CHAPTER 2

BACKGROUND AND PROBLEM FORMULATION

Graphs represent the complex and arbitrary relations among attributes of real-world data, such as users (nodes) and the relationships between them (edges) in social networks. There has been rising interest in the mining graph data.

Frequent subgraph mining is defined as finding all the subgraphs in a graph that appear more than a given support threshold. It is a widely studied problem as it results in the discovery of recurrent structures. Frequent subgraph mining process consists of two phases, i.e., candidate generation and support computation (Dhiman and Jain, 2016).

Recently, dynamic graph-based applications which deal with the dynamic data emerged i.e., social networks where friendships (i.e., edges of graph) are linked and dissolved over time, etc. So, the need for incremental frequent subgraph mining approaches has become essential. The increments can be represented in two different ways in a graph: (a) by a series of small graphs, and (b) as a stream of node and edge updates to the graph (Ray, Holder and Choudhury, 2014). Some works exist for dynamic frequent subgraph mining (Kuramochi and Karypis, 2004), (Abdelhamid *et al.*, 2017). Static algorithms assume that graphs do not change over time, those static algorithms try to find frequent subgraphs in the data. While dynamic frequent subgraph mining algorithms deal with change in the data, however most of them concentrate on exact output similar to static algorithms. Exact algorithms search for all the frequent patterns, which requires high execution time and memory consumption. Therefore, for faster results with lower accuracy in some cases; approximate results can serve the purpose. But to do approximation, sampling of the input data is needed, there are several sampling techniques are explained in this chapter.

In this chapter, basic terminology graph basics, frequent subgraph mining, graph sampling, and problem formulation are presented.

2.1. Graph Basics

A graph is defined as a set of vertexes (nodes) that are interconnected by a set of edges. An example of a graph is shown in Figure 2.1. A graph G is an ordered pair (V, E) consisting of a set of vertices $V = \{v_1, v_2, v_3, v_4, v_5\}$ and V is connected to each other by and a set of edges $E = \{e_1, e_2, e_3, e_4\}$. A label function, L , maps a vertex or an edge to a label.

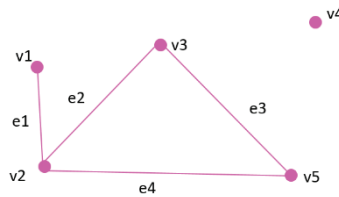


Figure 2.1. Example of Graph G

Assume subgraph $G'(V', E')$ is a subgraph of the graph $G(V, E)$, where edges and vertices are subsets of E and V respectively:

- $V' \subseteq V$
- $E' \subseteq E \wedge ((v_1, v_2) \in E' \rightarrow v_1, v_2 \in V')$

Figure 2.2. shows examples of subgraphs such that S_1 , S_2 , and S_3 are subgraphs of G in Figure 2.1.

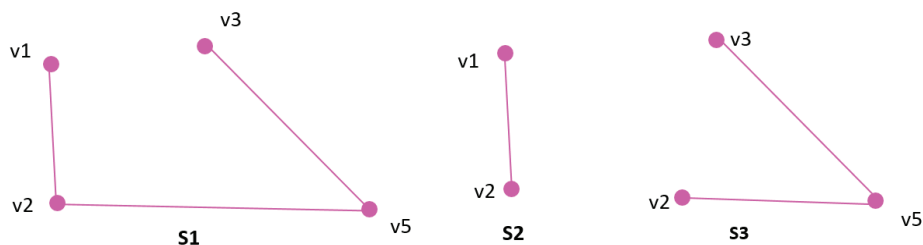


Figure 2.2. Subgraphs of the Graph G

Static graph: A graph $G = (V, E)$ consists of a set of nodes V , and a set of edges $E \subseteq V \times V$, where V and E do not change over time.

Dynamic graph (evolving graph): An evolving graph $G_D = (V_D, E_D)$ consists of a set of nodes V_D , and a set of edges $E_D \subseteq V_D \times V_D$. G_D is changed by node and edge additions or deletions over time. Figure 2.3 shows an example of a dynamic graph at three points of time (t_1 , t_2 and t_3). At time t_2 , there is a deletion of the edge u_7 - u_8 . At time t_3 , edge u_7 - u_8 and edge u_3 - u_{10} are added.

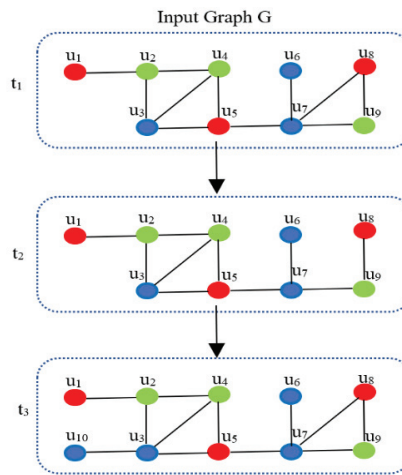


Figure 2.3. Dynamic graph G at different points of time

In incremental frequent subgraph mining approaches, the increments can be represented in two different ways in a graph: (a) by a series of small graphs, and (b) as a stream of node and edge updates to the graph. The following is explanation of the two kinds of increments (Ray, Holder and Choudhury, 2014).

A. Increments as a series of small graphs

In this kind of dynamic graphs, the increments are done by using a series of graph objects, i.e. each object in the stream is considered as a (static) graph snapshot.

Definition: (Series of Graphs) Given a sequence G_{ts} of n graphs $\{G_1, \dots, G_n\}$ with $G_i = (V_i, E_i)$ for $1 \leq i \leq n$. We define G_{ts} to be a time series of graphs if $V_1 = V_i$ for all $1 \leq i \leq n$. G_i is the i -th state of G_{ts} (Borgwardt, Kriegel and Wackersreuther, 2006).

Figure 2.4 shows an example of a dynamic graph when the input is a series of small graphs, it shows a graph input, where each incoming object is an entire graph.

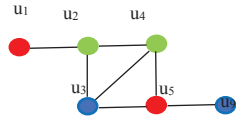
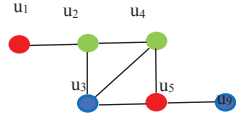
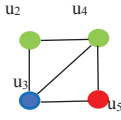
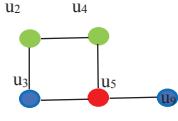
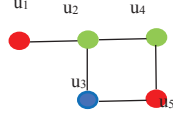
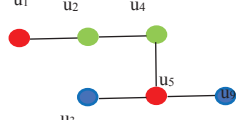
Transaction ID	Graph
1	
2	
3	
4	
5	
6	

Figure 2.4. Example of (graph inputs as a series of small graphs) Transformation of a time series

B. Increments as a stream of nodes and edges

In this kind of dynamic graphs, the increments consist of nodes and edges that change over time, and it can be addition or deletion.

Definition: Dynamic graph (evolving graph): An evolving graph $G_D = (V_D, E_D)$ consists of a set of nodes V_D , and a set of edges $E_D \subseteq V_D \times V_D$. G_D is changed by node additions or deletions, edge additions or deletions over time. Figure 2.5(a) shows an example of a dynamic graph at three points of time (t_1 , t_2 and t_3). At time t_2 , there is a deletion of the edge u_7 - u_8 . At time t_3 , edge u_7 - u_8 and edge u_3 - u_{10} are added.

Figure 2.5 shows an example of an edge updates, it shows the dynamic graph at different time points t_1 , t_2 and t_3 when edges are coming as updates.

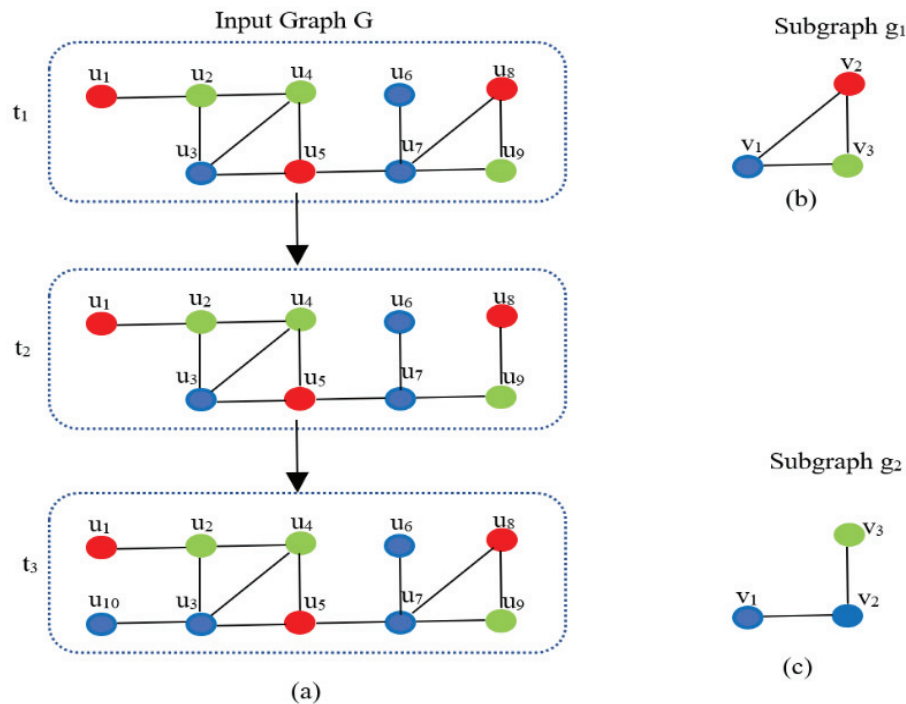


Figure 2.5. (a) Dynamic graph G at different points of time (b) Subgraph g_1 (c) Subgraph g_2

Subgraph isomorphism: Given two undirected graphs G and H. There is an isomorphism between G and H, if there is a bijection f between their vertices ($f: V(G) \rightarrow V(H)$). Hence, two vertices u, v are adjacent to each other in G if and only if $f(u), f(v)$ are adjacent in H. These two graphs are topologically identical in topology.

2.2. Frequent Subgraph Mining

The main task of frequent subgraph mining (FSM) is to find all frequent subgraphs in a given graph or a set of graphs for a given user-defined threshold (Aggarwal and Wang, 2010). The support of a subgraph (g) is defined the number of occurrences of this graph in a graph dataset.

Given a graph $G = (V_g, E_g)$, a graph $F = (V_f, E_f)$ will be a subgraph of G if and only if the vertices and edges of graph H are a subset of the vertices ($V_f \subseteq V_g$) and edges ($E_f \subseteq E_g$) of graph G . If the support of subgraph is equal or greater than the user-defined minimum support threshold, then this subgraph is considered as a frequent subgraph. If a graph is frequent, all its subsets must be frequent (downward closure property) (Dinari and Naderi, 2014).

Figure 2.6 illustrates an example of finding frequent subgraphs. Input is a database of graph transactions, undirected simple graph (no loops, no multiples edges), each graph transaction has labels associated with its edges and vertices, transactions might not be connected and a minimum support threshold σ ; example (60%). The Output is frequent subgraphs that satisfy the minimum support threshold, and each frequent subgraph is connected.

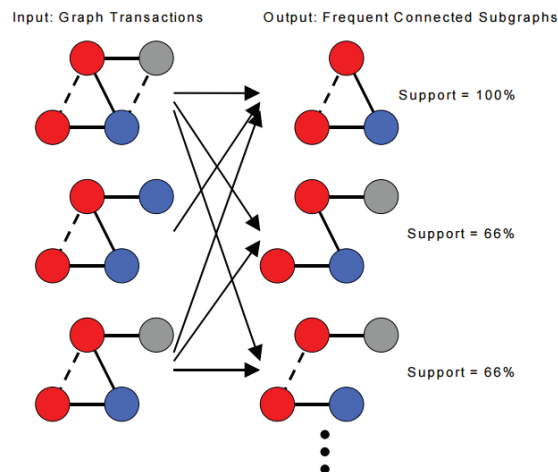


Figure 2.6. Finding Frequent Subgraphs (Input and Output)

Frequent subgraph mining process consists of two phases, i.e., candidate generation and support computation (Dhiman and Jain, 2016).

Frequent subgraph: Subgraph g to be frequent in an input graph G , if it has support larger than or equal to a user-defined threshold.

Frequent Subgraph Mining: It is the process of finding all frequent subgraphs in a given graph G . Finding isomorphic subgraphs is one the challenges of this process.

Dynamic Subgraph Mining: It is the process of finding all frequent subgraphs on evolving graph. In the example presented in Figure 2.5 given an input dynamic graph G and support threshold 2, let us see the status of subgraphs g_1 and g_2 . Addition of an edge to input graph increases the support of one or more subgraphs, removal of an edge of the input graph decreases the support of one or more subgraphs (Abdelhamid *et al.*, 2017). At time t_1 , the subgraph g_1 has 2 matches in G , however the subgraph g_2 has only one match. As a result, g_1 is frequent subgraph and g_2 is not frequent subgraph. At time t_2 , there is a deletion of the edge u_7-u_8 , so the number of embeddings of g_1 decreases to one, however the number of embedding of g_2 does not change. Therefore, both subgraphs g_1 and g_2 are not frequent. At time t_3 , there is an addition of edge u_3-u_{10} , this addition increases the number of matches of g_2 to two; so, g_2 becomes frequent.

2.3. Graph Sampling

Graph sampling is done by selecting a representative subset of the original graph as shown in Figure 2.7, so the graph sampling can make the graph size smaller while keeping the characteristics of the original graph (Sahu *et al.*, 2021) (Wang *et al.*, 2011).

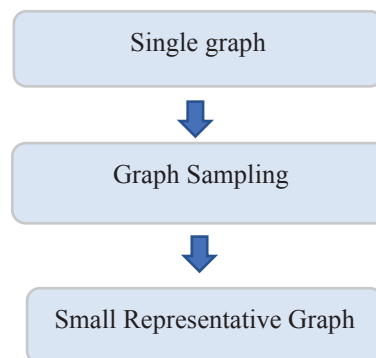


Figure 2.7. Graph sampling

Graph sampling: Graph sample of Graph $G = (V, E)$ is defined as $G_s = (V_s, E_s)$ where $V_s \subseteq V$, $E_s \subseteq E$ and $E_s \subseteq \{(u, v) \mid u \in V_s, \text{ and } v \in V_s\}$.

Approximate Frequent Subgraph Mining: If the frequent subgraph mining algorithm is applied on a sample graph of the original graph, the results is approximate.

To do approximation, sampling is needed, in the following, we will talk about graph sampling in context, need for Sampling, notations and definition, and finally graph sampling methods with examples.

Graph sampling is needed Social network analysis, to keep the graph scale small while capturing the properties of the original social graph, graph sampling provides an efficient, yet inexpensive solution for social network analysis (Wang *et al.*, 2011). Sampling can be used in graph analysis in applications such as security, high performance computing, etc (Zhang *et al.*, 2017). Also, survey hidden population in sociology, scale down Internet AS graph, graph sparsification, etc. (Stutzbach *et al.*, 2006). Sampling provides an abstract version of the original graph. Thus, visualizing sampling results is easier than visualizing the original. Secondly, the analysis of a large graph is costly. The third reason is incomplete graph data. In some cases, obtaining all data for a graph is not permitted or is very time-consuming. Thus, we must obtain the properties of the graph by sampling.

For the above reasons, sampling algorithms aims to reduce the complexity of graph drawing while preserving properties of the original graph, allowing analysis of the small sample to yield the characteristics similar to those of the original graph (Zhang *et al.*, 2017).

Graph Sampling Methods

There are four sampling methods, which are Node Sampling, Edge Sampling, Traversal-based sampling (Zhang *et al.*, 2017) and Sampling with Neighbourhood (VSN) (Hu and Lau, 2013). In the following we will explain each one with algorithmic example.

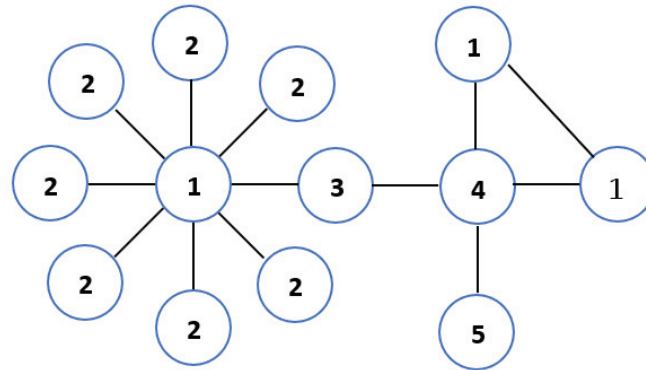
- **Node Sampling**

In node sampling, vertices are sampled. A subgraph is created from sampled nodes and existing edges of original graph. For example, Random Node (RN) sampling.

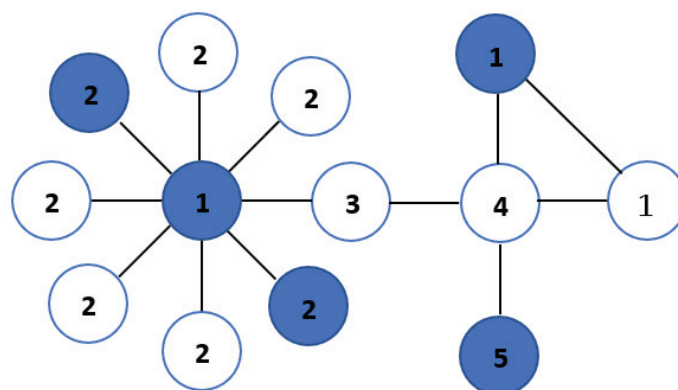
Example:

Random Node (RN) sampling (Leskovec and Faloutsos, 2006): it is the most common method; it selects a set of nodes uniformly at random from the graph. Using this set, an

induced subgraph can be created by including every edge that connects a pair of nodes in the set. RN is simple and efficient (Wu *et al.*, 2017), Figure 2.8 shows random node sampling from a graph.



(a) Original graph



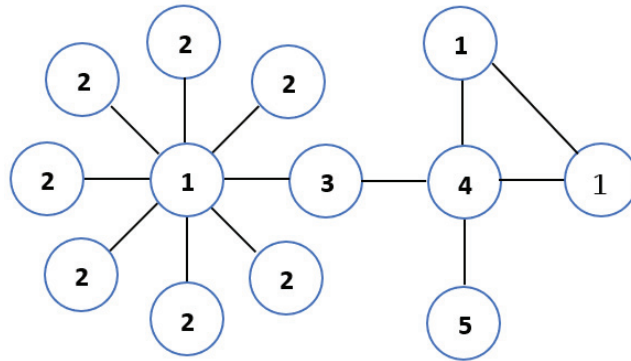
(a) Random Node

Figure 2.8. Random node sampling from a graph

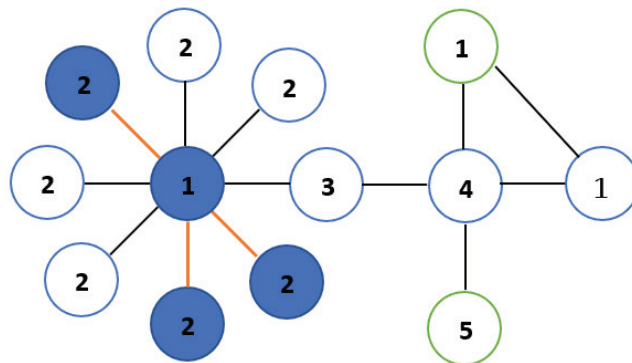
- **Edge Sampling**
- In edge sampling, edges are sampled, and then a subgraph is created from those edges. Induced edge sampling includes totally induced edge sampling and partially induced edge sampling. For example, Random Edge (RE) sampling, First (FS) sampling and Traversal-based sampling.

Example 1: Random Edge (RE) sampling

It generates an induced subgraph by selecting edges uniformly at random. The random edge deletion from graph is shown in Figure 2.9.



(a) Original graph



(b) Random Edge

Figure 2.9. Random edge sampling from a graph

Example 2: FS

- FS firstly randomly chooses a set of nodes, S , as seeds.
- Then FS will select a seed v from the set of seeds with the probability defined as follows:

$$P(v) = \frac{k_v}{\sum_{u \in S} k_u}$$

An edge (v, w) is selected uniformly from node v 's outgoing edges, and v will be replaced with w in the set of seeds and edge (v, w) will be added to the sequence of sampled edges.

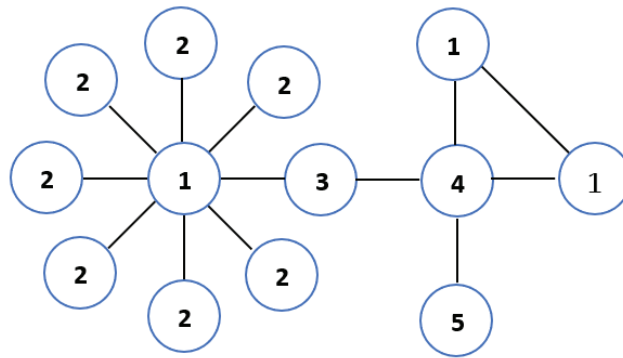
- FS repeats these steps until the budget is reached.

- FS requires that at least one of the in degree and out degree of the nodes is not 0.
- Otherwise, the node has neither incoming nor outgoing edges, which means, this node is isolated. In real OSNs the number of isolated nodes is small and in most researches isolated nodes are not considered (Wang *et al.*, 2011).

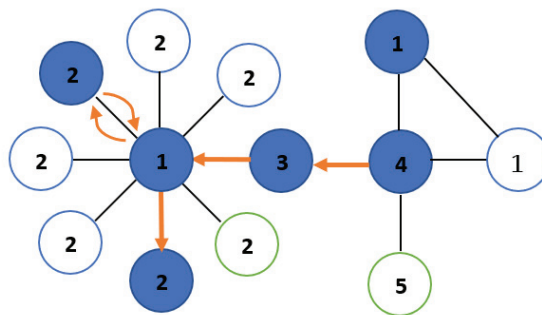
- Traversal-based sampling

Traversal-based sampling uses topology information to sample a subgraph. For example, Random walk sampling and Breadth-first sampling.

Example 1. Random walk sampling (Stutzbach *et al.*, 2006) starts at a seed vertex, and then chooses a vertex uniformly at random from the neighbours of the current vertex. A subgraph is created from the walking paths. Figure 2.10 shows random walk sampling.



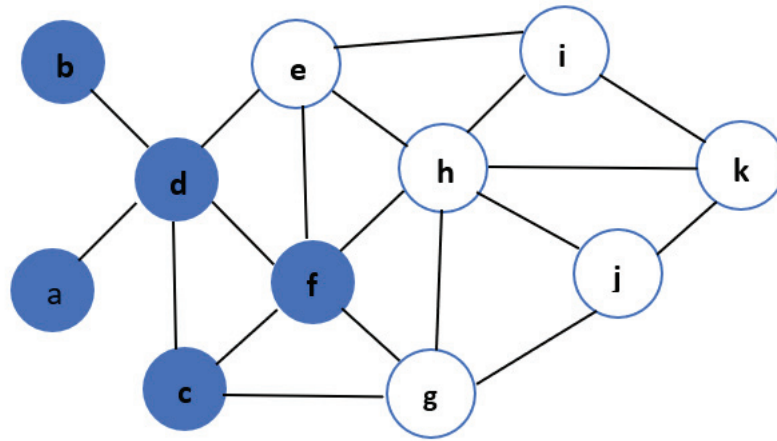
(a) Original graph



(b) Random Walk

Figure 2.10. Random walk sampling from a graph

Example 2. Breadth-first sampling (Hu and Lau, 2013) (Zhang *et al.*, 2017) is induced from the graph traversal algorithm breadth-first search. It begins with a random vertex and visits its neighbours iteratively. An example of BFS is shown in Figure 2.11.



d b a c f e h g k j
 Sampling budget

Figure 2.11. Breadth-first sampling of a large graph

Sampling algorithms in data streams

1. Simple random sampling (srs) (Yates *et. al.*, 2002)

In srs, a sample is chosen by picking each item in the data stream with an equal probability of being selected. The inclusion probability is provided as an input parameter to the sampling algorithm. Consider a $p = 0.5$, then each item in the data stream has half the chance to be included in the sample. srs is extremely simple, however one downside of srs is that the size of the sample grows along with size of the data stream (Anis and Nasir, 2018).

2. Reservoir sampling (rs) (Vitter, 1985)

It is a fixed-size randomized sampling scheme, which maintains a fixed-size uniform sample of the data stream. The size of the sample is provided as the input parameter. The algorithm initializes with a fixed-size input array, which initially gets filled by the items in the data stream. Once the array is filled, each i -th item is added to the sample with probability $1/i$ by replacing it with a randomly selected item from the sample. Random pairing (Gemulla *et al.*, 2006) is a fully dynamic algorithm for reservoir sampling, that compensate for item deletions using the future addition (Anis and Nasir, 2018).

2.4. Problem Definition

We consider the problem of finding approximate frequent subgraphs on a dynamic undirected graph where changes are additions of edges occurring over time. In other words, we observe a sample graph that changes over time t and its size does not exceed pre-defined threshold (maximum whole reservoir size). Assume the sample graph $G_t = (V_t, E_t)$, for any time $t \geq 0$, where V_t represents the vertices and E_t represents the edges. For any time instant $t \geq 0$, we receive an edge element e_{t+1} which consists of a pair of vertices (u, v) . The sample graph $G_{t+1} = (V_{t+1}, E_{t+1})$ is obtained by adding a new edge to the existing sample graph as follow: $E(t+1) = E(t) \cup (u, v)$. If u or v are not in $V(t)$; they will be added to $V(t+1)$.

The details of the management of fixed size reservoir reserved for the sample reservoir and heap reservoir are explained in the next chapter.

CHAPTER 3

RELATED WORK

In this chapter, a literature review on existing frequent subgraph mining approaches are presented. These algorithms are classified into two categories according to their environment: static and dynamic as in Figure 3.1.

In dynamic environment the graph datasets are evolving over time, while in static environment the state of the graphs doesn't change. In the literature many approaches are proposed to work in static environment. The Figure 3.1. shows that the frequent subgraph mining for dynamic data can be classified into two categories: exact and approximate. Most algorithms in dynamic environment concentrate on exact algorithms, where the exact approaches search for all the frequent patterns. However, for faster results users are willing to trade-off accuracy, whenever approximate results can serve the purpose. Those approximate algorithms use different sampling approaches. The basic idea in approximation is to execute the exact algorithm on a small portion of the data (sample) (Iyer, A. Liu, Z. Jin, X. Venkataraman, S. Braverman, V. Stoica, 2018). Sampling is done by selecting a representative subset of the original graph where the purpose is to reduce graph size while preserving the characteristics of the original graph (Wang *et al.*, 2011).

Frequent subgraph mining process consists of two phases, i.e., candidate generation and support computation (Dhiman and Jain, 2016). Several algorithms are proposed to solve this problem. In the following two sections, some graph mining algorithms for static and dynamic environment are presented.

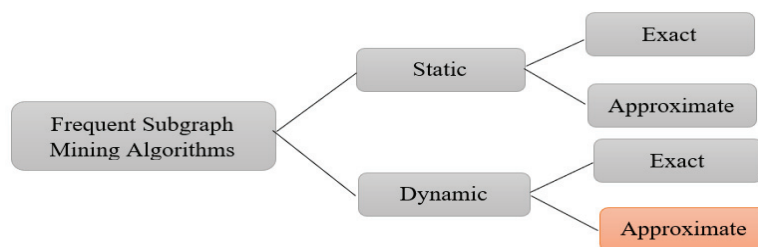


Figure 3.1. Frequent Subgraph Mining Algorithms

3.1. Graph Mining Algorithms for Static Environment

In static graph mining literature, several algorithms are proposed using different approaches with different attributes such as input and output type, graph type, graph representation, algorithmic approach, programming approach. Let us explain different comparison attributes.

The input type can be different from an algorithm to another. It can be a set of graphs that consist of a group of small graphs as chemical molecules. It is also possible to be a one single graph that is generated from associations of many small subgraphs as social networks. In frequent subgraph mining algorithms, the main difference between the two types is in frequency calculation.

The graph type of the input graphs can have one of the following possible types: undirected labelled graphs, undirected graphs, directed graphs, connected graphs, connected undirected graphs, or labeled graph.

The graph representation is considered as one of the most effective attributes on the consumption of runtime and memory. In general, the graphs can be represented by the adjacency matrix, adjacency list and canonical labelling.

Algorithmic approach shows pattern finding approach of the base algorithm where the possible values can be Apriori or Pattern growth. The Apriori based algorithms (Inokuchi, Washio and Motoda, 2000) generate candidates using breadth first strategy (BFS) and apply subgraph isomorphism testing to calculate frequencies of candidates. Pattern growth-based algorithms (Yan and Han, 2003) generate candidates based on depth first strategy (DFS). The pattern growth approach avoids the cost of generating candidates and subgraph isomorphism testing. The candidates are generated by extending frequent subgraphs starting from minimal frequent subgraphs by adding one edge at every step until they are still frequent.

Programming approach: some algorithms utilizes parallel programming, in order to take advantage of the existing multicore processor technology, or distributed, and as a result the time complexity is reduced. While other uses serial programming approach.

Type of output, the essential purpose of each algorithm is extracting a reduced set of frequent subgraphs. So, the nature of each output could be different from one algorithm to another, the retrieved output could be all, approximate, or closed frequent subgraphs.

Due to the existence of several static subgraph mining approaches. Table 3.1 summarizes a literature about some algorithms in static subgraph mining.

Table 3.1. Algorithms for static frequent subgraph mining

Algorithm	Input type	Graph type *	Graph representation	Algorithmic approach	Programming approach	Output type	Limitations
gSpan (Yan and Han, 2002)	set of graphs	U	Canonical label(min DFS)	Pattern growth	Serial	All frequent subgraphs	It is unable to process large datasets
FFSM (Huan, Wang and Prins, 2003)	set of graphs	U	Canonical label(CAM)	Apriori based & pattern growth	Serial	All frequent subgraphs	NP-complete problem
CloseGraph (Yan and Han, 2003)	set of graphs	U / D	Canonical label(min DFS)	Pattern growth	Serial	Closed frequent subgraphs	Failure detection takes lot of time overhead
AGM (Inokuchi, Washio and Motoda, 2000)	set of graphs	U / D	adjacency matrix	Apriori based	Serial	All Frequent subgraphs	High complexity due to multiple candidate generation
FSG (Kuramochi and Karypis, 2001)	set of graphs	U	adjacency list CAM	Apriori based	Serial	All frequent subgraphs	High complexity due to multiple candidate generation
FSM-H (Bhuiyan and Al Hasan, 2015)	set of graphs	U	adjacency list	Apriori based	Parallel	All Frequent subgraphs	High complexity due to multiple candidate generation
gSpan-H algorithm (Sangle and Bhavsar, 2016)	set of graphs	D	Canonical label(min DFS)	Apriori based	Parallel	All Frequent subgraphs	Multiple candidate generation
HSIGRAM (Kuramochi and Karypis, 2005)	single large graph undirected	U	Canonical label(CAM)	Apriori based	Serial	Approximate frequent subgraphs	Multiple candidate generation
VSIGRAM (Kuramochi and Karypis, 2005)	single large graph	U	Canonical label(CAM)	Pattern growth	Serial	Approximate frequent subgraphs	Some interesting patterns can be lost
SPIN (Huan <i>et al.</i> , 2004)	Set of graphs	U	Adjacency matrix	Pattern growth	Serial	Maximal frequent subgraphs	Interesting patterns may be lost
Ap-FSM (Bhatia and Rani, 2018)	single large graph	L	Canonical label(CAM)	Pattern growth	Parallel	Approximate frequent subgraphs	It handles only distributed systems
MaNIACS (Preti, De Francisci Morales and Riondato, 2021)	single large graph	L	Canonical label	Apriori based	Serial	Approximate frequent subgraphs	It is compared with the exact algorithm; it is preferred to be compared with recent approximate algorithms. Not scalable for small graphs, it uses random sampling for approximation, interesting patterns might be lost
Approximate GraMi (Sahu <i>et al.</i> , 2021)	single large graph		adjacency matrix	Apriori based	Serial	Approximate frequent subgraphs	Works only with static graphs

Graph type*:

U: Undirected graphs

D: Directed graphs

C: Connected graphs

L: Labeled graph

Frequent subgraph mining algorithms can be categorized into two main approaches according to their base algorithm, the two categories are Apriori based approach or pattern-growth based approach.

Algorithms based on Apriori: The Apriori based algorithms consist of two main steps: the first step is generating candidates using breadth first strategy (BFS), the second step is applying subgraph isomorphism testing to calculate frequencies of candidates. Apriori based algorithms are extended from Apriori algorithm (Inokuchi, Washio and Motoda, 2000). In the first step, the level-wise strategy for candidate generation is used. Apriori based approaches has a drawback because of the large number of candidates that are generated on large datasets. As a result, downward closure property is employed to minimize the search space, this property states that, if a subgraph is not frequent, its superset (set containing it) is considered not frequent. In the next step, there is no need to check whether any candidate graph containing this subgraph is frequent or not. In Apriori based algorithms the number of candidates is reduced, but on large datasets and minimum support threshold, these algorithms do not work well. This is due to the large number of generated candidates, also this process requires multiple scans for the database. Apriori based algorithms have challenges regard to subgraph isomorphism testing.

Algorithms based on FP-Growth: the main purpose of FP-Growth based algorithms (Han, Pei and Yin, 2000) is to discover frequent subgraphs without candidate generation and subgraph isomorphism testing. FP-Growth approach based on divide and conquer method. In this approach a frequent sub graph is extended by adding an extra edge in every possible position, this process continues until no more frequent subgraphs remains. This is instead of candidate generation. The mentioned extension of edges is done instead of candidate generation, but the drawback here is while adding an extra edge in every possible position, there is a probability that the same sub graph can be discovered several times, this results in duplication in candidate generation. There are works done to eliminate the duplication by using rightmost extension technique such as gSpan algorithm (Huan, Wang and Prins, 2003). Pattern growth-based FSM algorithms usually use the rightmost extension technique in the candidate generation process, and to avoid subgraph isomorphism testing in calculating the frequencies of subgraphs; the minimum DFS code is used.

One of the drawbacks of Apriori based algorithms is multiple scans for the database. To overcome this problem, pattern-growth based algorithms was developed, it handles a more compact and smaller data structure instead of working on the whole the database. In this approach, the number of generated candidates are reduced, also the subgraph isomorphism test is better than the Apriori based approaches. Pattern- Growth

approach algorithm include SPIN (Huan *et al.*, 2004), gSpan (Yan and Jiawei, 2002) and FFMS (Huan, Wang and Prins, 2003).

Type of output, the essential purpose of each algorithm is extracting a reduced set of frequent subgraphs. So, the nature of each output could be different from one algorithm to another, Some FSM algorithms retrieve all the frequent subgraphs, the output is called exact. while some algorithms retrieve part of the frequent patterns, in this case the algorithms are approximate. The exact and approximate types are presented in the following two subsections.

3.1.1. Exact Algorithms

Exact algorithms search for all the frequent patterns; this requires high execution time and memory consumption. Exact algorithms can be in dynamic or static environments. Several algorithms were proposed to serve this purpose e.g. (Huan, Wang and Prins, 2003), (Huan, Wang and Prins, 2003), (Elseidy, Abdelhamid and Skiadopoulou, 2014) and (Abdelhamid *et al.*, 2017). The following is an algorithm example the has an exact output.

The SSIGRAM (Spark based Single Graph Mining) algorithm in Qiao *et al.*, 2018; where Spark is an in-memory MapReduce-like general-purpose distributed computation platform which provides a high-level interface for users to build applications. Unlike (Zaharia *et al.*, 2010). The proposed method is based on parallel frequent subgraph mining algorithm in a single large graph. It approaches the two computational challenges of frequent subgraph mining, it conducts the subgraph extension and support evaluation parallel across all the distributed cluster worker nodes. Also, it utilizes a heuristic search strategy and three novel optimizations: load balancing, pre-search pruning and top-down pruning in the support evaluation process that significantly improve the performance. experiments using four different real-world datasets demonstrate that the proposed algorithm outperforms the existing GRAMI (Elseidy, Abdelhamid and Skiadopoulou, 2014) Graph Mining algorithm by an order of magnitude for all datasets and can work with a lower support threshold (Qiao *et al.*, 2018).

The AGM (Apriori graph based mining) (Inokuchi, Washio and Motoda, 2000) is employed to discover frequent subgraphs. Adjacency matrix is used to represent graph. A level wise search is used to discover the frequent subgraphs. It assumes that graph

contains only distinct vertexes. However, overall analysis showed that the time complexity of directed graphs is less than that of undirected graphs, this is since the possible edge directions in directed graphs results in more subgraph patterns, and their frequency will be less. In addition, the complexity of small graphs is less than larger graphs. In the experimentation of AGM on chemical carcinogenesis data; the output of AGM was subgraphs that are connected and subgraphs that are not connected with several isolated graph. It efficiently mined frequent subgraph, but complexity was high due to multiple candidate generation. Experiments reported in (Huan, Wang and Prins, 2003) showed that AGM performs good in dense synthetic graph datasets, and takes 40 minutes to 8 days (approx.) to tabulate all frequent sub graphs in a dataset containing 300 chemical compounds, when the minimum support threshold varies between 20% to 10%.

The FSG algorithm (Kuramochi and Karypis, 2001) finds all frequent connected subgraphs. It generates candidate subgraphs based on edges i.e., Candidate subgraphs are generated by adding edge to the previous subgraph. For frequency counting, it uses transaction identifiers list for frequent subgraphs, and it uses adjacency list for graph representation. Canonical labels (A canonical code is a unique code of a given graph and it is always be the same no matter how the graphs are represented, as long as those graphs have the same topological structure and the same labelling of edges and vertices) are used to check isomorphic graphs. It is very costly because it uses isomorphism testing, and it also generates a huge set of candidates. It requires multiple scans of database. It is inefficient for mining large sized subgraph patterns. It needs efficient finding of isomorphic graphs to count support. The advantages of FSG are that it can prune candidates without subgraph isomorphism. For large datasets, it checks only those graphs which may potentially contain the candidate.

The gSpan algorithm (Yan and Han, 2002) is a Graph-Based Substructure Pattern Mining, it discovers frequent substructure without candidate generation, it uses a canonical representation for graphs, which called "DFS-Code". gSpan maps subgraphs to a unique minimum depth-first search (DFS) code and uses a lexicographic order on these codes to order subgraphs. Based on this order, a DFS strategy is used to mine frequent subgraphs efficiently in gSpan. Such that, gSpan traverses the DFS Code Tree, where the code of a node corresponds to the parent's code is extended by one edge and the siblings are ordered according to the lexicographic order. Using this approach, the traversal starts from the smallest subgraphs and it backtracks if the corresponding subgraph is not frequent (Yan and Han, 2002). In gSpan; refinement generation is done in two ways: 1)

fragments can only be extended at nodes which lie on the rightmost path of the DFS tree. 2) Fragment generation is guided by occurrence in the appearance lists. Because these two pruning techniques cannot fully prevent isomorphism, gSpan calculates the canonical DFS code (lexicographically smallest) for each refinement by means of a permutation's series. Refinements with non-minimal DFS-code can be pruned. Since instead of embeddings, gSpan only stores appearance lists for each fragment, explicit subgraph isomorphism testing must be done on all graphs in these appearance lists. NP-completeness of the subgraph isomorphism leads to an exponential run time.

Luckily, graphs with diverse labels can decrease the runtime substantially when experiments are applied. Another problem of gSpan, and frequent subgraph mining algorithms in general, is that for large or dense graphs, the number of frequent subgraphs is very large, and as a result, it is not practical to mine all of them (Cook and Holder, 2007).

The FFSM algorithm (Fast Frequent Subgraph Mining) (Huan, Wang and Prins, 2003) considers large dense graphs with less labels. It represents graphs as triangle matrices (node labels on the diagonal, edge labels elsewhere). In this algorithm, vertical level search strategy is used to reduce the number of candidate generation. The main features of the proposed method are: First; a novel graph canonical form and two efficient candidate proposing operations are employed which are: FFSM-Join and FFSM-Extension, Second; suboptimal CAM (Canonical Adjacency Matrix) tree which is an algebraic graphical framework in order to ensure that all detected frequent subgraphs are enumerated unambiguously, and finally, avoiding subgraph isomorphism testing which is time consuming, this done by maintaining an embedding set for each frequent subgraph. However, FFSM only stores the matching nodes, while edges are ignored. As a result, this helps speeding up the join and extension operations, because the embedding lists of new fragments can be calculated by set operations on the nodes. Adjacency matrix is employed for graph representation. Limitation of FFSM algorithm is that it is NP-complete problem. Experimentation showed that FFSM outperformed gSpan.

3.1.2. Approximate Algorithms

There are two main drawbacks of exact algorithms, the first one is the requirement of high execution time, while the second one is the need for high memory consumption that required to search for all frequent patterns. Therefore, for faster results and lower memory consumption, approximate algorithms can serve the purpose, with approximation the user trade-off accuracy for much faster results. The main idea of approximate algorithms is executing the exact algorithm on a small subset of the data set, which is called samples, there are different approximate algorithms are proposed recently like (Iyer, A. Liu, Z. Jin, X. Venkataraman, S. Braverman, V. Stoica, 2018), (Bhatia and Rani, 2018), (Aslay *et al.*, 2018), (Preti, De Francisci Morales and Riondato, 2021) and (Sahu *et al.*, 2021). Explanation about an approximate algorithm is presented next.

The CloseGraph algorithm in (Yan and Han, 2003) is founded on gSpan. It uses an equivalent occurrence-based early termination in order to prune the search space. CloseGraph uses DFS strategy, lexicographic order, minimum DFS code and rightmost extension for finding closed frequent subgraphs. The concept of closed subgraph mining is not only reducing unnecessary subgraphs to be produced, but also substantially increasing the efficiency of mining, especially in the large graphs' patterns are presented. Experimental results demonstrated that CloseGraph performed better than gSpan and FSG (Yan and Han, 2003). Performance study shows that, CloseGraph not only reduces unnecessary subgraphs to be generated, but also increases the efficiency of mining, particularly in the presence of large graph patterns (Muttipati, 2015).

MaNIACS (Preti, De Francisci Morales and Riondato, 2021) is a sampling-based randomized algorithm for computing high quality approximations of the subgraph patterns, it works on a single, large, vertex labelled graph. It prunes the pattern search space, and thus to reduce the time spent in exploring subspaces containing no frequent patterns. It relies on uniform random sampling of vertices and on computing the patterns to which these vertices belong. It prunes parts of the search space that provably do not contain any frequent pattern, and to focus the exploration only on the “promising” subspaces, therefore avoiding expensive computations. Pruning leads to better bounds to the maximum frequency estimation error, which enables additional pruning. It is the first to use concepts from statistical learning theory for FPSM. In experimental evaluation, it is compared with exact algorithm, it shows that it returns high-quality collections of

frequent patterns in large graphs up to two orders of magnitude faster than the exact algorithm, it is scalable w.r.t. the size of the graph, Scalable for large graphs and no gain with small graphs.

Approximate GraMi (Sahu *et al.*, 2021) is an approximate algorithm, it works on static graphs. In this work they proposed three sampling techniques, these techniques are applied on a single large graph, the results are a sampled graph, which is used as an input to an existing static exact algorithm (GRAMI), the results are approximate subgraphs. Block diagram of the phase of approximate GraMi solution is shown in Figure 3.2.

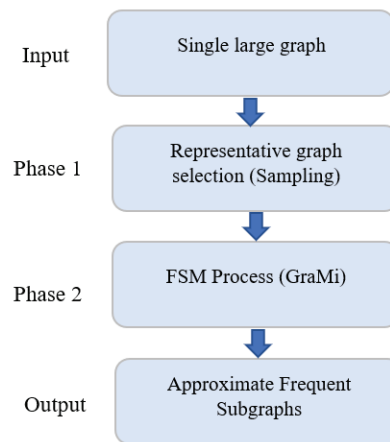


Figure 3.2. Block diagram of proposed solution of Approximate GRAMI (Sahu *et al.*, 2021)

The explanation of the three sampling techniques is follows:

The first sampling technique:

This technique samples the graph by picking up the vertices based on sampling rate. This is done according to the following three steps: First a list of vertices that have the same degree is created; this list is sorted in increasing order according to the degrees of the vertices. Second, the vertices that have the same degree are put into a single list, finally, the vertices are selected according to user defined sampling rate and then they are merged into a single list. The pseudo code of this sampling technique is as shown in Figure 3.3.

Input: A Graph $G (V, E)$ of size N , Sampling rate Δ
Output: A sampled graph $G' = (V', E')$ of size x

1. Rank N vertices of graph based on decreasing order of their degree d_i
2. Put the vertices with similar degree in list l_i
3. $G' \leftarrow \emptyset$
4. $h = \Delta * 100$
5. $x = h \% N$
6. While $|V'| \leq x$ do
7. For each list l_i do
8. Select $h \%$ vertices
9. End
10. $l'_i \leftarrow$ Selected $h \%$ vertices
11. End
12. $G' \leftarrow \sum_{i=0} l'_i$

Figure 3.3. The pseudo code of the first sampling technique (Sahu *et al.*, 2021)

The second sampling technique:

In this technique randomness is employed to create sample from the original graph. Sampling is done by selecting an edge randomly and added if it is not already existed in the list. The steps of this technique are as follows:

First, from the edges in the list, an edge is selected randomly. Edge selection is done up to a user defined sampling rate. The selected edge is added if it is not already in the list.

The pseudo code of the second sampling technique is as shown in Figure 3.4.

Input: A Graph $G (V, E)$ of size N , Sampling rate Δ .
Output: A sampled graph $G' = (V', E')$ of size x .

1. $V' = \phi$ and $E' = \phi$
2. $h = \Delta * 100$
3. $x = h \% N$
4. While $E' \leq x$, $k = 1$
5. do
6. Select $\text{rand}(e_k, (u, v)) = e_k$
7. If $u \notin V'$ and $v \notin V'$
8. Do nothing
9. else $E' = E' \cup e_k$
10. $V' = V' \cup (u, v)$
11. Increment k
12. Repeat from step 5.
13. End if
14. $G' \leftarrow (V', E')$
15. End

Figure 3.4. The pseudo code of the second sampling technique (Sahu *et al.*, 2021)

The third sampling technique:

This sampling technique selects certain number of top vertices from the sorted list of the vertices. This process starts by sorting the vertex list on increasing order according to the degrees of vertices. Then topmost vertices are selected according to the sampling rate, then they are merged into a single list. The pseudo code of the third sampling technique is as shown in Figure 3.5.

Input: A Graph $G (V, E)$ of size N , Sampling rate Δ

Output: A sampled graph $G' = (V', E')$ of size x

1. Rank N vertices of graph based on decreasing order of their degree d_i .
2. $G' \leftarrow \emptyset$
3. $h = \Delta * 100$
4. $x = h \% N$
5. While $|V'| \leq x$ do
6. Select top $h\%$ vertices
7. End
8. End

Figure 3.5. The pseudo code of the third sampling technique (Sahu *et al.*, 2021)

In this work, experimental results are done on one dataset (Mico), in the experiments; three sampling techniques compared with the exact static algorithm (GRAMI), But Approximate GraMi algorithms have disadvantages; they work only on static graph environment, and there is no limit for the memory size.

All mentioned algorithms in the previous sections assume that graphs are not changed over time, however; nowadays the emerging graph-based applications have the dynamicity nature. Examples include social networks, where friendships are formed and removed over time; protein-to-protein interaction networks, where information in biomedical databases is updated frequently. The following section presents the frequent subgraph mining in dynamic graphs.

3.2. Frequent Subgraph Mining Algorithms for Dynamic Environment

In dynamic graph mining literature, various algorithms have been proposed using different approaches in different fields such as input and output type, increments type, graph type, graph representation, data structure, algorithmic approach, programming approach, and Base algorithm. In this section, a classification among some algorithms is held according to the following attributes, let s explain each of them:

The Increments type; as the listed algorithms are incremental algorithms, the increments can be a series of small graphs or a stream of nodes and edges.

The graph type of the input graphs can have one of the following possible types: Undirected labelled graphs, undirected graphs, directed graphs, various kinds of graph data, connected graphs, connected undirected graphs, or labelled graph.

The graph representation is considered as one of the most effective attributes on the consumption of runtime and memory. In general, the graphs can be represented by the adjacency matrix, adjacency list and canonical labelling.

The increments can have nodes/edges where each column can have, A: Addition and D: Deletion.

The data structure represents data structure that used in the application of the mentioned algorithm. It can be DFS tree, Suffix trees, dictionary data structure, DS-Tree, DS-Table, DS-Matrix or, Array with hashed based index.

The algorithmic approach shows pattern finding approach of the base algorithm where the possible values can be Apriori or Pattern growth. The Apriori based algorithms (Inokuchi, Washio and Motoda, 2000) generate candidates using breadth first strategy (BFS) and apply subgraph isomorphism testing to calculate frequencies of candidates. Pattern growth-based algorithms (Yan and Han, 2003) generate candidates based on depth first strategy (DFS). The pattern growth approach avoids the cost of generating candidates and subgraph isomorphism testing. The candidates are generated by extending frequent subgraphs starting from minimal frequent subgraphs by adding one edge at every step until they are still frequent.

In programming approach some algorithms utilize parallel programming, to take advantage of the existing multicore processor technology, or distributed, and as a result the time complexity is reduced. While other uses serial programming approach.

Base algorithm, some incremental algorithms are developed based on static ones, the base algorithms represent static algorithm with which the dynamic algorithm is extended.

Output type, the essential purpose of each algorithm is extracting a reduced set of frequent subgraphs. So, the nature of each output could be different from one algorithm to another. It can be all, approximate or closed frequent subgraphs.

As there are currently many dynamic subgraph mining approaches. Table 3.2 summarizes a literature about some recent algorithms in dynamic subgraph mining, this summary is done to facilitate visualizing the main properties of each algorithm.

Table 3.2. Algorithms for dynamic frequent subgraph mining

Algorithm	Increments type	Graph type *	Graph representation	Increments				Data structure	Algorithmic approach	Programming approach	Base algorithm	Output type	Limitations
				A		D							
				Edge	Node	Edge	Node						
span (Lakshmi and Meyyappan, 2013)	Small graphs	U L	Adjacency list	✓	✓			DFS tree	Pattern growth	Parallel	Gspan	All frequent subgraphs	Not general for all classes, it focuses on a special class of undirected labelled simple graphs, graphs with unique no labels.
Germ (Berlingerio and Bonchi, 2009)	Stream of nodes and edges	U	Canonical label	✓	✓			DFS tree	Pattern growth	Serial	Gspan	All frequent subgraphs	It is assumed that node and edge labels do not change over time.
Dynamic GREW (Borgwardt, Kriegel and Wackersreuther, 2006)	Small graphs	L	adjacency matrix	✓		✓		Suffix trees	Apriori	Serial	Grew	All frequent subgraphs	Extra overhead to identify dynamic patterns. Misses some interesting patterns.
Time-evolving Graph (Miyoshi, Ozaki and Ohkawa, 2011)	Stream of nodes and edges	U	Canonical label	✓	✓			Directed Acyclic Graph	Pattern growth	Serial	Germ	All frequent subgraphs	Increments done by addition of nodes and edges only.
StreamFSM (Ray, Holder and Choudhury, 2014)	Stream of nodes and edges	V	Canonical label	✓	✓			dictionary data structure	Pattern growth	Serial	Gspan	Approximate frequent subgraphs	Simple heuristic and applicable only to incremental streams and without provable guarantees.
Triest (Stefani et al., 2016)	Stream of edges	C U	canonical label	✓		✓		Array with hash map	Apriori	Serial		Approximate frequent subgraphs	It counts only 3-node subgraphs
IncGM+ (Abdelhamid et al., 2017)	Stream of edges	D	Canonical label	✓		✓		index structure	Pattern growth	Serial	StreamFSM, moment	All frequent subgraphs	Still needs to enumerate and track an exponential number of candidate subgraphs.
FSM in an evolving graph (Aslay et al., 2018)	Stream of edges	C U	canonical label	✓		✓		Array with hashed based index	Apriori	Serial	StreamFSM	Approximate frequent subgraphs	Consider only edges, it uses BFS to explore the graphs and may multiple candidate generation is done. Some interesting patterns can be lost.
FP from dense graph streams (Braun et al., 2014)	Small graphs	U	canonical label	✓		✓		DS-Tree, DSTable, DSMatrix	Pattern Growth	Serial	FP-Growth	All frequent pattern	Handle edges addition and deletion only
edge-based FSM from graph streams (Cuzzocrea et al., 2015)	Small graphs	U	canonical label	✓		✓		DS-Tree, DSTable, DSMatrix	Pattern Growth	Serial	FP from dense graph streams (Braun et al., 2014)	All frequent subgraphs	Only handle edges addition and deletion without update or new node increments.
IncGraphMiner (Bifet and Gavaldá, 2011)	Small graphs	V	canonical label	✓	✓			DFS tree	Pattern Growth	Serial	CloseGraph()	Closed subgraphs	It takes overhead time to summarize the pattern
WinGraphMiner (Bifet and Gavaldá, 2011)	Stream of nodes and edges	V	canonical label	✓	✓			DFS tree	Pattern Growth	Serial	CloseGraph	Closed subgraphs	It is not able to adapt to changes on the stream since the right size of the sliding window should be known in advance. Time overhead to produce patterns.
AdaGraphMiner (Bifet and Gavaldá, 2011)	Stream of nodes and edges	V	canonical label	✓	✓			DFS tree	Pattern Growth	Serial	CloseGraph	Closed subgraphs	It takes overhead time to summarize the pattern

Graph type*:

- UL: Undirected labelled graphs
- U: Undirected graphs
- D: Directed graphs
- V: Various kinds of graph data
- C: Connected graphs
- CU: Connected undirected graphs
- L: Labeled graph

Most algorithms are programmed in serial approach while very few are developed in parallel manner like span (Lakshmi and Meyyappan, 2013). Some algorithms are exact like IncGM+ (Abdelhamid *et al.*, 2017) and moment (Chi *et al.*, 2004), but keeping track of all the possible changes in the graph is subject to combinatorial explosion, thus, is highly challenging (Aslay *et al.*, 2018). To overcome this challenge; some algorithms are proposed for the approximate purpose e.g. (Aslay *et al.*, 2018). Enumerate and extract the emerged subgraphs is a drawback of some existing algorithms. In the approximate algorithms there is a trade-off between time and accuracy.

Exact algorithms that focus on finding all subgraphs in the data require high execution times and high memory consumption. Therefore, for faster results users are willing to trade-off accuracy whenever approximate results can serve the purpose. Those approximate algorithms use different sampling approaches. The basic idea in approximation is to execute the exact algorithm on a small portion of the data (samples) (Anand Padmanabha Iyer, Zaoxing Liu, Xin Jin, Shivaram Venkataraman, Vladimir Braverman, Ion Stoica, 2018). Sampling is done by selecting a representative subset of the original graph where the idea is to reduce graph size while keeping the characteristics of the original graph (Wang *et al.*, 2011). Graph sampling provides an efficient, yet inexpensive solution for social network analysis where the graph size is huge (Wang *et al.*, 2011). Sampling can be used in graph analysis in applications such as security, high performance computing, etc (Zhang *et al.*, 2017). Also, survey hidden population in sociology; scale down Internet AS graph, graph sparsification, etc. (Stutzbach *et al.*, 2006).

There are two types of sampling algorithms in data streams; simple random sampling (SRS) (Yates, Moore and Starnes, 2002) and reservoir sampling (RS) (Vitter, 1985). In SRS, a sample is chosen by picking each item in the data stream with an equal probability of being selected. The inclusion probability is provided as an input parameter to the sampling algorithm. Consider a $p = 0.5$, then each item in the data stream has half the chance to be included in or excluded from the sample. SRS is extremely simple, however one downside of SRS is that the size of the sample grows along with size of the data stream (Anis and Nasir, 2018). On the other hand RS (Vitter, 1985) is a fixed-size randomized sampling scheme, which maintains a fixed-size uniform sample of the data stream. The size of the sample is provided as the input parameter. Random pairing (Gemulla, Lehner and Haas, 2006) is a fully dynamic algorithm as an example for

reservoir sampling, that compensate for item deletions using the future addition (Anis and Nasir, 2018).

When the literature of dynamic subgraph mining is analysed, it is observed that most of the algorithms are devised to produce exact output like span (Lakshmi and Meyyappan, 2013), Germ (Berlingerio and Bonchi, 2009), Dynamic GREW (Borgwardt, Kriegel and Wackersreuther, 2006), Time-evolving Graph (Miyoshi, Ozaki and Ohkawa, 2011), IncGM+ (Abdelhamid *et al.*, 2017), FP from dense graph streams (Braun *et al.*, 2014) and edge-based FSM from graph streams (Cuzzocrea *et al.*, 2015). However, keeping track of all the possible changes in the graph is subject to combinatorial explosion. On the other hand, there is a few work for approximate solutions like (Ray, Holder and Choudhury, 2014) (De Stefani *et al.*, 2016) (Aslay *et al.*, 2018). **These** approaches use simple heuristics and do not provide any correctness guarantee. Both solutions in (De Stefani *et al.*, 2016) and (Aslay *et al.*, 2018) use sampling technique based on the method which is proposed in (Vitter, 1985), this method is a randomized sampling schema. The algorithm in (De Stefani *et al.*, 2016) relies on sampling edges, while the algorithms in (Aslay *et al.*, 2018) are based on sampling subgraphs in order to gain more accuracy. These algorithms have limitations; the limitations are trade-off between time and accuracy. SR and OSR (Aslay *et al.*, 2018) are more accurate than Triest (De Stefani *et al.*, 2016) sacrificing time and space efficiency, while Triest (De Stefani *et al.*, 2016) is faster with the cost of low accuracy. Solutions that minimize time and space consumption while maximizing the accuracy at the same time are still needed.

In the following two subsections, exact and approximate approaches are introduced and discussed.

3.2.1. Exact Algorithms

The Span (Lakshmi and Meyyappan, 2013) is based on gSpan. It focuses on a special class of undirected labelled simple graphs, graphs with unique no labels. Aims reduce the time complexity, using parallel programming. If graph dataset can fit in main memory, the proposed method can be applied directly; the two techniques, DFS lexicographic order and minimum DFS code, introduced in gSpan are the best, which form, a novel canonical labeling system, to support DFS search. But still the problem of finding minimum DFS

code used in gSpan is also NP- complete. The proposed algorithm addresses this issue by using a modified DFS representation. It retains all the advantages of gSpan, while taking advantage of the multi core processing technology by using the concept of parallel programming to improve the performance of the algorithm. The number of duplicate graphs generated may be comparatively little more than gSpan algorithm, as mining of sub graphs from frequent single edge graphs are done in parallel. Span is quadratic but not general for all classes.

The GERM (Berlingerio and Bonchi, 2009) introduced Graph Evolution Rule Miner (GERM), a novel type of frequency based pattern that describe the evolution of large networks over time, at a local level. The input for this approach is a sequence of snapshots of an evolving graph, the main purpose is to mine the rules that describe the local changes in it. This approach uses the support based on minimum image to extract patterns which frequency is greater than a minimum support threshold. After that, graph-evolution rules are extracted from frequent patterns that satisfy a given minimum confidence constraint, the rules extraction framework is similar to it in classical rule mining. Experiments are done on four large real-world networks (two social networks, and two co-authorship networks), using different time granularities. Experiments approve feasibility and the utility of a framework. The limitations of GERM: it is designed for undirected graphs, nodes and edges are only added and never deleted. It assumed that node and edge labels do not change over time.

The Dynamic GREW (Borgwardt, Kriegel and Wackersreuther, 2006) investigates how pattern mining on static graphs can be extended to time series of graphs. Specifically, it handles dynamic graphs with edge insertions and edge deletions over time. They define a frequency and provide algorithmic solutions for finding frequent dynamic subgraph patterns. Existing subgraph mining algorithms can be easily integrated into this framework to make them handle dynamic graphs. Experimental results in the paper on real-world data confirm the practical feasibility of proposed approach, the limitations of Dynamic GREW are; it assumes that the input dynamic graph has a fixed set of nodes, and edges are inserted and deleted over time. Also, there is an extra overhead to identify dynamic patterns. It misses some interesting patterns.

In (Bifet and Gavaldà, 2011), the first work on close stream while only two frequent close graph algorithms on static graphs are introduced. Bifet et al. proposed new method for mining frequent closed subgraphs. The method is IncGraphMiner, it works on frequent weighted closed graph mining. this method works on coresets of closed

subgraphs, compressed representations of graph sets, and maintain these sets in a batch-incremental manner, it handles the potential concept drift. The proposed algorithm is based on close graph which takes time overhead to summarize the patterns.

The IncGM+ (Abdelhamid *et al.*, 2017) is a fast incremental approach for continuous frequent subgraph mining on a single large evolving graph. It adapts the notion of “fringe” to the graph context, which is the set of subgraphs that are on the border between frequent and infrequent subgraphs. IncGM+ maintains fringe subgraphs and utilize them in the search space pruning. In order to increase the efficiency, an efficient index structure is proposed to maintain selected embeddings, with minimal memory overhead. These embeddings are employed to avoid subgraph isomorphism operations. Furthermore, the proposed system supports batch updates. Experiments are done using large real-world graphs, it verifies that IncGM+ outperforms existing algorithms by up to three orders of magnitude, scales to much larger graphs, and consumes less memory. The limitation of IncGM+ that it still needs to enumerate and track an exponential number of candidate subgraphs.

The algorithm In (Miyoshi, Ozaki and Ohkawa, 2011) handles the problem of mining frequent patterns and valid rules representing graph evolutions (structural changes) in a graph with time information. They propose an efficient technique for extracting representative patterns and rules, they use graph-based summarization of discovered rules. This done by using certain measures provided by the summary, so it is expected to find more interesting information which are difficult to be discovered by the traditional support and confidence measures. Proposed algorithm based on gSpan and Germ, it differs from gSpan that it handles single graph input and work on graph patterns have time points, it differs from Germ that proposed method handle multi edges.

The FRISSMiner in (Inokuchi and Washio, 2012) defines subgraph subsequence class called an “induced subgraph subsequence” to enable the efficient mining of a complete set of frequent patterns from graph sequences containing large graphs and long sequences. In addition, it proposes an efficient method for mining frequent patterns, called “FRISSs (Frequent Relevant, and Induced Subgraph Subsequences)”, from graph sequences. The fundamental performance of the proposed method is evaluated using artificial datasets, and its practicality is confirmed by experiments using a real-world dataset.

In (Bifet and Gavaldà, 2011), the first work on close stream. Bifet et al. proposed frameworks are for studying graph pattern mining on time-varying streams. Two new

methods for mining frequent closed subgraphs are presented. The methods are WinGraphMiner and AdaGraphMiner, it works on frequent weighted closed graph mining. All methods work on coresets of closed subgraphs, compressed representations of graph sets, and maintain these sets in a batch-incremental manner but use different approaches to address potential concept drift. The above three algorithms are based on close graph which takes time overhead to summarize the patterns.

3.2.2. Approximate Algorithms

The StreamFSM (Ray, Holder and Choudhury, 2014) discovers the frequent subgraphs in a graph, represented by a stream of labeled nodes and edges. In this model, updates to the graph arrive in the form of batches that contain new nodes and edges. Proposed method continuously reports the frequent subgraphs that are estimated to be found in the entire graph as each batch arrives. It is evaluated using five large dynamic graph datasets: the Hetrec 2011 challenge data, Twitter, DBLP and two synthetic. It is evaluated against two popular static large graph miners, i.e., SUBDUE and GERM. Experimental results show that it can find the same frequent subgraphs as a non-incremental approach applied to snapshot graphs, and in less time. The drawback of the StreamFSM algorithm: In terms of several parameters that have to be tuned in order to get the optimal performance in terms of time and accuracy/interestingness of results. Also, it assumes that we have access to the entire graph as the graph grows. This assumption will not work in a real world streaming scenario. (Ray, Holder and Choudhury, 2014). Also it only handles increments with additions.

SR and OSR are two dynamic approximate algorithms (Aslay *et al.*, 2018), Both algorithms use reservoir sampling technique (RS) as introduced in (Vitter, 1985). RS is a fixed size randomized sampling technique; it maintains a fixed-size uniform sample of the data. The sample size is assigned as an input parameter. The algorithms initialize with a fixed size input array, that initially gets filled by the items in the input data gradually. Once the maximum sample size is reached or the reservoir is filled, each new item (i) is added to the sample with probability $1/i$ by replacing it with a randomly selected item from the sample.

The addition of an edge affects only the subgraphs in the local neighbourhoods up to specified neighbourhood. A uniform sample of subgraphs is maintained by iterating through the subgraphs in the neighbourhood of the newly inserted edge. Standard reservoir sampling is used as follows:

If the sample size less than fixed memory size, then the new subgraph is added to the sample. Otherwise, if the sample size greater than fixed memory size then a subgraph is removed randomly from the sample to insert the new one.

Triest algorithm (De Stefani *et al.*, 2016):

Triest counts triangles in incremental streams with fixed memory size, it uses standard reservoir sampling (Vitter, 1985) to maintain the edge sample:

The standard reservoir sampling is used as follows:

- If the sample size less than fixed memory size, then the new edge is added to the sample. Otherwise,
- if the sample size greater than fixed memory size then an edge is removed randomly from the sample to insert the new one.

The drawback of this algorithm that edge deletion from reservoir is done randomly when reservoir is full, by this random edge deletion some important patterns might be lost.

Example

Figure 3.6 illustrates an example of Triest after adding a new coming edge (FG), while the reservoir is full. Given an input graph of coming edges Figure 3.6(a), it is supposed that the maximum allowed size of the reservoir is assigned to a value of “6”. When the first five coming edges (AB, CX, DE, XY, XZ, XE) are received, they are inserted directly into the edge reservoir Figure 3.6(b), this insertion into the reservoir is done directly since it is not full. The edges in the reservoir are presented in a hash map as Figure 3.6(c). Now when a new edge (FG) is coming, the reservoir size is checked if it still has an empty space, at this time; it is found that it is full, in this case one edge from reservoir should be deleted, and replaced by the new edge (FG), the selection of a candidate edge is done randomly, in this example; the edge (DE) is selected as a candidate edge to be deleted from reservoir, and replaced with new edge (FG). After inserting the new edge (FG) in the edge reservoir, the node degree list is updated by adding the nodes

F and G with a degree of 1, since they are newly appeared, also the hash map is updated by adding F and G to it.

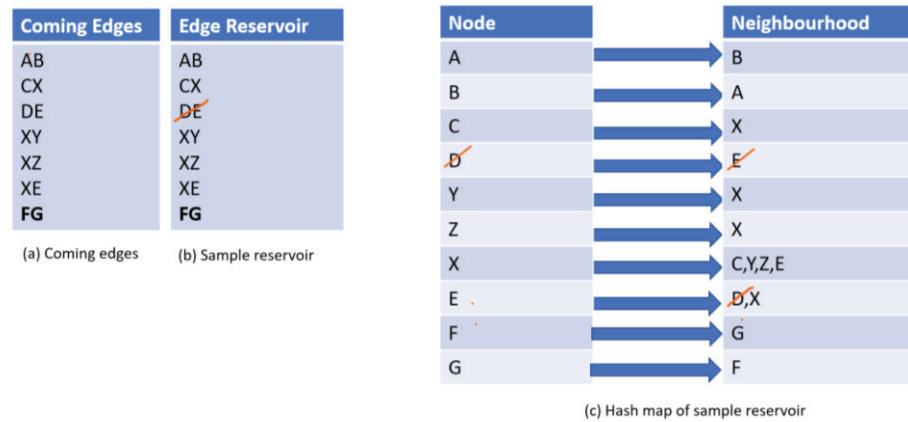


Figure 3.6. Example of Triest algorithm after adding a new coming edge (FG), when the sample reservoir is full

Example

Figure 3.7 shows an example of SR or OSR after adding a new coming edge (FG), while the reservoir is full. Suppose that an input graph of coming edges as given in Figure 3.7(a), let the maximum allowed size of the subgraph reservoir is assigned to a value of “5”. When the first coming edges are received, a three nodes subgraph is formed, then it is inserted into the subgraph reservoir Figure 3.7(b), this insertion into the reservoir is done directly since it is not full. The nodes of the subgraph in the reservoir are presented in a hash map as Figure 3.7(c). Now when a new edge (FG) is coming, the three nodes subgraph (FGB) is formed, then the reservoir size is checked if it still has an empty space or not, at this time; it is found that it is full, so one subgraph from reservoir should be deleted, and replaced by the new subgraph (FGB), the selection of a candidate subgraph is done randomly, in this example; the subgraph (XZE) is selected as a candidate subgraph to be deleted from reservoir, and replaced with new subgraph (FGB). After inserting the new subgraph (FGB) in the edge reservoir, the hash map is updated by adding F and G to it.

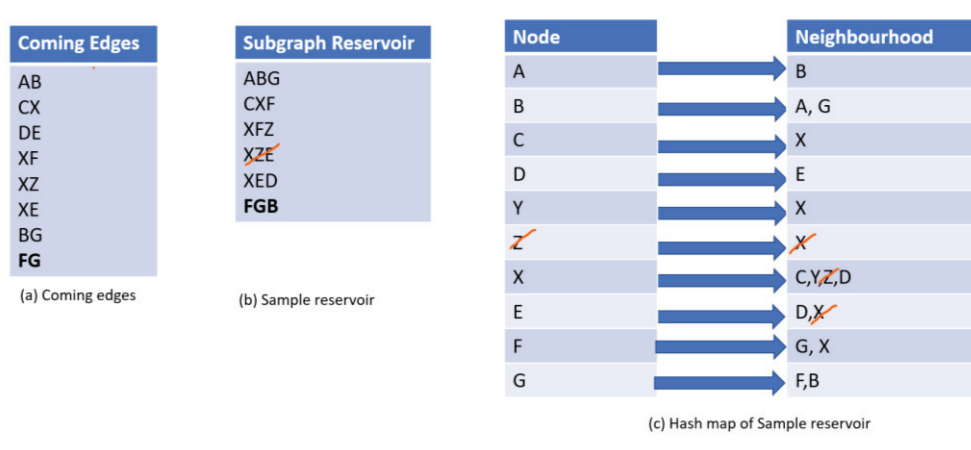


Figure 3.7. Example of SR or OSR algorithm after adding a new coming edge (FG), when the sample reservoir is full

Triest, SR and OSR are dynamic approximate algorithms, they use a fixed specified memory size (reservoir sampling), to keep this size of memory; random edge deletion from reservoir is employed when the sample reservoir is full, but this random edge deletion has a drawback; high connectivity edges might be deleted, as a result; these types of edge deletions can reduce recall. There are another limitation of SR and OSR algorithms, that they keep subgraphs in the reservoir instead of edges, which can increase the recall, but this expenses a high execution time, this time can be closed to the execution time of the exact algorithm.

CHAPTER 4

CONTROLLED RESERVOIR SAMPLING

In this chapter, three novel algorithms for approximate frequent subgraph mining on dynamic graphs are introduced. The first one is the Controlled Reservoir Sampling Algorithm with Unlimited heap size (UCRS), the second one is the Controlled Reservoir Sampling Algorithm with Limited heap size (LCRS), the third one is Maximum Controlled Reservoir Sampling (MCRS). All introduced algorithms use reservoir sampling technique (RS) as introduced in (Vitter, 1985). RS is a fixed size randomized sampling technique; it maintains a fixed-size uniform sample of the data. The sample size is assigned as an input parameter. The algorithms initialize with a fixed size input array, that initially gets filled by the items in the input data gradually. Once the maximum sample size is reached or the reservoir is filled, each new item (i) is added to the sample with probability $1/i$ by replacing it with a randomly selected item from the sample reservoir. But this random edge deletion has a drawback; high connectivity edges might be deleted, as a result; these types of edge deletions can reduce recall. For this reason, UCRS and LCRS are proposed in this work, both UCRS and LCRS use controlled deletion for minimum node degree edge from sample reservoir and deleting nodes from heap reservoir if their degrees are 1, the advantage of this controlled edge deletion is to keep more connected edges in the sample reservoir, by doing so; recall is expected to be increased. On the other hand, as a kind of heuristic, the third algorithm MCRS is proposed, it works in a similar manner to UCRS and LCRS, but the main difference is in determining the candidate edge to be deleted from sample reservoir and deleting nodes that has degree 1 from heap reservoir, whenever the whole reservoir is full, in this algorithm the candidate edge is the edge with maximum degree of its nodes, by this way, the high connectivity edges are deleted from the sample reservoir, as a result, recall is expected to be decreased. So, the results of MCRS motivate the need for the advantages of UCRS and LCRS algorithms.

UCRS, LCRS and MCRS methods use reservoir sampling technique with a modification. In all algorithms, random edge deletion from sample reservoir is replaced

by a controlled edge and node deletion from whole reservoir, in the case when the whole reservoir is full. To achieve the controlled edge deletion from sample reservoir, an additional minimum heap data structure is added to edge sampling schema, this heap is called heap reservoir. The whole reservoir keeps the edges of the sample reservoir together with the nodes of the heap reservoir. This heap reservoir contains the nodes in ascending/descending order of their degrees; this order helps to select the edge connecting the low/high degree nodes in deletion instead of deleting a random edge.

In the following sections first, our novel algorithm (UCRS) is introduced with motivating examples. Second, a modified version of UCRS is introduced, which is called (LCRS), is presented with illustrating examples. Third, MCRS algorithm is introduced with examples. And in the last section, an example of deleting an edge in UCRS, LCRS, MCRS and random algorithms is illustrated.

4.1. Controlled Reservoir Sampling Algorithm with Unlimited Heap Size (UCRS)

Controlled Reservoir Sampling with Unlimited heap (UCRS) algorithm is modified version of reservoir sampling, where an additional heap data structure is employed to manage the node degrees, which is called heap reservoir. Degree of a node indicates the number of connections of the node. When an edge deletion is required in other words when the fixed size reservoir is full (whole reservoir size), edge with lowest degree node is selected and removed from the sample reservoir, and if node degrees of this edge is 1, they are removed from heap reservoir, otherwise their degrees are decreased by 1. The idea is not to lose high degree nodes those of which might have more impact on accuracy. More accuracy is expected in sampling since more connected edges (higher degree nodes) remain in the sample reservoir and heap reservoir, while less connected nodes are deleted.

The pseudo code of UCRS is shown in Figure 4.1. This code is designed to manage the insertion process of new edges in controlled reservoir sampling. The input for UCRS is an incremental graph data d in an evolving graph environment. UCRS keeps sample reservoir S of edges and heap reservoir up to (M) from the input graph stream, where M is a positive integer parameter, it indicates fixed memory size in terms of nodes

in an abstract way. This size represents the maximum whole reservoir in nodes, which is equal to size of edges (2 nodes/edge) in the sample reservoir and the nodes that are in the heap reservoir in total. The output of the algorithm is a sample of the incoming edges (sample reservoir), which represents the characteristics of the whole graph. This sample is used to search for an approximate frequent pattern instead of searching whole graph.

Controlled Reservoir Sampling UCRS

Input: incremental graph data d , integer M /* M : maximum whole reservoir size in terms of nodes
Output: updated Sample, updated Heap

1: $s \leftarrow \emptyset, h \leftarrow \emptyset, i \leftarrow 0$, /* s : sample size, h : minimum heap reservoir size,

2: procedure addEdge(u, v)
3: for each edge (u, v) from d do
4: $i \leftarrow i+1$
5: if sampleEdge(u, v) then
6: addToSample (u, v)

7: procedure sampleEdge (u, v)
8: if sampleSize < M then /* whole reservoir is not full
9: return True
10: else
11: $\minEdge(x, y) \leftarrow \minEdge()$
12: removeFromSample (x, y)
13: return True

14: procedure minEdge ()
15: $x \leftarrow$ source node with lowest node degree
16: $y \leftarrow$ destination node with lowest node degree
 among neighbours of (x)
17: return edge (x, y) /* candidate edge for deletion is chosen

18: procedure addToSample (u, v)
19: $s \leftarrow s + \{(u, v)\}$
20: addToHeap (u, v)

21: procedure removeFromSample (u, v)
22: $s \leftarrow s - \{(u, v)\}$
23: removeFromHeap (u, v)

24: procedure addToHeap (u, v)
25: $h \leftarrow h + \{(u, v)\}$

26: procedure removeFromHeap (u, v)
27: $h \leftarrow h - \{(u, v)\}$

Figure 4.1. Pseudo code of the UCRS algorithm

UCRS algorithm that is represented in Figure 4.1 works as follows; (Line 2-6) when new increment d arrives; each edge (u, v) in the increment is added to the sample by `addToSample (u,v)` procedure, to do this; edge sampling is applied by `sampleEdge (u, v)` procedure (Line 7-13). This procedure works as follows:

- If the sample size is less than (M) , then the new edge is added to the sample directly. Otherwise,
- If the sample size is greater than or equal to (M) , then an existing edge is removed from the sample to insert the new one.

The edge that should be removed is the edge that has the minimum node degree (Line 14-17), it is determined from the min heap, which classifies the nodes according to their node degrees. First, it detects the node with the minimum degree as the source node. Second, from the neighbours of the selected source node; the neighbour node with the minimum node degree is selected as the destination node. The result of the previous two steps finds an edge connecting the minimum node degrees. This edge is the candidate edge to be deleted from the sample reservoir, to replace it with a new incoming edge in the sample reservoir.

The required updates are done on the sample reservoir (Line 18-23). This update can be addition to the sample reservoir or deletion from the sample reservoir, the addition to the sample reservoir is done through the procedure `addToSample (u, v)`, while the edge deletion from the sample reservoir is done using the procedure `removeFromSample (u, v)`.

In the same manner heap reservoir is updated either by node addition or by node deletion (Line 24-27). In UCRS there is no limit in the size of the heap reservoir, all incoming nodes are added to the heap reservoir if they do not exist before. Heap reservoir contains all the nodes of the sample reservoir.

Illustrating Examples

Example 1

The example in Figure 4.2 shows the deletion process of an edge when the whole reservoir is full as it is done by minimum controlled edge deletion in (UCRS) and random edge deletion as in Triest (De Stefani *et al.*, 2016), SR (Aslay *et al.*, 2018) and OSR

(Aslay *et al.*, 2018). In another words an example of edge deletion with random edge deletion and controlled edge deletion is represented.

Suppose we have an original graph Figure 4.2(a), the left table in the figure shows the nodes and their degrees of the original graph. By applying random edge deletion, suppose the candidate edge is: (W, U), when it is removed; three triangle patterns of the original graph are lost as shown in Figure 4.2(b). On the other hand, in controlled edge deletion method that is used in UCRS and LCRS, the edge (V, Y) is a candidate to be removed since V is the lowest degree node and Y is the lowest degree neighbour of V. When the edge (V, Y) is removed from the original graph, only one triangle patterns of six is lost as illustrated in Figure 4.2(c). As a result, less patterns are affected by controlled deletion. The table below the figure shows, the number of lost triangles, the number of remained triangles, and the percentage lost of random and minimum controlled edge deletion processes, respectively.

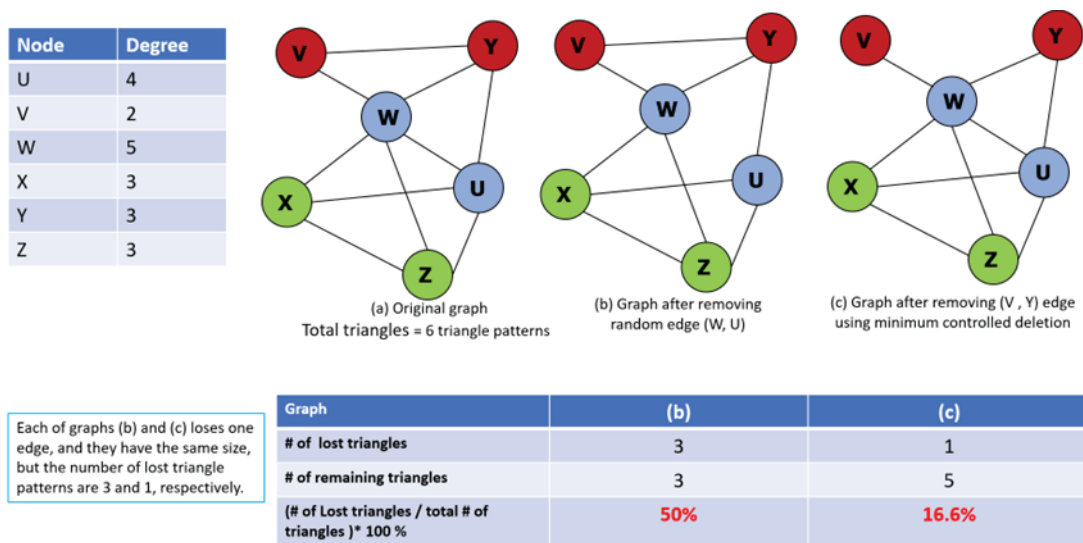


Figure 4.2. Deleting edge by random edge deletion and minimum controlled edge deletion

Example 2

The example in Figure 4.3 shows the addition process of the coming edges when the whole reservoir is not full, as it is done by UCRS algorithm. Suppose the maximum allowed size of the whole reservoir is assigned to a value of “6”. Given an input graph of coming edges Figure 4.3(a), when those coming edges are received, they are inserted in

the sample reservoir Figure 4.3(b), this insertion is done into the sample reservoir as long as the whole reservoir is not full (it doesn't reach the maximum allowed size), in this example; the whole reservoir is not full, and all coming edges are inserted directly in it. The nodes of the edges in the sample reservoir are enrolled in the node degree list (heap reservoir) Figure 4.3(c), this list represents the nodes and their degrees, however the degree of each node is considered as the number of connections of each node in the reservoir, this node degrees list is presented in UCRS algorithm by the minimum heap data structure, where the nodes are ordered according to their degrees in ascending order, while the root node holds the node with the minimum degree among all the other nodes in the list. The edges in the sample reservoir are presented in a hash map as Figure 4.3(d), as seen in the hash; the nodes are on the left list and the neighbours of each node are on the right list.

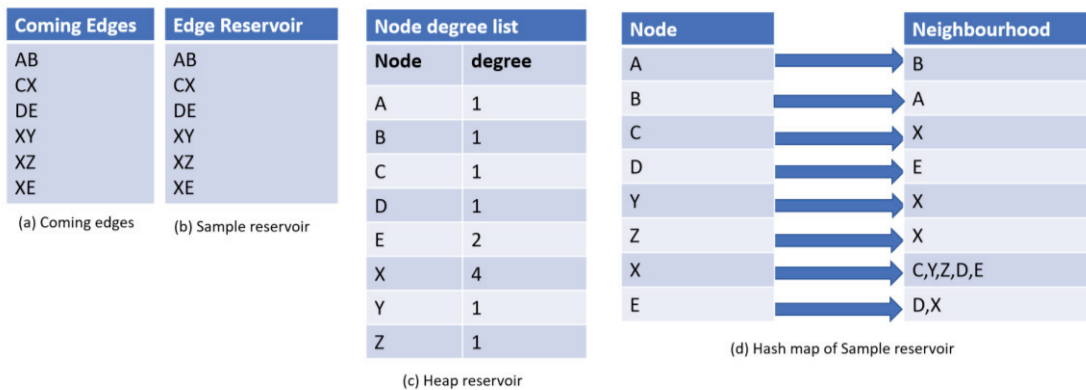


Figure 4.3. Example of the UCRS algorithm when whole reservoir (sample and heap reservoirs) is not full

Example 3

Figure 4.4 illustrates an example of UCRS after adding a new coming edge (FG), while the whole reservoir is full. Given an input graph of coming edges Figure 4.4(a), it is supposed that the maximum allowed size of the whole reservoir is assigned to a value of “6”. When the first five coming edges (AB, CX, DE, XY, XZ, XE) are received, they are inserted directly into the sample reservoir Figure 4.4(b), this insertion into the sample reservoir is done directly since the whole reservoir is not full. The nodes of the edges in the sample reservoir and their degrees are listed in the node degree list (heap reservoir)

Figure 4.4(c), where the degree of each node represents the number of connections of each node with other nodes in the sample reservoir, the node degree list is implemented in UCRS algorithm by the minimum heap. The edges in the sample reservoir are presented in a hash map as Figure 4.4(d). Now when a new edge (FG) is coming, the whole reservoir size is checked if it still has an empty space, at this time; it is found that it is full, in this case one edge from sample reservoir should be deleted, and replaced by the new edge (FG), but the selection of a candidate edge couldn't be done randomly in UCRS, the candidate edge should be the edge with the minimum node degree of its source and destination nodes, in this example; the node (A) is marked as a source node, since it has the minimum node degree, then from the neighbours of node (A), the neighbour node with the minimum node degree is selected as a destination node, since there is only one neighbour node which is (B), so it selected as a destination node, now the edge (AB) is formed as a candidate edge to be deleted from sample reservoir, and replaced with new edge (FG), as a consequent steps of deleting the edge (AB), the nodes A and B are deleted from node degree list if they have degree of 1, or their degrees are decremented by 1 if their degrees are greater than 1. Also, the nodes are deleted from the hash map if they don't have connections with any other nodes of the sample reservoir. After inserting the new edge (FG) in the sample reservoir, the node degree list is updated by adding the nodes F and G with a degree of 1, since they are newly appeared, also the hash map is updated by adding F and G to it.

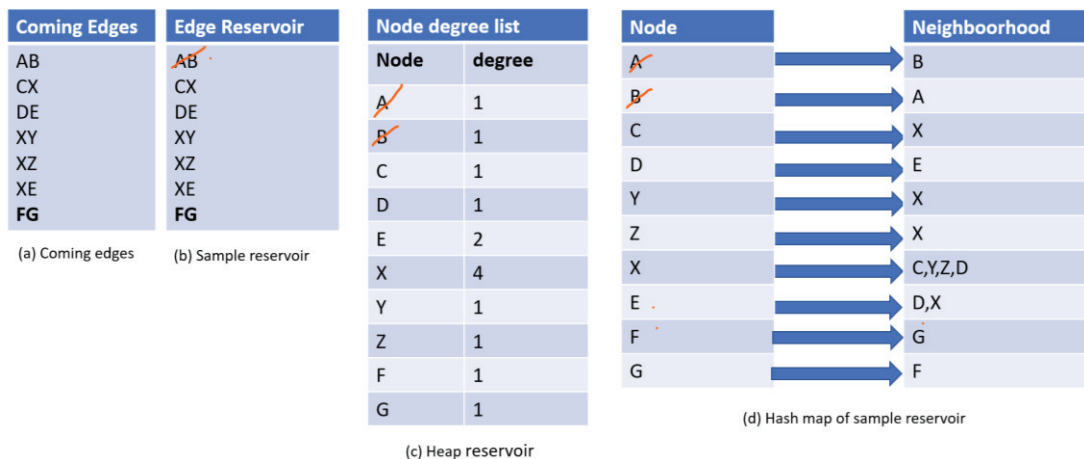


Figure 4.4. Example of UCRS after adding a new coming edge (FG), while the whole reservoir (sample and heap reservoirs) is full

4.2. Controlled Reservoir Sampling Algorithm with Limited Heap Size (LCRS)

In UCRS algorithm heap keeps all the degrees of the nodes that are presented in the sample. It is noticed from the empirical results of the UCRS algorithm; the heap size becomes large especially with higher density datasets. As a result, space left for the sample decreases. This results in lower number of retrieved patterns, so if the heap is pruned, the efficiency of the UCRS algorithm is expected to increase in terms of number of patterns, execution time and memory usage.

A modified version of UCRS is introduced Controlled Reservoir Sampling with Limited heap (LCRS). As the name indicates in LCRS the heap size is minimized. If heap size is minimized or limited, then it will be possible to store more edges in the sample. Increased sample size leads to larger number of patterns that are retrieved, lower execution time than UCRS due to the management of smaller heap.

LCRS algorithm that is represented in Figure 4.5 works as follows; (Line 2-6) when new increment d arrives; each edge (u, v) in the increment is added to the sample by `addToSample (u,v)` procedure, to do this; edge sampling is applied by `sampleEdge (u, v)` procedure (Line 7-13). This procedure works as follows : If the sample size is less than (M) , then the new edge is added to the sample directly. On the other hand, if the sample size is greater than or equal to (M) , then an existing edge is removed from the sample to insert the new one.

The edge that should be removed is the edge that has the minimum node degree (Line 14-17), it is determined from the min heap, which classifies the nodes according to their node degrees. First, it detects the node with the minimum degree as the source node. Second, from the neighbours of the selected source node; the neighbour node with the minimum node degree is selected as the destination node. The result of the previous two steps finds an edge connecting the minimum node degrees. This edge is the candidate edge to be deleted from the sample reservoir, to replace it with a new incoming edge in the sample reservoir.

The required updates are done on the sample (Line 18-23). This update can be addition of an edge to the sample reservoir or deletion of an edge from the sample reservoir, the addition to the sample is done through the procedure `addToSample (u, v)`,

while the deletion from the sample reservoir is done using the procedure `removeFromSample (u, v)`.

By the same way, heap reservoir is updated either by node addition or by node deletion (Line 24-31). In LCRS there is a limit in the size of the heap reservoir, so the addition of nodes to the heap reservoir is done only if its size is doesn't exceed the maximum allowed node capacity of the heap reservoir.

Controlled Reservoir Sampling LCRS

Input: incremental graph data d , integer M /* M : maximum whole reservoir size in terms of nodes
Output: updated Sample, updated Heap

1: $s \leftarrow \emptyset, h \leftarrow \emptyset, i \leftarrow 0, x \leftarrow \emptyset, \text{maxHeapSize} \leftarrow \emptyset$ /* s : sample size, h : minimum heap size, x : maximum limited heap size,

2: procedure `addEdge((u, v))`
3: for each edge (u,v) from d do
4: $i \leftarrow i+1$
5: if `sampleEdge(u, v)` then
6: `addToSample (u, v)`

7: procedure `sampleEdge (u, v)`
8: if `sampleSize < M` then /* whole reservoir is not full
9: return True
10: else
11: $\text{minEdge}(x,y) \leftarrow \text{minEdge}()$
12: `removeFromSample (x, y)`
13: return True

14: procedure `minEdge ()`
15: $x \leftarrow$ source node with lowest node degree
16: $y \leftarrow$ destination node with lowest node degree among neighbours of (x)
17: return edge (x, y) /* candidate edge for deletion is chosen

18: procedure `addToSample (u, v)`
19: $s \leftarrow s + \{(u, v)\}$
20: `addToHeap (u, v)`

21: procedure `removeFromSample (u, v)`
22: $s \leftarrow s - \{(u, v)\}$
23: `removeFromHeap (u, v)`

24: procedure `addToHeap (u, v)`
25: if `maxHeapSize < x` do /* there is space in the heap
26: $h \leftarrow h + \{(u, v)\}$

27: procedure `removeFromHeap (u, v)`
28: $h \leftarrow h - \{(u, v)\}$

Figure 4.5. Pseudo code of the LCRS algorithms

Illustrating Examples

Example 1

The example in Figure 4.6 illustrates an example of the LCRS algorithms. It shows the addition process of the coming edges when the sample reservoir is not full, and the heap reservoir is not full also. In this situation LCRS algorithm behaves in the same manner as UCRS algorithm.

Suppose we have an input graph of coming edges Figure 4.6(a), suppose the maximum allowed whole reservoir size is assigned to 6, and the maximum allowed node capacity of the heap reservoir is assigned to 6. When the coming edges are received, they are inserted in the sample reservoir Figure 4.6(b), at each time of inserting a new edge, the size of the whole reservoir is checked, to make sure if it is still having a free space to insert the new coming edge. In this example, the 6 coming edges (AB, BC, CX, DX, DE, AC, DX) are inserted directly one by one into the sample reservoir, since it doesn't exceed the maximum allowed size (6). After inserting each edge in the sample reservoir; the two nodes of each inserted edge are enrolled in the node degree list if they aren't enrolled before, if they are already exist in the degree list (heap reservoir), their degrees are incremented by one as shown in Figure 4.6(c), this list represents the nodes and their degrees, while the degree of each node is the number of connections of each node in the sample reservoir, before insertion of the nodes and their degrees in node degree list is done as long as it doesn't exceed the maximum allowed size of the list (maximum heap reservoir size). The edges in the sample reservoir are presented in a hash map as Figure 4.6(d).

Let's take an example of inserting the coming edge (DX), first, the size of sample reservoir is checked, the sample reservoir size now is equal to 5, so there is a chance to insert DX directly into the sample reservoir Figure 4.6(a), then the size of the node degree list is checked also, and it is found 6, so it's size is equal to the maximum allowed size of the heap reservoir (node degree list), however the both nodes of the edge (DX) are already exist before in the heap reservoir, so the degrees of the nodes D and X are just needed to be incremented by one as in Figure 4.6(b). The hash map in Figure 4.6(c) is updated, in the map, node D is added as a neighbour of node X, and node X is added as a neighbour of node D.

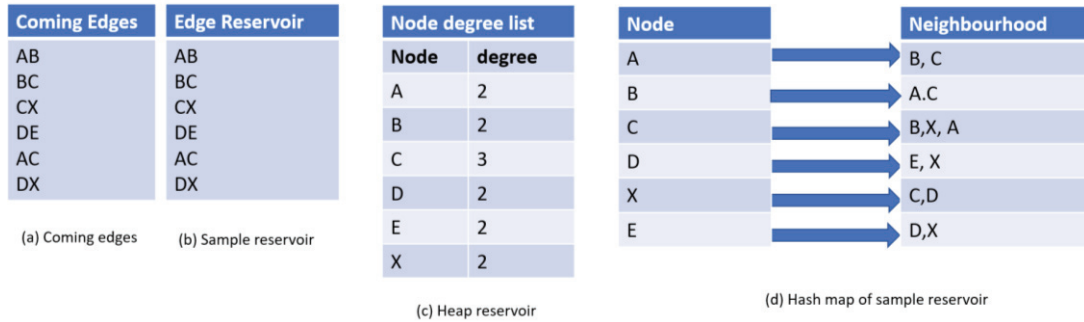


Figure 4.6. Example of the LCRS algorithm when whole reservoir (sample and heap reservoirs) is not full

Example 2

Figure 4.7 illustrates an example of the LCRS algorithms. The example is shown when the sample reservoir is full, and the heap reservoir reaches its maximum allowed limit. The example in this figure is based on the previous example of Figure 4.6. But in here when a new edge (FG) comes. Given an evolving graph, the incoming edges are listed as in Figure 4.7(a).

Suppose the maximum allowed size of the sample reservoir is 6 edges, and the maximum allowed size for the heap is assigned to 6 nodes. When the coming edges are received, they are inserted in the sample reservoir if it is not full (Figure 4.7(b)), their new nodes are inserted in the node degree list if they are not in the list, otherwise their degrees are incremented. The records of these coming nodes are updated in the hash map also. In the example of Figure 4.7; the first six coming edges (AB, BC, CX, DX, DE, AC, DX) are inserted directly one by one into the sample reservoir, since the sample reservoir is not full, and the heap reservoir is not full also. On the other hand, when the edge (FG) comes, the sample reservoir size reaches the maximum allowed size (6), so it is full, as a result, one edge from sample reservoir should be removed, this edge is the edge with the minimum degrees of its two nodes, so the edge AB is the candidate edge for removal, with removing the edge AB from sample reservoir, the counts of the nodes A and B in heap reservoir are decreased, and it is removed from the hash map of the sample graph. Now, there is a space in the sample reservoir to insert the new edge (FG), after that, the size of heap (node degree list) is checked, its size (6) so it is full, and the new nodes F and G

will not be added to the list Figure 4.7(c). The hash map is updated by adding F and G to it Figure 4.7(d).

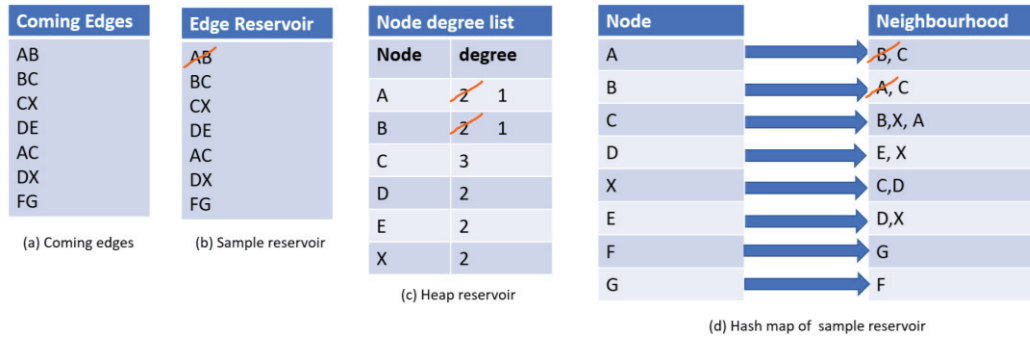


Figure 4.7. Example of the LCRS algorithms, adding a coming node (FG) while the whole reservoir (sample reservoir and heap reservoir) is full

4.3. Maximum Controlled Reservoir Sampling (MCRS) Algorithm

A modified version of controlled reservoir sampling is introduced Maximum Controlled Reservoir Sampling (MCRS). As the name indicates, in MCRS, when the whole reservoir is full, an edge should be deleted to be replaced by a new edge, this candidate edge is an edge with maximum node degrees of its source and destination, while in UCRS the candidate edge for deletion is an edge with minimum node degrees of its source and destination. It is very similar to UCRS, but instead of deleting edges with minimum node degrees, it deletes edges with maximum node degrees.

The pseudo code of MCRS algorithm is represented in Figure 4.8. It works as follows; (Line 2-6) represents the addition of an edge, such that when new increment d arrives; each edge (u, v) in the increment is added to the sample, this addition is done by the procedure $\text{addToSample}(u, v)$, to do this; edge sampling is done by $\text{sampleEdge}(u, v)$ procedure (Line 7-13). There are two case for this procedure to work, the cases are as follows:

- If the sample size is less than (M) , then the new edge is added to directly to the sample. Otherwise,

- If the sample size is greater than or equal to (M) , then an existing edge is removed from the sample to insert the new one.

The candidate edge for removal is the edge that has the maximum node degree (Line 14-17), this edge is detected from the heap, that keeps the nodes according to their node degrees. This detection is done as follows: First, it determines the node with the maximum degree as the source node. Second, from the neighbours of the determined source node; the neighbour node with the maximum node degree is selected as the destination node. The result of the previous two steps determines an edge connecting the maximum node degrees. This edge is the candidate edge to be deleted from the sample reservoir, to be replaced with a new incoming edge in the sample reservoir.

There are consequent modifications required after the previous steps on the sample (Line 18-23). This modification can be addition of an edge to the sample reservoir or deletion of an edge from the sample reservoir, the addition to the sample is done by the procedure `addToSample (u, v)`, while the deletion from the sample reservoir is done through the procedure `removeFromSample (u, v)`.

By the same way, heap is updated either by node addition or by node deletion (Line 24-27).

Maximum Controlled Reservoir Sampling (MCRS)

Input: incremental graph data d , integer M /* M : maximum whole reservoir size in terms of nodes
Output: updated Sample, updated Heap

1: $s \leftarrow \emptyset, h \leftarrow \emptyset, i \leftarrow 0, x \leftarrow \emptyset$
/* s : sample size, h : minimum heap size

2: procedure addEdge((u, v))
3: for each edge (u,v) from d do
4: $i \leftarrow i+1$
5: if sampleEdge(u, v) then
6: addToSample (u,v)

7: procedure sampleEdge ((u, v))
8: if sampleSize $< M$ then
9: return True
10: else
11: $\maxEdge(x,y) \leftarrow \maxEdge()$
12: removeFromSample (x,y)
13: return True

14: procedure maxEdge ()
15: $x \leftarrow$ source node with highest node degree
16: $y \leftarrow$ destination node with highest node degree
 among neighbours of (x)
17: return edge(x,y)

18: procedure addToSample (u,v)
19: $s \leftarrow s + \{(u, v)\}$
20: addToHeap (u,v)

21: procedure removeFromSample (u,v)
22: $s \leftarrow s - \{(u, v)\}$
23: removeFromHeap (u,v)

24: procedure addToHeap (u,v)
25: $h \leftarrow h + \{(u, v)\}$

26: procedure removeFromHeap (u,v)
27: $h \leftarrow h - \{(u, v)\}$

Figure 4.8. Pseudo code of the MCRS algorithm

Illustrating Examples on MCRS

Example 1

The example in Figure 4.9 shows the addition process of the coming edges when the whole reservoir is not full, as it is done by MCRS algorithm.

Given an input graph of coming edges Figure 4.9(a), each coming edge is inserted in the sample reservoir Figure 4.9(b), this insertion is done into the sample reservoir as long as the whole reservoir is not full, in this example; the whole reservoir is not full, and all coming edges are inserted directly into the sample reservoir. Then; the nodes of each edge in the sample reservoir are registered in the node degree list (heap reservoir) if they are not in the list before, otherwise the degrees are incremented as in Figure 4.9(c), this list consists of the nodes and their degrees, however the degree of each node is considered as the number of connections of each node in the sample reservoir, this node degrees list (heap reservoir) is presented in MCRS algorithm by the heap data structure, where the nodes are ordered according to their degrees in ascending order, the root node of the heap holds the node with the minimum degree among all the other nodes in the list. The edges in the sample reservoir are presented in a hash map as Figure 4.9(d).

Let's explain the process of inserting the coming edge (XE), first, the size of sample reservoir is checked, the sample reservoir size now is equal to 5, so there is a space to insert the edge XE directly into the sample reservoir Figure 4.9(b), then the node degree list is checked for the nodes X and E, it is found that both of them are already in the list, the degrees of X and E are 3 and 1 respectively, now both of them are incremented by 1, to become 4 for X and 2 for E Figure 4.9(c). The hash map in Figure 4.9(c) is updated, in the map, node E is added as a neighbour of node X, and node X is added as a neighbour of node E.

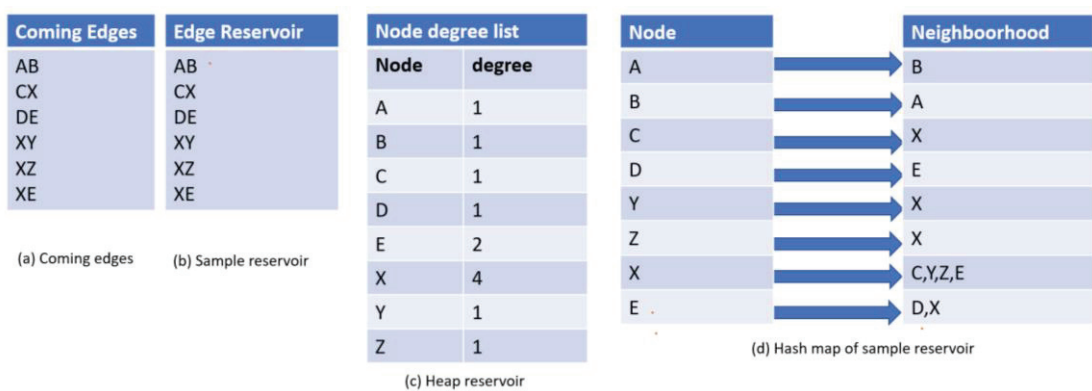


Figure 4.9. Example of MCRS after adding a new coming edge (XE), while the whole reservoir (sample and heap reservoirs) is not full

Example 2

Figure 4.10 shows an example of the MCRS algorithms. The example is illustrated when the whole reservoir is full. The example in this figure based on the previous example of Figure 4.9. Given an evolving graph, the incoming edges are listed as in Figure 4.10(a). This example is explained when a new edge (FG) comes. Suppose the maximum allowed size of the sample reservoir is 6 edges. When the coming edges appears, they are inserted one by one in the sample reservoir if it is not full Figure 4.10(b), the new nodes of the coming edges are inserted in the node degree list (heap reservoir) if they are not already in the list, otherwise their degrees are updated by increasing the degree of each node by one. The records of these coming nodes are updated in the hash map also. In the example of Figure 4.10; the first six coming edges (AB, CX, DE, XY, XZ, XE) are inserted directly one by one into the sample reservoir, since the sample reservoir is not full. On the other hand, when the edge (FG) comes, the sample reservoir size reaches the maximum allowed size (6), so it is full, as a result, one edge from sample reservoir should be removed, this edge is the edge with the maximum degrees of it two nodes, so the edge XE is the candidate edge for removal, the degrees of with removing the edge XE from sample reservoir, the counts of the nodes X and E in the node degree list are decreased, and the connection edge between E and X is removed from the hash map of the sample graph. Now, there is a space in the sample reservoir to insert the new edge (FG), then, and the new nodes F and G are added to the list, the degrees of both nodes are one, since they are newly added to the list Figure 4.10(c). The hash map is updated by adding F and G to it Figure 4.10(d).

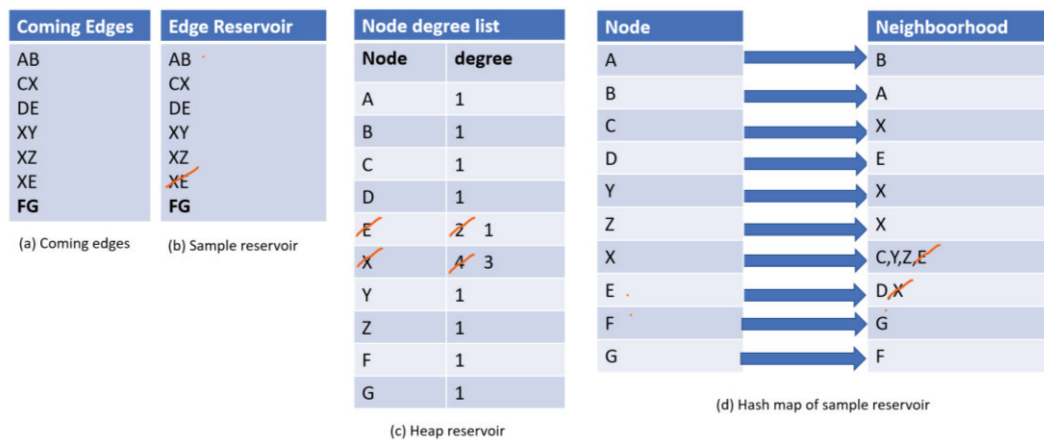


Figure 4.10. Example of MCRS after adding a new coming edge (FG), while the whole reservoir is full

4.4. Deleting an Edge in UCRS, LCRS, MCRS and Random algorithms

The example in Figure 4.11 illustrates the deletion process of an edge when the whole reservoir is full, in this example; three types of edge deletion are illustrated, the first one is random edge deletion as in Triest (De Stefani *et al.*, 2016), SR (Aslay *et al.*, 2018) and OSR (Aslay *et al.*, 2018), the second one is controlled deletion of an edge with minimum degree of nodes, as it is done by controlled edge deletion in (UCRS and LCRS) and the third one is controlled deletion of an edge with maximum degree of nodes as in MCRS. In another words an example of edge deletion with random edge deletion and controlled edge deletion is represented.

Given an original graph Figure 4.11(a), the table in the figure shows the nodes and their degrees of the original graph. By applying random edge deletion, the method that is used in (De Stefani *et al.*, 2016), SR (Aslay *et al.*, 2018) and OSR (Aslay *et al.*, 2018), suppose the candidate edge is: (X, Z), when it is removed; two triangle patterns of the original graph are lost as shown in Figure 4.11(b). On the other hand, in controlled deletion of an edge with minimum degree of nodes; the method that is used in UCRS and LCRS, the edge (S, V) is a candidate to be removed since S is the lowest degree node and V is the lowest degree neighbour of V. When the edge (S, V) is removed from the original graph, no triangle pattern of six triangle patterns is lost as illustrated in Figure 4.11(c). However, in controlled deletion of an edge with maximum degree of nodes; the method that is used in MCRS, the edge (W, U) is a candidate to be removed since W is the highest degree node and U is the highest degree neighbour of W. When the edge (W, U) is removed from the original graph, three triangle patterns are lost as illustrated in Figure 4.11(d). The table under the figure shows, the number of lost triangles, the number of remained triangles, and the percentage loss of random, minimum controlled edge deletion and maximum controlled edge deletion processes, respectively. As a result, less patterns are lost by minimum controlled edge deletion (UCRS and LCRS) and more patterns are lost by maximum edge controlled deletion (MCRS). While the number of patterns that affected by random edge deletion be between the other two mentioned types of edge deletion.

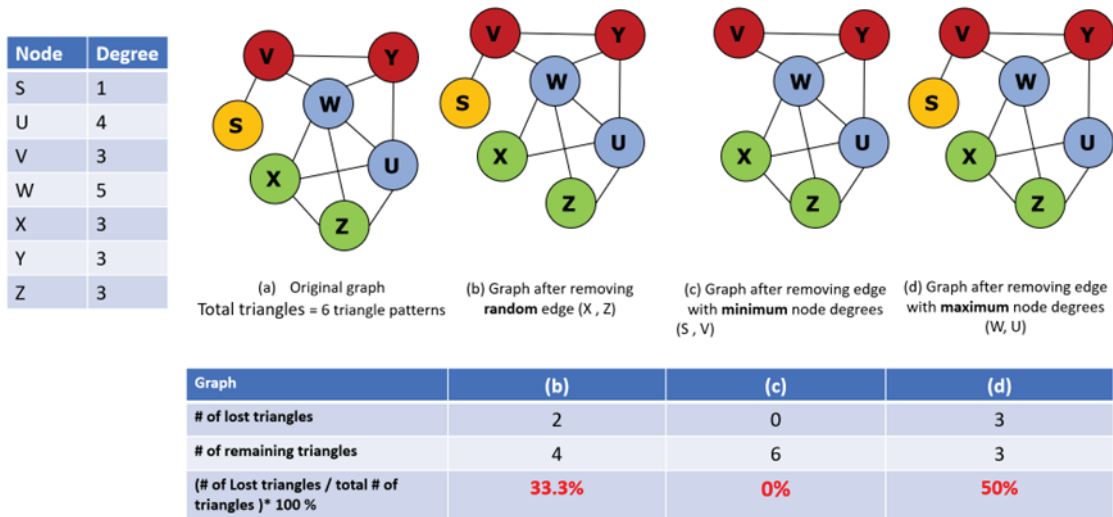


Figure 4.11. Deleting edge with minimum node degrees, maximum node degrees and random edge

From the above example, it is noticed that; removing edge with minimum degree of its nodes, very few patterns are affected by this edge deletion.

By removing edge with maximum node degrees, most patterns are expected to be lost, so this way of edge deletion supports the idea of deleting edge with minimum degrees of the nodes. By using random edge deletion of nodes, the number of left patterns after edge deletion are expected to be lower than it in the case of removing edge with minimum node degrees, and higher than it in the case removing edge with maximum node degrees. So removing edge with minimum node degrees is preferred to keep most patterns in the graph.

In this Chapter, three algorithms (UCRS, LCRS and MCRS) are proposed for approximate frequent-subgraph mining (FSM) in evolving graphs, where edge/vertex can be arbitrary added using a fixed memory size of whole reservoir. Whole Reservoir keeps the sample reservoir dynamic graph and heap reservoir, sample reservoir which represents the characteristics of the original graph, and allows dynamic algorithms to work on reduced sized graph, heap reservoir keeps the node degrees of the nodes that are in the reservoir according to the type of algorithm. Both UCRS and LCRS use controlled edge deletion for minimum node degree edge from sample reservoir, the advantage of this controlled edge deletion is to keep more connected edges in the sample reservoir, by doing so; recall is expected to be increased. While, the third algorithm MCRS is proposed, it

works in a similar manner to UCRS and LCRS, but the main difference is in determining the candidate edge to be deleted from sample reservoir, whenever the whole reservoir is full, in MCRS algorithm the candidate edge is the edge with maximum degree of its nodes, by doing so, the high connectivity edges are deleted from the sample reservoir, as a result, recall is expected to be decreased. So, the results of MCRS support the need for the advantages of UCRS and LCRS algorithms.

CHAPTER 5

PERFORMANCE EVALUATION

The experimental evaluation of the proposed algorithms Unlimited Controlled Reservoir Sampling (UCRS), Limited Controlled Reservoir Sampling (LCRS) and Maximum Controlled Reservoir Sampling (LCRS) are conducted in comparison to three existing algorithms: Triest (De Stefani *et al.*, 2016), Subgraph Reservoir (Aslay *et al.*, 2018) and Optimized Subgraph Reservoir (OSR) (Aslay *et al.*, 2018). These three algorithms are devised to find the approximate frequent patterns in dynamic graphs as UCRS, LCRS and MCRS. All algorithms including UCRS, LCRS and MCRS use reservoir sampling technique. Triest, SR and OSR implement reservoir technique of (Vitter, 1985), where edge deletion from sample reservoir is done randomly when sample reservoir is full. However, UCRS, LCRS and MCRS use a heap reservoir to manage the node degrees, UCRS and LCRS delete the low degree nodes when whole reservoir is full, while MCRS delete the high degree nodes when the whole reservoir is full. Trade-off between execution time and accuracy is measured in the following subsections as execution time, scalability, recall and heap size measurement experiments.

All experimental results are reported as an average of 3 runs. The properties of the datasets used in all experiments are shown in the Table 5.1. D1 is Patents dataset which is publicly available (Hall, Jaffe and Trajtenberg, 2001) and contains citations among US Patents from January 1963 to December 1999. D2 is LastFM is Asia Social Network dataset (<https://snap.stanford.edu/index.html>); social network of LastFM users which was collected from the public API. Nodes are LastFM users from Asian countries and edges are mutual follower relationships between them. D2 has approximately double density of the Patents dataset, where the density of the datasets is calculated as $D = \frac{|E|}{|V|(|V|-1)}$ where $|E|$ and $|V|$ represents the total number of edges and vertices (nodes) in the dataset (Chakraborty, Byshkin and Crestani, 2020).

Table 5.1. Properties of the datasets

Dataset	V	E	Density
D1 (Patents)	3M	14M	2.3
D2 (LastFM)	7.6	27.8	4.8

In the following subsections the performance evaluation of Triest, UCRS, LCRS, MCRS, SR, and OSR algorithms is measured; experiments are conducted on D1 (Patents) and D2 (LastFM).

5.1. Scalability

To measure the scalability performances of Triest (De Stefani *et al.*, 2016), UCRS, LCRS, MCRS, SR (Aslay *et al.*, 2018), and OSR (Aslay *et al.*, 2018); an experiment is conducted on D1 and D2 datasets. In this experiment, the data size changes between 4000 to 20000 transactions. Execution time is measured; the maximum whole reservoir size (M) is kept constant. Whole reservoir size M is measured in terms of number of nodes as an abstract common metric for all the algorithms. Triest algorithm keeps edges in the sample reservoir (Whole reservoir in Triest); each edge is counted as 2 nodes. SR and OSR algorithms keep subgraphs of 3 nodes in the sample reservoir (Whole reservoir in SR and OSR). UCRS, LCRS and MCRS keeps edges (2 nodes) in the sample reservoir and the nodes that are in the heap reservoir. In another words, Triest spends 2 nodes with each edge addition, SR and OSR spends 3 nodes with each subgraph addition, UCRS, LCRS and MCRS spend 2 nodes and 0/1/2 node(s) depending on the existence/absence of the node(s) of the sample in the heap reservoir. The whole reservoir in UCRS, LCRS and MCRS consists of both of sample reservoir and heap reservoir. Heap reservoir size in LCRS is limited to a specified size. In this experiment maximum allowed heap size is 0.4 of the maximum whole reservoir size M .

The graph of Figure 5.1.A illustrates the scalability of Triest, UCRS, LCRS, MCRS, SR, and OSR on dataset D1 when M is assigned to 600 nodes. The graph of the Figure 5.1.B describes the same experiment but since Triest, UCRS, LCRS and MCRS have a very small execution time compared with SR and OSR, and can't appear in the

first graph, they are shown separately in the second graph. It is noticed from the figures that; as the data size increases; the execution times of all algorithms increase. SR and OSR have the highest execution time, since they search for subgraphs of neighbourhood of incoming edges and store them in sample reservoir, searching and storing subgraphs have higher complexity than searching edges only as in (Triest, UCRS, LCRS and MCRS). UCRS and LCRS have higher execution time than Triest, this is because of the heap management cost. UCRS has a higher execution time than LCRS because of increasing heap size. The limited size of the heap in LCRS requires less time for heap management. While MCRS has the highest execution time among the algorithms in Figure 5.1.B. Triest has the lowest execution time, since it has no heap management like UCRS, LCRS and MCRS, MCRS has higher execution time than UCRS and LCRS since the heap sizes in the recent two algorithms are smaller than it in MCRS, this is because the after each deletion of edges with lowest node degrees at least one of the nodes (root node) is deleted from the heap reservoir, while in MCRS the nodes of the deleted edge are decremented without deletion, so the time required to manage the a heap with larger size is more in MCRS. Also searching for minimum node degree is much cheaper than searching for maximum node degrees in the minimum heap.

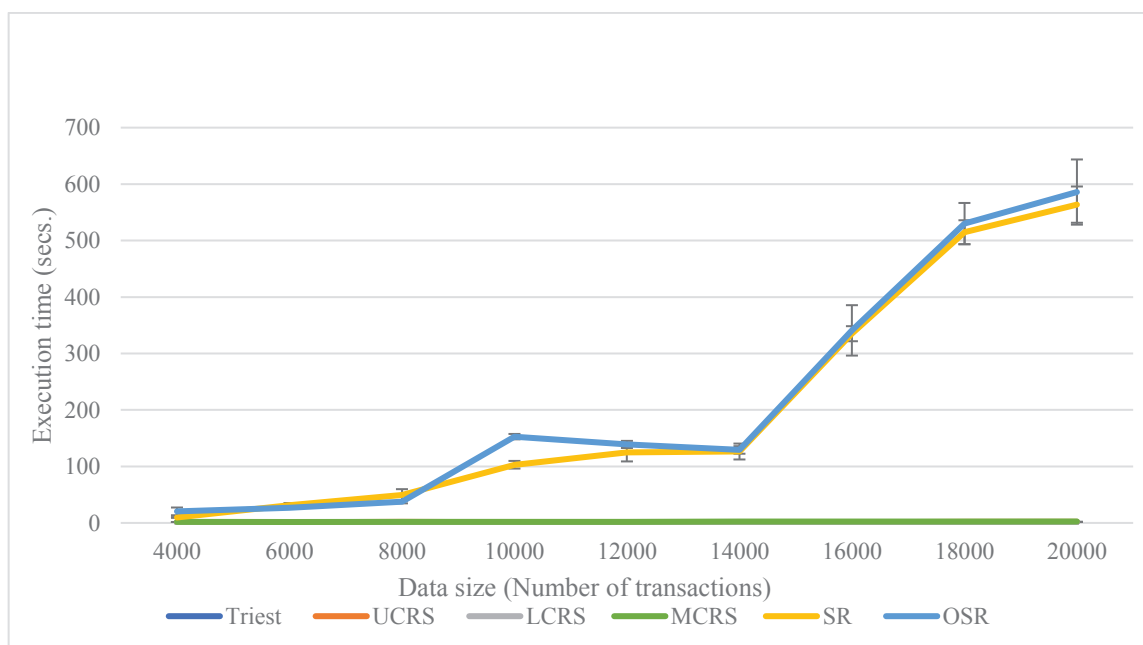


Figure 5.1.A. Scalability performance of the algorithms while changing the dataset size on Datasets D1 for M= 600

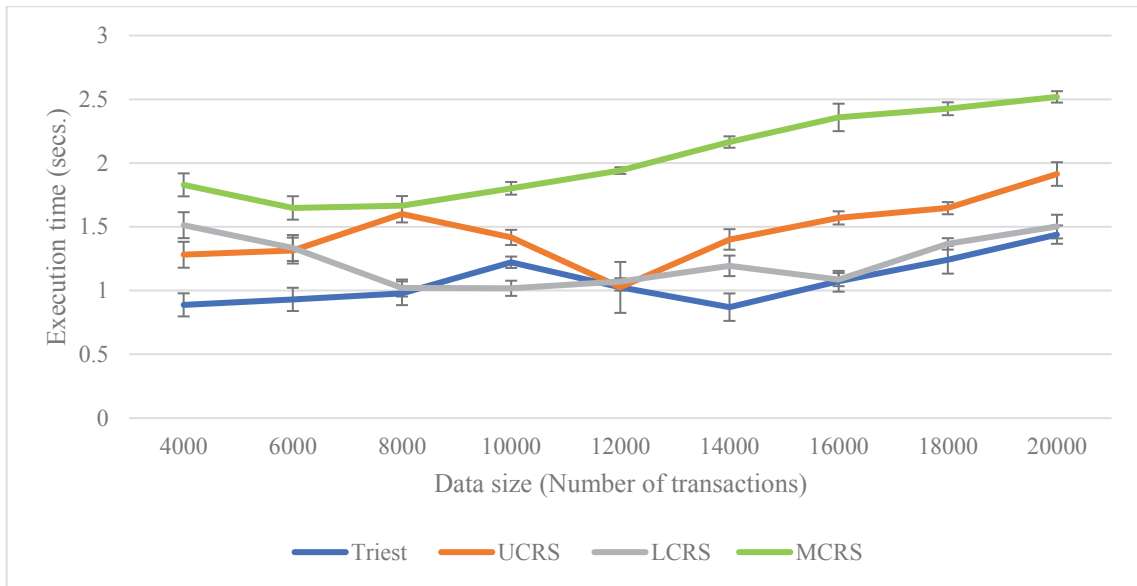


Figure 5.1.B. Scalability performance of the algorithms while changing the dataset size on Datasets D1 for M= 600

Figure 5.2.A and Figure 5.2.B show the scalability performance of Triest, UCRS, LCRS, MCRS, SR, and OSR algorithms, this experiment is done on dataset D1 when M is assigned to 1200 nodes. This experiment is similar to previous experiment only the maximum whole reservoir size is larger now (1200 nodes). The first graph presents all algorithms whereas the second graph focuses on Triest, UCRS, LCRS and MCRS execution times. It is shown in the figure that; as the data size increases; the execution times of all algorithms increase as in the previous experiment. SR and OSR have the highest execution time, since they search for subgraphs instead of edges. UCRS, LCRS and MCRS have higher execution time than Triest due to additional heap management cost. UCRS has a higher execution time than LCRS. Limited size of the heap in LCRS requires less time for its management. MCRS has the highest execution time among Triest, UCRS and LCRS. Triest has the lowest execution time, this is due to the nonexistence of heap management like UCRS, LCRS and MCRS, the execution time of MCRS is higher than UCRS and LCRS execution times, the reason for that is: the heap sizes in the UCRS and LCRS are smaller than the heap size in MCRS, in UCRS and LCRS; after each deletion of edges with lowest node degrees, at least one of the nodes (root node) is deleted from the heap, while in MCRS the nodes of the deleted edge are decremented without deletion, so the time required to manage the a heap with larger size

is more in MCRS. Also searching for minimum node degree is much cheaper than searching for maximum node degrees in the minimum heap. The heap size is limited in LCRS, while the heap size is unlimited in UCRS and MCRS.

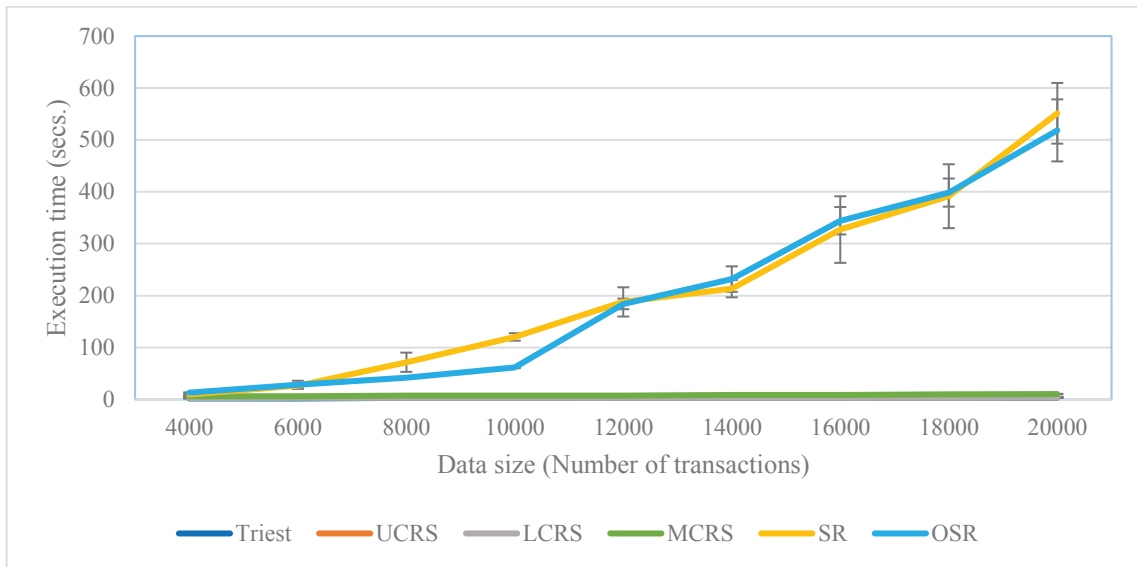


Figure 5.2.A. Scalability performance of the algorithms while changing the dataset size on Datasets D1 for M= 1200

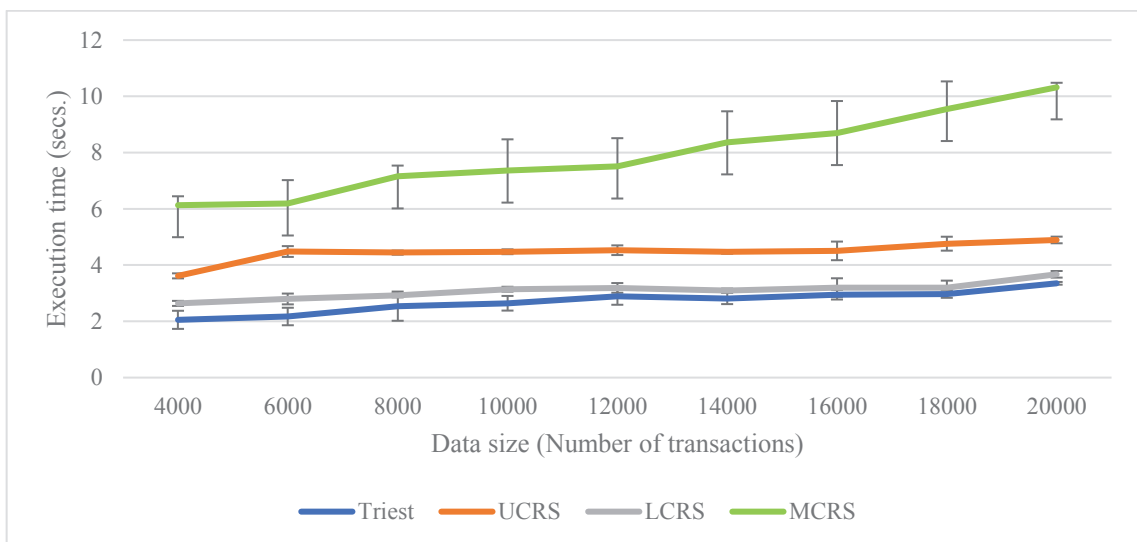


Figure 5.2.B. Scalability performance of the algorithms while changing the dataset size on Datasets D1 for M= 1200

The scalability performance of Triest, UCRS, LCRS, MCRS, SR, and OSR algorithms is illustrated in Figure 5.3.A and Figure 5.3.B, this experiment is done on dataset D1 when M is assigned to 1800 nodes. This experiment is similar to previous two experiments, but in this experiment, the maximum whole reservoir size is assigned to a larger value (1800 nodes). The first graph presents all algorithms while the second graph focuses on Triest, UCRS, LCRS and MCRS execution times. It is shown in the figure that; as the data size increases; the execution times of all algorithms increase as in the previous experiments. SR and OSR have the highest execution time, since they search for subgraphs instead of edges. UCRS, LCRS and MCRS have higher execution time than Triest due to additional heap management cost. UCRS has a higher execution time than LCRS. Limited size of the heap in LCRS requires less time for its management. MCRS has the highest execution time among Triest, UCRS and LCRS. Triest has the lowest execution time, since it has no heap management like UCRS, LCRS and MCRS, the execution time of MCRS is higher than UCRS and LCRS execution times, the reason for that is: the heap sizes in the UCRS and LCRS are smaller than the heap size in MCRS, in UCRS and LCRS; after each deletion of edges with lowest node degrees, at least one of the nodes (root node) is deleted from the heap, but in MCRS the nodes of the deleted edge are decremented without deletion, so the time required to manage the a heap with larger size is higher in MCRS. In addition to that, searching for minimum node degree is much cheaper than searching for maximum node degrees in the minimum heap. In LCRS the heap size is limited, however the heap size is unlimited in UCRS and MCRS.

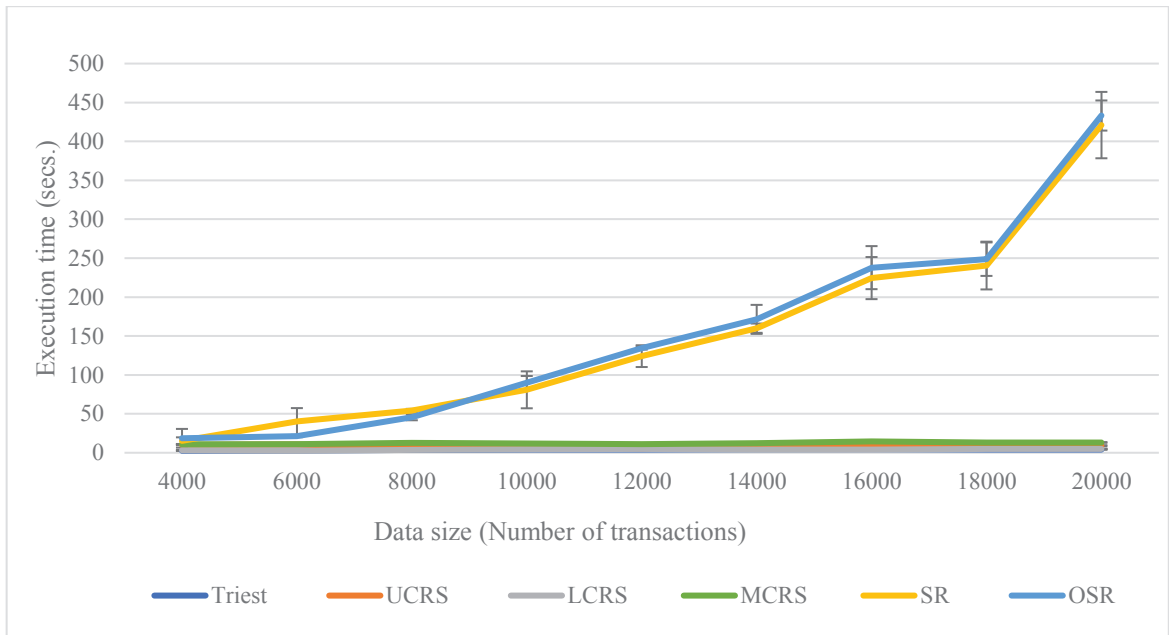


Figure 5.3.A. Scalability performance of the algorithms while changing the dataset size on Datasets D1 for M= 1800

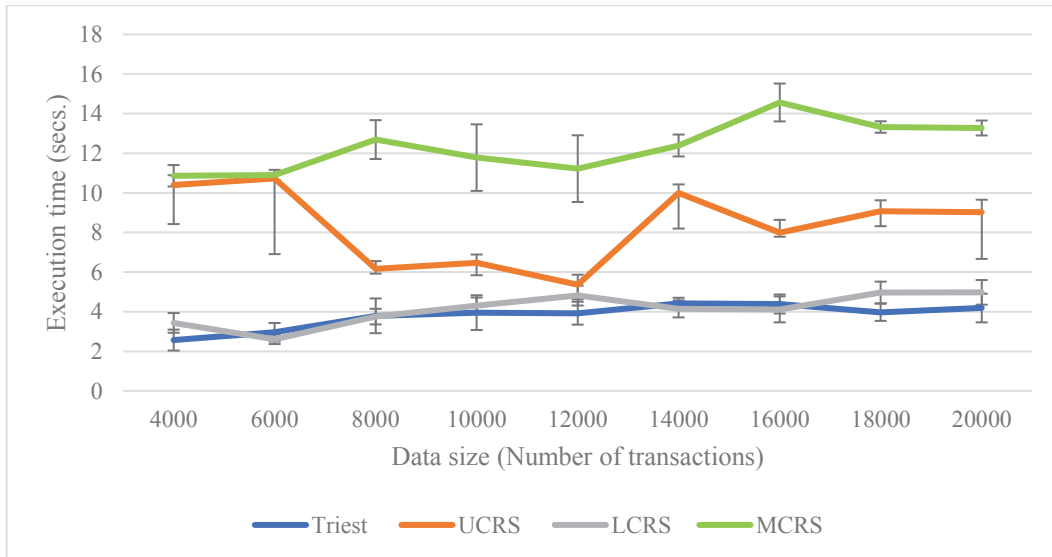


Figure 5.3.B. Scalability performance of the algorithms while changing the dataset size on Datasets D1 for M= 1800

The graph of Figure 5.4.A illustrates the scalability of Triest, UCRS, LCRS, MCRS, SR, and OSR on dataset D2 and M is assigned to 600 nodes. The graph of Figure 5.4.B describes the same experiment showing Triest, UCRS, LCRS and MCRS results since they are not seen well in the first graph due to scale difference. It is noticed from the Figure 5.4.A graph that; as the data size increases; the execution times of all algorithms increase. SR and OSR have the highest execution time, since they search for subgraphs instead of edges. It is noticed from the Figure 5.4.B graph that; UCRS, LCRS and MCRS have higher execution time than Triest, this is because of the heap management cost. UCRS has a higher execution time than LCRS since LCRS requires less time for smaller heap. MCRS has higher execution time than UCRS and LCRS, since the nodes of the deleted edge are decremented without node deletion from the heap reservoir, so the time required to manage a heap with larger size is more in MCRS. Also searching for maximum node degree is more expensive than searching for minimum node degrees in the minimum heap.

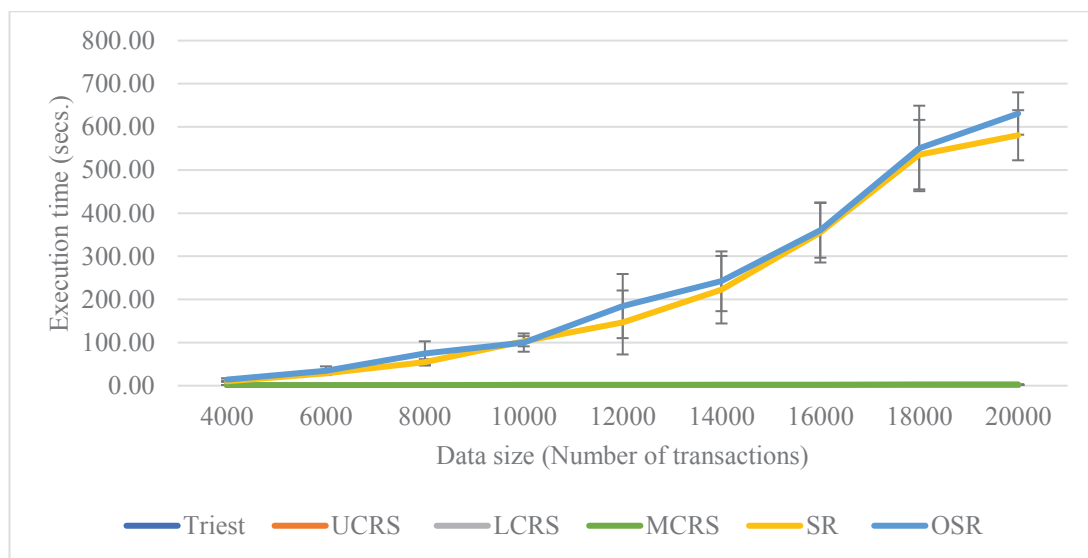


Figure 5.4.A. Scalability performance of the algorithms while changing the dataset size on Datasets D2 for M= 600

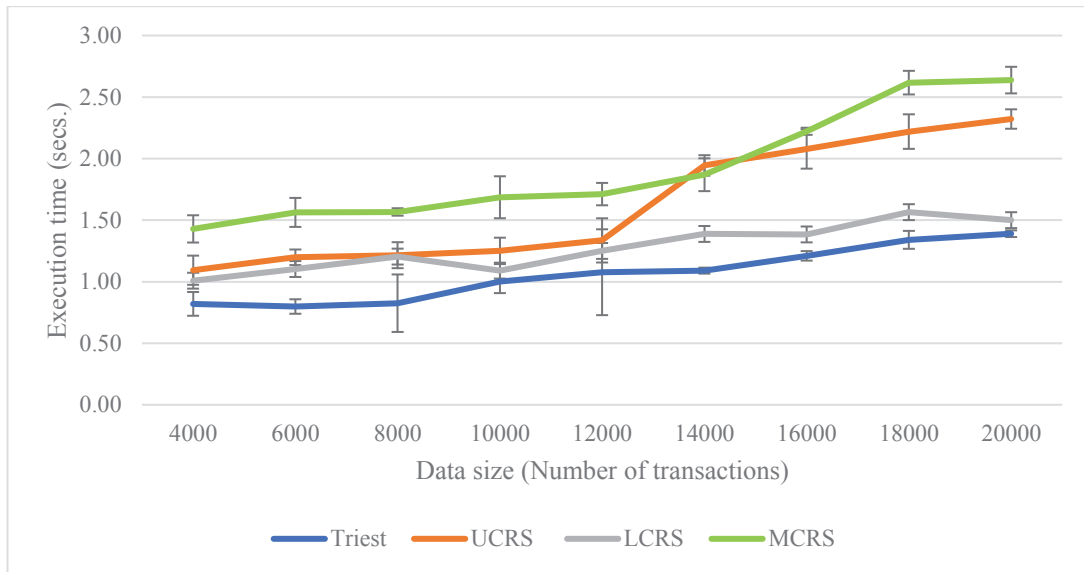


Figure 5.4.B. Scalability performance of the algorithms while changing the dataset size on Datasets D2 for M= 600

The Graph of Figure 5.5.A illustrates the scalability of Triest, UCRS, LCRS, MCRS, SR, and OSR on dataset D2 and M is assigned to 1200 nodes. The graph of the Figure 5.5.B describe the same experiment, but it shows only Triest, UCRS, LCRS and MCRS since they are not seen well on the graph of Figure 5.5.A. It is shown in the first graph that; as the data size increases; the execution times of all algorithms increase. SR and OSR have the highest execution time, since they search for subgraphs of neighbourhood of incoming edges and store them in sample reservoir. Searching and storing subgraphs have higher execution time than searching edges only. In Figure 5.5.B, UCRS, LCRS and MCRS have higher execution time than Triest due to additional heap management cost. UCRS has a higher execution time than LCRS since the heap size is not limited as LCRS. UCRS and MCRS have lower execution time than MCRS since the heap reservoir size in UCRS and LCRS always shrinks with each edge deletion process from sample reservoir, because the root node always deleted, while the candidate node that have maximum degree is decremented since it is still connected with other nodes.

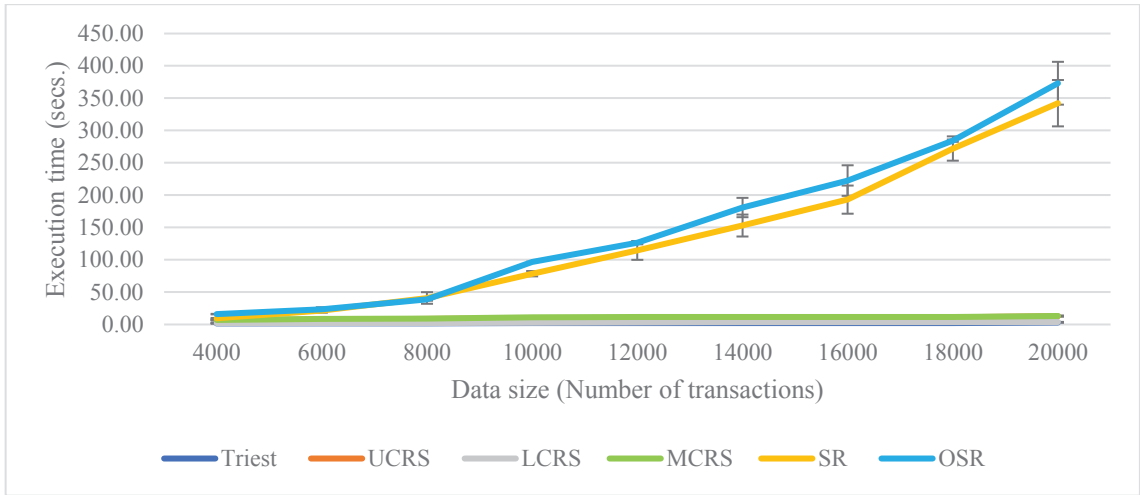


Figure 5.5.A. Scalability performance of the algorithms while changing the dataset size on Datasets D2 for M= 1200

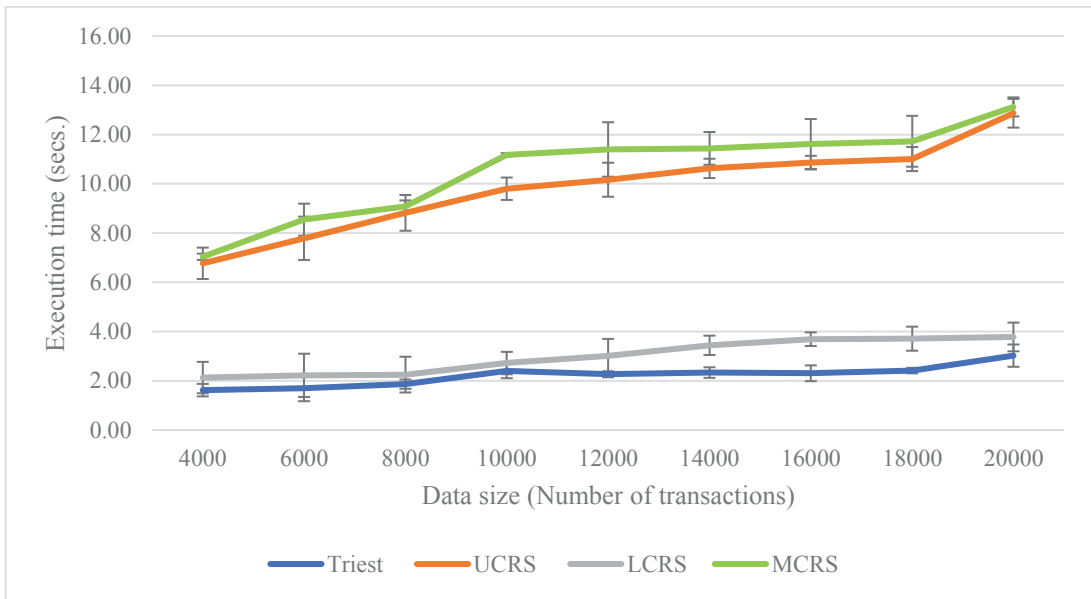


Figure 5.5.B. Scalability performance of the algorithms while changing the dataset size on Datasets D2 for M= 1200

The Graph of Figure 5.6.A illustrates the scalability of Triest, UCRS, LCRS, MCRS, SR, and OSR on dataset D2 and M is assigned to 1200 nodes. The graph of the Figure 5.6.B describe the same experiment, but it shows only Triest, UCRS, LCRS and

MCRS since they are not seen well on the graph of Figure 5.6.A. It is shown in the first graph that; as the data size increases; the execution times of all algorithms increase. SR and OSR have the highest execution time, since they search for subgraphs of neighbourhood of incoming edges and store them in sample reservoir. Searching and storing subgraphs have higher execution time than searching edges only. In Figure 5.6.B, UCRS, LCRS and MCRS have higher execution time than Triest due to additional heap management cost. UCRS has a higher execution time than LCRS since the heap size is not limited as LCRS. UCRS and MCRS have lower execution time than MCRS since the heap reservoir size in UCRS and LCRS always shrinks with each edge deletion process from sample reservoir, because the root node always deleted, while the candidate node that have maximum degree is decremented since it is still connected with other nodes.

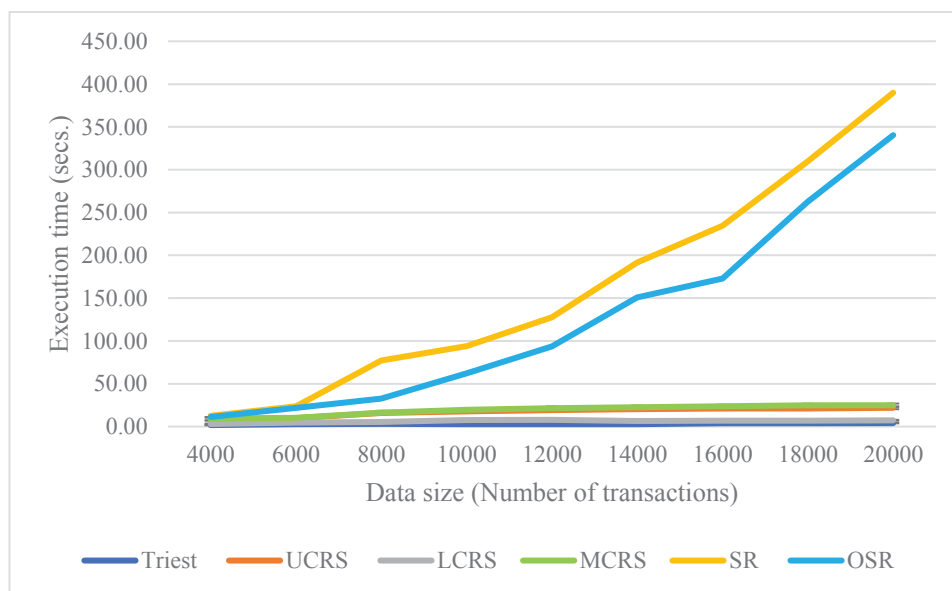


Figure 5.6.A. Scalability performance of the algorithms while changing the dataset size on Datasets D2 for M= 1800

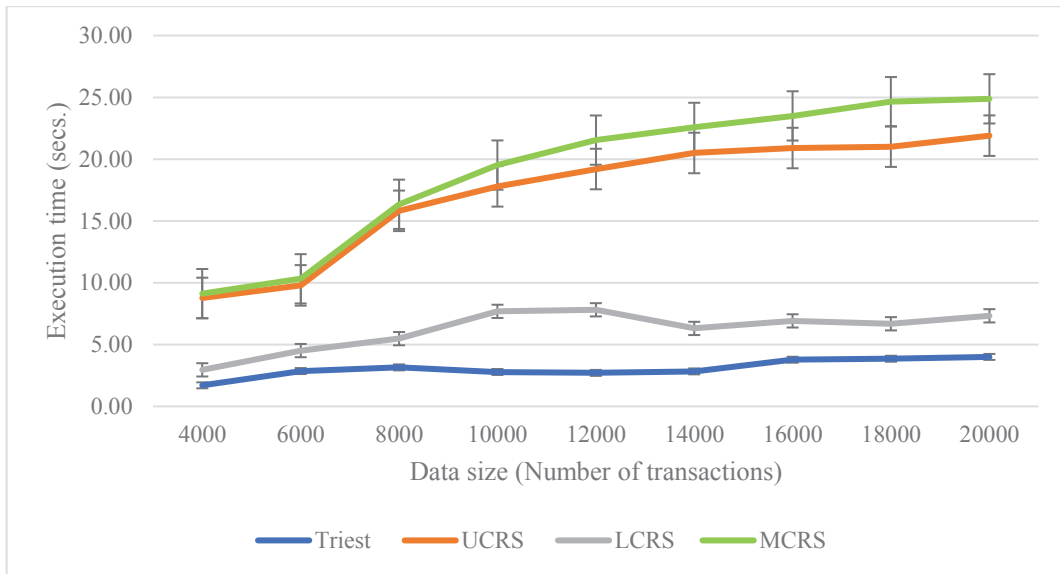


Figure 5.6.B. Scalability performance of the algorithms while changing the dataset size on Datasets D2 for M= 1800

Table 5.2 shows the scalability speed-up of the algorithms while varying the dataset size. This speed up is calculated with respect to the slowest algorithm (OSR). From the table, it is shown that, Triest has the highest speed up since it processes edges without managing heap. LCRS is close to Triest in terms of speed-up. LCRS and UCRS have noticeable speed-up over SR and OSR. LCRS has additional speed-up over UCRS algorithms since LCRS has an extra heap pruning method that minimize the space of the heap, and as a result, the required execution time for the heap is minimized. MCRS has lower speed-up than UCRS, since the heap size in MCRS is larger than the heap size in UCRS, and it needs more time to manage it.

Table 5.2. Scalability speed-up of the algorithms while varying the dataset size

Dataset	M (nodes)	Speed-up with Triest ^[a]	Speed-up with UCRS ^[b]	Speed-up with LCRS ^[c]	Speed-up with MCRS ^[d]	Speed-up with SR ^[e]
D1	600	22.99 – 407.17	15.94 – 306.12	20.42 – 390.09	11.16 – 207.59	0.47 – 0.96
	1200	6.37 – 154.622	3.62 – 105.00	4.96 – 141.00	2.13 – 50.22	1.38 - 0.94
	1800	7.25 – 108.55	1.79 – 47.97	5.43 – 87.06	2.36 – 32.63	1.2 – 1.03
D2	600	17.32 – 453.89	12.99 – 271.75	14.08 – 420.61	9.93 – 239.16	0.72 – 0.92
	1200	9.91 – 123.51	2.37 – 29.01	7.54 – 98.71	2.28 – 28.43	0.60 – 0.91
	1800	12.99 – 107.17	1.5.94 – 206.12	2.04 – 39.09	1.11 – 14.32	0.47 – 0.96

^[a] Speed-up = Execution time of OSR algorithm / Execution time of Triest algorithm.

^[b] Speed-up = Execution time of OSR algorithm / Execution time of UCRS algorithm.

^[c] Speed-up = Execution time of OSR algorithm / Execution time of LCRS algorithm.

^[d] Speed-up = Execution time of OSR algorithm / Execution time of MCRS algorithm.

^[e] Speed-up = Execution time of OSR algorithm / Execution time of SR algorithm.

5.2. Recall

The algorithms Triest, UCRS, LCRS, MCRS, SR and OSR are compared and evaluated in terms of recall. Recall is the ratio of the number of patterns found in the sample by the algorithm to the number of patterns found by an exact algorithm that works on whole data instead of the sample.

The experiment in Figure 5.7, Figure 5.8 and Figure 5.9 is conducted to measure the recall. The data sizes are varied from 4000 to 20000 transactions, and the maximum whole reservoir size M (number of nodes) is kept constant as 600, 1200 or 1800 nodes. The used dataset is D1. From the figures it is shown that; LCRS has the highest recall among the other five tested algorithms. The reason for that is the novel method in UCRS and LCRS, which is designed to replace the random edge deletion in sample reservoir with controlled edge deletion. The idea behind this method is not to lose high degree nodes, which have more impact on recall. Higher recall is gained since more connected (higher degree) nodes remain in the sample reservoir, while in Triest the edge deletion from sample reservoir is done randomly, by this way of edge deletion; patterns that are more important can be lost. On D1, LCRS has higher recall than UCRS, this is because of the pruning method that is done on the heap size, which give a chance to have a smaller size of the heap reservoir and larger size of the sample reservoir (sample of incoming

edges), by this way, the number of retrieved patterns increases and as a result the recall increases. On D1, with $M=1800$ nodes, UCRS and LCRS have higher recall than the other algorithms in the experiment, the reason for that is as the sample reservoir size increases, the number on non-important edges were deleted and more important edges that can affect the patterns are increases. MCRS has lowest recall among the other five algorithms, MCRS works in a similar way to UCRS, but instead of removing the less connected edge whenever the whole reservoir is full, in MCRS the most conned edge is removed, this way supports the idea of UCRS and LCRS, which is: removing less connected edges can keep important patterns in the sample reservoir, and as a result the recall increases.

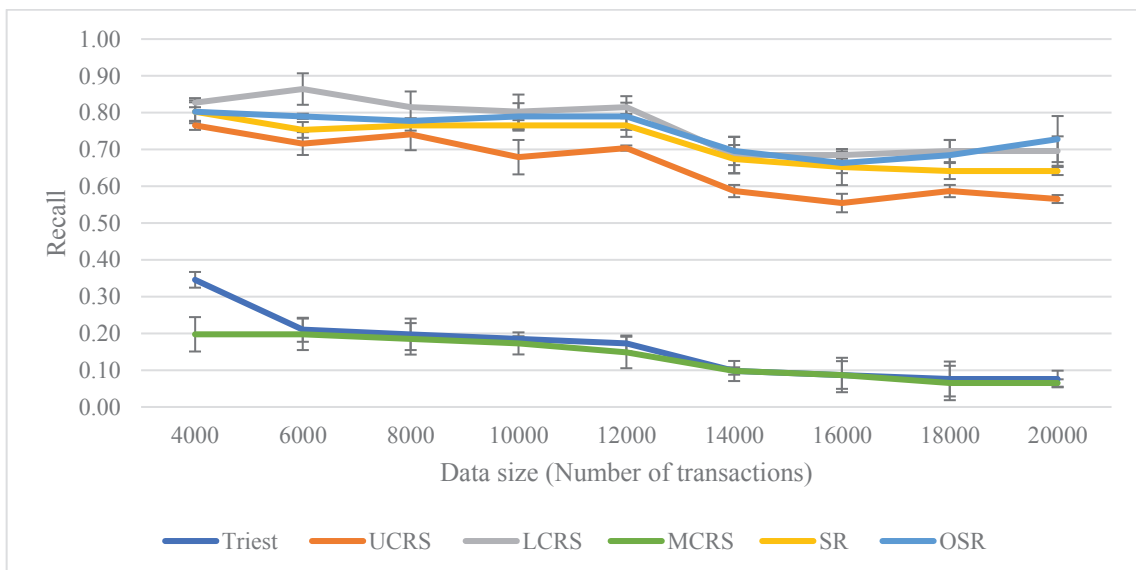


Figure 5.7. Recall of the algorithms while changing the dataset size on Dataset D1 ($M=600$ nodes)

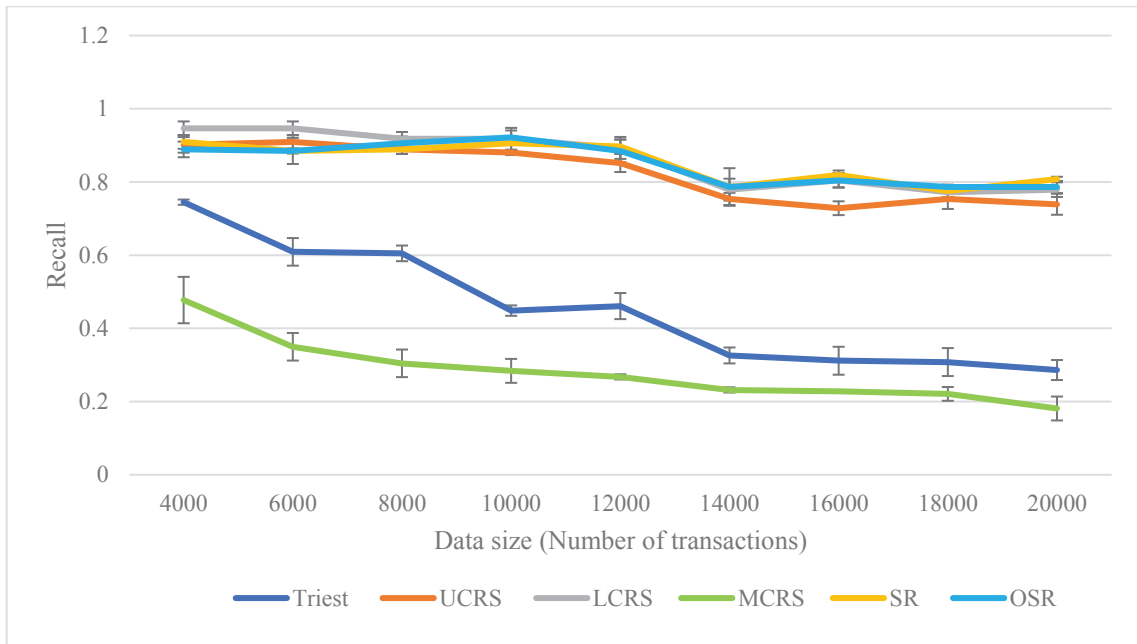


Figure 5.8. Recall of the algorithms while changing the dataset size on Dataset D1 (M=1200 nodes)

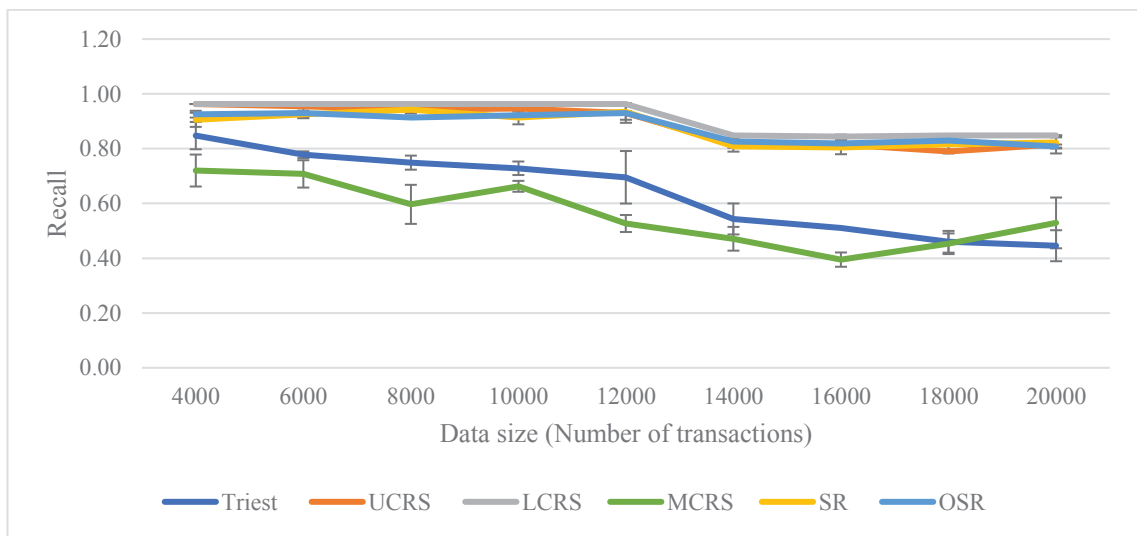


Figure 5.9. Recall of the algorithms while changing the dataset size on Dataset D1 (M=1800 nodes)

The experiment in Figure 5.10, Figure 5.11 and Figure 5.12 is conducted to measure the recall. The data sizes are varied from 4000 to 20000 transactions and the maximum whole reservoir size M (number of nodes) is kept constant as 600, 1200 or 1800 nodes. The used dataset is D2. It is noticed that SR and OSR have the highest recalls,

the reason for that is the dense nature of D2 and the storing subgraphs in sample reservoir instead of edges. UCRS and LCRS have higher recalls than Triest since a controlled edge deletion from sample reservoir is used instead of random edge deletion. LCRS has higher recall than UCRS because of the heap reservoir pruning strategy that is used in LCRS, which gives a chance to store a greater number of edges in the sample.

MCRS has lowest recall among the other five algorithms, since in MCRS the most conned edge is removed, while in UCRS and LCRS the less connected edge is removed whenever the whole reservoir is full, this way supports the idea of UCRS and LCRS, which is: removing less connected edges that can keep important patterns in the sample, and as a result the recall increases.

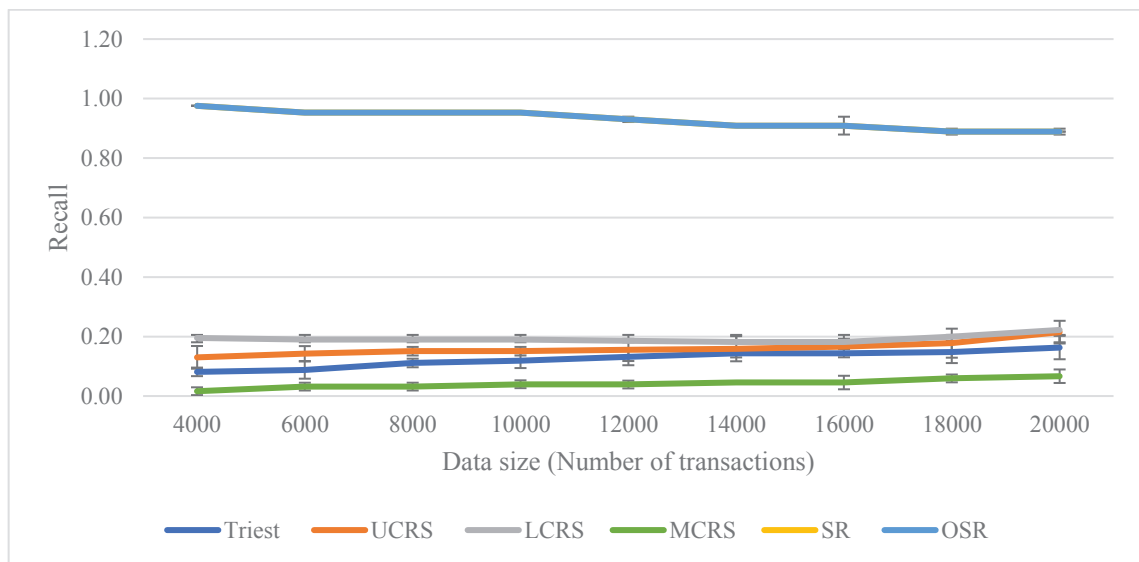


Figure 5.10. Recall of the algorithms while changing the dataset size on Dataset D2 (M=600 nodes)

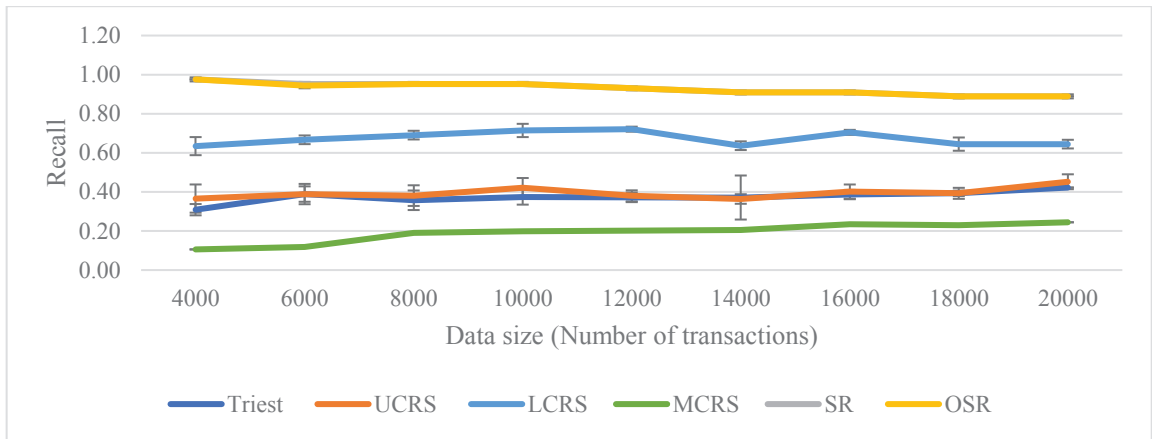


Figure 5.11. Recall of the algorithms while changing the dataset size on Dataset D2 (M=1200 nodes)

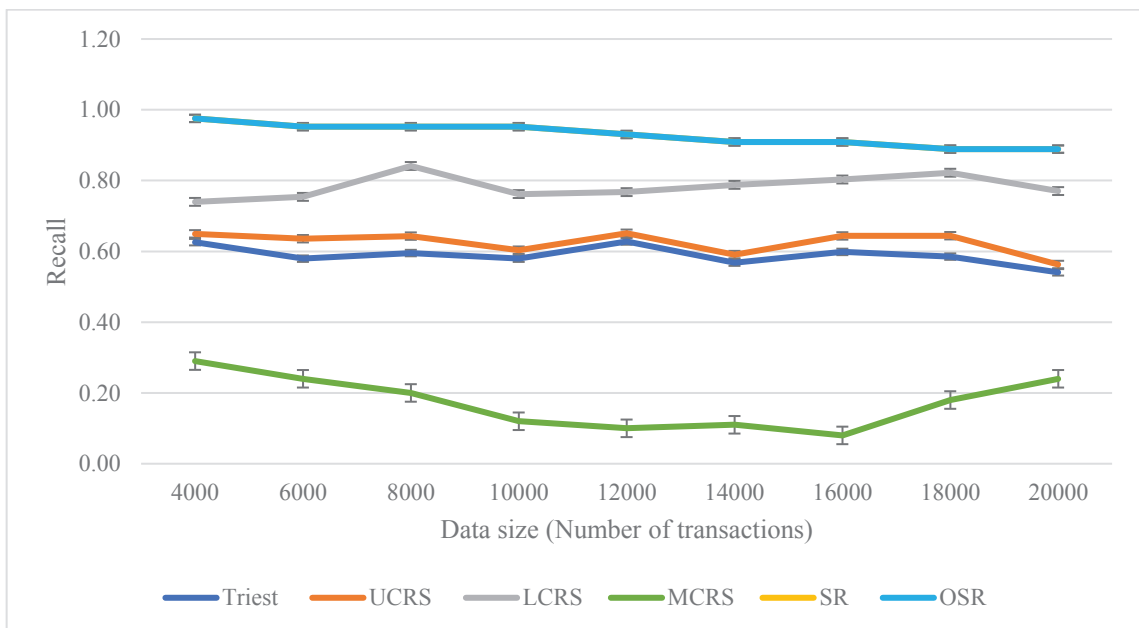


Figure 5.12. Recall of the algorithms while changing the dataset size on Dataset D2 (M=1800 nodes)

From Table 5.3, it is noticed that; on D1, UCRS and LCRS has a distinguished recall with average (0.66 and 0.76) for M=600, (0.82 and 0.86) for M= 1200, (0.89 and 0.91) for M= 1800 respectively. Increasing the maximum whole reservoir size has positive impact on the recall. Another positive impact comes with the controlled edge deletion strategies of the UCRS and LCRS algorithms. However, the average recall

decreases on D2, e.g. (0.16 and 0.19) respectively for (M=600), (0.39 and 0.67) respectively for (M=1200) and (0.62 and 0.78) respectively for (M=1800) this is because of the nature of the datasets. D2 is denser than D1. In addition, it is noticed that; for higher values of total number of nodes in the sample e.g., 1800 or 1200 instead of 600, higher recall is achieved. This is because increased maximum whole reservoir size brings increased sample reservoir size, which represents the original graph better.

When the total number of nodes in the whole reservoir is 1800 nodes, the recall of LCRS becomes higher and closer to the recalls of SR and OSR, if the whole reservoir sizes increases more and more, a higher recall can be gotten, until an equal recall to SR and OSR at the whole reservoir size (M) 4510 nodes. of course, with similar behaviour of execution time which is less than the execution time of SR and OSR. MCRS has lowest recall among the other five algorithms, since in MCRS the most connected edge is removed, so important patterns are expected to be lost while in UCRS and LCRS the less connected edge is removed whenever the whole reservoir is full.

Therefore, LCRS is recommended for sparse datasets, SR and OSR are recommended for dense datasets with small whole reservoir size, with a trade-off of high execution time. For dense dataset LCRS is recommended with a high number of nodes in the whole reservoir.

Table 5.3. Recall of the algorithms while changing the datasets sizes

Dataset	M (nodes)	Average Recall					
		Triest	UCRS	LCRS	MCRS	SR	OSR
D1	600	0.16	0.66	0.76	0.14	0.72	0.74
	1200	0.64	0.82	0.86	0.28	0.85	0.85
	1800	0.64	<u>0.89</u>	0.91	0.56	0.87	0.88
D2	600	0.13	0.16	0.19	0.04	0.93	0.93
	1200	0.37	0.39	0.67	0.19	0.93	0.93
	1800	0.59	0.62	0.78	0.27	0.93	0.93

5.3. Heap size

This experiment is to measure the space that the heap occupies in both algorithms UCRS and LCRS, the heap size is measured in terms of number of nodes. D1 and D2 datasets are used in these experiments. The maximum whole reservoir size (M) in term of (number of nodes) is constant.

Figure 5.13, Figure 5.14 and Figure 5.15 show the number of nodes in heap while changing the dataset sizes from 4000 to 20000 transactions on Dataset D1. M is assigned to 600 nodes in Figure 5.13 ,1200 nodes in Figure 5.14 and 1800 nodes in Figure 5.15. From the figures, it is shown that, the heap size is growing until a certain data size, this size depends on the nature of dataset and the whole reservoir size, in the experiments; for UCRS and MCRS, the curve grows fast for the data sizes (4000 - 14000) and it becomes slower after 12000. While for LCRS; the curve grows for the data sizes (4000 - 14000) and it becomes constant after 14000 according to the maximum assigned heap size.

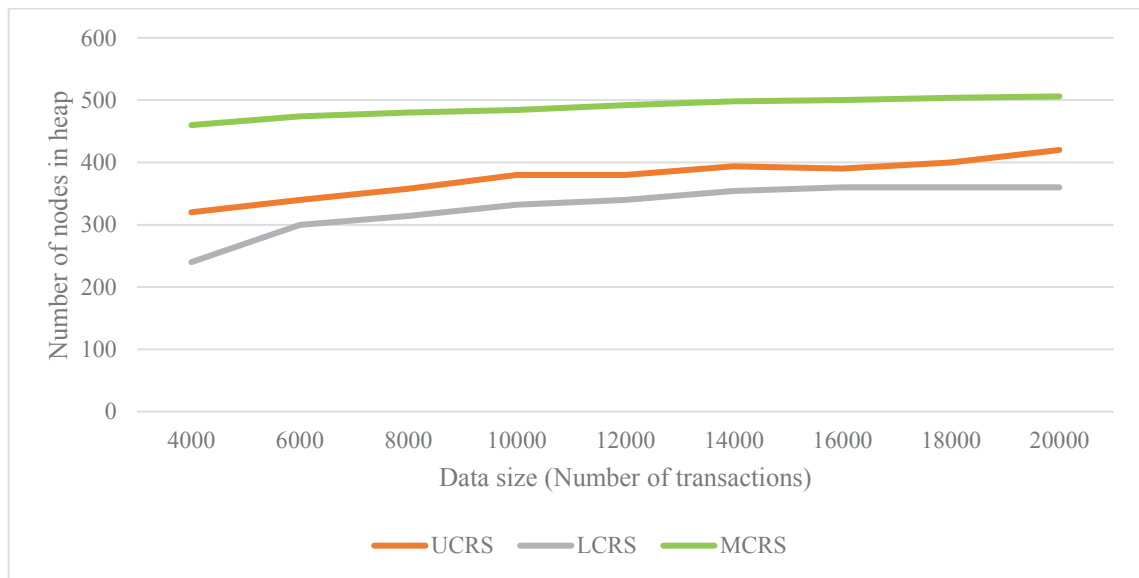


Figure 5.13. Number of nodes in heap while changing the dataset size on Dataset D1 (M = 600 nodes)

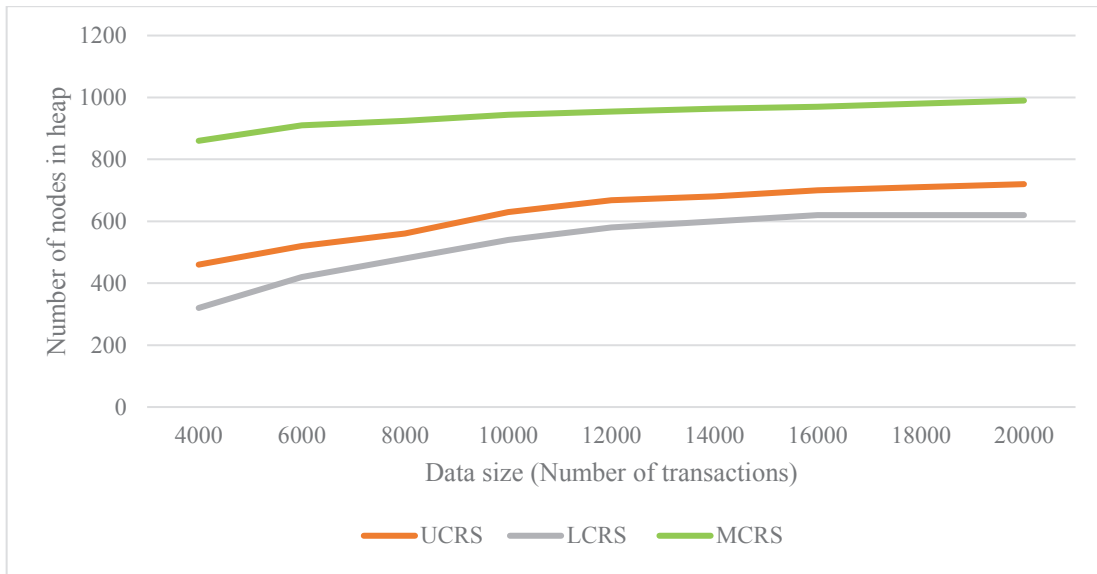


Figure 5.14. Number of nodes in heap while changing the dataset size on Dataset D1 (M = 1200 nodes)

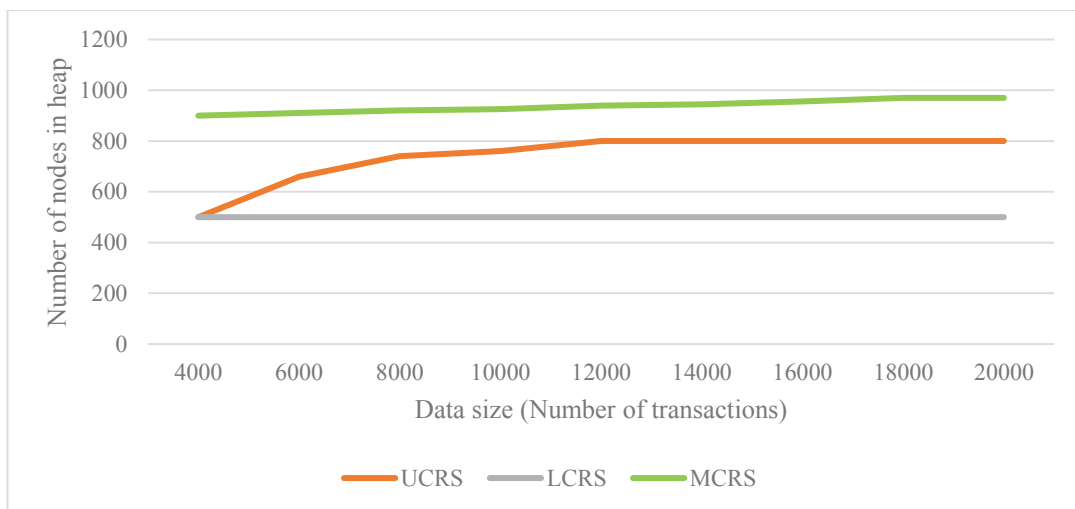


Figure 5.15. Number of nodes in heap while changing the dataset size on Dataset D1 (M = 1800 nodes)

Figure 5.16, Figure 5.17 and Figure 5.18 show the number of nodes in heap while changing the dataset sizes from 4000 to 20000 transactions on Dataset D2, and M is assigned to 600 nodes in Figure 5.16, 1200 nodes in Figure 5.17 and 1800 nodes in Figure 5.18. From the figures, it is shown that, the heap size is growing until a certain data size, according to the nature of dataset and whole reservoir size, in the experiments; for UCRS

and MCRS, the curve grows fast for the data sizes (4000 - 14000) and it becomes slower after 12000. While for LCRS; the curve grows for the data sizes (4000 - 14000) and it becomes constant after 14000 according to the maximum assigned heap size.

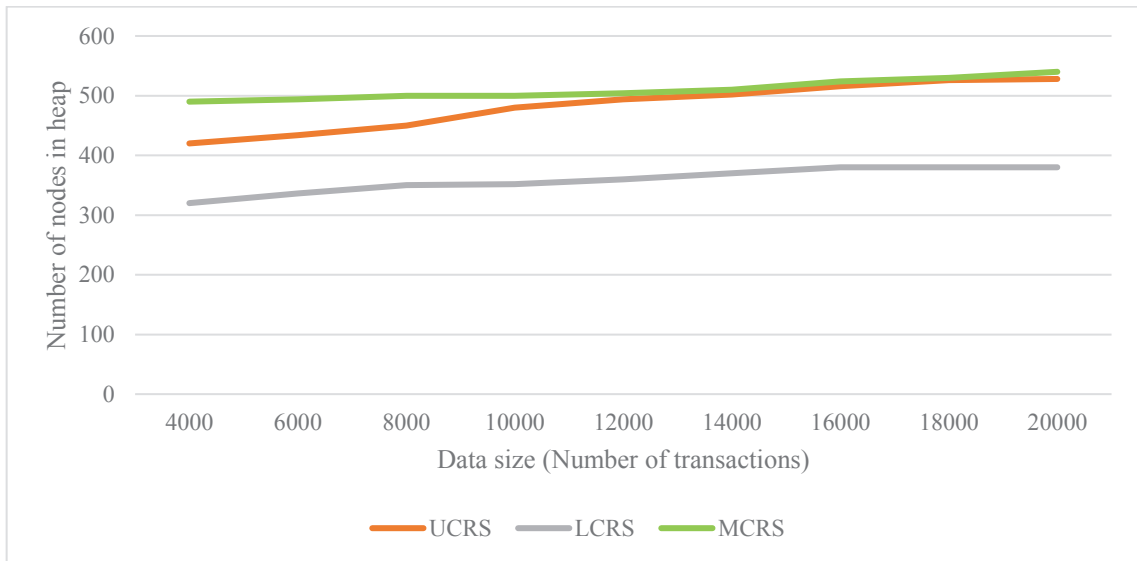


Figure 5.16. Number of nodes in heap while changing the dataset size on Dataset D2 (M = 600 nodes)

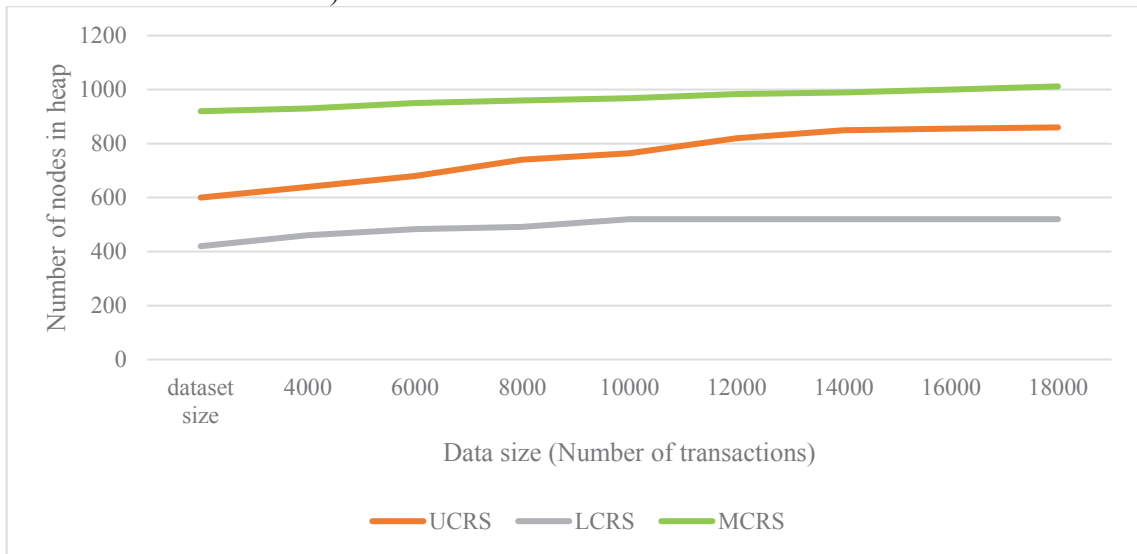


Figure 5.17. Number of nodes in heap while changing the dataset size on Dataset D2 (M = 1200 nodes)

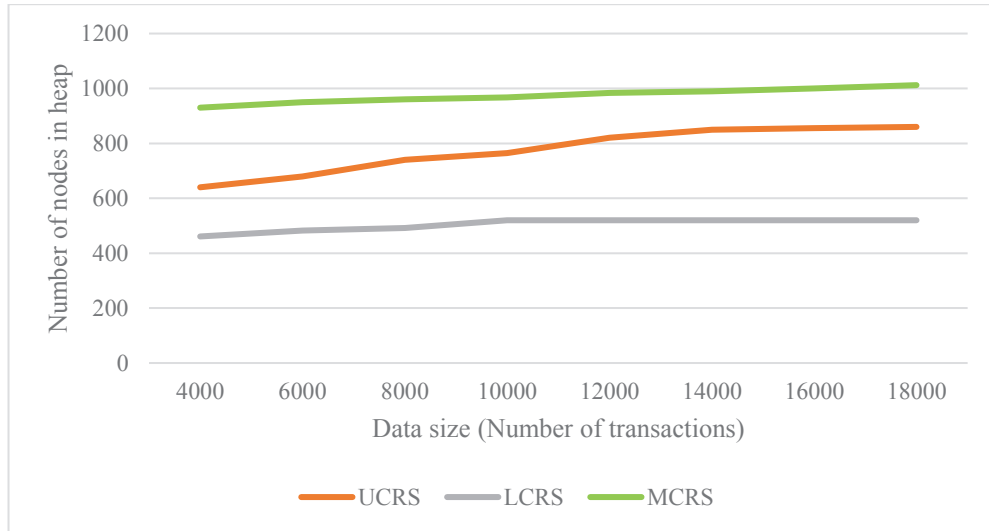


Figure 5.18. Number of nodes in heap while changing the dataset size on Dataset D2 (M = 1800 nodes)

From Figures 5.13, 5.14, 5.15, 5.16, 5.17 and 5.18 it is noticed that; the heap sizes when D2 is used are greater than heap sizes when D1 is used, this is due to the larger density of D2 over the density of D1. As a result, increasing heap sizes when D2 is used has a negative effect on the retrieved number of patterns. When the heap reservoir size gets larger, the sample reservoir size in the reservoir gets smaller. As a solution for the larger heap reservoir size and lower recall, LCRS is proposed, it has a limited heap size, so it allows more space to sample in the reservoir. Also, it is noticed that the heap reservoir size of MCRS is larger than the heap reservoir sizes in UCRS and LCRS, and it can have more than a double size of LCRS, as result this has a negative effect on the retrieved number of patterns, so MCRS supports the idea of UCRS and LCRS.

Table 5.4 shows the correlation between heap reservoir size and whole reservoir sizes for UCRS, LCRS and MCRS algorithms, the used datasets are D1 and D2. M is assigned to 600 and 1200 nodes. From the table is noticed that heap sizes on D2 are greater than heap sizes on D1 for the same algorithm. The heap size is growing until a certain data size, this size depends on the nature of dataset and whole reservoir size, e.g., for UCRS and MCRS, the curve grows fast for the data sizes (4000 - 14000) and it becomes slower after 12000. While for LCRS; the curve grows for the data sizes (4000 - 14000) and it becomes constant after 14000 according to the maximum assigned heap size. heap sizes in LCRS are lower than it in UCRS and MCRS, this is because of the

pruning method that is added to LCRS. heap sizes in MCRS are larger than it in UCRS and LCRS, this is because of deleting strategy in MCRS, which deletes the maximum connected edge, and as a result these edges are deleted but its nodes are still connected with other edges, so these nodes still in heap and keep it with a large size, this behaviour of MCRS motivates the idea of LCRS and UCRS.

Table 5.4. Correlation between heap reservoir size and whole reservoir sizes for UCRS, LCRS and MCRS

Dataset	M (nodes)	(Heap reservoir size / whole reservoir size) *100%		
		UCRS	LCRS	MCRS
D1	600	53% - 70%	40% - 60%	77% - 84%
	1200	38% - 60%	26% - 51%	72% - 82%
	1800	27% - 44%	27% - 28%	50% - 53%
D2	600	70% - 88%	23% - 18%	81% - 90%
	1200	50% - 72%	35% - 43%	76% - 84%
	1800	33% - 48%	23% - 29%	51% - 56%

5.4. Discussion on Experiments

Table 5.5 shows the summary of execution time speed-ups and recall results for Triest, UCRS, LCRS, SR and OSR algorithms on D1 and D2 datasets. M is assigned to 600, 1200 and 1800 nodes.

Triest always achieves best speed-up but worst recall on both datasets D1 and D2. Since it manipulates edges without heap management like UCRS and LCRS, processing edges has less time than processing subgraphs like SR and OSR.

LCRS has the closest speed-up to Triest on both datasets and outperforms all the algorithms in terms of recall on D1. The reason for that is the novel method in UCRS and LCRS, which is designed to replace the random edge deletion in whole reservoir with controlled edge deletion. The idea behind this method is not to lose high degree nodes, which have more impact on recall. Higher recall is gained since more connected (higher degree) nodes remain in the sample reservoir, while in Triest the edge deletion from

sample reservoir is done randomly, by this way of edge deletion; patterns that are more important can be lost. for higher values of total number of nodes in the sample e.g., 1800 instead of 1200 or 600, higher recall is achieved. This is because increased maximum whole reservoir size brings increased sample reservoir size, which represents the original graph better.

UCRS again is very close to Triest in speed-up on both datasets. It has much better recall than Triest on D1, and a slightly better recall on D2. The reason for these results is the novel method in UCRS that is designed to replace the random edge deletion in sample reservoir with controlled edge deletion.

MCRS has lower speed-up than UCRS, since the heap size in MCRS is larger than the heap size in UCRS, and it needs more time to manage it, also it has the lowest recall among all the other five algorithms, since in MCRS the higher connectivity edge is a candidate to be removed to keep a fixed memory size and replaced with edges, while in UCRS and LCRS the less connected edge is removed whenever the whole reservoir is full, this way supports the motivation for UCRS and LCRS, where less connected edges are candidates to be removed, that can keep important patterns in the sample reservoir, and as a result the recall increases.

SR is slightly faster than the slowest algorithm (OSR) over all the speed ups of OSR, so it is the second slower algorithm after OSR on D1 and D2. The reason for that is; both of them processing subgraphs and not edges, while handling subgraphs needs higher time complexity than handling edges. SR achieves best recall on D2 only, since D2 is denser than D1.

OSR is the slowest algorithm on both datasets D1 and D2 with best recall or close to best on D2 only, OSR recall to SR recall is $(0.74/0.72 - 0.85/.85 - 0.88/.87)$ on D1 and equal recall on D2(0.93). OSR best recall on D2 only since D2 is denser than D1. The reason for these results since OSR handling subgraphs which consumes higher execution time than handling edges, in addition it is a modified version of SR, that it has an added procedure over SR, which is called “skip optimization” procedure to explore neighbourhood efficiently.

Table 5.5. Summary of Execution Time Speed-ups and Recall Results

Dataset	M (nodes)		Triest	UCRS	LCRS	MCRS	SR	OSR
D1	600	Speed-up	183.43	137.78	173.04	95.74	0.94	
		Recall	0.16	0.66	0.76	0.14	0.72	0.74
	1200	Speed-up	67.87	43.51	61.63	22.74	0.96	
		Recall	0.64	0.82	0.86	0.28	0.85	0.85
	1800	Speed-up	38.08	19.16	34.54	12.13	0.99	
		Recall	0.64	<u>0.89</u>	0.91	0.56	0.87	0.88
D2	600	Speed-up	200.76	126.56	172.65	109.78	0.88	
		Recall	0.13	0.16	0.19	0.04	0.93	0.93
D2	1200	Speed-up	61.76	13.82	44.57	12.96	0.88	
		Recall	0.37	0.39	0.67	0.19	0.93	0.93
	1800	Speed-up	35.08	12.03	28.26	7.715	0.72	
Recall		0.59	0.62	0.78	0.07	0.93	0.93	

Figure 5.19 shows the summary of execution time speed-ups and recall results for Triest, UCRS, LCRS, SR and OSR algorithms on the datasets D1 and D2. M is assigned to 600, 1200 and 1800 nodes.

Triest always scores the best speed-up but worst recall on both datasets D1 and D2. This is because it handles edges without heap management like UCRS and LCRS, working on edges has less time than handling subgraphs like SR and OSR.

LCRS has the closest speed-up to Triest on both datasets and outperforms all the algorithms in terms of recall on D1. The reason for that is the novel method in UCRS and LCRS, which is designed to replace the random edge deletion in sample reservoir with controlled edge deletion. The working idea of this method is not to lose high connected edges, which have more impact on recall. Higher recall is gained since more connected (higher degree) nodes remain in the sample reservoir, while in Triest the edge deletion from sample reservoir is done randomly, by this way of edge deletion; patterns that are more important can be lost. for higher values of total number of nodes in the sample e.g., 1800 instead of 1200 or 600, higher recall is achieved. This is because increased

maximum whole reservoir size brings increased sample reservoir size, which represents the original graph better.

UCRS achieves a very close speed-up to Triest on both datasets. It has much better recall than Triest on D1, and a slightly better recall on D2. The reason for these results is the novel method in UCRS that is designed to replace the random edge deletion in sample reservoir with controlled edge deletion.

MCRS has lower speed-up than UCRS, since the heap size in MCRS is larger than the heap size in UCRS, and it needs more time to manage it, also it has the lowest recall among all the other five algorithms., since in MCRS the higher connectivity edge is a candidate to be removed to keep a fixed memory size and replaced with edges, while in UCRS and LCRS the less connected edge is removed whenever the whole reservoir is full, this way supports the motivation for UCRS and LCRS, where less connected edges are candidates to be removed, that can keep important patterns in the sample, and as a result the recall increases.

SR is slightly faster than the slowest algorithm (OSR), so it is the second slower algorithm after OSR on D1 and D2. The reason for that is; both of them processing subgraphs in the sample reservoir and not edges, handling items of subgraphs needs higher time complexity than handling items of edges. SR has best recall on D2 only, since D2 is denser than D1.

OSR is the slowest algorithm on both datasets D1 and D2 with best recall or close to best on D2 only. OSR best recall on D2 only since D2 is denser than D1. The reason for these results since OSR handling subgraphs which consumes higher execution time than handling edges, in addition it is a modified version of SR, that it has an added procedure over SR, which is called “skip optimization” procedure to explore neighbourhood efficiently.

On D1 and $M=1800$; *LCRS and UCRS achieve higher recall* than the other compared algorithms, it noticed that as the M size increases, the recall of LCRS increases, and on D2, and $M=1800$; the recall of LCRS becomes closer to SR and OSR, by applying extreme values of M on D2, *LCRS can achieve an equal recall to SR and OSR*, the reason for that is as the whole reservoir size increases, the sample reservoir size increases and the number on non-important edges were deleted and more important edges that can affect the patterns are increases.

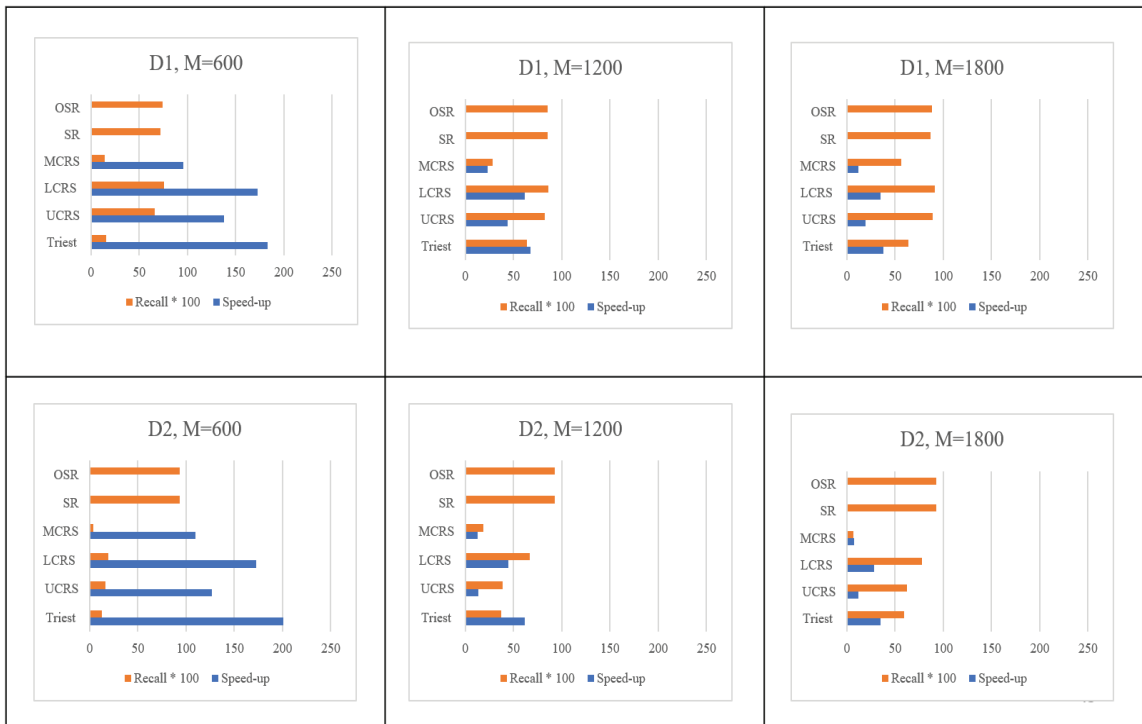


Figure 5.19. Summary of Recall and Speed-up on Datasets D1 and D2

CHAPTER 6

CONCLUSION

Frequent subgraph mining is defined as finding all the subgraphs in a graph that appear more than a given support threshold. It is a widely studied problem as it results in the discovery of recurrent structures. Frequent subgraph mining process consists of two phases, i.e., candidate generation and support computation (Dhiman and Jain, 2016).

There are several challenges of Frequent Subgraph Mining, they are as follows.

- 1) the total number of frequent subgraphs can become too large to allow a full enumeration using reasonable computational resources.
- 2) Subgraph isomorphism process is the most expensive step since it is an NP-complete problem.
- 3) Graph related operations such as subgraph testing, generally have higher time complexity than the corresponding operations on itemsets, sequences, and trees have higher time complexity.
- 4) candidate generation which occur in Apriori based approach, candidate generation is a very complicated and costly process and second, the pruning of the false positives is costly as subgraph isomorphism is NP- Complete.
- 5) mining frequent patterns from a large data set, such mining often generates a huge number of patterns satisfying the minimum support threshold, A large pattern will contain an exponential number of smaller, frequent sub-patterns.
- 6) distributed FSM from single massive graphs, due to not only the special constraints of FSM algorithm design, but also the deficient support from existing distributed programming frameworks.
- 7) large sizes of graphs, therefore, a natural solution is to reduce the size of the call graph with the use of a compression-based approach. This naturally results in loss of information,
- 8) privacy preserving data mining of graphs is especially challenging, because background information about many structural characteristics such as the node degrees or structural distances can be used in order to mount identity-attacks on the nodes.
- 9) Dynamic graphs is challenging since most existing frequent subgraph mining algorithms are devised to handle static graphs.
- 10) Response time in continuous evolving graphs. Such dynamic applications require quick responses to queries to a number of traditional applications such as the shortest path problem or connectivity queries. Such queries are an enormous challenge, since it is

impossible to restore the massive volume of the data for future analysis. 11) Subgraph matching in dynamic graphs due to the emerging use of dynamic graphs. 12) single large graph is challenging due to the very high complexity of handling it. This work focuses on the last four challenges, to minimize the challenges in such dynamic environments, sampling is used to produce approximate algorithms.

In this work, three algorithms (UCRS, LCRS and MCRS) are proposed for approximate frequent subgraph mining (FSM) in evolving graphs, where edge/vertex can be arbitrary added using a fixed sized whole reservoir. Whole reservoir keeps the sample reservoir and the heap reservoir, the sample reservoir represents the characteristics of the original dynamic graph and allows dynamic algorithms to work on reduced sized graph, while the heap reservoir consists of distinct nodes of the edges in the sample reservoir. The three algorithms manage the edges in the sample reservoir with the help of an auxiliary heap reservoir. This heap reservoir keeps the degrees of the nodes corresponding to the edges in the sample reservoir, the node degrees are kept in ascending order starting from the root node, and the node with minimum degree is retrieved directly whenever necessary. In UCRS and LCRS the edges of low connectivity nodes are deleted from the sample, which can maximize accuracy without sacrificing time and space. The third algorithm MCRS is proposed as a kind of heuristic, it works in a similar manner to UCRS and LCRS, while the main difference is in selecting the candidate edge to be deleted from sample reservoir, whenever the whole reservoir is full, in this algorithm the candidate edge is the edge with maximum degree of its nodes, by this way, the high connectivity edges are deleted from the sample reservoir, as a result, the recall is expected to be decreased. So, the results of MCRS motivate the need for the advantages of UCRS and LCRS algorithms.

Our experimental evaluation reveals that; UCRS and LCRS can outperform state-of-the-art approaches in terms of recall and execution time. The findings are, i) on sparse datasets; LCRS achieves highest recall in comparison to recent works, ii) on dense datasets; UCRS and LCRS can't achieve the highest recall, but it can be higher than one of recent works, the recall of LCRS is getting higher and closer to the recall as the total number of nodes in the sample is getting higher, iii) on sparse and dense datasets; LCRS outperforms UCRS in recall and scalability, UCRS and LCRS achieve high scalability, they can be as good as the fastest competitor algorithm, the speed up of LCRS can be up to 99% faster than the exact algorithm. iv) on sparse datasets and high values of whole reservoir size, UCRS and LCRS can achieve highest recall with smaller closer time to

fastest algorithm, v) on dense datasets and higher values of whole reservoir sizes, the recall of UCRS and LCRS getting closer to the highest recall until it achieves the highest recall. LCRS is recommended for sparse datasets, and it is recommended for dense datasets with large values of whole reservoir size. MCRS has the worst speed-up and recall among the other proposed and competitor algorithms.

This research can be continued with some challenges left to explore as follows:

- The UCRS, LCRS and MCRS algorithms are designed to handle 3 nodes subgraph patterns, they can be extended to retrieve 4 or 5 node subgraph patterns.
- Applying different heuristics in controlling the edge deletion from sample reservoir (other than edges with minimum or maximum node degrees), then the performance evaluation can be measured and compared with UCRS, LCRS and MCRS algorithms.
- Shuffling the used datasets in the experiments, then monitoring the new results and comparing them with the previous results.
- Determining the proper sample reservoir size for the UCRS, LCRS and MCRS algorithms.
- Modifying a new CRS algorithm based on subgraph sample reservoir and trying to find a way that can reduce the execution time.

REFERENCES

- Abdelhamid, E. et al. (2017) ‘Incremental frequent subgraph mining on large evolving graphs’, *IEEE Transactions on Knowledge and Data Engineering*, 29(12), pp. 2710–2723. doi: 10.1109/TKDE.2017.2743075.
- Aggarwal, C. and Wang, H. (2010) *Managing and Mining Graph Data*. doi: 10.1007/978-1-4419-6045-0.
- Anand Padmanabha Iyer, Zaoxing Liu, Xin Jin, Shivaram Venkataraman, Vladimir Braverman, Ion Stoica, U. (2018) ‘ASAP: Fast, Approximate Graph Pattern Mining at Scale’, *Proceedings of the 13th USENIX Symposium on Operating System Design and Implementation (OSDI’18)*, pp. 745–761.
- Anis, M. and Nasir, U. (2018) *Mining Big and Fast Data: Algorithms and Optimizations for Real-Time Data Processing*. Available at: <http://www.diva-portal.org/smash/get/diva2:1195589/FULLTEXT02.pdf>.
- Aslay, C. et al. (2018) ‘Mining Frequent Patterns in Evolving Graphs’, *The 27th ACM International Conference on Information and Knowledge Management (CIKM ’18)*, pp. 923–932. doi: 10.1145/3269206.3271772.
- Berlingerio, M. and Bonchi, F. (2009) ‘Mining graph evolution rules’, *Proceedings of the European Conference on Machine Learning and Knowledge Discovery in Databases: Part I, (ECML PKDD, pp. 115–130*.
- Bhatia, V. and Rani, R. (2018) ‘Ap-FSM: A parallel algorithm for approximate frequent subgraph mining using Pregel’, *Expert Systems with Applications*. Elsevier Ltd, 106, pp. 217–232. doi: 10.1016/j.eswa.2018.04.010.
- Bifet, A. and Gavaldà, R. (2011) ‘Mining Frequent Closed Graphs on Evolving Data Streams’, *Intelligent Data Analysis*, 15(1), pp. 29–48. doi: 10.3233/IDA-2010-0454.
- Borgwardt, K. M., Kriegel, H. P. and Wackersreuther, P. (2006) ‘Pattern mining in frequent dynamic subgraphs’, *Proceedings - IEEE International Conference on Data Mining, ICDM*, pp. 818–822. doi: 10.1109/ICDM.2006.124.
- Braun, P. et al. (2014) ‘Effectively and efficiently mining frequent patterns from dense graph streams on disk’, *Procedia Computer Science*, 35(C), pp. 338–347. doi: 10.1016/j.procs.2014.08.114.
- Chakrabarti, D. and Faloutsos, C. (2006) ‘Graph Mining : Laws , Generators , and Algorithms’, *ACM Computing Surveys*, 38(1), pp. 2-es. doi: 10.1145/1132952.1132954.
- Chakraborty, M., Byshkin, M. and Crestani, F. (2020) ‘Patent citation network analysis: A perspective from descriptive statistics and ERGMs’, *PLoS ONE*, 15(12 December), pp. 1–28. doi: 10.1371/journal.pone.0241797.
- Chi, Y. et al. (2004) ‘Moment : Maintaining Closed Frequent Itemsets over a Stream Sliding Window’, *Proceedings - Fourth IEEE International Conference on Data Mining, ICDM 2004*, pp. 59–66.

- Cook, D. J. and Holder, L. B. (2007) ‘Mining graph data’.
- Cuzzocrea, A. et al. (2015) ‘Edge-based mining of frequent subgraphs from graph streams’, *Procedia - Procedia Computer Science*. Elsevier Masson SAS, 60, pp. 573–582. doi: 10.1016/j.procs.2015.08.184.
- Dhiman, A. and Jain, S. K. (2016) ‘Frequent subgraph mining algorithms for single large graphs - A brief survey’, *Proceedings - 2016 International Conference on Advances in Computing, Communication and Automation, ICACCA 2016*. doi: 10.1109/ICACCA.2016.7578886.
- Dinari, H. and Naderi, H. (2014) ‘A Survey of Frequent Subgraphs and Subtree Mining Methods’, 14(1), pp. 39–57.
- Elseidy, M., Abdelhamid, E. and Skiadopoulos, S. (2014) ‘GRAMI: Frequent Subgraph and Pattern Mining in a Single Large Graph’, *Proceedings of the VLDB Endowment*, 7(7), pp. 517–528. doi: 10.14778/2732286.2732289.
- Fournier-Viger, P. et al. (2019) ‘TKG: efficient mining of top-K frequent subgraphs’, *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 11932 LNCS(December), pp. 209–226. doi: 10.1007/978-3-030-37188-3_13.
- Fournier-Viger, P. et al. (2020) ‘A survey of pattern mining in dynamic graphs’, *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 10(6). doi: 10.1002/widm.1372.
- Gemulla, R., Lehner, W. and Haas, P. J. (2006) ‘A dip in the reservoir: Maintaining sample synopses of evolving datasets’, *VLDB 2006 - Proceedings of the 32nd International Conference on Very Large Data Bases*, pp. 595–606.
- Hall, B., Jaffe, A. and Trajtenberg, M. (2001) ‘The NBER Patent Citation Data File: Lessons, Insights and Methodological Tools’, (October 2014). doi: 10.3386/w8498.
- Han, J., Pei, J. and Yin, Y. (2000) ‘Mining Frequent Patterns without Candidate generation’, *ACM SIGMOD Record*, 29(2), pp. 1–12.
- Hu, P. and Lau, W. C. (2013) ‘A Survey and Taxonomy of Graph Sampling’, pp. 1–34. Available at: <http://arxiv.org/abs/1308.5865>.
- Huan, J. et al. (2004) ‘Spin: mining maximal frequent subgraphs from graph databases’, *Proceedings of the 10th ACM SIGKDD international conference on Knowledge discovery and data mining*, (1), pp. 581–586. doi: 10.1145/1014052.1014123.
- Huan, J., Wang, W. and Prins, J. (2003) ‘Efficient mining of frequent subgraphs in the presence of isomorphism’, *The Proceedings of 3rd IEEE International Conference on Data Mining*, pp. 2–5. doi: 10.1109/ICDM.2003.1250974.
- Inokuchi, A. and Washio, T. (2012) ‘FRISSMiner: Mining frequent graph sequence patterns induced by vertices’, *IEICE Transactions on Information and Systems*, E95-D(6), pp. 1590–1602. doi: 10.1587/transinf.E95.D.1590.
- Inokuchi, A., Washio, T. and Motoda, H. (2000) ‘An Apriori-Based Algorithm for Mining Frequent Substructures from Graph Data’, pp. 13–23. doi: 10.1007/3-540-45372-5_2.

- Iyer, A. Liu, Z. Jin, X. Venkataraman, S. Braverman, V. Stoica, I. (2018) ‘ASAP: Fast, Approximate Graph Pattern Mining at Scale’, Proceedings of the 13th USENIX Symposium on Operating System Design and Implementation (OSDI’18), pp. 745–761.
- Jiang, C., Coenen, F. and Zito, M. (2004) ‘A Survey of Frequent Subgraph Mining Algorithms’, The Knowledge Engineering Review, 000, pp. 1–31. doi: 10.1017/S0000000000000000.
- Jiang, C., Coenen, F. and Zito, M. (2013) ‘A survey of frequent subgraph mining algorithms’, Knowledge Engineering Review, 28(1), pp. 75–105. doi: 10.1017/S0269888912000331.
- Kuramochi, M. and Karypis, G. (2001) ‘Frequent subgraph discovery’, Proceedings 2001 IEEE International Conference on Data Mining, pp. 313–320. doi: 10.1109/ICDM.2001.989534.
- Kuramochi, M. and Karypis, G. (2004) ‘GREW - A scalable frequent subgraph discovery algorithm’, Proceedings - Fourth IEEE International Conference on Data Mining, ICDM 2004, pp. 439–442. doi: 10.1109/ICDM.2004.10024.
- Kuramochi, M. and Karypis, G. (2005) ‘Finding Frequent Patterns in a Large Sparse Graph*’, Journal of Data Mining and Knowledge Discovery, 11(3), pp. 243–271. doi: 10.1007/s10618-005-0003-9.
- Lakshmi, K. and Meyyappan, T. (2013) ‘Efficient Algorithm for Mining Frequent Subgraphs (Static and Dynamic) based on gSpan’, International Journal of Computer Applications, 63(19), pp. 9–12.
- Leskovec, J. and Faloutsos, C. (2006) ‘Sampling from large graphs’, Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 2006, pp. 631–636. doi: 10.1145/1150402.1150479.
- Miyoshi, Y., Ozaki, T. and Ohkawa, T. (2011) ‘Mining Interesting Patterns and Rules in a Time-evolving Graph’, Proceedings of the International MultiConference of Engineers and Computer Scientist, I, pp. 1–6.
- Preti, G., De Francisci Morales, G. and Riondato, M. (2021) ‘MaNIACS: Approximate Mining of Frequent Subgraph Patterns through Sampling’, Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 1348–1358. doi: 10.1145/3447548.3467344.
- Qiao, F. et al. (2018) ‘A Parallel Approach for Frequent Subgraph Mining in a Single Large Graph Using Spark’, Applied Sciences, 8(2), p. 230. doi: 10.3390/app8020230.
- Ranu, S. and Singh, A. K. (2009) ‘GraphSig: a scalable approach to mining significant subgraphs in large graph databases’, Proceedings - International Conference on Data Engineering, pp. 844–855. doi: 10.1109/ICDE.2009.133.
- Ray, A., Holder, L. and Choudhury, S. (2014) ‘Frequent Subgraph Discovery in Large Attributed Streaming Graphs’, JMLR: Workshop and Conference Proceedings, 36, pp. 166–181.
- Sahu, S. et al. (2021) ‘Mining approximate frequent subgraph with sampling techniques’, Materials Today: Proceedings. Elsevier Ltd, (xxxx). doi:

10.1016/j.matpr.2021.03.425.

- De Stefani, L. et al. (2016) ‘TRIÈST: Counting local and global triangles in fully-dynamic streams with fixed memory size’, Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 13-17-Aug(X), pp. 825–834. doi: 10.1145/2939672.2939771.
- Stutzbach, D. et al. (2006) ‘Sampling techniques for large, dynamic graphs’, Proceedings - IEEE INFOCOM. doi: 10.1109/INFOCOM.2006.39.
- Vitter, J. S. (1985) ‘Random Sampling with a Reservoir’, ACM Transactions on Mathematical Software (TOMS), 11(1), pp. 37–57. doi: 10.1145/3147.3165.
- Wang, T. et al. (2011) ‘Understanding graph sampling algorithms for social network analysis’, Proceedings - International Conference on Distributed Computing Systems, pp. 123–128. doi: 10.1109/ICDCSW.2011.34.
- Wu, Y. et al. (2017) ‘Evaluation of Graph Sampling: A Visualization Perspective’, IEEE Transactions on Visualization and Computer Graphics, 23(1), pp. 401–410. doi: 10.1109/TVCG.2016.2598867.
- Yan, X. and Han, J. (2003) ‘CloseGraph : Mining Closed Frequent Graph Patterns’, Proc. of the 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD-2003), 6, pp. 1–10.
- Yan, X. and Jiawei, H. (2002) ‘gSpan: Graph-based substructure pattern mining’, Journal of Chemical Information and Modeling, 53(9), pp. 1689–1699. doi: 10.1017/CBO9781107415324.004.
- Yates, D. S., Moore, D. S. and Starnes, D. S. (2002) The practice of statistics. Macmillan.
- Zaharia, M. et al. (2010) ‘Spark : Cluster Computing with Working Sets’.
- Zhang, F. et al. (2017) ‘A visual evaluation study of graph sampling techniques’, IS and T International Symposium on Electronic Imaging Science and Technology, pp. 110–117. doi: 10.2352/ISSN.2470-1173.2017.1.VDA-394.

VITA

Nourhan N. I. El-Dabba Abuzayed received her BSc degree in Computer Engineering from the Islamic University of Gaza (IUG), Palestine. From 2005 to 2010 she worked as a software engineer in Palestinian National Internet Naming Authority (PNINA). From 2014 to 2016 she completed her master study in Computer Engineering from Izmir Institute of Technology (IzTech), during her master study; she worked on a project of The Scientific and Technological Research Council of Turkey (TÜBİTAK) under ARDEB 3501 Project No: 114E779. In 2016 she joined the PhD program at (IzTech), during her PhD study she was accepted as a PhD scholar from the Islamic Development bank (IDB).

The List of Her Publications:

- **Abuzayed, Nourhan & Ergenç, Belgin.** (2022) “Approximate Frequent Subgraph Mining on Dynamic Graphs”, Knowledge and Information Systems (KAIS) Journal, Submitted
- **Abuzayed, Nourhan & Ergenç, Belgin.** (2017) “Comparison of Dynamic Itemset Mining Algorithms for Multiple Support Thresholds”, 21st International Database Engineering and Applications Symposium IDEAS 2017, Byte Press, 12-14 July, Bristol, England, pp 309-316
- **Abuzayed, Mazen, El-Dabba, Nourhan, Frary, Anne and Doganlar, Sami.** (2016) “GDdom: An online tool for calculation of dominant marker gene diversity”, Biochem. Genet. 55, pp 155-157
- **Abuzayed, Nourhan & Ergenç, Belgin.** (2016) “Dynamic Itemset Mining under Multiple Support Thresholds”, In: The 2nd International Conference on Fuzzy Systems and Data Mining, China Macau, pp 11-14