2019

Incremental and parallel algorithms for dense subgraph mining

Apurba Das
Iowa State University

Incremental and parallel algorithms for dense subgraph mining

by

Apurba Das

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Major: Computer Engineering (Software Systems)

Program of Study Committee:
Srikanta Tirthapura, Major Professor
Pavan Aduri
Suraj Kothari
Chinmay Hegde
Neil Zhenqiang Gong

Iowa State University

Ames, Iowa

2019

DEDICATION

I would like to dedicate this thesis to my wife Joshita and to my son Rishan without whose support I would not have been able to complete this work. I would also like to thank my friends and family for their loving guidance during this work.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGMENTS

I would like to take this opportunity to express my thanks to those who helped me with various aspects of conducting research and the writing of this thesis. First and foremost, Dr. Srikanta Tirthapura for his guidance, patience and support throughout this research and the writing of this thesis. His insights and words of encouragement have often inspired me and renewed my hopes for completing my graduate education. I would also like to thank my committee members for their efforts and guidance to this work. I would additionally like to thank all my teachers who inspired me to enter into this research world and taught me the art of asking questions which is the seed of a successful research endeavor.

I would like to thank all the staff members in the Department of Electrical and Computer Engineering as well as in Graduate College for prompt responses to all my questions and doubts.

Next I would like to thank all my colleagues at our lab with whom I spent wonderful times for these five precious years of my life through discussing interesting problems and loads of fun. Special thanks goes to Vahid with whom I worked on multiple problems. I appreciate his thoughtful comments on the works and the code review.

I would like to thank all my friends and neighbors in Ames, IA. Life would become very hard away from home without their company.

I would like to thank my family and friends for their love, care, and support. Last but not the least, I would like to thank my wonderful wife Joshita who shared with me the ups and downs throughout my graduate career and never complained, but always encouraged. I appreciate her patience throughout this journey.

Finally, I am grateful to the support by the National Science Foundation under Grant IIS-1527541 and CCF-1725702.

ABSTRACT

The task of maintaining densely connected subgraphs from a continuously evolving graph is important because it solves many practical problems that require constant monitoring over the continuous stream of linked data often represented as a graph. For example, continuous maintenance of a certain group of closely connected nodes can reveal unusual activity over the transaction network, identification, and evolution of active groups in the social network, etc. On the other hand, mining these structures from graph data is often expensive because of the complexity of the computation and the volume of the structures (the number of densely connected structures can be of exponential order on the number of vertices in the graph). One way to deal with the expensive computations is to consider parallel computation.

In this thesis, we advance the state of the art by developing provably efficient algorithms for mining maximal cliques and maximal bicliques; two fundamental dense structures.

First, we consider the design of efficient algorithms for the maintenance of maximal cliques and maximal bicliques in an evolving network. We observe that it is important to locate the region of the graph in the event of the update so that we can maintain the structures by computing the changes exactly where it is located. Following this observation, we design efficient techniques that find appropriate subgraphs for identifying the changes in the structures. We prove that our algorithms can maintain dense structures efficiently. More specifically, we show that our algorithms can quickly compute the changes when it is small irrespective of the size of the graph. We empirically evaluate our algorithms and show that our algorithms significantly outperform the state of the art algorithms.

Next, we consider parallel computation for efficient utilization of the multiple cores in a multicore computing system so that the expensive mining tasks can be eased off and we can achieve better speedup than their efficient sequential counterparts. We design shared memory parallel algorithms for the mining of maximal cliques and maximal bicliques and we prove the efficiency of the parallel algorithms through showing that the total work performed by the parallel algorithm is equivalent to the time complexity of the best sequential algorithm for doing the same task. Our experimental study shows that we achieve good speedup over the prior state of the art parallel algorithms and significant speedup over the state of the art sequential algorithms. We also show that our parallel algorithms scale almost linearly with the increase in the processor cores.

CHAPTER 1. OVERVIEW

1.1 Introduction

Graphs are ubiquitous in representing relationships among data points in social network, telecommunication network, transaction network etc. Changes in these networks over time are well represented using dynamic graph where new edges/vertices are added and existing edges/vertices are deleted. For example, consider friendship network where a new edge is added when two unknown persons become friend and an existing edge is deleted when two friends become unfriend.

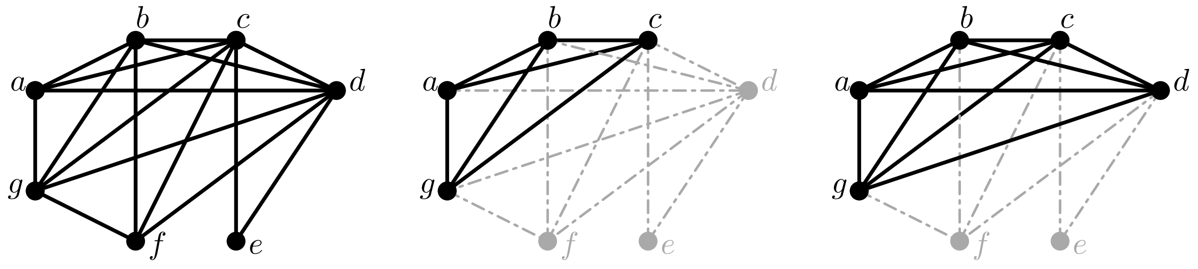
At a high level, dense structure in a graph is a subgraph where the nodes are tightly connected. Mining dense structures from a dynamic graph is an important task as it reveals many interesting properties and structures in a network. For examples, real-time identification of the stories from twitter is possible through the mining of dense subgraphs from appropriately defined graph using twitter data [10]. Broadly speaking, identifying dense structures in a graph is applicable to any task that needs to identify and analyze communities among users in micro-blogging platforms [68], to identify groups of closely linked people in a social network [61, 85, 94], to identify web communities [56, 82, 120], to construct the Phylogenetic Tree of Life [43, 124, 154] etc.

Many graph mining tasks are based upon *maximal clique* and *maximal biclique*, which are among the fundamental dense structures in a network. Examples where maximal cliques are used include the work of Palla et al. [113] on clustering and community detection in social and biological network, Rokhlenko et al. [119] on the study of co-expression of genes under stress, Harley et al. [62] on integrating different types of genome mapping data, Chateau et al. [27] on maintaining common intervals of genomes, Koichi et al. [79] on discovering chemical structures from large-scale chemical databases, various problems on mining biological data [60, 64, 102, 28, 72, 119, 157], inference in graphical models [80] etc. The work of Kumar et al. [82] on detecting cyber-communities from the web graph, Murata et al. [108] on identifying user communities from web log data, Lehmann et al. [85] on community detection in collaboration networks, Braun et al. [22] on detecting credit-card fraud in transaction networks, various problems in bioinformatics [159, 91, 15], social network analysis [85, 57] are all based on mining maximal bicliques in an appropriately defined bipartite graph. Another application of maximal biclique is in the task of mining closed item-sets from transactional databases [114]. One approach to closed item-sets is to enumerate maximal bicliques from a bipartite graph representing the transactional database where the different transactions are in one partition and the set of items are in the other partition, with edges connecting a transaction to an item if the item was included in that transaction [88].

1.2 Dense Structures in a Graph

There are many different types of dense subgraphs in the literature. In this work we focus on two fundamental dense structures *maximal cliques* and *maximal bicliques* and will discuss here about these two structures in greater detail and then we will discuss about other dense structures briefly.

Maximal Clique: The maximal clique is perhaps the most fundamental and widely studied dense subgraph. Let $G = (V, E)$ be an undirected unweighted graph on vertex set V and edge set E. A clique in G is a set of vertices $C \subseteq V$ such that any two vertices in C are connected to each other in G. A clique is called maximal if it is not a proper subset of any other clique (see Fig. 1.1). In this work we study the problem of Maximal Clique Enumeration (MCE) from a graph, which requires to enumerate all cliques (complete subgraphs) in the graph that are maximal.



(a) Input Graph G **(b)** Non-Maximal Clique in G **(c)** Maximal Clique in G

Figure 1.1: Maximal clique in a graph

MCE is a computationally hard problem since it is harder than the *maximum* clique problem, a classical NP-complete combinatorial problem that asks to find a clique of the largest size in a graph. Note that maximal clique and maximum clique are two related, but distinct notions. A maximum clique is also a maximal clique, but a maximal clique need not be a maximum clique. The computational cost of enumerating maximal cliques can be higher than the cost of finding the maximum clique, since the output size (set of all maximal cliques) can itself be very large. Moon and Moser [104] showed that a graph on n vertices can have as many as $3^{n/3}$ maximal cliques, which is proved to be a tight bound. Real-world networks typically do not have cliques of such high complexity and as a result, it is feasible to enumerate maximal cliques from large graphs. The literature is rich on sequential algorithms for MCE. Bron and Kerbosch [23] introduced a backtracking search method to enumerate maximal cliques. Tomita et. al [143] introduced the idea of "pivoting" in the backtracking search, which led to a significant improvement in the runtime. This has been followed up by further work such as due to Eppstein et al. [49], who used a degeneracy-based vertex ordering on top of the pivot selection strategy.

Maximal Biclique: A bipartite graph $G = (L, R, E)$ is a graph whose vertex set can be partitioned into two sets L and R such that each edge in E connects a vertex in L with a vertex in R. Given a bipartite graph $G = (L, R, E)$, a biclique (X, Y), $X \subseteq L$, $Y \subseteq R$ is a subgraph of G where for every vertex in X is connected to every vertex in Y (see Figure 1.2 for an example). A biclique is called maximal if it is not a subgraph of another biclique. Along with the study of maximal clique enumeration, In this work we study the problem of Maximal Biclique Enumeration (MBE) from a graph, which requires to enumerate all bicliques (complete bipartite subgraphs) in a bipartite graph that are maximal.

The worst-case complexity of any algorithm for MBE is necessarily high, since the number of maximal bicliques can be exponential in the number of vertices [117]. However, the number of maximal bicliques in real world graphs is typically much smaller. For example, the number of maximal bicliques is linear in the graph size for graphs with bounded arboricity [48]. Sequential methods for MBE have been studied for many decades [48, 6, 155, 97, 88, 109, 159]. The algorithm

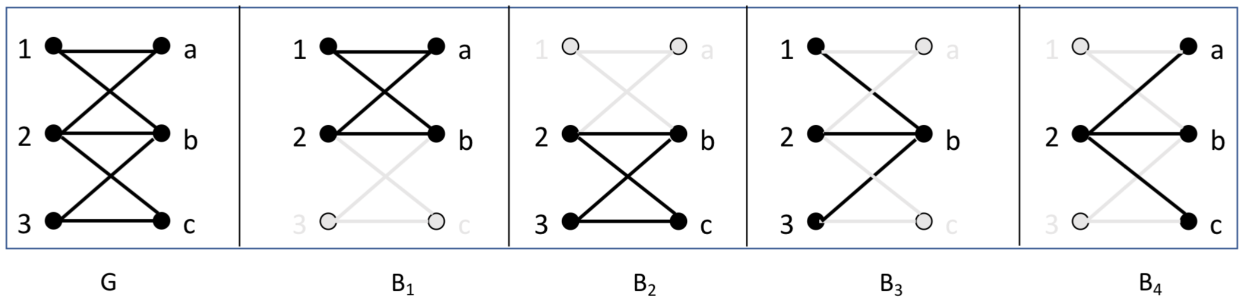


Figure 1.2: G is a bipartite graph with four maximal bicliques B_1, B_2, B_3, and B_4.

that seems to have the best practical performance is a branch-and-bound algorithm due to Liu et al. [92], which we call `MineLMBC`.

k-Core, k-Truss: A k-core is a maximally connected subgraph in which every vertex is connected to at least k other vertices (see Figure 1.3 for an example). This structure can be found in polynomial time [14] and is an important building block in many graph mining and visualization applications [7, 42, 59]. A closely related concept is the core number k of a vertex v which is the maximum k such that v is contained in a k-core but in no $k+1$ core. A recent application of k-core is due to Cheng et al. [31] for finding cluster centers of the k-means clustering algorithm [63]. Similar to the k-core, in a k-truss, every edge is connected to at least $(k-2)$-triangles. These structures are also sometimes called triangle core [121] due to analogy with the k-core. Huang et al. [65] use k-truss for community discovery in dynamic network.

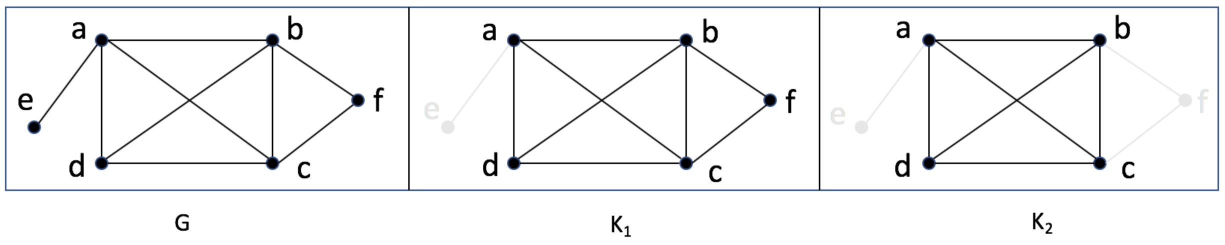


Figure 1.3: G is a simple undirected graph (which is also a 1-core) with a 2-core K_1 and a 3-core K_2.

γ-Quasi-Clique: A connected subgraph q is a γ-quasi-clique if the ratio of total number of edges in q to the number of edges of a complete graph of size $|q|$ is at least γ (called the density of a quasi-clique). A quasi-clique is maximal if it is not a subgraph of another quasi-clique. Unlike maximal clique and maximal biclique, deciding maximality of a quasi-clique is an NP-complete problem [149]. Also, it is NP-complete to decide whether there exists a γ-quasi-clique of size at least k [115]. Brunato et al. [24] use this structure and design heuristic algorithm for finding overlapping community in a network. In [17], the authors tried to find important proteins in human body by finding large quasi-clique in the protein network. Another definition of quasi-cliques [93] is based on the degree threshold γ where a subgraph q is γ-quasi-clique if degree of each vertex in q is at least $\gamma(|q|-1)$. Note that all degree based quasi-cliques are also density based quasi-cliques, but the reverse is not always true. See Figure 1.5 for an example of degree based γ-quasi-clique.

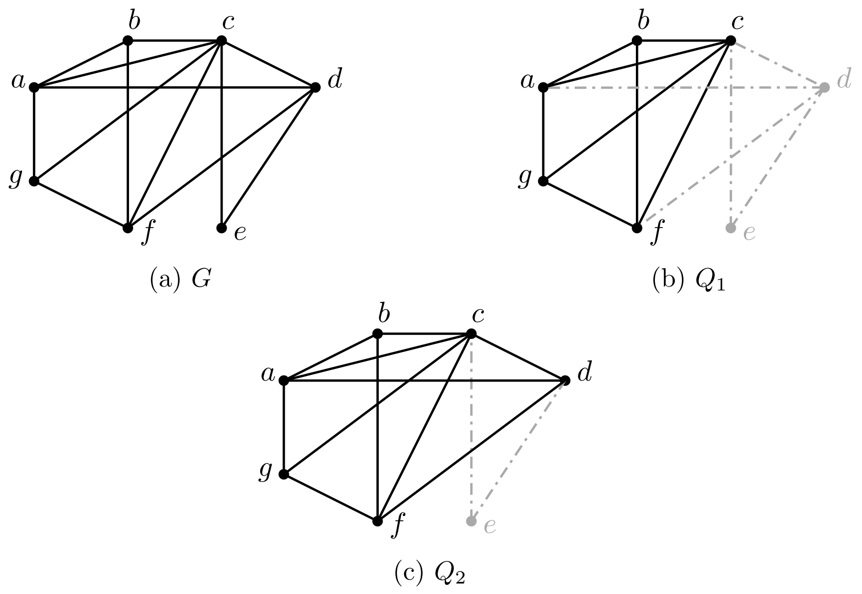


Figure 1.5: Example of degree based γ-quasi-clique with $\gamma = 0.6$. (a) the original graph G (b) vertices $\{a, b, c, f, g\}$ form a γ-quasi-clique Q_1. (c) vertices $\{a, b, c, d, f, g\}$ form a maximal γ-quasi-clique Q_2.

Quasi-Biclique [132]: A quasi-biclique is a bipartite subgraph of an undirected bipartite graph $G = (L \cup R, E)$ induced by $L' \subseteq L$ and $R' \subseteq R$ such that $\forall v \in L'$, $|R'| - |\Gamma_{R'}(v)| \leq \epsilon$ and $\forall v \in R'$,

$|L'| - |\Gamma_{L'}(v)| \leq \epsilon$ for small integer values of ϵ where $\Gamma_V(u)$ denote all the vertices in V that are adjacent to u. This is one of many definitions of a quasi-biclique [100, 154, 89, 133]. Sim et al. [132] use this type of quasi-bicliques in modeling correlation between stocks and financial ratios.

Densest Subgraph: A densest subgraph of the original graph $G = (V, E)$ is a subgraph G_U of G induced by U with density $d^* = \max_{U \subseteq V} \frac{E(G_U)}{|U|}$. This structure has been widely studied for over a decade [26, 9, 77] along with many variants starting from 1984 when Goldberg [58] first designed a polynomial time algorithm for finding a densest subgraph in a graph using the technique of maximum flow algorithm. However when size of the subgraph is imposed, the problem of finding densest subgraph becomes NP-hard [9]. This structure is useful in many applications including index construction for graph reachability and distance queries [33, 128, 70], decomposing the graph into locally-dense components [139] etc.

Densest Bipartite Subgraph: A densest bipartite subgraph is a bipartite subgraph induced by $L' \subseteq L$ and $R' \subseteq R$ of an undirected bipartite graph $G = (L \cup R, E)$ with density $d^* = \max_{L' \subseteq L, R' \subseteq R} \frac{e(L', R')}{\sqrt{|L'||R'|}}$ where $e(L', R')$ denote the total number of edges whose one endpoint is in L' and the other endpoint is in R'. In [8], the author uses this definition for developing a general technique that can be used for finding dense subgraph near a targeted vertex by exploring only a portion of the graph.

Triangle Densest Subgraph (TDS): A triangle densest subgraph is a subgraph of the original graph $G = (V, E)$ induced by $S \subseteq V$ with triangle density $\tau^* = \max_{S \subseteq V} t(S)/|S|$ where $t(S)$ is the number of triangles induced by S. In [147], the author defined and used this structure for finding those subgraphs which are dense (near-cliques, that misses a few edges from being cliques) but not detected using algorithms for densest subgraph. The main purpose of defining such structure is that these can be found in polynomial time unlike near-clique finding problem which is NP-hard [149].

k-Clique Densest Subgraph: A k-clique densest subgraph is a subgraph of the original graph $G = (V, E)$ induced by $S \subseteq V$ with k-clique density $h_k^* = \max_{S \subseteq V} \frac{c_k(S)}{|S|}$ where $c_k(S)$ is the number of k-cliques induced by vertex set S. Unlike densest subgraph problem with size restriction, this problem can be exactly solvable in polynomial time [101]. The purpose of introducing such definition of dense structure is to formulate a tractable problem for finding dense structure when the size is predetermined.

(p, q)-Biclique Densest Subgraph: A (p, q)-biclique densest subgraph is a bipartite subgraph induced by a vertex set $S \subseteq L \cup R$ of a undirected bipartite graph $G = (L \cup R, E)$ such that the (p, q)-biclique density $\rho_{p,q}^*(S) = \max_{S \subseteq L \cup R} \frac{c_{p,q}(S)}{|S|}$ where $c_{p,q}(S)$ is the number of (p, q)-bicliques induced by S. Mitzenmacher et al. [101] introduce this notion for characterizing dense bipartite graph and use the techniques similar to that of k-clique densest subgraph for finding these dense bipartite subgraphs.

1.3 Dynamic Graph Algorithms

Most of the applications discussed above are for batch processing meaning that given a static network, the goal is to mine the dense structures for understanding and exploring useful and interesting properties of the network. Suppose the above problems are presented in the dynamic setting where the graph evolves over time due to the addition/deletion of edges. To maintain the dense structures in an evolving graph, if we need to use the static algorithm for enumerating the structures, we need to recompute all the structures from scratch once the graph is updated. In doing so, we may end up enumerating a large number of dense structures that are not affected due to the changes in the graph. Clearly there are wasteful computations, because, we end up computing unaffected structures more than once. Thus we need new algorithms for the maintenance of dense structures that can enumerate only the structures that are (1) new dense structures that emerges due to the addition of new edges and (2) old dense structures that are no more dense due to the changes in the graph.

There are two broad classes of algorithms for dealing with the dynamic datasets. One is the *dynamic algorithm* and another is the *streaming algorithm*. The main difference between a dynamic algorithm and a streaming algorithm is that in streaming algorithm the space cost must be sub-linear on the size of the input observed so far but in dynamic algorithm space cost need not necessarily be of sub linear order but the update and query time should be as minimal as possible. A dynamic algorithm is called *incremental* when only the insertion of new data elements are considered, *decremental* when only the deletion of existing data elements are considered, and *fully-dynamic* when both insertions and deletions are considered. It poses several challenges in the maintenance of dense structures in a dynamic graph:

- The maintenance of dense structure is a kind of enumeration in the dynamic graph each time the graph is updated. It is challenging to know the maximum possible running time without knowing the maximum possible size of the changes in the structure when the graph is updated because, the computation time should be at least the time required to enumerate the structures.

- It is challenging to compute the maximum possible number of changes in the structures when the graph is updated without knowing the newly added edges or existing edges to be removed in advance.

- One goal in the dynamic algorithm is to reduce the re-computation as much as possible when the graph is updated. A strategy for the reduction in re-computation is to find a subgraph as small as possible where we can find all the new/deleted structures only due to the changes in the graph. However, it is challenging to find such a subgraph. This is because, search space changes when the definition of the structure changes. For example, search space for maximal clique and maximal biclique are different.

1.4 Parallel Algorithms

The number of dense structures in a large graph are usually large. For example, we observed that the number of maximal cliques in `Orkut` network is 2.3 billion and it takes around $29K$ seconds to enumerate these maximal cliques using one of the most efficient sequential algorithm for maximal clique enumeration such as `TTT` in a 4 core machine as well as in a 32 core machine. Clearly, the sequential algorithm can not take advantage of the power of multiple processors in a multicore computing system for such a costly computation. Here comes the power of parallel and distributed algorithms that can take advantage of the multiple processing units in enumerating the dense structures in parallel. There are mainly two paradigms of distributing the tasks: (1) *Distributed Algorithms:* distributing the graph to multiple processing units so that each unit processes on its part of the graph and result is accumulated from all the units at the end of the computations; (2) *Shared Memory Parallel Algorithms:* keep the graph in a single shared global memory and leverage the computation tasks to multiple processing units of a single multi-core computer so that each processor can access the graph from the global location and enumerate the structures in parallel. Shared memory parallelism has several advantages over distributed memory parallelism. Firstly, we need not think about partitioning the graph in shared memory parallel computation as opposed to distributed memory computation because graph is accessed by multiple processors from a single shared global memory location in shared memory computation. Secondly, shared memory has no network communication overhead as all the processors reside in a single computer. Thirdly, updating the graph becomes easy in shared memory computation in computing on dynamic graph.

There are mainly two types of models for shared memory algorithm design: (1) PRAM Model and (2) Work-Depth Model.

PRAM Model: In Parallel Random Access Model (PRAM), every computing core can access the single share global memory simultaneously at each unit of time. In this model, the total number of operations performed by the algorithm is characterized by the product of its computation time T and the number of processors P involved in the computations. We need to account the number of processors for analyzing a parallel algorithm in this model. PRAM model supports *flat-parallelism* meaning that it can run the sequential components of the algorithm in parallel.

Work-Depth Model: This model supports *nested fork-join* parallelism. The algorithm using this model can be expressed as a Directed Acyclic Graph (DAG) where each node in the graph represents a task and the directed edges represents the order or computations. The complexity of the algorithm in this model is analyzed by computing its work W, which is the cumulative cost of all the tasks in the DAG, and the depth D which is the maximum cost of all the tasks on a directed path in the DAG. This is a formal model used for the analysis of a parallel algorithm and the analysis is independent of the number of processors. The amount of parallelism can be expressed as $\frac{W}{D}$. In our research we will focus on the work-depth model in designing parallel algorithms.

Roadmap: The chapters in this thesis are organized in the following way: We summarize our contributions in this thesis in Chapter 2, related and previous works in Chapter 3. We present our work on incremental maintenance of maximal cliques in a dynamic graph in Chapter 4, shared memory parallel algorithm for maximal clique enumeration in Chapter 5, incremental maintenance of maximal bicliques in a dynamic bipartite graph in Chapter 6, and work on shared memory parallel algorithm for maximal biclique enumeration in a bipartite graph in Chapter 7.

CHAPTER 2. CONTRIBUTIONS

In this section we discuss about the problems that we solve in this thesis with high level discussions on the solution approaches and experimental observations. Broadly, we focus on the enumeration of dense subgraphs from large static and dynamic graphs and we design efficient sequential and parallel algorithms for solving such enumeration problems. We empirically evaluate our algorithms on real world and synthetic graphs with millions of vertices and tens of million of edges to show that our algorithms are substantial improvement over the state-of-the-art algorithms for solving the same problems that are magnitude of order slower than our algorithms. Here are the summary of contributions:

2.1 Incremental Maintenance of Maximal Cliques

When the graph keeps changing over time due to addition or deletion of edges, the set of maximal cliques in the original graph changes. In the event of the addition of new edges the changes in the set of maximal cliques include (1) the set of new maximal cliques that are newly formed and (2) the set of maximal cliques of the original graph that become subgraph (subsumed) of larger new maximal cliques in the updated graph. Thus the changes in the set of maximal cliques is the union of the set of new maximal cliques and the set of subsumed cliques. Our contributions are as follows:
(A) Magnitude of Change in the Set of Maximal Cliques: We present a tight analysis of the magnitude of change in the set of maximal cliques in a graph, when a set of edges are added. When a set of edges H is added to graph $G = (V, E)$ resulting in graph $G' = G \cup H = (V, E \cup H)$.

(A.1): We present nearly matching upper and lower bounds on the maximum size of $\Lambda(G, G \cup H)$, taken across all possible graphs G and edge sets H. Let $f(n)$ denote the maximum number of maximal cliques in a graph on n vertices. A result of Moon and Moser [104] shows that $f(n)$ is approximately $3^{n/3}$. We show that by the addition of a small number of edges to the graph G on n vertices, it is possible to cause a change of nearly $2f(n) \approx 2 \cdot 3^{n/3}$. We also note that this is an upper bound on the magnitude of $\Lambda(G, G')$. We present this analysis in Theorem 3.

(A.2): We encountered an error in the 50-year old result of Moon and Moser [104] on the number of maximal cliques in a graph, which is directly relevant to our bounds on the change in the set of maximal cliques. We present our correction to their result in Observation 1.

It is easy to see that the set of maximal cliques can change by very little upon the addition of edges. For instance, adding a single edge between two vertices that are part of different components can lead to only a single new maximal clique being added (the clique consisting of a single edge), and no maximal cliques subsumed, so that the total change in the set of maximal cliques is 1. Thus, we note that the magnitude of the change can vary significantly from one input instance to another.

(B) Algorithm for Maintaining Maximal Cliques: We present incremental and decremental algorithms for maintaining the set of maximal cliques of a dynamic graph. We describe our results on incremental algorithms. Results for decremental algorithms are similar. The key algorithmic contributions in this work are as follows:

(B.1) We present algorithms that take as input G and H, and enumerate the elements of $\Lambda(G, G')$ in time proportional to the size of $\Lambda(G, G')$, i.e. the magnitude of the change in the set of maximal cliques. We refer to such algorithms as *change-sensitive* algorithms. To our knowledge, these are the first provably change-sensitive algorithms for maintaining the set of maximal cliques in a dynamic graph. The time taken for enumerating newly formed cliques $\Lambda^{new}(G, G')$ is $O(\Delta^3 \rho |\Lambda^{new}(G, G')|)$ where Δ is the maximum degree of a vertex in G' and ρ is the number of edges in H. The time taken for enumerating subsumed cliques $\Lambda^{del}(G, G')$ is $O(2^\rho |\Lambda^{new}(G, G')|)$. Note that when ρ, the size of a batch of edges, is logarithmic in Δ, the cost of enumerating subsumed cliques is of the same order as that of enumerating new cliques.

Our algorithm for enumerating the change (in the set of maximal cliques) is based on an exploration of a carefully chosen subgraph of G that is local to the set of edges that have been added. Importantly, it does not iterate through existing maximal cliques in the graph to enumerate the change, either for enumerating new maximal cliques, or for subsumed maximal cliques. The key aspect of the algorithm is that through exploring this subgraph, it is able to directly "zero in" on maximal cliques that have changed (either added or subsumed). This approach reduces wasteful effort in enumeration, when compared with an approach that iterates through the set of existing maximal cliques. Based on our theoretically-efficient algorithms, we present a practical algorithm `IMCE` for enumerating new and subsumed cliques, and an efficient implementation.

(B.2) Our methods extend to the decremental case, to handle deletion of edges from the graph. They can also be applied to the fully dynamic case, where the change includes both the addition and deletion of edges from the graph. However, the fully dynamic case is not provably change-sensitive, as discussed in Section 4.4.4.

(C) Experimental Evaluation: We present empirical evaluation of our algorithm using real world dynamic graphs as well as synthetic graphs. Our experimental study shows that `IMCE` can enumerate change in maximal cliques in a large graph with of the order of a hundred thousand vertices and millions of edges within a few seconds. Our comparison with prior and recent works show that `IMCE` significantly outperform prior solutions, including ones due to Stix [135], Ottosen and Vomlel [112], and Sun et al. [136]. For example, on the `flickr-growth` graph, our algorithms are faster than [135, 112, 136] by a factor of more than a thousand. On the `flickr-growth` graph, in order to maintain the set of maximal cliques over the insertion of 250 batches of 100 edges each, `IMCE` took about 40 ms, while prior techniques took anywhere from 5 mins to 2 hrs. Further details are in Section 4.6.

2.2 Parallel Maximal Clique Enumeration on Static and Dynamic Graphs

When the number of maximal cliques in the input graph becomes very large, it becomes costly to enumerate all maximal cliques using an efficient sequential algorithm. Also, running those sequential algorithms in a multicore computing system does not help in gaining performance by utilizing the power of multiple processors when it becomes difficult to parallelize the operations in the sequential algorithm. In this work, we design shared memory parallel algorithms for the enumeration of maximal cliques from static graph and the maintenance of maximal cliques in a dynamic graph. We make the following contributions towards enumerating all maximal cliques in a simple graph.

Theoretically Efficient Parallel Algorithm: We present a shared-memory parallel algorithm `ParTTT` that takes as input a graph G and enumerates all maximal cliques in G. `ParTTT` is an efficient parallelization of the algorithm due to Tomita, Tanaka, and Takahashi [143]. Our analysis of `ParTTT` using a work-depth model of computation [18] shows that it is work-efficient when compared with [143] and has a low parallel depth. To our knowledge, this is the first shared-memory parallel algorithm for MCE with such provable properties.

Optimized Parallel Algorithm: We present a shared-memory parallel algorithm `ParMCE` that builds on `ParTTT` and yields improved practical performance. Unlike `ParTTT`, which starts with a single task at the top level that spawns recursive subtasks as it proceeds, which leads to a lack of parallelism at the top level of recursion, `ParMCE` spawns multiple parallel tasks at the top level. To achieve this, `ParMCE` uses per-vertex parallelization, where a separate sub-problem is created for each vertex and different sub-problems are processed in parallel. Each sub-problem is required to enumerate cliques that contain the assigned vertex, and care is taken to prevent overlap between sub-problems. Each per-vertex sub-problem is further processed in parallel using `ParTTT` – this additional (recursive) level of parallelism using `ParTTT` is important since different per-vertex sub-problems may have significantly different computational costs, and having each run as a separate sequential task may lead to uneven load balance. To further address load balance, we use a vertex ordering in assigning cliques to different per-vertex sub-problems. For ordering the vertices, we use various metrics such as degree, triangle count, and the degeneracy number of the vertices.

Incremental Parallel Algorithm: Next, we present a parallel algorithm `ParIMCE` that can maintain the set of maximal cliques in a dynamic graph, when the graph is updated due to the addition of new edges. When a batch of edges are added to the graph, `ParIMCE` can (in parallel) enumerate the set of all new maximal cliques that emerged and the set of all maximal cliques that are no longer maximal (subsumed cliques). `ParIMCE` consists of two parts: `ParIMCENew` for enumerating new maximal cliques, and `ParIMCESub` for enumerating subsumed maximal cliques. We analyze `ParIMCE` using the work-depth model and show that it is work-efficient relative to an efficient sequential algorithm, and has a low parallel depth. A summary of our algorithms is shown in Table 2.1.

Table 2.1: Summary of shared-memory parallel algorithms for MCE.

Algorithm	**Type**	**Description**
`ParTTT`	Static	A work-efficient parallel algorithm for MCE on a static graph
`ParMCE`	Static	A practical parallel algorithm for MCE on a static graph dealing with load imbalance
`ParIMCE`	Dynamic	`ParIMCENew`: A work-efficient parallel algorithm for enumerating new maximal cliques in a dynamic graph.
		`ParIMCESub`: A work-efficient parallel algorithm for enumerating subsumed maximal cliques in a dynamic graph.

Experimental Evaluation: We implemented all our algorithms and our experiments show that `ParMCE` yields a speedup of **15x-21x** when compared with an efficient sequential algorithm (due to Tomita et al. [143]) on a multicore machine with 32 physical cores and 1 TB RAM. For example, on the `Wikipedia` network with around 1.8 million vertices, 36.5 million edges, and around 131.6 million maximal cliques, `ParTTT` achieves a **16.5x** parallel speedup over the sequential algorithm, and the optimized `ParMCE` achieves a **21.5x** speedup, and completed in approximately two minutes. In contrast, prior shared-memory parallel algorithms for MCE [158, 45, 87] failed to handle the input graphs that we considered, and either ran out of memory ([158, 87]) or did not complete in 5 hours ([45]).

On dynamic graphs, we observe that `ParIMCE` gives a **3x-19x** speedup over a state-of-the-art sequential algorithm `IMCE` [38] on a multicore machine with 32 cores. Interestingly, the speedup of the parallel algorithm increases with the magnitude of change in the set of maximal cliques – the "harder" the dynamic enumeration task is, the larger is the speedup obtained. For example, on a dense graph such as `Ca-Cit-HepTh` (with original graph density of 0.01), we get approximately a **19x** speedup over the sequential `IMCE`. More details are presented in Section 5.5.

Techniques for Load Balancing: Our parallel methods can effectively balance the load in solving parallel MCE. As shown in Fig. 5.1, "natural" sub-problems of MCE are highly imbalanced and therefore load balancing is not trivial. In our algorithms, sub-problems of MCE are broken down into smaller sub-problems, according to the search used by the sequential algorithm [143], and this process continues recursively. As a result, the final sub-problem that is solved in a single task is not so large as to create load imbalances. Our experiments demonstrate that the recursive splitting of sub-problems in MCE is essential for achieving a high speedup over existing algorithms [143]. In order to efficiently assign these (dynamically created) tasks to threads at runtime, we utilize a *work stealing scheduler* [19, 20].

2.3 Incremental Maintenance of Maximal Bicliques

In this work we design incremental algorithms for maintaining the set of maximal bicliques in a bipartite graph when the graph keeps changing over time due to addition of edges. Similar to the maintenance of maximal cliques, we maintain the set of maximal bicliques by maintaining the set of new maximal bicliques and the set of subsumed bicliques separately. We make the following contributions towards the maintenance of maximal bicliques in a bipartite graph:

Magnitude of Change: Let $g(n)$ denote the maximum number of maximal bicliques possible in an n vertex bipartite graph. A result due to Prisner [117] shows that $g(n) \leq 2^{n/2}$, where equality occurs when n is even. We show that the change in the number of maximal bicliques when a single edge is added to the graph can be as large as $3g(n-2) \approx 1.5 \times 2^{n/2}$, which is exponential in the number of vertices in the graph. This shows that the addition of even a single edge to the graph can lead to a large change in the set of maximal bicliques in the graph. We further show that this bound is tight for the case of the addition of a single edge – the largest possible change in the set of maximal bicliques upon adding a single edge is $3g(n-2)$. For the case when more edges can be added to the graph, it is easy to see that the maximum possible change is no larger than $2g(n)$.

Enumeration Algorithm: From our analysis, it is clear that the magnitude of change in the set of maximal bicliques in the graph can be exponential in n in the worst case. On the flip side, the magnitude of change can be as small as 1 – for example, consider the case when a newly arriving edge connects two isolated vertices in the graph. Thus, there is a wide range of values the magnitude of change can take. When the magnitude of change is very large, an algorithm that enumerates the change must inevitably pay a large cost, if only to enumerate the change. On the other hand, when the magnitude of change is small, it will ideally pay a smaller cost. This motivates our search for an algorithm whose computational cost for enumerating the change is proportional to the magnitude of the change in the set of maximal bicliques.

We present an incremental algorithm, `DynamicBC`, for enumerating the change in the set of maximal bicliques when a set of edges H are added to the bipartite graph G. `DynamicBC` has two parts, `NewBC`, for enumerating new maximal bicliques, and `SubBC`, for enumerating subsumed maximal bicliques. When a batch of new edges H of size ρ is added to the graph, the time complexity of `NewBC` for enumerating Υ^{new}, the set of new maximal bicliques, is $O(\Delta^2\rho|\Upsilon^{new}|)$ where Δ is the maximum degree of the graph after update. The time complexity of `SubBC` for enumerating Υ^{del}, the set of subsumed bicliques, is $O(2^\rho|\Upsilon^{new}|)$. Note that when ρ is a constant, the time complexity of enumerating the change is $O(\Delta^2|\Upsilon^{new}|)$, which is linear in the number of bicliques that are output, times a factor related to the size of the graph. To the best of our knowledge, these are the first change-sensitive algorithms for maintaining maximal bicliques in a dynamic graph.

Experimental Evaluation: We present an empirical evaluation of `DynamicBC` on real bipartite graphs with million of nodes and compare our algorithm with baseline approaches. Our results show that the performance of `DynamicBC` is many orders of magnitude faster than directly applying a static algorithm (BaselineBC) and many times faster than an improved baseline we devised (BaselineBC*). For example, on the lastfm-song-init graph, DynamicBC took about 93 sec. for computing the change due to addition of 625 batches each of size 100, whereas BaselineBC took about $7,920$ sec. and BaselineBC* about $1,740$ sec.

2.4 Parallel Maximal Biclique Enumeration on Static Bipartite Graphs

In this work we design parallel algorithms for maximal biclique enumeration from a static bipartite graph. We are motivated by the fact that the runtime of the state-of-the-art sequential algorithms for MBE remain almost the same if we increase the number of processor cores. Also, it takes large amount of time to enumerate all the maximal bicliques in a large graph. For example, it takes more than 8 hours to enumerate around 54 million maximal bicliques from a real world bipartite graph called BookCrossing with around 445 thousand vertices and 1.1 million edges. Clearly, the sequential algorithm cannot utilize the power of multiple cores in a multicore computing system. Therefore, we develop work-efficient parallel algorithms based on the state-of-the-art sequential algorithm using parallel techniques. Our contributions are the following:

Efficient Parallel Algorithm: We present a shared-memory parallel algorithm for MBE ParLMBC that takes as input a bipartite graph G and enumerates all maximal bicliques in G. ParLMBC is an efficient parallelization of MineLMBC [92] that parallelizes different steps in the algorithm. Our analysis of ParLMBC using a work-depth model of computation [18] shows that ParLMBC is work-efficient, i.e. its total work across all processors is of the same order as the work of the sequential algorithm MineLMBC. We also show that ParLMBC has a low parallel depth.

Optimized Parallel Algorithm: We design another shared memory parallel algorithm ParMBE that builds on ParLMBC and yields substantially improved practical performance. At a high level, ParMBE constructs a cluster (subgraph) for each vertex v using vertices in the 2-neighborhood of v, and applies ParLMBC within each cluster, in parallel. This brings in a different level of parallelization at the level of subgraphs, in addition to ParLMBC. This approach significantly reduces the parallel enumeration time compared with ParLMBC because the computational cost for enumerating the bicliques is directly related to the size of the candidate set and the adjacency vertex set (vertices adjacent to the vertices in the candidate set) which is much smaller in each subproblem in ParMBE than that of executing ParLMBC directly on the input graph. Thus, the size of the problem instances is reduced in ParMBE while keeping the total number of recursive calls same as that of ParLMBC as each recursive call is followed by the generation of a maximal biclique. If we simply enumerate all maximal bicliques from each subproblem then a maximal biclique will be enumerated more than once. We prevent this by assuming an ordering of the vertices using a rank function so that a highly ranked vertex will contain more maximal bicliques than a low ranked vertex. Clearly, there will be imbalance of the load if we enumerate the maximal bicliques from the subproblems corresponding to the highly ranked vertices. We distribute the load by delegating the task of enumerating maximal bicliques to the vertices adjacent to a highly ranked vertex. For doing this, we create subproblems in a manner that all maximal bicliques enumerated from the subproblem for vertex w will have w as the least ranked vertex. However, computing the exact rank of the vertices beforehand is difficult and therefore we heuristically consider degree of a vertex for computing the rank. This way, in our optimized algorithm ParMBE we ensure (1) non duplicate enumeration of all maximal bicliques and (2) load distribution.

Experimental Evaluation: We empirically evaluated all our algorithms and the experiment shows that on a 16 core machine, ParLMBC yields a **4x-7.5x** parallel speedup over MineLMBC. On the other hand, ParMBE yields a **8x-8500x** parallel speedup when compared with MineLMBC, which was surprising. **We found that the size of each subproblem is reduced in a large extent in ParMBE and as a result the time taken for pruning the search space is reduced considerably with respect to the algorithms MineLMBC and ParLMBC.** We also show that the parallel speedup of ParMBE is almost a linear function of the number of processors.

Improved Sequential Algorithm: With the parallel algorithm ParMBE yielding super-linear speedups compared to MineLMBC, it is possible that there is room for improvement in MineLMBC. We design a better sequential algorithm FMBE, which executes parallel portions of ParMBE sequentially, one after another. On some of the input graphs, FMBE is an order of magnitude faster than MineLMBC, and on each graph, `FMBE` runs at least as fast as `MineLMBC`. On a 16 core machine, `ParMBE` still provides up to **10x** speedup when compared with `FMBE`.

CHAPTER 3. PREVIOUS WORKS

Here we will discuss about the prior and related works on the dense structures in large static and dynamic graphs. We discuss on the fundamental dense structures such as maximal cliques and maximal bicliques in detail as this thesis is about the enumeration and maintenance of these two fundamental dense structures and then we will discuss briefly on the other dense structures such as densest subgraph, k-core etc.

3.1 Maximal Cliques

In this section we will discuss about the prior works on the sequential and parallel algorithms for solving Maximal Clique Enumeration (MCE) problem on both static and dynamic graphs.

3.1.1 Sequential Algorithms

On Static Graph: There is substantial prior work on enumerating maximal cliques in a static graph, starting from the algorithm based on depth-first-search due to Bron and Kerbosch [23]. A significant improvement to [23] is presented in Tomita et al. [143], leading to worst-case optimal time complexity $O(3^{n/3})$ for an n vertex graph [104]. Other work on refinements of [143, 23] include [78], who presents several strategies for pivot selection to enhance the algorithm in [23], and a fixed parameter tractable algorithm parameterized by the graph degeneracy [49, 50].

There is a class of algorithms for enumerating structures (such as maximal cliques) in a static graph whose time complexity is proportional to the size of the output – such algorithms are called "output-sensitive" algorithms. Many output-sensitive structure enumeration algorithms for static graphs, including [148, 32, 97], can be seen as instances of a general technique called "reverse search" [11]. The current best bound on the time complexity of output-sensitive maximal clique enumeration on a dense graph $G = (V, E)$ is due to [97] which runs with $O(M(n))$ time delay (the interval between outputting two maximal cliques), where $M(n)$ is the time complexity for multiplying two $n \times n$ matrices, which is $O(n^{2.376})$. Further work in this direction includes [81] and [71], which consider the enumeration of maximal independent sets in lexicographic order, [30], which considers the external memory model, and [107], which considers uncertain graphs. Extensions to parallel frameworks such as MapReduce, MPI, or Shared Memory are presented in [137, 106, 37]. Note that there is a long line of prior work on finding the maximum clique in a graph. e.g. [141, 142, 144] on finding a maximum clique in a graph. However, these algorithms are not directly useful to our work on enumerating maximal cliques, since the maximum clique is a related, but different concept. While every maximum clique is a maximal clique, there maybe maximal cliques that are not maximum.

On Dynamic Graph: In [135], the authors present algorithms for tracking new and subsumed maximal cliques in a dynamic graph when a single edge is added to the graph. These algorithms are not proved to be change-sensitive, even for a single edge. The algorithm due to Stix [135] for enumerating new maximal cliques needs to consider (and filter out) maximal cliques in the original graph that remain unaffected due to addition of new edge. This can be wasteful, in terms of update time. Hence, such an algorithm cannot be change-sensitive. For example, consider the case of a graph growing from an empty graph on 10 vertices to a clique on 10 vertices. Only one new maximal clique has been formed by this batch, but numerous maximal cliques arise during intermediate steps – if all these are enumerated, then the time complexity of enumeration is inherently large, even though the magnitude of change is small.

Ottosen and Vomlel [112] present an algorithm to enumerate the change in set of maximal cliques, based on running a maximal clique enumeration algorithm on a smaller graph. Their algorithm supports addition of a set of edges all at once. In contrast with our work, there are no provable performance bounds for this algorithm. Another difference is that the algorithm of [112] may not maintain the exact change in the set of maximal cliques, in certain cases, while our algorithms can maintain the change in the set of maximal cliques exactly. Sun et al. [136] present an algorithm for enumerating the change in set of maximal cliques, based on iterating over the set of maximal cliques of the original graph to derive the set of maximal cliques of the updated graph. This need to iterate over currently existing cliques makes the algorithm expensive, especially for cases when the set of maximal cliques does not change significantly due to the update in edge set. Prior algorithms for maximal clique enumeration on a dynamic graph are not proved to be change-sensitive, and do not provide a provable bound on the cost to enumerate the change, or on the magnitude of the change.

3.1.2 Parallel Algorithms

There are multiple prior works on parallel algorithms for MCE [158, 44, 151, 129, 95, 138]. We first discuss shared memory algorithms and then distributed memory algorithms. Zhang et al. [158] presented a shared memory parallel algorithm based on the sequential algorithm due to Kose et al. [81]. This algorithm computes maximal cliques in an iterative manner, and in each iteration, it maintains a set of cliques that are not necessarily maximal and for each such clique, maintains the set of vertices that can be added to form larger cliques. This algorithm does not provide a theoretical guarantee on the runtime and suffers for large memory requirement. Du et al. [44] present a output-sensitive shared-memory parallel algorithm for MCE, but their algorithm suffers from poor load balancing as also pointed out by Schmidt et al. [129]. Lessley et al. [87] present a shared memory parallel algorithm that generates maximal cliques using an iterative method, where in each iteration, cliques of size $(k-1)$ are extended to cliques of size k. The algorithm of [87] is memory-intensive, since it needs to store a number of intermediate non-maximal cliques in each iteration. Note that the number of non-maximal cliques may be far higher than the number of maximal cliques that are finally emitted, and a number of distinct non-maximal cliques may finally lead to a single maximal clique. In the extreme case, a complete graph on n vertices has $(2^n - 1)$ non-maximal cliques, and only a single maximal clique. We present a comparison of our algorithm with [87, 158, 44] in later sections.

Distributed memory parallel algorithms for MCE include works due to Wu et al. [151], designed for the MapReduce framework, Lu et al. [95], which is based on the sequential algorithm due to Tsukiyama et al. [148], and Svendsen et al. [138].

All the works discussed above are the parallel and distributed algorithms for solving MCE on static graph. **To the best of our knowledge, there is no prior work on the parallel algorithms for the maintenance of the maximal cliques in a dynamic graph**.

3.2 Maximal Bicliques

3.2.1 Sequential Algorithms

On Static Graphs: There has been substantial prior work on enumerating maximal bicliques from a static graph. Alexe et al. [6] present an algorithm for MBE from a static graph based on the consensus method, whose time complexity is proportional to the size of the output (number of maximal bicliques in the graph) - termed as an *output-sensitive algorithm*. Liu et al. [92] present an algorithm for MBE based on depth-first-search (DFS). Damaschke [36] present an algorithm for bipartite graphs with a skewed degree distribution. Gély et al. [55] present an algorithm for MBE through a reduction to maximal clique enumeration (MCE). However, in their work, the number of edges in the graph used for enumeration increases significantly compared to the original graph. Makino & Uno [97] present an algorithm for MBE based on matrix multiplication, which provides the current best time complexity for dense graphs. Eppstein [48] presented a linear time algorithm for MBE when the input graph has bounded arboricity. Other works on sequential algorithms for MBE and MCE on a static graph include [40, 41, 143, 107], and on parallel algorithms include [105, 106, 153, 137]. Li et al. [88] show a correspondence between closed itemsets in a transactional database and maximal bicliques in an appropriately defined graph.

On Dynamic Graphs: There have been some prior works related to maintenance of dense structures similar to maximal bicliques in dynamic graphs. Kumar et al. [82] define an (i, j)-core which is a biclique with i vertices in one partition and j vertices in another partition, and present a dynamic algorithm for extracting non-overlapping sets of (i, j)-cores for interesting communities.

To the best of our knowledge, there is no prior work on the maintenance of maximal bicliques in a dynamic graph. In this thesis, we first propose an incremental algorithm for the maintenance of maximal bicliques in a dynamic bipartite graph.

3.2.2 Parallel Algorithms

Nataraj and Selvan [111] first propose a shared memory parallel algorithm (not mentioned clearly in the literature) for enumerating all maximal bicliques from a static graph. However, this work neither explore the load balancing nor the work efficiency of the enumeration process.

3.3 Other Dense Structures

3.3.1 Algorithms for Static Graphs

k-core: Batagelj and Zaversnik [14] first proposed an $O(m)$ (m is the number of edges in the graph) time algorithm for core decomposition which is to identify the core number of every vertex in the graph using a bottom-up approach where vertices are processed from smaller degree to larger degree. This algorithm cannot handle the graphs larger than the size of the main memory. Cheng et al. [29] address this problem and develop an external memory algorithm for core decomposition using a top-down approach where the vertices with large degree are processed first. Following this work, Khaouid et al. [75] propose efficient implementations of the prior algorithms harnessing the power of GraphChi [84] for vertex centric computation and the power of Webgraph [21] for graph compression. Montresor et al. [103] first propose a distributed memory parallel algorithm for k-core decomposition using a message passing model that takes $O(n)$ rounds to complete the computation where n is the number of vertices in the network. Following this work, Dasari et al. [39] propose a technique to parallelize the distributed algorithm in a shared memory multicore processors. Kabir and Madduri [73] propose another shared memory parallel algorithm improving upon the prior shared memory parallel algorithm.

k-truss: Cohen [34] first proposes a sequential algorithm for truss decomposition and following that a MapReduce algorithm [35] based on parallel triangle counting technique. Following these works, Wang and Cheng [150] propose an efficient external memory algorithm for graphs that cannot fit in the main memory. Quick et al. [118] proposed a distributed memory truss decomposition algorithm based on vertex centric computation. Following this work, Shao et al. [131] propose a more efficient distributed parallel algorithm that substantially improve the prior work. In this work, the authors propose a construct called triangle complete subgraph and based on this construct the truss decomposition is carried out through local computation at each computing node. Kabir and Madduri [74] propose a shared memory truss decomposition algorithm by performing a level-synchronous parallelization of the algorithm by Wang and Cheng [150]. Recently, Sariyüce et al. [127] propose an efficient generalized algorithm based on iterative h-index computation [96] that guarantees exact truss decomposition along with core decomposition and nucleus decomposition.

quasi-clique: Abello et al. [5] first proposed a heuristic algorithm for finding density based large quasi clique and developed an external memory algorithm. Pei et al. [116] propose a heuristic algorithm for extracting cross graph degree based quasi cliques which is defined on the same vertex set but different γ values across different graphs (with same vertex set). Zeng et al. [156] consider discovering frequent quasi-cliques that occur in many graphs with the same γ value. In their work, the authors consider degree based quasi clique and develops efficient pruning techniques for reducing the futile search paths as much as possible. Following this work, Zhang et al. [160] parallelize the algorithm using a message passing model. Liu and Wong [93] first propose exact algorithm for enumerating degree based all maximal quasi cliques followed by several pruning techniques that they use in designing the algorithm. Uno [149] considers density based definition of quasi-clique and designs an exact enumeration algorithm for all quasi-cliques. Additionally, the author proved that it is NP-complete to find a quasi-clique containing a give set of vertices. Khosraviani and Sharifi [76] propose a MapReduce algorithm for degree based quasi-clique enumeration combining the diameter based pruning strategies from the work of Pei et al. [116] and degree based pruning strategies from the work of Liu and Wong [93]. Tsourakakis et al. [146] considers density based quasi-clique as in [149] and develop a greedy approximation algorithm by reducing the quasi-clique search problem into an optimization problem.

Works on other dense structures on static graphs include the works on quasi-biclique [100, 154, 132, 89, 133], densest subgraph [58, 26, 9, 77], densest bipartite subgraph [8, 13], triangle densest subgraph [147], k-clique densest subgraph [145, 101], (p, q)-biclique densest subgraph [101] etc.

3.3.2 Algorithms for Dynamic Graphs

k-core: Miorand and Pellegrini [99] first propose the maintenance of the core number of each vertex when the graph evolves over time where the authors apply standard k-core decomposition algorithm on different snapshots of the time evolving graph. Li et al. [90] improves upon the previous work by identifying a small set of vertices to consider on each update of the graph (by addition and deletion of the edges) for the maintenance of the core numbers and proposed an efficient algorithm. Independently, Sariyüce et al. [125, 126] propose another incremental algorithm for maintaining the core values of the vertices through identifying a small subgraph where that contains all the nodes whose core values to be updated when a new edge is added or an old edge is deleted to/from the graph. Zhang et al. [161] improve upon the prior core maintenance algorithm by identifying the deficiency in the prior algorithm and proposing an efficient algorithm based upon the idea of maintaining an ordering of the vertices when the graph is updated for updating the core value of the vertices and experimentally show the significant improvement over the prior work. Jin et al. [69] propose an incremental parallel algorithm for the core maintenance inspired by the single edge addition/deletion case as in [90]. Most recently, Esfandiari et al. [52] propose a distributed streaming algorithm for the maintenance of approximate core decomposition based on a sketching technique.

k-truss: Zhou et al. [162] studied the problem of truss maintenance on dynamic network in when the graph evolves due to addition/deletion of the edges.

Works on other dense structures on dynamic graph includes dynamic and streaming algorithms on finding densest subgraph [12, 47, 98, 16, 110].

CHAPTER 4. MAINTENANCE OF MAXIMAL CLIQUES

4.1 Introduction

Most current methods for identifying dense subgraphs are designed for a static graph. Suppose we used a method designed for a static graph to handle a dynamic graph. If the input graph changes slightly, say, by the addition of a few edges, it is necessary to enumerate all dense subgraphs all over again, even though the set of dense subgraphs may have only changed slightly due to the addition of the new edges. This repeated and redundant work is a source of serious inefficiency, so that methods designed for static graphs are not applicable to a graph that is changing frequently. Different methods are needed, which can handle changes to a graph more efficiently. From a foundational perspective, identifying dense structures in a graph has been a problem of long-standing interest in computer science, but even basic questions remain unanswered on dynamic graphs.

We consider the maintenance of the set of *maximal cliques* in a dynamic graph. Many applications benefit from efficient maintenance of maximal cliques in a dynamic graph, such as described in the work of Chateau et al. [27] on maintaining common intervals among genomes, Duan et al. [46] on incremental k-clique clustering, Hussain et al. [67] on maintaining the maximum range-sum query over a point stream.

Suppose that we started from a graph $G = (V, E)$ and the state of the graph changed to $G' = (V, E \cup H)$ through an addition of a set of new edges H to the set of edges in the graph G. See Figure 4.1 for an example. Let $\mathcal{C}(G)$ denote the set of maximal cliques in G and $\Lambda^{new}(G, G') = \mathcal{C}(G') \setminus \mathcal{C}(G)$ denote the set of maximal cliques that were newly formed when going from G to G', and $\Lambda^{del}(G, G') = \mathcal{C}(G) \setminus \mathcal{C}(G')$ denote the set of cliques that were maximal in G but are no longer maximal in G'. Let $\Lambda(G, G') = \Lambda^{new}(G, G') \cup \Lambda^{del}(G, G')$ denote the symmetric difference of $\mathcal{C}(G)$ and $\mathcal{C}(G')$. We ask the following questions:

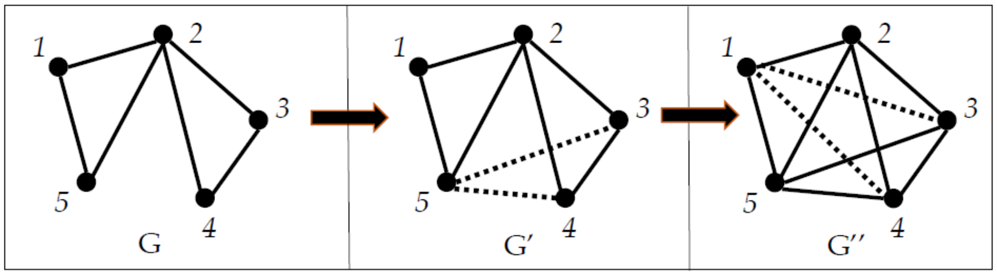


Figure 4.1: Change in maximal cliques due to addition of edges. On the left is the initial graph G with maximal cliques $\{1, 2, 5\}$ and $\{2, 3, 4\}$; On the middle is the graph G' after adding edges $(3, 5)$ and $(4, 5)$ to G resulting in new maximal clique $\{2, 3, 4, 5\}$ and only subsumed maximal clique $\{2, 3, 4\}$; On the right is the graph G'' after adding edges $(1, 3)$ and $(1, 4)$ to G' resulting in new maximal clique $\{1, 2, 3, 4, 5\}$ and subsumed cliques $\{1, 2, 5\}$ and $\{2, 3, 4, 5\}$.

– How large can the size of $\Lambda(G, G')$ be? To systematically study the problem of maintaining maximal cliques in a dynamic graph, we first need to understand the magnitude of change in the set of maximal cliques.

– What are efficient methods to compute $\Lambda(G, G')$? Can we compute $\Lambda(G, G')$ quickly in cases when the size of $\Lambda(G, G')$ is small, and take longer when it is large? Do these methods scale to large graphs?

Roadmap: We present preliminaries in Section 4.2, followed by bounds on magnitude of change in Section 4.3, algorithms for enumerating the change in Section 4.4, discussions in Section 4.5, and experimental results in Section 4.6.

4.2 Preliminaries

We consider a simple undirected graph without self loops or multiple edges. For graph G, let $V(G)$ denote the set of vertices in G and $E(G)$ denote the set of edges in G. Let n denote the size of $V(G)$, and m denote the size of $E(G)$. For vertex $u \in V(G)$, let $\Gamma_G(u)$ denote the set of vertices adjacent to u in G. When the graph G is clear from the context, we use $\Gamma(u)$ to mean $\Gamma_G(u)$. For edge $e = (u, v) \in E(G)$, let $G - e$ denote the graph obtained by deleting e from $E(G)$, but retaining vertices u and v in $V(G)$. Similarly, let $G + e$ denote the graph obtained by adding edge e to $E(G)$.

For edge set H, let $G + H$ $(G - H)$ denote the graph obtained by adding (subtracting) all edges in H to (from) $E(G)$. Let $\Delta(G)$ denote the maximum degree of a vertex in G. When the context is clear, we use Δ to mean $\Delta(G)$. For vertex $v \in V(G)$, let $G - v$ denote the induced subgraph of G on the vertex set $V(G) - \{v\}$, i.e. the graph obtained from G by deleting v and all its incident edges. Let $\mathcal{C}_v(G)$ denote the set of maximal cliques in G containing v.

Algorithm `TTT`: The algorithm due to Tomita, Tanaka, and Takahashi [143], which we call `TTT`, is a recursive backtracking-based algorithm for enumerating all maximal cliques in a static undirected graph, with a worst-case time complexity of $O(3^{n/3})$ where n is the number of vertices in the graph. In practice, this is one of the most efficient sequential algorithms for MCE.

In any recursive call, `TTT` maintains three disjoint sets of vertices K, `cand`, and `fini` where K is a candidate clique to be extended, `cand` is the set of vertices that can be used to extend K, and `fini` is the set of vertices that are adjacent to K, but need not be used to extend K (these are being explored along other search paths). Each recursive call iterates over vertices from `cand` and in each iteration, a vertex $q \in$ `cand` is added to K and a new recursive call is made with parameters $K \cup \{q\}$, $\texttt{cand}_q$, and $\texttt{fini}_q$ for generating all maximal cliques of G that extend $K \cup \{q\}$ but do not contain any vertices from $\texttt{fini}_q$. The sets $\texttt{cand}_q$ and $\texttt{fini}_q$ can only contain vertices that are adjacent to all vertices in $K \cup \{q\}$. The clique K is a maximal clique when both `cand` and `fini` are empty.

The ingredient that makes `TTT` different from the algorithm due to Bron and Kerbosch [23] is the use of a "pivot" where a vertex $u \in \texttt{cand} \cup \texttt{fini}$ is selected that maximizes $|\texttt{cand} \cap \Gamma(u)|$. Once the pivot u is computed, it is sufficient to iterate over all the vertices of $\texttt{cand} \setminus \Gamma(u)$, instead of iterating over all vertices of `cand`. The pseudo code of `TTT` is presented in Algorithm 1. For the initial call, K and `fini` are initialized to an empty set, `cand` is the set of all vertices of G.

Definition 1 (Change-Sensitive Algorithm)**.** *An algorithm for a property P on a dynamic graph is said to be change-sensitive if the time complexity of enumerating the change in P due to a change*

Algorithm 1: $\mathtt{TTT}(\mathcal{G}, K, \mathtt{cand}, \mathtt{fini})$

```
    Input: $\mathcal{G}$ - The input graph
           $K$ - a clique to extend,
    cand - Set of vertices that can be used extend $K$,
    fini - Set of vertices that have been used to extend $K$
    Output: Set of all maximal cliques of $G$ containing $K$ and vertices from cand but not
            containing any vertex from fini
 1  if $(\mathtt{cand} = \emptyset)$ & $(\mathtt{fini} = \emptyset)$ then
 2      Output $K$ and return
 3  pivot $\leftarrow (u \in \mathtt{cand} \cup \mathtt{fini})$ such that $u$ maximizes the size of $\mathtt{cand} \cap \Gamma_{\mathcal{G}}(u)$
 4  ext $\leftarrow \mathtt{cand} - \Gamma_{\mathcal{G}}(\mathtt{pivot})$
 5  for $q \in$ ext do
 6      $K_q \leftarrow K \cup \{q\}$
 7      $\mathtt{cand}_q \leftarrow \mathtt{cand} \cap \Gamma_{\mathcal{G}}(q)$
 8      $\mathtt{fini}_q \leftarrow \mathtt{fini} \cap \Gamma_{\mathcal{G}}(q)$
 9      $\mathtt{cand} \leftarrow \mathtt{cand} - \{q\}$
10      $\mathtt{fini} \leftarrow \mathtt{fini} \cup \{q\}$
11      $\mathtt{TTT}(\mathcal{G}, K_q, \mathtt{cand}_q, \mathtt{fini}_q)$
```

in the set of edges of the graph is linear in the magnitude of change (in P), and polynomial in the the size of the input graph, and the number of edges added to or deleted from the graph.

By the phrase "magnitude of change", we mean the number of structures that have changed with respect to the property P. Note that the notion of a change-sensitive algorithm for a dynamic graph is similar to the notion of an "output-sensitive" algorithm for a static graph, whose time complexity depends on the size of the output and the size of the graph.

An algorithm for a dynamic graph is called *incremental* if it can efficiently handle insertion of edges, *decremental* if it can handle deletion of edges, and *fully dynamic* if it can handle both insertions and deletions. For example, a parallel algorithm due to Simsiri et al. [134] is an incremental algorithm for graph connectivity, an algorithm due to Thorup [140] is a decremental algorithm, and one due to Wulff-Nilsen [152] is a fully dynamic algorithm. We present change-sensitive incremental and decremental algorithms for maximal clique maintenance. Our fully dynamic algorithm for maximal clique maintenance is however not change-sensitive.

Results for Static Graphs: We present some known results about maximal cliques on static graphs. Nearly 50 years ago, Moon and Moser [104] considered the question: "what is the maximum number of maximal cliques that can be present in an undirected graph on n vertices", and gave the following answer. Let $f(n)$ denote the maximum possible number of maximal cliques in a graph on n vertices. A graph on n vertices that achieves $f(n)$ maximal cliques is called a "Moon-Moser" graph.

Theorem 1 (Theorem 1, Moon and Moser, [104])**.**

$$\begin{aligned} f(n) &= 3^{\frac{n}{3}} && \textit{if } n \mod 3 = 0 \\ &= 4 \cdot 3^{\frac{n-4}{3}} && \textit{if } n \mod 3 = 1 \\ &= 2 \cdot 3^{\frac{n-2}{3}} && \textit{if } n \mod 3 = 2 \end{aligned}$$

We use as a subroutine an output-sensitive algorithm for enumerating all maximal cliques within a (static) graph, using time proportional to the number of maximal cliques. There are multiple such algorithms, for example, due to Tsukiyama et al. [148], and due to Makino and Uno [97]. We use the following result (Theorem 2) due to Chiba and Nishizeki since it provides one of the best possible time complexity bounds for general graphs. Better results are possible for dense graphs [97] and our algorithm can use other methods as a subroutine also. Let the arboricity of a graph be defined as the minimum number of forests into which the edges of the graph can be partitioned. The arboricity of a graph is no more than the maximum vertex degree, but could be significantly smaller [32].

Theorem 2 (Chiba and Nishizeki, [32])**.** *There is an algorithm* $\texttt{MCE}(G)$ *that enumerates all maximal cliques in graph G in time $O(\alpha m \mu)$ where μ is the number of maximal cliques in G, and α and m are respectively the arboricity of G and the number of edges in G. The space complexity of the algorithm is $O(n + m)$, where n is the number of vertices in G.*

Note that the space complexity excludes the size of the output, which may be much larger.

4.3 Magnitude of Change

From prior work [104], the maximum number of maximal cliques in an n-vertex graph, denoted by $f(n)$ is known (see Theorem 1). The result of [104] is relevant for static graphs. In the case of a dynamic graph, a different question is more relevant: *what is the maximum change in the set of maximal cliques, that can result from the addition of edges to the graph?* This will give us a bound on the worst case complexity of enumerating the change in the set of maximal cliques.

4.3.1 Maximum Possible Change in Maximal Cliques

We consider the maximum change in the set of maximal cliques upon the addition of edges to the graph. For an integer n, let $\lambda(n)$ be the maximum size of $\Lambda(G, G+H)$ taken over all possible n vertex graphs G and edge sets H. We present the following result with nearly tight bounds on the value of $\lambda(n)$. Interestingly, our results show that it is possible to change the set of maximal cliques by as much as $\approx 2 \cdot 3^{n/3}$ by the addition of only a few edges to the graph.

Theorem 3.

$$\begin{array}{rll} \frac{16}{9} f(n) \leq & \lambda(n) & < 2f(n) \qquad \textit{if} \ (n \mod 3) = 0 \\ & \lambda(n) & = 2f(n) \qquad \textit{if} \ (n \mod 3) = 1 \\ \frac{11}{6} f(n) \leq & \lambda(n) & < 2f(n) \qquad \textit{if} \ (n \mod 3) = 2 \end{array}$$

Proof. We first note that $\lambda(n) \leq 2f(n)$ for any integer n. To see this, note that for any graph G on n vertices and edge set H, it must be true from Theorem 1 that $|\mathcal{C}(G)| \leq f(n)$ and $|\mathcal{C}(G+H)| \leq f(n)$. Since $|\Lambda^{new}(G, G+H)| \leq |\mathcal{C}(G+H)|$ and $|\Lambda^{del}(G, G+H)| \leq |\mathcal{C}(G)|$, we have $|\Lambda(G, G+H)| = |\Lambda^{new}(G, G+H)| + |\Lambda^{del}(G, G+H)| \leq |\mathcal{C}(G)| + |\mathcal{C}(G+H)| \leq 2f(n)$.

The result of Moon and Moser [104] states that for $n \geq 2$, there is only one graph H_n on n vertices (subject to isomorphism) that has $f(n)$ maximal cliques. We show below that there is an error in this result for the case $(n \mod 3) = 1$. Note that the result is still true in the cases $(n$

mod 3) equals 0 or 2. Thus for the cases where ($n \mod 3$) is 0 or 2, adding or deleting edges from H_n leads to a graph with fewer than $f(n)$ maximal cliques, so that we can never achieve a change of $2f(n)$ maximal cliques. Thus we have that for ($n \mod 3$) equal to 0 or 2, $\lambda(n)$ is strictly less than $2f(n)$. The case of ($n \mod 3) = 1$ is discussed separately (see Observation 1 below).

Next, we show that there exists a graph G on n vertices and an edge set H such that the size of $\Lambda(G, G + H)$ is large. See Figure 4.2 for an example. Graph G is constructed on n vertices as follows. Let $\varepsilon > 3$ be an integer. Choose ε vertices in $V(G)$ into set V_1. Let $V_2 = V \setminus V_1$. Edges of G are constructed as follows.

- Each vertex in V_1 is connected to each vertex in V_2.
- Edges are added among vertices of V_2 to make the induced subgraph on V_2 a Moon-Moser graph on $(n - \varepsilon)$ vertices. Let G_2 denote this induced subgraph on V_2, which has $f(n - \varepsilon)$ maximal cliques.
- There are no edges among vertices of V_1 in G.

It is clear that for each maximal clique c in G_2 and vertex $v \in V_1$, there is a maximal clique in G by adding v to c. Thus the number of maximal cliques in G is $|V_1| \cdot |\mathcal{C}(G_2)|$. Hence we have

$$|\mathcal{C}(G)| = \varepsilon \cdot f(n - \varepsilon) \tag{4.1}$$

We add edge set H to the graph as follows. H consists of edges connecting vertices in V_1, to form a Moon-Moser graph on ε vertices. Let $G' = G + H$. We note that $\mathcal{C}(G)$ and $\mathcal{C}(G')$ are disjoint sets. To see this, note that each maximal clique in G contains exactly one vertex from V_1, since no two vertices in V_1 are connected to each other in G. On the other hand, each maximal clique in G' contains more than one vertex from V_1, since each vertex $v \in V_1$ is connected to at least one other vertex in V_1 in G'. Hence, $\Lambda(G, G') = \mathcal{C}(G) \cup \mathcal{C}(G')$, and

$$|\Lambda(G, G')| = |\mathcal{C}(G)| + |\mathcal{C}(G')| \tag{4.2}$$

To compute $|\mathcal{C}(G')|$, note that since each vertex in V_1 is connected to each vertex in V_2, for each maximal clique in $G'(V_1)$ and each maximal clique in $G'(V_2)$, we have a unique maximal clique in

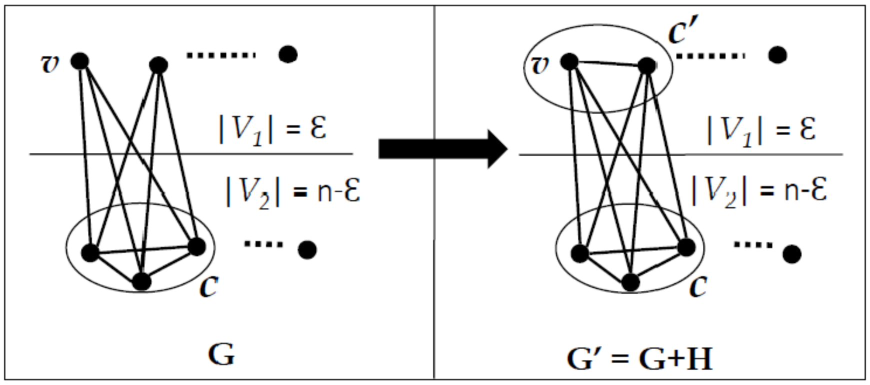


Figure 4.2: A large change in set of maximal cliques when a few edges are added. The vertex set is partitioned into V_1 and V_2. On the left is G, the original graph on n vertices where each vertex in V_1 is connected to each vertex in V_2, and V_1 is an independent set. In G, the induced subgraph G_2 on vertex set V_2 forms a `Moon-Moser` graph. On the right is G', the graph formed after adding edge set H to G such that the induced subgraph on vertex set V_1 becomes a `Moon-Moser` graph. Let c be a clique in G_2, and c' a new clique in G' formed among vertices in V_1. Note that $c \cup \{v\}$ was a maximal clique in G, and is now subsumed by a new maximal clique $c \cup c'$.

G'. There are $f(\varepsilon)$ maximal cliques in $G'(V_1)$ and $f(n-\varepsilon)$ maximal cliques in $G'(V_2)$, and hence we have

$$|\mathcal{C}(G')| = f(\varepsilon) \cdot f(n-\varepsilon) \tag{4.3}$$

Putting together Equations 4.1, 4.2, and 4.3 we get

$$|\Lambda(G, G')| = (\varepsilon + f(\varepsilon)) \cdot f(n-\varepsilon) \tag{4.4}$$

Let $F(\varepsilon) = (\varepsilon + f(\varepsilon))f(n-\varepsilon)$. We compute the value of $\varepsilon(> 3)$ at which $F(\varepsilon)$ is maximized. To do this, we consider three different cases depending on the value of ($n \mod 3$), and omit the calculations. If $n \mod 3 = 0$, $F(\varepsilon)$ is maximized at $\varepsilon = 4$ and the maximum value $F(4) = \frac{16}{9}f(n)$. If $n \mod 3 = 1$, $F(\varepsilon)$ is maximized at $\varepsilon = 4$ and $F(4) = 2f(n)$. And finally if $n \mod 3 = 2$, $F(\varepsilon)$ is maximized at $\varepsilon = 5$ and $F(5) = \frac{11}{6}f(n)$. This completes the proof. □

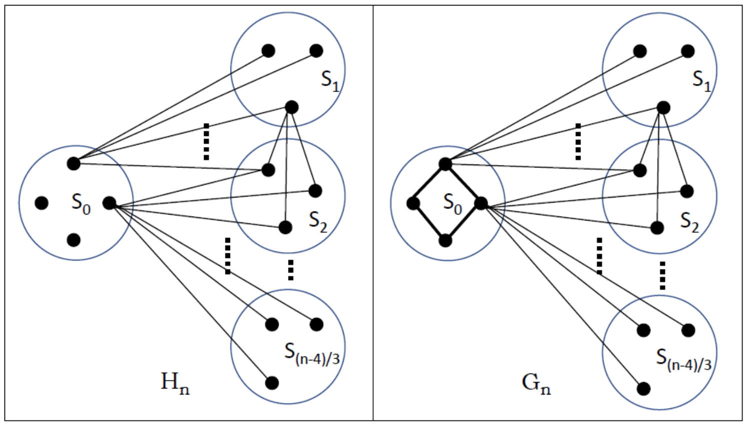


Figure 4.3: On the left is H_n where each vertex v in S_i is connected to each vertex u in S_j, $i \neq j$. On the right is G_n which is formed from H_n by adding four edges to S_0. For the case $(n \mod 3) = 1$, H_n and G_n are non-isomorphic graphs on n vertices, with $f(n)$ maximal cliques each, showing a counterexample to Theorem 2 of Moon and Moser [104].

4.3.2 An Error in a Result of Moon and Moser (1965)

Moon and Moser [104], in Theorem 2 of their paper, claim "For any $n \geq 2$, if a graph G has n nodes and $f(n)$ cliques, then G must be equal to H_n", where H_n is a specific graph, described below. We found that this theorem is incorrect for the case when $(n \mod 3) = 1$.

The error is as follows (see Figure 4.3). For $(n \mod 3) = 1$, the graph H_n is constructed on vertex set $V_n = \{1, 2, \ldots, n\}$ by taking vertices $\{1, 2, 3, 4\}$ into a set S_0 and dividing the remaining vertices into groups of three, as sets $S_1, S_2, \ldots, S_{\frac{n-4}{3}}$. In graph H_n, edges are added between any two vertices u, v such that $u \in S_i$, $v \in S_j$ and $i \neq j$. This graph H_n has $4 \cdot 3^{\frac{n-4}{3}}$ maximal cliques, since we can make a maximal clique by choosing a vertex from S_0 (4 ways), and one vertex from each $S_i, i > 0$ (3 ways for each such $S_i, i = 1 \ldots (n-4)/3$).

Contradicting Theorem 2 in [104], we show there is another graph G_n that is different from H_n, but still has the same number of maximal cliques. G_n is the same as H_n, except that the vertices within S_0 are connected by a cycle of length 4. In this case, we can still construct $4 \cdot 3^{\frac{n-4}{3}}$ maximal cliques, since we can make a maximal clique by choosing two connected vertices in S_0 (4 ways to do this), and one vertex from each $S_i, i > 0$ (3 ways for each such $S_i, i = 1 \ldots (n-4)/3$).

Observation 1. *For the case* $(n \mod 3) = 1$*, there are two distinct non-isomorphic graphs* H_n *and* G_n *described above, that have* $4 \cdot 3^{\frac{n-4}{3}}$ *maximal cliques, which is the maximum possible. This is a correction to Theorem 2 of Moon and Moser [104], which states that there is only one such graph,* H_n*.*

This observation enables us to have $\lambda(n) = 2f(n)$ for the case $(n \mod 3) = 1$. By starting with graph H_n and by adding edges to make it G_n, we remove $f(n)$ maximal cliques and introduce $f(n)$ maximal cliques, leading to a total change of $2f(n)$.

4.3.3 Bound on the size of change parameterized by max degree Δ of G

Lemma 1. *For any* $v \in V(G)$*,* $|\mathcal{C}_v(G)| \leq f(\Delta)$ *where* Δ *is the maximum degree of* G*.*

Proof. Each maximal clique c of G that contains v must be within the neighborhood of v and the size of this neighborhood can be at most Δ. Thus, there can be at most $f(\Delta)$ maximal cliques each of which contains v. □

Lemma 2. *For any* $e \in E(G)$*, the number of maximal cliques in* G *containing* e *is at most* $f(\Delta - 1)$*.*

Proof. Suppose each of the vertices u and v of an edge e has degree Δ and that u and v has the same neighborhood. Thus the size of the neighborhood except u and v is $\Delta - 1$. There can be at most $f(\Delta - 1)$ maximal cliques within these vertices. Now when we add u and v to each of such maximal clique, a new maximal clique will be formed that contains the edge e. Thus there can be at most $f(\Delta - 1)$ such maximal cliques. □

Lemma 3. *For a graph* G *on* n *vertices and edge* $e \notin E(G)$*, the size of* $\Lambda(G, G + e)$ *can be no larger than* $3f(\Delta)$*.*

Proof. Proof by contradiction. Suppose there exists a graph G and edge $e \notin E(G)$ such that $|\Lambda(G, G + e)| > 3f(\Delta)$. Then either $|\mathcal{C}(G + e) \setminus \mathcal{C}(G)| > f(\Delta)$ or $|\mathcal{C}(G) \setminus \mathcal{C}(G + e)| > 2f(\Delta)$.

Case 1: $|\mathcal{C}(G + e) \setminus \mathcal{C}(G)| > f(\Delta)$: This means that the total number of new maximal cliques is more than $f(\Delta)$. Assume that $e = (u, v)$, $U = \Gamma_G(u) = \Gamma_G(v)$, and $|U| = \Delta$. Then there can be at most $f(\Delta)$ maximal cliques in the subgraph of G induced by U. For each such maximal clique, there will be a new maximal clique by adding e. Thus the number of new maximal cliques is at most $f(\Delta)$. A contradiction.

Case 2: $|\mathcal{C}(G)\setminus\mathcal{C}(G+e)| > 2f(\Delta)$: First note that each subsumed clique contains either u or v but not the both. Next, the number of maximal cliques in G is at most $f(\Delta)$ containing u and $f(\Delta)$ containing v. Thus the total number of subsumed cliques is at most $2f(\Delta)$. A contradiction. □

Lemma 4. *For any integer $n > 2$ there exists a graph G on n vertices with maximum degree Δ and an edge $e \notin E(G)$ such that $|\Lambda(G, G+e)| = 3f(\Delta)$.*

Proof. We use proof by construction. Let G be a graph on n vertices and X be a subset of $V(G)$ of size Δ. Also assume that X forms a Moon-Moser graph.Note that degree of every vertex of X is at most $\Delta - 1$ within X. Fix two vertices u and v and connect from each of u and v to all the vertices of X. Connect among rest of the vertices in such a way that the maximum degree in G does not exceed Δ. This completes the construction of G.

First note that total number of maximal cliques containing u in G is $f(\Delta)$ and total number of maximal cliques containing v in G is $f(\Delta)$. This is because, each maximal clique in X will be maximal in G when adding u and when adding v. Total number of such maximal cliques is thus $2f(\Delta)$ that contains either u or v but not both.

Next note that total number of maximal cliques in $G + e$ containing $e = (u, v)$ is $f(\Delta)$. This is because, each maximal clique in X becomes a new maximal clique in $G + e$ when e is added and these are all inclusive new maximal cliques because, neither u nor v is connected to any other vertex in G except X. Also note that each maximal clique in G containing either u or v will becomes part of a new maximal clique and thus will be subsumed.

Thus, $|\Lambda(G, G+e)| = |\mathcal{C}_u(G)| + |\mathcal{C}_v(G)| + f(\Delta) = 3f(\Delta)$. □

4.3.4 Bound on the size of change parameterized by degeneracy d of G

Graph degeneracy is a measure of graph sparsity. Degeneracy is the smallest value d such that every subgraph of G has at least a vertex of degree at most d. In other words, it is the maximum over minimum degrees of all the subgraphs of G. When the degeneracy d is known, the maximum number of maximal cliques of d-degenerate graph is different from the moon-moser bound as shown in [49]. First we prove a lower bound on $|\Lambda(G, G+e)|$ as follows:

Lemma 5. *For any integer $n > 2$, there exists a graph G with n vertices and degeneracy d such that $|\Lambda(G, G+e)|$ is at least $3(n-d)f(d-2)$ where $e \notin E(G)$.*

Proof. We prove by construction. Suppose there is a graph G with n vertices and $V = \{1, 2, 3, ..., n-1, n\}$. Now assume that first $n-d$ vertices (denote as set A) form an independent set, followed by next $(d-2)$ vertices (denote as set B) that forms a moon-moser graph, followed by vertices $n-1$ and n that are not connected but each is connected to the rest $n-1$ vertices. Each vertex in set A is connected to each vertex in set B. Clearly, the degeneracy of G is d. Now there are $(n-d)f(d-2)$ maximal cliques in $G' = G - \{n-1, n\}$ and assume $\mathcal{C}(G')$ denotes the set of maximal cliques in G'. For each clique c' in $\mathcal{C}(G')$, we get two maximal cliques in G, once by adding $n-1$ to c' and then by adding n to c'. Thus, $|\mathcal{C}(G)| = 2(n-d)f(d-2)$. When we add an edge e between $n-1$ and n each clique in $\mathcal{C}(G')$ is extended by both $n-1$ and n and becomes a maximal clique in $G+e$. Note that $\mathcal{C}(G)$ and $\mathcal{C}(G+e)$ are disjoint set. To see this, observe that each maximal clique in G contains either $n-1$ or n since $n-1$ and n are not connected in G. On the other hand, each maximal clique in $G+e$ contains both $n-1$ and n and $|\mathcal{C}(G+e)| = |\mathcal{C}(G')| = (n-d)f(d-2)$. Hence, $\Lambda(G, G+e) = \mathcal{C}(G) \cup \mathcal{C}(G+e)$ and $|\Lambda(G, G+e)| = |\mathcal{C}(G)| + |\mathcal{C}(G+e)| = 3(n-d)f(d-2)$. □

Next, we will prove an upper bound on $|\Lambda(G, G+e)|$ in Lemma 10. For proving an upper bound, we use the following results and some supporting lemmas in obtaining bound on the size of change in our case:

Lemma 6 (Theorem 3 of [49])**.** *Let d be a multiple of 3 and $n \geq d+3$. Then the largest possible number of maximal cliques in an n-vertex graph with degeneracy d is $(n-d)3^{\frac{d}{3}}$.*

This result assumes that d is multiple of 3. However, we can generalize the bound that works for any value of the degeneracy d as follows:

Lemma 7. *Let d be the degeneracy of an n vertex graph G. Then maximum number of maximal cliques in G is $(n-d)f(d)$.*

Proof. **Upper Bound:** We will show that number of maximal cliques in a graph G with n vertices and degeneracy d is no more than $(n-d)f(d)$. For this, assume that there is a degeneracy ordering of the vertices of G for which there are at most d neighbors of each vertex that come later in the ordering. Consider first $(n-d)$ vertices in this ordering and denote this set of vertices as S. Now, each vertex in S is connected to at most d vertices in set $V \setminus S$. Also, there are precisely d vertices in the set $S' = V \setminus S$. Now, the maximum possible number of maximal cliques formed using the vertices in S' is $f(d)$. As each vertex in S is connected to at most d vertices in S', each such vertex can participate in at most $f(d)$ maximal cliques. Thus, maximum number of maximal cliques in G is $(n-d)f(d)$.

Lower Bound: Next we will show that there exists a graph G with n vertices and degeneracy d such that there are $(n-d)f(d)$ maximal clique. We will show this using a construction. Assume that vertices in S' forms a moon-moser graph. Next, the vertex set S is an independent set and each vertex in S is connected to all vertices in S'. This completes the construction of G. Note that, number of maximal cliques in the induced subgraph with vertex subset S' is $f(d)$. Each maximal clique in that subgraph together with each vertex in S forms a distinct maximal clique in G. Thus, total number of maximal clique in G is $|S|f(d)$ which is $(n-d)f(d)$. This completes the proof. □

Lemma 8. *For any $v \in V(G)$, $|\mathcal{C}_v(G)| \leq (n-d-1)f(d)$ where d is the degeneracy of the graph G*

Proof. Suppose there is a vertex v such that v is connected to $n-1$ vertices. Also, assume that the degeneracy of G is d. Now, the degeneracy of $G - v$ can be at most d. Now, the maximum number of maximal cliques in $G - v$ is $(n-d-1)f(d)$. Each of these maximal clique when includes v becomes a maximal clique in G. Thus maximum number of maximal cliques containing a vertex is at most $(n-d-1)f(d)$. □

Lemma 9. *For any $e \in E(G)$, the number of maximal cliques containing e is at most $(n-d-2)f(d)$ where d is the degeneracy of G.*

Proof. Suppose that $e = (u, v)$ and each of u and v is connected to rest of the vertices of G. Also the degeneracy of $G-u-v$ can be at most d. Now, the maximum number of maximal cliques in $G-u-v$ is $(n-d-2)f(d)$. Each of these maximal cliques, when includes u and v, becomes a maximal cliques in G. Thus, maximum number of maximal cliques containing e is $(n-d-2)f(d)$. □

Lemma 10. *For a graph G on n vertices and edge $e \notin E(G)$, the size of $\Lambda(G, G+e)$ can be no larger than $3(n-d-2)f(d)$ where d is the degeneracy of the graph G.*

Proof. Proof by contradiction. Suppose there exists a graph G and an edge $e \notin E(G)$ such that $|\Lambda(G, G+e)| > 3(n-d-2)f(d)$. Then, either $|\mathcal{C}(G+e)\setminus\mathcal{C}(G)| > (n-d-2)f(d)$ or $|\mathcal{C}(G)\setminus\mathcal{C}(G+e)| > 2(n-d-2)f(d)$.

Case 1: $|\mathcal{C}(G+e)\setminus\mathcal{C}(G)| > (n-d-2)f(d)$: Assume that $e = (u, v)$ and each of u and v are connected to the rest $n-2$ vertices of G. Note that the degeneracy of $G-u-v$ is at most d. So, maximum number of maximal cliques in $G-u-v$ is $(n-d-2)f(d)$. When we add e, each maximal clique in $G-u-v$ together with u and v becomes a maximal clique in G and each of these maximal clique contains e. Thus $|\mathcal{C}(G+e)\setminus\mathcal{C}(G)| \leq (n-d-2)f(d)$. A contradiction.

Case2: $|\mathcal{C}(G)\setminus\mathcal{C}(G+e)| > 2(n-d-2)f(d)$: Note that each maximal clique $c \in \mathcal{C}(G)\setminus\mathcal{C}(G+e)$ must contain either u or v, but not both. Suppose that c contains u but not v. Then c must be maximum clique in $G-v$. Using Lemma 8 we see that the number of maximal cliques in $G-v$ that contains a specific vertex u can be no more than $(n-d-2)f(d)$; hence the number of possible maximal cliques that contain u is no more than $(n-d-2)f(d)$. In a similar way, the number of possible maximal cliques that contain v is at most $(n-d-2)f(d)$. Therefore, the total number of maximal cliques in $\mathcal{C}(G)\setminus\mathcal{C}(G+e)$ is at most $2(n-d-2)f(d)$. This is a contradiction. □

4.4 Enumeration of Change in the Set of Maximal Cliques

In this section we present algorithms for enumerating the change in the set of maximal cliques. In Section 4.4.1, we first present an algorithm with provable theoretical properties for enumerating new maximal cliques that arise due to the addition of a batch of edges, followed by an algorithm with good practical performance in Section 4.4.2. In Section 4.4.3, we present an algorithm for enumerating subsumed cliques due to the addition of new edges. We then consider the decremental case where edges are deleted from the graph in Section 4.4.4. For graph G and edge set H, when the context is clear, we use Λ^{new} to mean $\Lambda^{new}(G, G + H)$ and similarly Λ^{del} to mean $\Lambda^{del}(G, G + H)$.

4.4.1 Enumeration of New Maximal Cliques

When a set of edges H is added to the graph G, let G' denote the graph $G + H$. One approach to enumerating new cliques in G' is to simply enumerate all cliques in G' using an output-sensitive algorithm such as [32], suppress cliques that were also present in G, and output the rest. The above approach is not change-sensitive. To see why, consider a case when the initial graph G is the union of a Moon-Moser graph on $(n - 3)$ vertices, along with three isolated vertices a, b, and c. Suppose three edges are added in H so that vertices $\{a, b, c\}$ form a triangle. In going from graph G to $G + H$, the set of maximal cliques has changed as follows: A new clique $\{a, b, c\}$ has been formed and three existing (trivial) cliques $\{a\}$, $\{b\}$, and $\{c\}$ have been subsumed. Following the above approach, all cliques in G and in $G + H$ are enumerated, which has a cost of $\Omega(3^{\frac{n}{3}})$, since there are $\Theta(3^{\frac{n}{3}})$ cliques in G and in $G + H$. The size of change is small (a constant), while the cost of enumeration is very large (exponential in the number of vertices). Approaches that involve enumerating maximal cliques in a certain graph, followed by suppressing cliques that do not belong to Λ^{new}, run the risk of sometimes having to suppress most of the cliques that were enumerated, and such approaches will not be change-sensitive.

In the following, we present a simple approach that leads to a change-sensitive algorithm for new cliques. At its core, our algorithm constructs a set of subgraphs of $G + H$ such that each maximal clique in any of these subgraphs is a new maximal clique, i.e. belongs to $\Lambda^{new}(G, G')$. Further, each element of $\Lambda^{new}(G, G')$ is a maximal clique in one of these subgraphs. This construction allows the algorithm to directly output cliques from $\Lambda^{new}(G, G')$, without enumerating cliques that do not belong to $\Lambda^{new}(G, G')$. This can form the basis of a change-sensitive algorithm. There

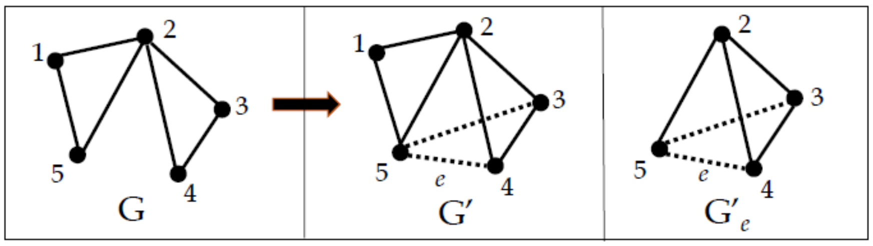


Figure 4.4: Illustration of Lemma 12 that, the set of new maximal cliques in G' containing $e = (4,5)$, i.e. the single clique $\{2,3,4,5\}$, is exactly the set of all maximal cliques in G'_e.

is an additional duplicate elimination step in our algorithm, whose goal is to only to suppress enumerating the same clique multiple times.

For edge $e \in H$, let $C'(e)$ denote the set of maximal cliques in G' that contain edge e. We first present the following observation that Λ^{new}, the set of all new maximal cliques, is precisely the set of all maximal cliques in G' that contain at least one edge from H.

Lemma 11.

$$\Lambda^{new}(G, G') = \cup_{e \in H} C'(e)$$

Proof. We first note that each clique in Λ^{new} must contain at least one edge from H. We use proof by contradiction. Consider a clique $c \in \Lambda^{new}$. If c does not contain an edge from H, then c is also a clique in G, and hence cannot belong to Λ^{new}. Hence, $c \in C'(e)$ for some edge $e \in H$, and $c \in \cup_{e \in H} C'(e)$. This shows that $\Lambda^{new} \subseteq \cup_{e \in H} C'(e)$. Next consider a clique $c \in \cup_{e \in H} C'(e)$. It must be the case that $c \in C'(h)$ for some h in H. Thus c is a maximal clique in G'. Since c contains edge $h \in H$, c cannot be a clique in G. Thus $c \in \Lambda^{new}$. This shows that $\cup_{e \in H} C'(e) \subseteq \Lambda^{new}$. □

We now consider efficient ways of enumerating cliques from $\cup_{e \in H} C'(e)$. For an edge $e \in H$, the enumeration of cliques in $C'(e)$ is reduced to the enumeration of *all* maximal cliques in a specific subgraph of G', as follows. Let u and v denote the endpoints of e, and let G'_e denote the induced subgraph of G' on the vertex set $\{u, v\} \cup \{\Gamma_{G'}(u) \cap \Gamma_{G'}(v)\}$ i.e. the set of vertices adjacent to both u and v in G', in addition to u and v. For example, see Figure 4.4 for construction of G'_e.

Lemma 12.

$$For\ each\ e \in H,\ C'(e) = \mathcal{C}(G'_e)$$

Proof. First we show that $C'(e) \subseteq \mathcal{C}(G'_e)$. Consider a clique c in $C'(e)$, i.e. a maximal clique in $G' = G + H$ containing edge e. Hence c must contain both u and v. Every vertex in c (other than u and v) must be connected to both u and to v in G', and hence must be in $\Gamma_{G'}(u) \cap \Gamma_{G'}(v)$. Hence c must be a clique in G'_e. Since c is a maximal clique in G', and G'_e is a subgraph of G', c must also be a maximal clique in G'_e. Hence we have that $c \in \mathcal{C}(G'_e)$, leading to $C'(e) \subseteq \mathcal{C}(G'_e)$.

Next, we show that $\mathcal{C}(G'_e) \subseteq C'(e)$. Consider any maximal clique d in G'_e. We note the following in G'_e: (1) every vertex in G'_e (other than u and v) is connected to u as well as v (2) u and v are connected to each other. Due to these conditions, d must contain both u and v, and hence also edge $e = (u, v)$. Clearly, d is a clique in G' that contains edge e. We now show that d is a maximal clique in G'. Suppose not, and we could add vertex v' to d and it remained a clique in G'. Then, v' must be in $\Gamma_{G'}(u) \cap \Gamma_{G'}(v)$, and hence v' must be in G'_e, so that d is not a maximal clique in G'_e, which is a contradiction. Hence, it must be that d is a maximal clique in G' that contains edge e, and $d \in C'(e)$. □

Following Lemma 12, in Figure 4.4, $\{2, 3, 4, 5\}$ is a new maximal clique in G' that contains $e = (4, 5) \in H, H = \{(3, 5), (4, 5)\}$. Note that $\{2, 3, 4, 5\}$ is also a maximal clique in G'_e.

Our change-sensitive algorithm, `IMCENewClq` (Algorithm 2) is based on the above observation, and uses an output-sensitive algorithm `MCE`, due to [32], to enumerate all maximal cliques in G'_e.

We now present the time space complexity analysis of `IMCENewClq`. Our analysis shows that `IMCENewClq` is a change-sensitive algorithm for enumerating new maximal cliques, since its time complexity is polynomial in the maximum degree Δ and linear in the number of new maximal cliques $|\Lambda^{new}|$ and in the number of new edges ρ.

Theorem 4. `IMCENewClq` *enumerates the set of all new cliques arising from the addition of H in time $O(\Delta^3 \rho |\Lambda^{new}|)$ where Δ is the maximum degree of a vertex in G'. The space complexity is $O(|E(G + H)| + |V(G + H)|)$.*

```
Algorithm 2: IMCENewClq(G, H)
   Input: $G$ - Input graph, $H$ - Set of $\rho$ edges added to $G$
   Output: All cliques in $\Lambda^{new}$, each clique output once
1  Consider edges of $H$ in an arbitrary order $e_1, e_2, \ldots, e_\rho$
2  $G' \leftarrow G + H$
3  for $i = 1 \ldots \rho$ do
4      $e \leftarrow e_i = (u, v)$
5      $V_e \leftarrow \{u, v\} \cup \{\Gamma_{G'}(u) \cap \Gamma_{G'}(v)\}$
6      $G'_e \leftarrow$ graph induced by $V_e$ on $G'$
7      Generate cliques using MCE($G'_e$). For each clique $c$ thus generated, output $c$ only
         if $c$ does not contain an edge $e_j$ for $j < i$
```

Proof. We first prove the correctness of the algorithm. From Lemmas 11 and 12, we have that by enumerating $\mathcal{C}(G'_e)$ for every $e \in H$, we enumerate Λ^{new}. Our algorithm does exactly that, and enumerates $\mathcal{C}(G'_e)$ using Algorithm MCE. Note that each clique $c \in \Lambda^{new}$ is output exactly once though c maybe in $\mathcal{C}(G'_e)$ for multiple edges $e \in H$. This is because c is output only for edge e that occurs earliest in the pre-determined ordering of edges in H.

For the runtime, consider that the algorithm iterates over the edges in H. In an iteration involving edge e, it constructs a graph G'_e and runs MCE(G'_e). Note that the number of vertices in G'_e is no more than $\Delta + 1$, and is typically much smaller, since it is the size of the intersection of two vertex neighborhoods in G'. Since the arboricity of a graph is less than its maximum degree, $\alpha' \leq \Delta$ where α' is the arboricity of G'_e. Further, the number of edges in G'_e is $O(\Delta^2)$. The set of maximal cliques generated in each iteration is a subset of Λ^{new}, hence the number of maximal cliques generated from each iteration is no more than $|\Lambda^{new}|$. Applying Theorem 2, we have that the runtime of each iteration is $O(\Delta^3|\Lambda^{new}|)$.

Within each iteration, the time taken to generate the subgraph $G'(e)$ is $O(\Delta^2)$, which is dominated by the term $O(\Delta^3|\Lambda^{new}|)$. For each new edge added, there must be a new maximal clique that contains this edge. Hence, as long as $\rho > 0$ i.e. at least one new edge is added, Λ^{new} is a non-empty set. The overall runtime of each iteration is bounded by $O(\Delta^3|\Lambda^{new}|)$. Since there are ρ iterations, the result on runtime follows.

For the space complexity, we note that the algorithm does not store the set of new cliques in memory at any point. The space required to construct G'_e is linear in the size of $G' = (G + H)$, and so is the space requirement of Algorithm `MCE`(G'_e), from Theorem 2. Hence the total space requirement is linear in the number of edges in $G + H$. □

4.4.2 Practical Algorithm for Enumerating New Maximal Cliques

Now we present an efficient algorithm `FastIMCENewClq` for enumerating new maximal cliques when new edges are added. This is based on the theoretically efficient algorithm `IMCENewClq`, and incorporates improvements that we discuss next.

The algorithm `IMCENewClq` uses as a subroutine Algorithm `MCE` (Chiba and Nishizeki [32]) to enumerate maximal cliques within a subgraph of G. While `MCE` is theoretically output-sensitive, in practice, it is not the most efficient algorithm for maximal clique enumeration. The most efficient algorithms for maximal clique enumeration in a static graph are typically based on depth-first search using a technique called "pivoting", such as the algorithm `TTT` due to Tomita et al. [143] as we have explained in Section 4.2. It is possible to directly improve the performance of the `IMCENewClq` algorithm by using `TTT` in place of `MCE`. In the following, we show how to do even better.

Reducing Redundant Clique Computation: Note that `IMCENewClq` (Algorithm 2) may compute the same clique c multiple times. For example, if $c \in C'(e_1)$ and $c \in C'(e_2)$ for two distinct edges e_1 and e_2, c will be enumerated (at least) twice, once when considering e_1 in the for loop, and once while considering edge e_2. In Line 7, duplicates are suppressed prior to emitting the cliques, by outputting c only for one of the edges among $\{e_1, e_2\}$. However, the algorithm still pays the computational cost of computing a clique such as c multiple times.

We now present a method, Algorithm `FastIMCENewClq` to avoid such redundant clique computation. The idea is to consider the edges in H in a specific order $e_1, e_2, \ldots$. When enumerating

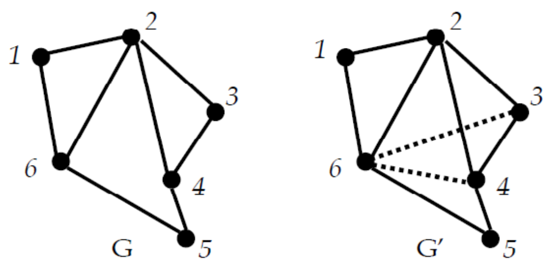
Algorithm 3: `TTTExcludeEdges`$(\mathcal{G}, K, \texttt{cand}, \texttt{fini}, \mathcal{E})$

```
Input: $\mathcal{G}$ - The input graph, $K$ - a non-maximal clique to extend
cand - Set of vertices that may extend $K$, fini - vertices that have been used to
extend $K$
$\mathcal{E}$ - set of edges to exclude
1  if $(\texttt{cand} = \emptyset)$ & $(\texttt{fini} = \emptyset)$ then
2      Output $K$ and return
3  pivot $\leftarrow$ $(u \in \texttt{cand} \cup \texttt{fini})$ such that $u$ maximizes the size of $\texttt{cand} \cap \Gamma_{\mathcal{G}}(u)$
4  ext $\leftarrow$ $\texttt{cand} - \Gamma_{\mathcal{G}}(\texttt{pivot})$
5  for $q \in$ ext do
6      $K_q \leftarrow K \cup \{q\}$
7      if $K_q \cap \mathcal{E} \neq \emptyset$ then
8          $\texttt{cand} \leftarrow \texttt{cand} - \{q\}$ ; $\texttt{fini} \leftarrow \texttt{fini} \cup \{q\}$
9          continue
10     $\texttt{cand}_q \leftarrow \texttt{cand} \cap \Gamma_{\mathcal{G}}(q)$ ; $\texttt{fini}_q \leftarrow \texttt{fini} \cap \Gamma_{\mathcal{G}}(q)$
11     TTTExcludeEdges$(\mathcal{G}, K_q, \texttt{cand}_q, \texttt{fini}_q, \mathcal{E})$
12     $\texttt{cand} \leftarrow \texttt{cand} - \{q\}$ ; $\texttt{fini} \leftarrow \texttt{fini} \cup \{q\}$
```

all cliques in $C(e_i)$, the algorithm prunes out search paths that lead to cliques containing edge $e_j, j < i$. This way, each new clique is enumerated exactly once.

For this purpose, Algorithm `FastIMCENewClq` uses as a subroutine Algorithm `TTTExcludeEdges` (Algorithm 3), an extension of the `TTT` algorithm, which enumerates all maximal cliques of an input graph that avoid a given set of edges. While `TTT` simply takes a graph as input and enumerates all maximal cliques within the graph, `TTTExcludeEdges` takes an additional input, a set of edges $\mathcal{E}$, and only enumerates those cliques within the graph that do not contain any edge from $\mathcal{E}$. We present a recursive version of `TTTExcludeEdges`, which takes as input five parameters – an input graph G, three sets of vertices K, `cand`, and `fini`, and a set of edges $\mathcal{E}$. The algorithm outputs every maximal clique in G that (a) contain all vertices in K, (b) zero or more vertices in `cand`, (c) none of the vertices in `fini`, and (d) none of the edges in $\mathcal{E}$.

A description of `TTTExcludeEdges` is presented in Algorithm 3, and an example of its its output in Figure 4.5. This algorithm follows the structure of the recursion in the `TTT` algorithm, and



new maximal cliques

TTT	TTTExcludeEdges
edge (3,6): {2,3,4,6}	edge (3,6): {2,3,4,6}
edge (4,6): {2,3,4,6}, {4,5,6}	edge (4,6): {4,5,6}

Figure 4.5: Enumeration of new maximal cliques from G to G' due to addition of new edges $(3, 6)$ and $(4, 6)$. Order the new edges as $(3, 6)$ followed by $(4, 6)$. There are two new maximal cliques containing edge $(4, 6)$, $\{4, 5, 6\}$ and $\{2, 3, 4, 6\}$. With `TTTExcludeEdges`, only $\{4, 5, 6\}$ is enumerated when considering edge $(4, 6)$, since $\{2, 3, 4, 6\}$ has already been enumerated while considering edge $(3, 6)$.

incorporates additional pruning of search paths, by avoiding paths that contain an edge from $\mathcal{E}$. In particular, in Line 7 of `TTTExcludeEdges`, if the clique K_q (formed after adding vertex q to K) contains an edge from $\mathcal{E}$, then the rest of the search path, which will continue adding more vertices, is not explored further. Instead the algorithm backtracks and tries to extend the clique K by adding other vertices.

Our algorithm for enumerating new maximal cliques `FastIMCENewClq` (Algorithm 4) is an adaptation of `IMCENewClq` (Algorithm 2) where we use `TTTExcludeEdges` instead of the output-sensitive `MCE`. In particular, while enumerating all new cliques containing edge e_i, `FastIMCENewClq` enumerates only those cliques that exclude edges $\{e_1, e_2 \ldots, e_{i-1}\}$. Note that in `FastIMCENewClq`, there is no further duplicate suppression required, since the call to `TTTExcludeEdges` does not return any cliques that contain an edge from $\mathcal{E}$. This is more efficient than first enumerating duplicate cliques, followed by suppressing duplicates before emitting them. This idea makes `FastIMCENewClq` more efficient in practice than `IMCENewClq`.

The correctness of `FastIMCENewClq` follows in a similar fashion to that of Algorithm `IMCENewClq` proved in Theorem 4, except that we also need a proof of the guarantee provided by Algorithm `TTTExcludeEdges`, which we establish in the following lemma.

Algorithm 4: `FastIMCENewClq`(G, H)

```
Input: $G$ - input graph
       $H$ - Set of $\rho$ edges being added to $G$
Output: Cliques in $\Lambda^{new} = \mathcal{C}(G+H) \setminus \mathcal{C}(G)$
1  $G' \leftarrow G + H$ ; $\mathcal{E} \leftarrow \phi$
2  Consider edges of $H$ in an arbitrary order $e_1, e_2, \ldots, e_\rho$
3  for $i \leftarrow 1, 2, \ldots, \rho$ do
4      $e \leftarrow e_i = (u, v)$
5      $V_e \leftarrow \{u, v\} \cup \{\Gamma_{G'}(u) \cap \Gamma_{G'}(v)\}$
6      $\mathcal{G} \leftarrow$ Graph induced by $V_e$ on $G'$
7      $K \leftarrow \{u, v\}$
8      cand $\leftarrow V_e \setminus \{u, v\}$ ; fini $\leftarrow \emptyset$
9      $S \leftarrow$ TTTExcludeEdges($\mathcal{G}, K,$ cand, fini, $\mathcal{E}$)
10     $\Lambda^{new} \leftarrow \Lambda^{new} \cup S$
11     $\mathcal{E} \leftarrow \mathcal{E} \cup e_i$
```

Lemma 13. `TTTExcludeEdges`$(\mathcal{G}, K,$ `cand`, `fini`$, \mathcal{E})$ *(Algorithm 3) returns all maximal cliques c in $\mathcal{G}$ such that (1) c contains all vertices from K, (2) remaining vertices in c are chosen from* `cand`*, (3) c contains no vertex from* `fini` *and (4) c does not contain any edges in $\mathcal{E}$.*

Proof. We note that `TTTExcludeEdges` matches the original `TTT` algorithm, except for lines 7 to 9. Hence, if we do not consider lines 7 to 9 in `TTTExcludeEdges`, the algorithm becomes `TTT`, and by the correctness of `TTT` (Theorem 1 [143]), all maximal cliques c in $\mathcal{G}$ are returned. Now consider lines 7 to 9 in `TTTExcludeEdges`. Clearly, (1), (2), (3) are preserved for each maximal clique c generated by `TTTExcludeEdges`. Now to complete the correctness proof of `TTTExcludeEdges`, along with proving (4), we also need to prove that each maximal clique c in $\mathcal{G}$ that does not contain any edge in $\mathcal{E}$ is generated by `TTTExcludeEdges`. Assume there exists a maximal clique c in $\mathcal{G}$, which contains an edge in $\mathcal{E}$, which is output by the `TTTExcludeEdges` algorithm, assume the offending edge is $e = (q, v)$. Suppose that vertex v was added to our expanding clique first. Then, as q is processed, line 7 of the algorithm will return back true as $e \in K_q$ and $\mathcal{E}$, thus q will not be added to the clique, and c will not be reported as maximal, a contradiction.

Next, we show if a maximal clique does not contain an edge from $\mathcal{E}$, the clique will be generated. Consider a maximal clique c in $\mathcal{G}$ that contains no edge from $\mathcal{E}$ but c is not generated by `TTTExcludeEdges`. The only reason for c not being generated is the inclusion of lines 7 to 9 (of `TTTExcludeEdges`) to `TTT` resulting in `TTTExcludeEdges`, because, otherwise, c would be generated due to correctness of `TTT`. So in `TTTExcludeEdges`, during the expansion of K towards c, there exists a vertex $q \in c$ such that line 7 in `TTTExcludeEdges` is satisfied and c never gets a chance to be generated as q is excluded from `cand` and included in `fini` (line 8). This implies that c contains at least an edge in $\mathcal{E}$, because otherwise, condition at line 7 would never be satisfied. This is a contradiction, and completes the proof. □

4.4.3 Enumeration of Subsumed Maximal Cliques

We now consider the enumeration of subsumed cliques, i.e. the set $\mathcal{C}(G) \setminus \mathcal{C}(G+H)$. A subsumed clique c' still exists in $G' = G + H$, but is now a part of a larger clique in G'. Such a larger clique must be a part of Λ^{new}. Thus, an algorithm idea is to check each new clique c in Λ^{new} to see if c subsumed any maximal clique c' in G. In order to see which maximal cliques c may have subsumed, we note that any maximal clique subsumed by c must also be a maximal clique within subgraph $c - H$. Thus, one approach is to enumerate all maximal cliques in $c - H$ and for each such generated clique c', we check whether c' is maximal in G by verifying maximality of c' in G. This algorithm can be implemented in space proportional to the size of $G + H$, since it can directly use an algorithm for maximal clique enumeration such as `MCE`.

However, in practice, checking each potential clique for maximality is a costly operation since it potentially needs to consider the neighborhood of every vertex of the clique. An alternative approach to avoid this costly maximality check is to store the set of maximal cliques $\mathcal{C}(G)$ and check if c' is in $\mathcal{C}(G)$. The downside of this approach is that the space required to store the clique set can be high.

Hence, we considered another approach to subsumed cliques, where we reduce the memory cost by storing signatures of maximal cliques as opposed to the cliques themselves. The signature is computed by representing a clique in a canonical fashion (for instance, representing the clique as a list of vertices sorted by their ids.) as a string followed by computing a hash of this string. By storing only the signatures and not the cliques themselves, we are able to check if a clique is a current maximal clique, and at the same time, pay far lesser cost in memory when compared with storing the clique itself. The procedure is as described in Algorithm 5. With this approach of storing signatures instead of storing the cliques themselves, there is a (small) chance of collision of signatures, which means that the signatures of two different cliques C_1 and C_2 might be the same. This might result in false positives meaning that some cliques might wrongly be concluded as subsumed cliques. However, the probability of the event that the hash values of two different cliques are same is extremely low with the use of a hashing algorithm such as 64-bit murmur hash [1]. In our experiments, we observed that the set of subsumed cliques reported with the use of signature is always the same as the actual set of subsumed cliques. If it is extremely important to avoid false positives, we can explicitly check a potential subsumed clique for maximality in the original graph.

In Algorithm 5, Lines 4 to 12 describes the procedure for computing S, the set of all maximal cliques in $c - H$ and Lines 13 to 15 decide which among the maximal cliques in S are subsumed. For computing maximal cliques in $c - H$, we only consider the edges in H that are present in c as we can see in Line 4. We prove that S is the set of all maximal cliques in $c - H$ in the following lemma using an induction on the number of edges in H those are present in c:

Lemma 14. *In Algorithm 5, for each $c \in \Lambda^{new}$, S contains all maximal cliques in $c - H$.*

Proof. Note that we only consider the set of all edges $H_1 \subseteq H$ which are present in c (line 4). It is clear that computing maximal cliques in $c - H$ is equivalent to to computing maximal cliques in $c - H_1$.

We prove the lemma using induction on k, the number of edges in H_1. Suppose $k = 1$ so that H_1 is a single edge, say $e_1 = \{u, v\}$. Note that $c - H_1$ has two maximal cliques, $c \setminus \{u\}$ and $c \setminus \{v\}$. It can be verified that in Algorithm 5 cliques $c \setminus \{u\}$ and $c \setminus \{v\}$ are inserted into S, thus proving the base case.

[1]https://sites.google.com/site/murmurhash/

Suppose that for any set H_1 of size k, it is true that all maximal cliques in $c - H_1$ have been generated using induction hypothesis. Consider a set $H_1' = \{e_1, e_2, ..., e_{k+1}\}$ with $(k+1)$ edges. Now each maximal clique c' in $c - H_1$ either remains a maximal clique within $c - H_1'$ (if at least one endpoint of e_{k+1} is not in c'), or leads to two maximal cliques in $c - H_1'$ (if both endpoints of e_{k+1} are in c'). Thus lines 4 to 12 in Algorithm 5 generate all maximal cliques in $c - H$. □

Algorithm 5: `IMCESubClq`(G, H, D, Λ^{new})

```
Input: $G$ - Input Graph
       $H$ - Edge set being added to $G$
       $D$ - Set of maximal cliques in $G$
       $\Lambda^{new}$ - set of new maximal cliques in $G + H$
Output: All cliques in $\Lambda^{del} = \mathcal{C}(G) \setminus \mathcal{C}(G + H)$
1  $\Lambda^{del} \leftarrow \emptyset$
2  for $c \in \Lambda^{new}$ do
3      $S \leftarrow \{c\}$
4      for $e = (u, v) \in E(c) \cap H$ do
5          $S' \leftarrow \phi$
6          for $c' \in S$ do
7              if $e \in E(c')$ then
8                  $c_1 = c' \setminus \{u\}$ ; $c_2 = c' \setminus \{v\}$
9                  $S' \leftarrow S' \cup c_1$ ; $S' \leftarrow S' \cup c_2$
10             else
11                 $S' \leftarrow S' \cup c'$
12         $S \leftarrow S'$
13     for $c' \in S$ do
14         if $c' \in D$ then
15             $\Lambda^{del} \leftarrow \Lambda^{del} \cup c'$
16             $D \leftarrow D \setminus c'$
```

We show that the above is a change-sensitive algorithm for enumerating Λ^{del} in the case when the number of edges ρ in H is a constant. In the following lemma (Lemma 15), we present the time and space complexity of `IMCESubClq` where we use an induction on ρ for proving the time complexity. Note that the time complexity is change-sensitive when ρ is a constant because, the time complexity is linear on the size of Λ^{new}.

Lemma 15. *Algorithm* `IMCESubClq` *(Algorithm 5) enumerates all cliques in $\Lambda^{del} = \mathcal{C}(G) \setminus \mathcal{C}(G')$ using time $O(2^{\rho}|\Lambda^{new}|)$. The space complexity of the algorithm is $O(|E(G')|+|V(G')|+|\mathcal{C}(G)|)$. The algorithm can also be adapted to run in time $O(2^{\rho}|E(G)||\Lambda^{new}|)$, and space $O(|E(G')| + |V(G')|$.*

Proof. We first show that every clique c' enumerated by the algorithm is indeed a clique in Λ^{del}. To see this, note that c' must be a maximal clique in G, due to explicitly checking the condition. Further, c' is not a maximal clique in G', since it is a proper subgraph of c, a maximal clique in G'. Next, we show that all cliques in Λ^{del} are enumerated. Consider any subsumed clique $c_1' \in \Lambda^{del}$. It must be contained within $c_1 - H$, where $c_1 \in \Lambda^{new}$. Moreover, c_1' will be a maximal clique within $c_1 - H$, and will be enumerated by the algorithm according to Lemma 14.

For the time complexity we show that for any $c \in \Lambda^{new}$, the maximum number of maximal cliques in $c^{-H} = c - H$ is 2^{ρ}. Proof is by induction on ρ. Suppose $\rho = 1$ so that H is a single edge, say $e_1 = \{u, v\}$. Then clearly c^{-H} has two maximal cliques, $c \setminus \{u\}$ and $c \setminus \{v\}$, proving the base case. Suppose that for any set H of size k, it was true that c^{-H} has no more than 2^k maximal cliques. Consider a set $H'' = \{e_1, e_2, \ldots, e_{k+1}\}$ with $(k + 1)$ edges. Let $H' = \{e_1, e_2, \ldots, e_k\}$. Subgraph $c - H''$ is obtained from $c - H'$ by deleting a single edge e_{k+1}. By induction, we have that $c - H'$ has no more than 2^k maximal cliques. Each maximal clique c' in $c - H'$ either remains a maximal clique within $c - H''$ (if at least one endpoint of e_{k+1} is not in c') , or leads to two maximal cliques in $c - H''$ (if both endpoints of e_{k+1} are in c'). Hence, the number of maximal cliques in $c - H''$ is no more than 2^{k+1}, completing the inductive step.

Thus, for each cliques $c \in \Lambda^{new}$, we need to check maximality for no more than 2^{ρ} cliques in G. Note that a clique c' is maximal in G if it is contained in $\mathcal{C}(G)$, the set of maximal cliques in G. This can be done in constant time by storing the signatures of maximal cliques and checking if the signature of c' is in the set of signatures of maximal cliques of G.

For the space bound, we first note that all operations in Algorithm 5 except maximality check can be done in space linear in the size of G'. For maximality check we need space $O(|\mathcal{C}(G)|)$ as we need to store the (signatures of) maximal cliques of G. The only remaining space cost is the size of Λ^{new}, which can be large. Note that the algorithm only iterates through Λ^{new} in a single pass. If elements of Λ^{new} were provided as a stream from the output of an algorithm such as `IMCENewClq`, then they do not need to be stored within a container, so that the memory cost of receiving Λ^{new} is reduced to the cost of storing a single maximal clique within Λ^{new} at a time.

An alternative algorithm does not store $\mathcal{C}(G)$ (or hashes of elements in $\mathcal{C}(G)$). Instead, each time a potential subsumed clique c' is generated that is contained in a new clique $c \in \Lambda^{new}$, we simply check c' for maximality in G. This can be done in time $O(|E(G)|)$, by checking the intersections of the different vertex neighborhoods – typical runtime for maximality checking can be much smaller. □

4.4.4 Decremental Case

Next, we consider the case when a set of edges H is deleted from G. A set of edges H is deleted from graph G, and we are interested in efficiently enumerating $\Lambda(G, G-H)$. The decremental case can be reduced to the incremental case through the following observation.

Observation 2. $\Lambda^{del}(G, G-H) = \Lambda^{new}(G-H, G)$ *and* $\Lambda^{new}(G, G-H) = \Lambda^{del}(G-H, G)$

Proof. Consider the first equation: $\Lambda^{del}(G, G-H) = \Lambda^{new}(G-H, G)$. Let $c \in \Lambda^{del}(G, G-H)$. This means that $c \in \mathcal{C}(G)$ and $c \notin \mathcal{C}(G-H)$. Equivalently, c is not a maximal clique in $G-H$, but upon adding H to $G-H$, c becomes a maximal clique in G. Hence, it is equivalent to say that $c \in \Lambda^{new}(G-H, G)$. Hence, we have $\Lambda^{del}(G, G-H) = \Lambda^{new}(G-H, G)$. The other equation, $\Lambda^{new}(G, G-H) = \Lambda^{del}(G-H, G)$, can be proved similarly. □

The decremental algorithm for maximal cliques is outlined in Algorithm 6 (`IMCED`).

4.4.5 Fully Dynamic Case

Consider the *fully dynamic case*, where there is a set of insertions (edge set H) as well as deletions (edge set H') from a graph. This can be processed as follows. First, we ensure there is

```
Algorithm 6: IMCED($G, H$)
   Input: $G$ - Input Graph, $H$ - Set of $\rho$ edges being deleted
   Output: All cliques in $\Lambda^{new}(G, G - H) \cup \Lambda^{del}(G, G - H)$
1  $\Lambda^{new} \leftarrow \emptyset$, $\Lambda^{del} \leftarrow \emptyset$, $G'' \leftarrow G - H$
2  $\Lambda^{del} \leftarrow$ IMCENewClq($G'', H$)
3  $\Lambda^{new} \leftarrow$ IMCESubClq($G'', H, \mathcal{C}(G''), \Lambda^{del}$)
```

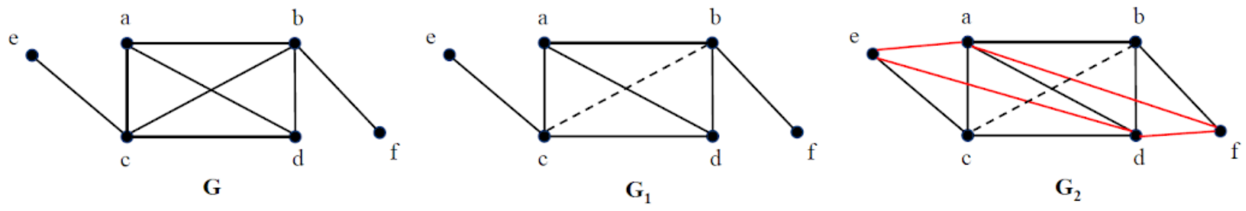


Figure 4.6: Change in the maximal cliques due to both addition and deletion of edges. The initial graph G, graph G_1 after deleting edge (b, c) from G, resulting in new maximal cliques $\{a, c, d\}$ and $\{a, b, d\}$ and one deleted maximal clique $\{a, b, c, d\}$, graph G_2 after adding edges (a, e), (e, d), (a, f), and (d, f) from G_1 resulting in new maximal cliques $\{a, e, d, c\}$ and $\{a, b, d, f\}$ and subsumed cliques $\{a, c, d\}$, $\{a, b, d\}$, $\{c, e\}$, and $\{b, f\}$. Note that, the intermediate new cliques (at state G_1) $\{a, c, d\}$ and $\{a, b, d\}$ are only "transient" maximal cliques, and are not in the final graph G_2.

no overlap between H and H', i.e. $H \cap H' = \emptyset$. If this is not the case, we can simply remove overlapping elements since they have no effect on the final graph. Next, we enumerate the change following all the edge deletions, followed by enumerating the change upon edge insertions. Note however, that this may not lead to a change-sensitive algorithm. Intermediate cliques that are output may not be in the final set of new or subsumed cliques. See Figure 4.6 for an example.

4.5 Discussion

Our incremental algorithm IMCE can adapted to related problems such as maintaining top-k maximal cliques. Also, the techniques developed in the work can be used for the maintenance of maximal cliques with additional search context such as graph with labels at nodes or vertices.

Maintenance of top-k maximal cliques: Observe that the vertices corresponds to the top-k maximal cliques are of a high degree. More precisely, if the smallest size of the clique among the current top-k maximal cliques is s, then we only need to consider the vertices of the original graph whose degree is at least $(s-1)$. Thus, given top-k maximal cliques of G, we can update the top-k maximal cliques of the graph $G' = G + H$ using our incremental algorithm as follows: (1) For computing new maximal cliques, we only enumerate those with size at least s. We can do this by recursively deleting vertices of degree smaller than $s-1$ from the subgraph used ($\mathcal{G}$ at Line 6 of Algorithm 4), adding these vertices to the `fini` set (Line 8 of Algorithm 4) and then enumerating maximal cliques of the updated graph containing the rest of the vertices (by adding them to the `cand` set at Line 8 of Algorithm 4). This ensures that each maximal clique such enumerated, is of size at least s. (2) It is possible that some of the maximal cliques in top-k may be subsumed by larger new maximal cliques when new edges are added. The algorithm for subsumed cliques can deal with this situation in the following manner: instead of checking for the containment in the set of maximal cliques of the original graph, check for the containment in the set of top-k maximal cliques (Line 14 of Algorithm 5 where the set C contains only top-k maximal cliques of G instead of all maximal cliques of G). Eventually, as a result of subsumption, it might happen that, the number of maximal cliques is less than k in the final set. For handling this situation, it is required to generate the set of all new maximal cliques and sort them in decreasing order of sizes.

Maintenance of maximal cliques with search context: Search contexts are relevant in "keyword" based or "topic" based searches, or in the combinations of the two, such as finding communities with people interested in a specific topic [54, 66]. The network in this context contains labels attached to nodes/edges. We can use `IMCE` for maintaining maximal cliques such that each of the nodes in the maximal cliques contains one or a group of specified keywords/labels. There might be two cases.

First, consider the case when vertices contain labels. Here, we should consider only those vertices to add to `cand` set (Line 8 of Algorithm 4) from the graph G'_e that contains the specified labels and put rest of the vertices in `fini` set. This way, we can use `IMCE` for maintaining only those maximal cliques that contain specified labels to the vertices. Next we consider the case when edges contain labels. For generating maximal cliques with each of the edges containing specified labels, we consider Algorithm 3, and add an additional check for constraints on edge labels whenever we add a vertex q to K (in Line 6 of the algorithm).

Boundedness of incremental computation: In the context of a recent theoretical framework for incremental graph algorithms [53], the time complexity of `IMCE` is *bounded* when the size of the batch ρ is fixed. Our analysis shows that when the original graph is large and the changes in the graph is small, the time complexity of computing the change is proportional to the size of the change in the set of maximal cliques, which follows the definition of bounded computation as defined in [53]. Also, `IMCE` admits *localizable computations* [53] as we focus on the subgraphs local to the changes in the graph structure for enumerating the changes.

4.6 Experimental Evaluation

In this section, we present results from empirical evaluation of the performance of algorithms proposed in this paper. We address the following questions: (1) What is the computation time and memory usage of our algorithms? (2) How does the computation time compare with the magnitude of the change when new edges are added (incremental algorithm), when existing edges are deleted (decremental algorithm), and in the fully dynamic case, when edges are both added and deleted? (3) What is the impact on the computation time of our incremental algorithm when the stream of new edges are located around high/low degree vertices of the original graph? (5) In the incremental case, can we achieve a space-time trade-off in subsumed clique computation, depending on whether or not we store the (signatures) of the set of maximal cliques? (6) How does our algorithms compare with prior works?

4.6.1 Datasets

We consider graphs from the Stanford large graph database [86], KONECT- The Koblenz Network Collection [2], and Network Repository [3]: `dblp-coauthor` is a co-authorship network where each vertex represents an author and there is an edge between two authors if they have a common publication. `flickr-growth` is a social network of Flickr users where each vertex represents a user and there exists a directed edge if two users are friends. `ca-cit-HepTh` is a citation network in high energy physics theory in a period from January 1993 to April 2003 where each vertex represents a paper and there is an edge from "a" to "b" if paper "a" cited paper "b". `wikipedia-growth` is a hyperlink network of the English Wikipedia where each vertex represents a wikipedia page and there is an edge from a page $wiki_1$ to a page $wiki_2$ if there is a hyperlink of $wiki_2$ from $wiki_1$. `facebook-friendship` is a friendship network where vertex represents user and there is an edge between two users if they are friends. In each graph, edges have time-stamps of creation. We convert all these graphs into simple undirected graphs. If there are multiple time-stamp edges between two vertices, we take the edge with the earliest time-stamp. `soc-livejournal` is a social network of LiveJournal where vertex represents user and there is an edge between two users when they are friends. As the original graph does not contain timestamps at its edges, we synthetically generate timestamps of edges by assigning an integer uniformly chosen at random between 0 and the number of edges in the network to each edge. A summary of the graphs used in this experiment is given in Table 4.1. In our experiments, for incremental computation we start with the empty graph and at each iteration, we add a batch of new edges (in the increasing order of timestamps), and enumerate the change in maximal cliques after the addition. For decremental computation, we start with the original graph and it each iteration, delete a batch of existing edges (in the decreasing order of timestamps), and enumerate the change in maximal cliques after the deletion.

Next we generate three synthetic RMAT [25] graphs. An RMAT-n-m graph has n vertices and m edges. We use `RMAT-50K-5M` and `RMAT-100K-10M` for addressing graphs with specific edge stream patterns (edge stream around high/low degree vertices) and a high density graph `RMAT-100-4000`

[2] http://konect.uni-koblenz.de/
[3] http://networkrepository.com/

for addressing the behavior of the maintenance algorithms with the change in the density of the graph.

For generating the edge stream for the fully dynamic case, we used the first two RMAT graphs. We first randomly assign a label of either 0 (for edge deletion) or 1 (for edge addition) to each edge. We then assign a randomly chosen timestamp to each edge of the graph. For creating the initial graphs `RMAT-50K-5M-INIT` and `RMAT-100K-10M-INIT`, we remove all the edges from the original graph those are marked 1. We then arrange all edges in increasing order of timestamps for creating the edge stream for `RMAT-50K-5M-INIT`. For `RMAT-100K-10M-INIT`, we order all the edges by grouping them based on the source vertex. Note that a batch of edges contains a mix of new edges to add and existing edges to delete. For experimenting with the stream of edges around high degree vertices, we choose the 1000 highest degree vertices of the initial graph and consider all the edges with a label of 1 that are attached to at least one high degree vertex. Similarly for experimenting with the stream of edges around low degree vertices, we choose the 10,000 lowest degree vertices of the initial graph and consider all the edges with label 1 that are attached to at least a low degree vertex. For creating the edge stream of `RMAT-100-4000`, we follow an approach similar to `soc-livejournal`.

We also considers a variant of the Erdős-Rényi random graph model $G(n, N)$ graph for our experiments where n is the number of vertices and N is the number of edges. In these, we first generate graphs according to the standard Erdős-Rényi random graph model [51], and we "plant" cliques of a certain size. We call these graphs `ER-1M-20M` with 1M vertices, 20M edges and `ER-2M-15M` with 2M vertices and 15M edges. We plant 10 random cliques each of size 20 on `ER-1M-20M` and 10 random cliques each of size 30 on `ER-2M-15M`, with the goal of finding the planted cliques through incremental computation.

Table 4.1: Input graphs and their aggregate statistics.

Dataset	Nodes	Edges	Density	# Maximal Cliques	Maximum Degree	Degeneracy
`dblp-coauthor`	1,282,468	5,179,996	6.3×10^{-6}	1,219,320	1522	118
`flickr-growth`	2,302,925	22,838,276	8.6×10^{-6}	> 400B	27,937	600
`wikipedia-growth`	1,870,709	36,532,531	2.08×10^{-5}	131,652,971	226,073	206
`soc-livejournal`	4,033,137	27,933,062	3.4×10^{-6}	38,413,665	2651	213
`ca-cit-HepTh`	22,908	2,444,798	0.0093	> 400B	8718	561
`facebook-friendship`	63,731	817,035	4×10^{-4}	1,539,038	1098	52
`RMAT-100-4000`	100	4000	0.8	10,180	99	65
`RMAT-50K-5M`	50K	5M	0.004	232,400,002,455	10,496	328
`RMAT-100K-10M`	100K	10M	0.002	144,600,002,154	15,408	371
`ER-1M-20M`	1M	20M	4×10^{-5}	19,978,809	81	29
`ER-2M-15M`	2M	15M	7.5×10^{-6}	14,998,954	59	29

4.6.2 Experimental Setup and Implementation Details

We implemented all the algorithms in Java on a 64-bit Intel(R) Xeon(R) CPU with 16G DDR3 RAM with 13G JVM heap memory.

Algorithm Implementations: We first evaluate our incremental algorithm `IMCE` for maintenance of maximal cliques when new edges are added. `IMCE` consists of `FastIMCENewClq` for enumerating new maximal cliques and `IMCESubClq` for enumerating subsumed maximal cliques. We also implemented the theoretically efficient algorithm `IMCENewClq` for enumerating new maximal cliques. Since `FastIMCENewClq` performed better in all cases, we present results for `FastIMCENewClq`. We also implemented a variant of `IMCENewClq` by replacing `MCE` with `TTT` and name this variant as `IMCENewClqTTT`.

We evaluate a variant of `IMCE` where we use a different strategy for computing subsumed cliques in `IMCESubClq`. Note that deciding subsumed cliques by checking if it is in the set of all maximal cliques of the graph before update requires us to store the set of maximal cliques (or their signatures) of the original graph as in `IMCESubClq`. In this variant, we modify Line 14 of `IMCESubClq` where instead of checking for containment, we directly check for maximality of each c' (Line 13 of algorithm `IMCESubClq`). We name this variant of `IMCE` as $\texttt{IMCE} - \texttt{NoCliqueStore}$. $\texttt{IMCE} - \texttt{NoCliqueStore}$

uses less memory than IMCE, but has to pay an additional overhead to check for maximality for each candidate subsumed clique.

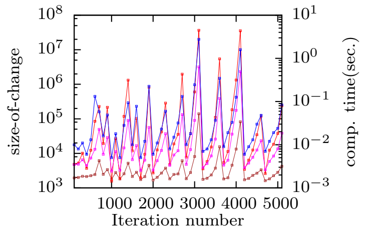
Next we evaluate Algorithm 6 (IMCED) which handles the decremental case. When the graph changes to $G - H$ starting from G due to the deletion of a batch H, we compute the new maximal cliques and deleted maximal cliques following Observation 2. We also experimentally evaluate the fully dynamic case with a mixture of addition and deletion of edges. For dealing with the fully dynamic case, we first remove all edges that are both added and deleted (as these edges do not contribute to the change in the set of maximal cliques). We then run IMCED for computing the changes due to the deletion of edges, followed by IMCE − NoCliqueStore for computing the changes due to the addition of edges. Finally, we generate the overall changes in the set of maximal cliques due to the deletion and addition of edges.

We consider the following prior algorithms for comparison with IMCE: (1) STIX (Stix [135]) computes on a dynamic graph by incrementally adding one edge at a time; (2) OV (Ottosen and Vomlel [112]) computes on a dynamic graph by incrementally adding a set of edges; (3) MCMEI (Sun et al. [136]) computes on a dynamic graph by incrementally adding one edge at a time. We also consider the following prior algorithms for comparison with our decremental algorithm IMCED: (1) STIXD (Stix [135]) computes on a dynamic graph by deleting one edge at a time; (2) MCMED (Sun et al. [136]) computes on a dynamic graph by deleting one edge at a time. For the algorithms (STIX, MCMEI, STIXD, MCMED) that support only single edge addition/deletion, we simulate the addition (deletion) of a batch of edges by inserting (deleting) the edges one at a time. We also compare IMCE and IMCED with baseline algorithms Naive and NaiveD respectively where Naive handles the incremental case by running a static algorithm TTT each time a set of new edges is added to the graph and explicitly computing the symmetric difference; NaiveD similarly handles the decremental case by running TTT each time a set of existing edges is deleted from the graph.

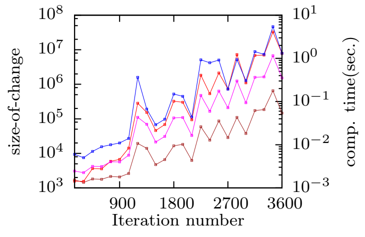
Metrics: We evaluate the performance of algorithms through the following metrics: **(1)** total computation time for determining new maximal cliques and subsumed maximal cliques when a batch of new edges is added to the graph; **(2)** change-sensitiveness, i.e, total computation time as a function of the size of the total change. For defining the size of change in the theoretical analysis (Section 4.4), we used the total number of cliques that were added and deleted. We call this metric as "change-in-number". Note that there are other natural ways to quantify the size of change. If one were to actually enumerate the change, each clique that is added (or deleted) could be written as a set of its constituent vertices. Hence it is natural to consider another metric for the size of change, equal to the sum of the sizes of all cliques that are a part of the change. We call this metric as "change-in-nodes". We further consider another metric "change-in-edges", defined as the sum of the numbers of edges in all the cliques that are a part of the change. For example, suppose there are two new maximal cliques of sizes 3 and 4, and one subsumed clique of size 2. The change-in-number is 3, since there are a total of three cliques to enumerate. The change-in-nodes is $3 + 4 + 2 = 9$. The change-in-edges is $\binom{3}{2} + \binom{4}{2} + \binom{2}{2} = 10$, since a clique on k vertices has $\binom{k}{2}$ edges. We consider all three metrics, change-in-number, change-in-nodes, and change-in-edges, to measure the size of change. **(3)** memory cost, which includes the space required to store the graph as well as additional data structures used by the algorithm; and **(4)** cumulative computation time (through a series of incremental updates) as a function of the size of the batch.

4.6.3 Discussion of Experimental Results

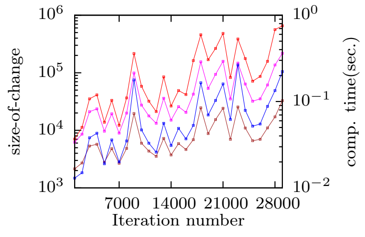
Incremental computation time: Figure 4.7 shows the computation time of `IMCE` for computing the change in the set of maximal cliques when batches of edges are added. The batch size is set to $\rho = 1000$. The size of the change is shown on the left y-axis, and the time for computing the change is shown on the right y-axis. We see that the time for computing the change in the set of maximal cliques becomes greater as iterations progress for graphs `flickr-growth, soc-livejournal, and facebook-friendship`, and remains roughly the same for other graphs. Figure 4.8 shows the breakdown of computation time of `IMCE` into computation time for new maximal cliques (`FastIMCENewClq`) and computation time for subsumed maximal cliques (`IMCESubClq`).



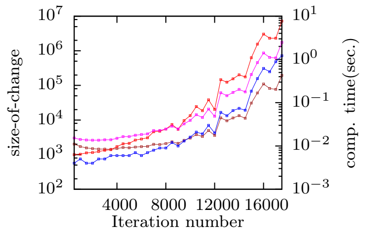
(a) `dblp-coauthor`



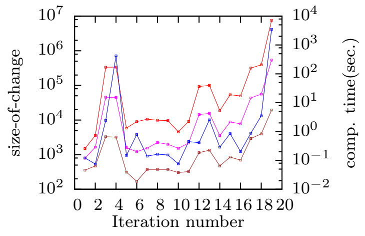
(b) `flickr-growth`



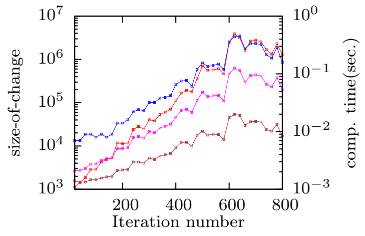
(c) `wikipedia-growth`



(d) `soc-livejournal`



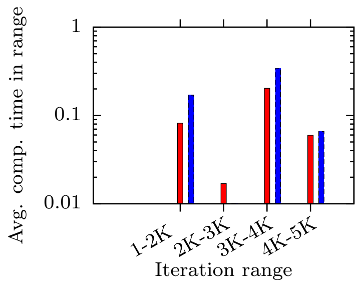
(e) `ca-cit-HepTh`



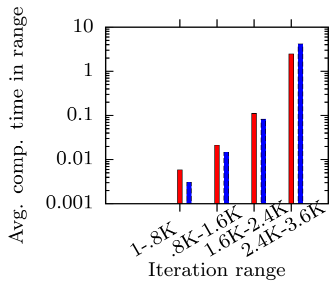
(f) `facebook-friendship`

change-in-number —— **change-in-nodes** ——
change-in-edges —— **time** ——

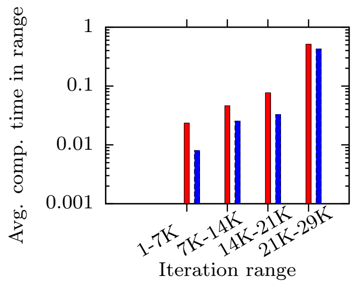
Figure 4.7: Computation time for enumerating the change in set of maximal cliques for `IMCE`, and size-of-change per batch (batch size $\rho = 1000$). The left y axis shows the size of change and the right y axis shows the computation time in seconds.



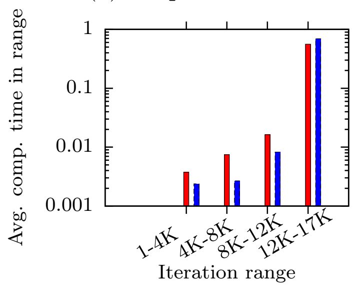
(a) `dblp-coauthor`



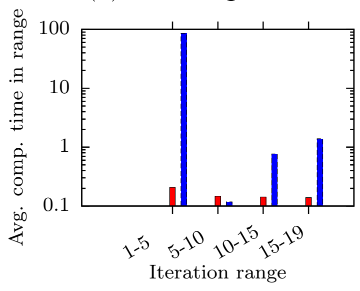
(b) `flickr-growth`



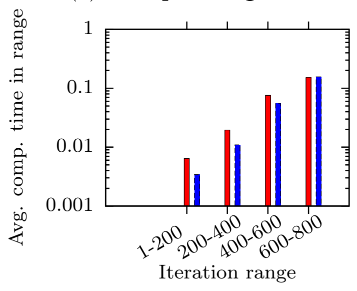
(c) `wikipedia-growth`



(d) `soc-livejournal`



(e) `ca-cit-HepTh`



(f) `facebook-friendship`

`FastIMCENewClq` 🟥 **`IMCESubClq`** 🟦

Figure 4.8: Computation time (in sec.) broken down into time for new and subsumed cliques with batch size $\rho = 1000$. Average time in the y-axis is the average taken over the total computation times (new + subsumed) of the iterations in each of the ranges on the x-axis.

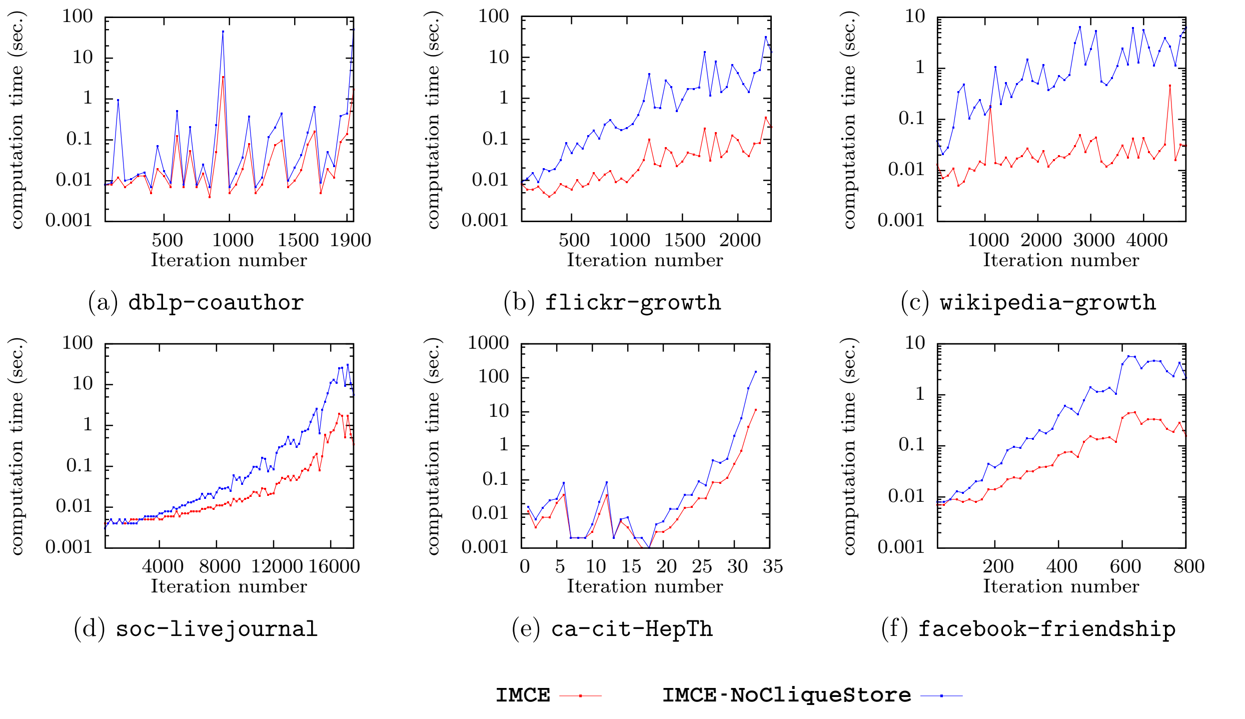


Figure 4.9: Difference in computation time due to different strategies for subsumed cliques computation: once by storing the maximal cliques and another by directly checking for maximality (without storing the maximal cliques). We use batch size 1000 for all graphs except for `ca-cit-HepTh` where we use batch size of 100.

Strategies for subsumed cliques: Next we compare the computation time of `IMCE` with `IMCE` $-$ `NoCliqueStore` as shown in Figure 4.9. Clearly, `IMCE` is faster than `IMCE` $-$ `NoCliqueStore`, because, cost of checking for maximality in computing subsumed cliques as in `IMCE` $-$ `NoCliqueStore` is higher than checking for containment in the set of maximal cliques in computing subsumed cliques as in `IMCE`. We do not observe much difference in computation time for the graphs `dblp-coauthor` and `ca-cit-HepTh` because `dblp-coauthor` is small and sparse; `ca-cit-HepTh` is small and sparse at the initial states of the computation compared to the other graphs. Therefore the sizes of neighborhood of the vertices are small that causes the maximality checking easier.

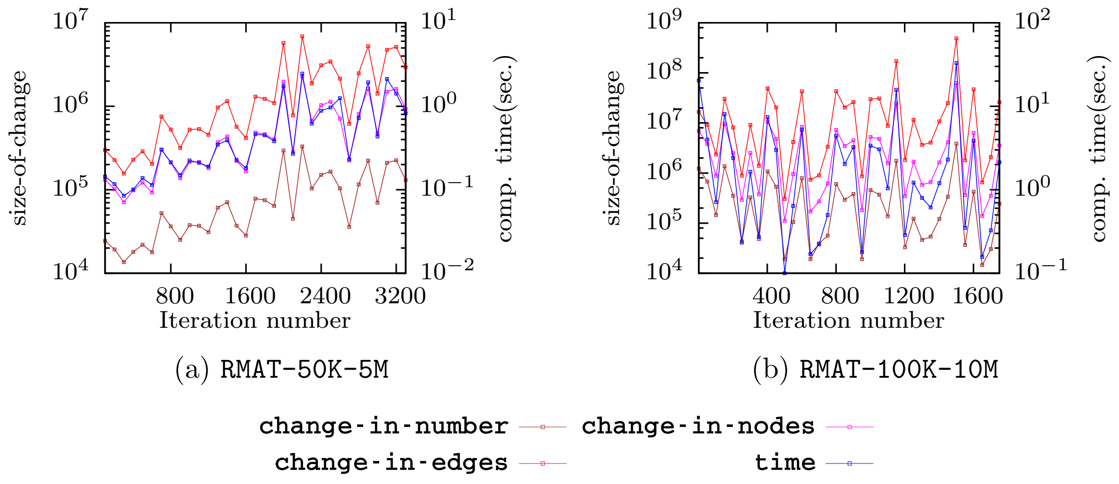


Figure 4.10: Performance of `IMCE` with edge stream centering around 1K highest degree vertices considering batch size 100.

Impact of edge insertion pattern on computation time: We study the computation time of `IMCE` when the new edges centers around high degree and low degree vertices. We have used synthetic graphs `RMAT-50K-5M-INIT` and `RMAT-100K-10M` for this evaluation. We consider new edges around 1K highest degree nodes for creating the edge stream around high degree nodes and consider new edges around 10K least degree nodes for creating the edge stream around low degree nodes. We observe that the changes in the set of maximal cliques are large (Figure 4.10) when the new edges centers around high degree vertices of the initial graph. On the other hand, the changes in the set of maximal cliques is small when the new edges centers around low degree vertices of the initial graph and the computation time is small. For instance, addition of 655 batches (with batch size 100) of edges around low degree vertices, starting with `RMAT-50K-5M-INIT` takes around 0.6 second with cumulative size of change (in the number of maximal cliques) 93K whereas addition of 655 batches of new edges (with the same batch size) centering around high degree vertices takes around 137 sec. with cumulative size of change (in the number of maximal cliques) 1.8×10^7 starting with the same initial graph.

Table 4.2: Cumulative computation time (in sec.) for new maximal cliques with batch size $\rho = 100$. The number of batches for which the cumulative time is computed is in the parenthesis.

Dataset	`IMCENewClq`	`IMCENewClqTTT`	`FastIMCENewClq`
`dblp-coauthor` (9603)	7774	62	24
`flickr-growth` (25,000)	7161	343	125
`wikipedia-growth` (26,795)	446	310	30
`soc-livejournal` (151,997)	7200	474	302
`ca-cit-HepTh` (233)	7464	71	15
`facebook-friendship` (8171)	2100	89	55

Benefits of using* `TTTExcludeEdges` *for new maximal cliques: We compare the computation times of `IMCENewClq` (which uses `MCE` to enumerate cliques), `IMCENewClqTTT` (which uses `TTT`) and `FastIMCENewClq` (which uses `TTTExcludeEdges`) and the results are shown in Table 4.2. We observe that `FastIMCENewClq` is significantly faster than `IMCENewClqTTT` – the difference can be attributed to the additional pruning in `TTTExcludeEdges` when compared with `TTT`. Further, `IMCENewClqTTT` is much faster than `IMCENewClq` – the difference can be attributed to the use of `TTT` which is faster than `MCE`.

On finding planted cliques in synthetic graphs: We observe that `IMCE` can find all "planted" cliques in the synthetic $G(n, N)$ graphs in approximately 20 min. where as the other algorithms (`STIX`, `OV`, `MCMEI`) could not find a single planted clique in an hour. Results are shown

Table 4.3: Total time taken to find all the planted cliques incrementally ($\rho = 100$). Other algorithms (`STIX`, `OV`, `MCMEI`) cannot find a single planted clique within an hour.

Dataset	IMCE	IMCENewClq
ER-1M-20M	19 min.	24 min.
ER-2M-15M	15 min.	15 min.

in Table 4.3.

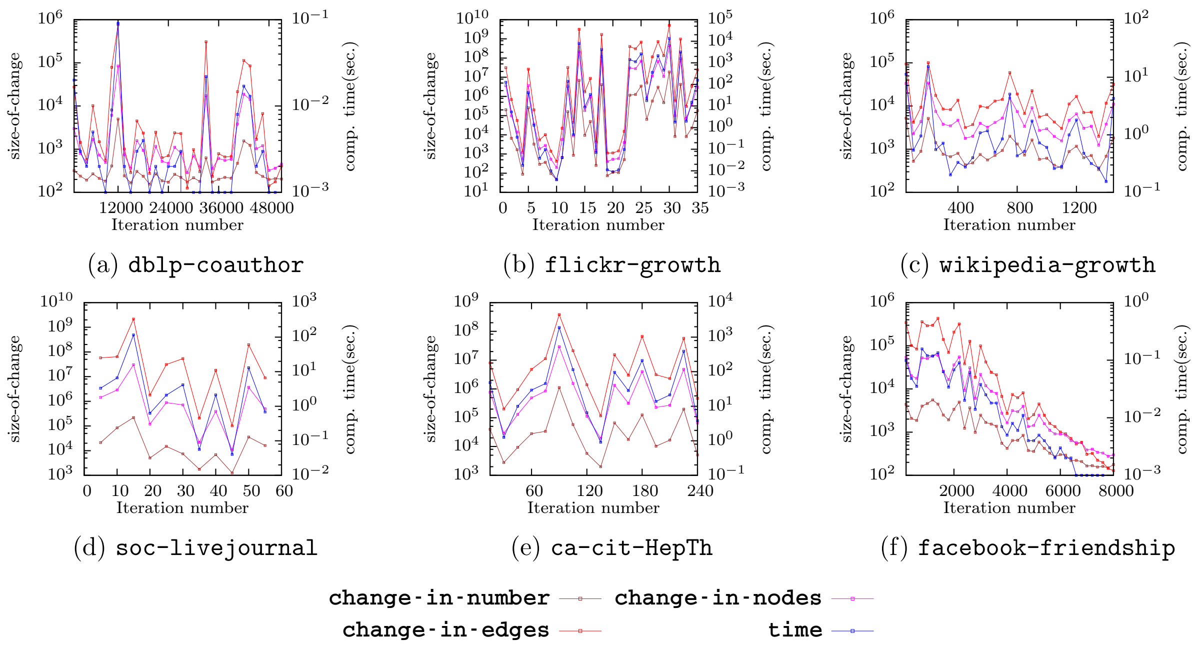


Figure 4.11: Computation time for enumerating the change in set of maximal cliques for decremental case when the edges are deleted from the graph instead of insertion, and size-of-change per batch (batch size $\rho = 100$ except for `flickr-growth` where the batch size is 10). The left y axis shows the size of change and the right y axis shows the computation time in seconds.

Decremental computation time: Figure 4.11 shows the computation time of `IMCED` for computing the changes in the set of maximal cliques when batches of edges are deleted. For this experiment, we choose a batch size of 100 edges. For the graph `flickr-growth`, we had to prune the graph down by 15 million edges, to get a reasonable turnaround time for the computation. We used a batch size of 10 for this graph. Similarly, in the case of `ca-cit-HepTh`, we had to prune the graph down by 1.9 million edges and used the rest of the graph as the initial graph and used the batch size of 100 for performing the decremental computation starting from that point.

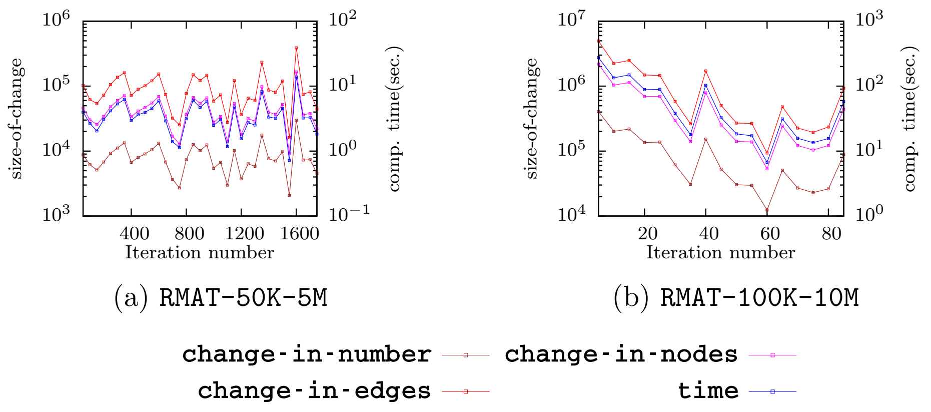


Figure 4.12: Fully dynamic case where both addition and deletion of edges are performed in a streaming manner. Each batch (of size 100) in the stream consists of mixed edges.

Fully dynamic computation time: Figure 4.12 shows the behavior of the algorithm in a fully dynamic setting, where a batch contains both the edges for addition and edges for deletion. For this experiment, we considered synthetic RMAT graphs `RMAT-50K-5M` and `RMAT-100K-10M`.

Impact of graph density on incremental case: Figure 4.13 shows the computation time of `IMCE – NoCliqueStore` upon adding a batch of 100 edges of the initial graphs with different densities. For generating the initial graphs, at every iteration i, we add $100000 \times i$ edges in stream to the empty graph and we insert the batch of next 100 edges to evaluate the performance of `IMCE – NoCliqueStore`. For this experiment, we do not use `IMCE` because for most of the graphs (`flickr-growth, wikipedia-growth, soc-livejournal, ca-cit-HepTh`) the number of maximal cliques at different densities are so large that those cannot fit in the main memory and `IMCE` requires the set of maximal cliques of the initial graph for computing the subsumed cliques. We observe that the cost of computing the changes increases as the graph becomes denser. The changes

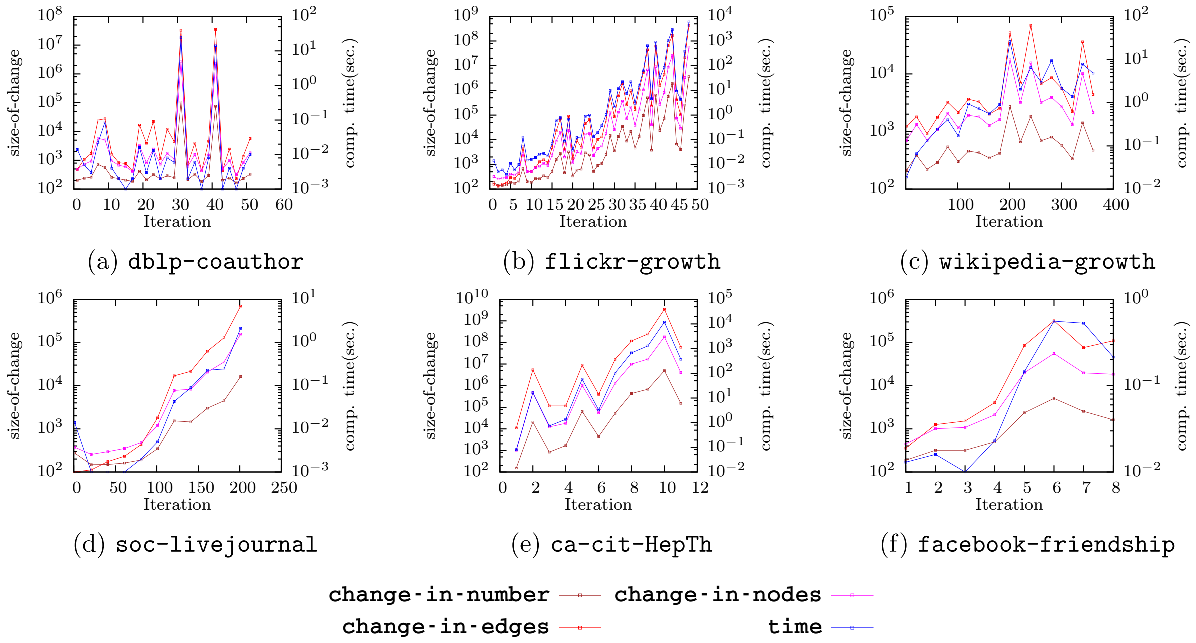


Figure 4.13: Performance of `IMCE` when the density of the graph changes over time.

are especially noticeable for graphs `flickr-growth`, `soc-livejournal`, `ca-cit-HepTh`. Sometimes, the computation time is lower in the denser state of the graph. This is because, (1) the changes in the set of maximal cliques due to the insertion of a batch are smaller than the others and (2) the computation time is dominated by the size of the changes in the set of maximal cliques.

Impact of graph density on decremental case: Table 4.4 and Table 4.5 shows the computation time of the algorithm `IMCED` when the density of the initial graph changes. For doing this, at each iteration, we remove the edges from the original graph for decreasing the density of the graph and then perform the decremental computation on deleting a batch of next 100 edges. For example, for `dblp-coauthor` graph, at first iteration we remove 1 million initial edges and on the rest of the graph we perform the decremental computation, and in second iteration, we remove 2 million initial edges and then perform the decremental computation. Similarly, we remove multiple of 500 edges from the original graph `RMAT-100-4000` in each iteration. We observe that for `RMAT-100-4000` graph, the decrease in computation time is noticeable when the density of the initial graph changes

Table 4.4: Decremental computation time (in sec.) of different algorithms upon changing density of `RMAT-100-4000` by deleting edges in reverse order (of the stream for incremental computation) starting from the original graph. The reported computation time is for deleting a batch of next 100 edges from initial graph (at different density).

Initial edges	Initial density	`IMCED`	`STIXD`	`MCMED`
3.5K	0.7	243.8	1496	$>$ hour
3K	0.6	4.2	15	94
2.5K	0.5	0.3	1.2	5
2K	0.4	0.06	0.6	0.6
1K	0.2	0.003	0.5	0.05

Table 4.5: Decremental computation time (in sec.) of different algorithms upon changing density of `dblp-coauthor` by deleting edges in reverse order (of the stream for incremental computation) starting from the original graph. The reported computation time is for deleting a batch of next 100 edges from initial graph (at different density).

Initial edges	Initial density	`IMCED`	`STIXD`	`MCMED`
4,179,996	5×10^{-6}	0.001	7.8	26.5
3,179,996	3.9×10^{-6}	0.001	6.7	21.2
2,179,996	2.7×10^{-6}	0.002	5.7	20.4
1,179,996	1.4×10^{-6}	0.005	4.9	15.8
179,996	2.2×10^{-7}	0.002	4.1	11.4

whereas, for `dblp-coauthor` graph, no such trend is observable. This is because, `RMAT-100-4000` graph is very dense (with density more than 0.8) whereas, `dblp-coauthor` graph is very sparse (with density 6.3×10^{-6}). Therefore, the change in the density due to the deletion of edges is not significant in `dblp-coauthor` as in `RMAT-100-4000`.

Table 4.6: Cumulative computation time for adding the same set of edges once in incremental computation and then in decremental computation. The initial state of each graph for the incremental computation is the final state for the same graph in the decremental computation and vice versa. Batch size is 1000 for all graphs except RMAT, where batch size is 100.

Dataset	`IMCE`	`IMCED`
`dblp-coauthor`	1113	7219
`wikipedia-growth`	224	7627
`facebook-friendship`	101	342
`RMAT-100-4000`	564	4274

Incremental vs. decremental computation: Table 4.6 shows the comparison of `IMCE` and `IMCED` on `dblp-coauthor, facebook-friendship, wikipedia-growth, and RMAT-100-4000`. In this study, we started the incremental computation from the empty graph, and then starting from the point where we stopped the incremental computation, we started decremental computation with reverse order of the edges of the incremental stream. We observe that the overall computation time for decremental computation is higher than the incremental computation on the same set of edges. This is because, the computation cost for generating subsumed cliques is lower in the incremental computation than the computation cost of generating new cliques (that are subsumed in the incremental computation) in the decremental computation as in the decremental computation, we directly check for maximality instead of containment check by presenting the set of maximal cliques of the graph before update as in the incremental computation.

Table 4.7: Cumulative time (in sec.) for enumerating new and subsumed cliques. The number of batches is shown in parentheses. Batch size $\rho = 100$ except `ca-cit-HepTh`, where $\rho = 10$ edges.

Dataset	STIX	OV	MCMEI	IMCE
`dblp-coauthor` (499)	3870	295	7212	**0.6**
`flickr-growth` (288)	3911	306	7214	**0.2**
`wikipedia-growth` (305)	4258	309	7216	**0.3**
`soc-livejournal` (156)	4077	299	7234	0.1
`ca-cit-HepTh` (310)	7291	17	11	1
`facebook-friendship` (1895)	7776	193	6853	1

Change-sensitiveness: The change in the computation time as a function of the size of change can be seen in Figure 4.7, Figure 4.13, Figure 4.10 for incremental computation, in Figure 4.11 for decremental computation, and in Figure 4.12 for fully dynamic computation.

We observe that the computation time is almost proportional to the size of change. Note that the metrics change-in-nodes and change-in-edges better capture the notion of change-sensitiveness than the metric change-in-number because, the actual cost of computation depends on the neighborhood structure of the vertices.

As we have discussed in Section 4.4 that the fully dynamic case might not become change-sensitive because there might be many intermediate cliques computed that are not in the final output, we tried to reproduce this case on `RMAT-100K-10M` graph by creating edge stream grouped by the source vertex. Indeed, we observed that there are many intermediate cliques generated as a result of the change in the graph that are not in the final output, but this size (of intermediate cliques that are not in the final output) is much smaller than the actual changes that are in the final output. Therefore, this wasteful computation time is dominated by the time for computing the actual change and thus we see the change-sensitive behavior in this case as well.

Memory consumption: Figure 4.15 shows the main memory used by `IMCE`. For this experiment, we consider two different versions of the algorithm – one with storing the clique set explicitly, and one with only storing the hashes of the cliques. As expected, the use of a hash

Table 4.8: Cumulative computation time (in sec.) of IMCE with different batch sizes. Note that Δ is the maximum degree of the graph before update. Numbers in the parenthesis indicates the total number of edges inserted incrementally.

Dataset	$\rho = 1$	$\rho = 10$	$\rho = 100$	$\rho = 1000$	$\rho = 3\log_2 \Delta$
`dblp-coauthor` (5,179,996)	1659	1335	1198	1289	1252
`flickr-growth` (3649×10^3)	7028	6784	6465	7159	6062
`wikipedia-growth` ($28{,}798 \times 10^3$)	6973	6567	7160	6995	6513
`soc-livejournal` ($17{,}633 \times 10^3$)	6892	6876	7176	7095	6871
`ca-cit-HepTh` (19,000)	29	24	1076	3728	22
`facebook-friendship` (817,035)	100	97	97	94	96

function reduces the memory consumption considerably. The difference in memory consumption between the two versions is especially visible in graphs `flickr-growth`, `wikipedia-growth`, `soc-livejournal` and `facebook-friendship`, where the sizes of the maximal cliques are considerably larger. We used the 64-bit `murmur`[4] hash function on the canonical string representation of a clique, for computing the hash signature. Note that there are some "spikes" in the plot for `dblp-coauthor`, where the memory consumption suddenly increased. On this graph, we observed that the number of maximal cliques at the point corresponding to the spike in memory usage also increased suddenly and then subsequently decreased. We do not show the memory consumption for `ca-cit-HepTh` because the number of maximal cliques are small compared to the other graph till the state we executed the incremental computation for this graph, and therefore, the difference in memory consumption with storing the maximal cliques and with storing the hashes of the maximal cliques is not noticeable (less than 1 MB).

Cumulative computation time vs. batch size: We also studied the effect of the batch size (ρ) on the cumulative computation time of IMCE, while keeping the total number of edges added the same. For example a total of 10,000 edges would lead to 1000 batches if we used a batch size of 10, and 100 batches if we used a batch size of 100. Table 4.8 shows the results for different batch sizes. There is no observable trend found by varying the batch size for all the input graphs except for

[4]https://sites.google.com/site/murmurhash/

Table 4.9: Incremental computation time (in sec.) of different algorithm upon changing the density of `dblp-coauthor` at each computation with batch size 100.

Initial edges	Initial density	`IMCE`	`STIX`	`OV`	`MCMEI`
1M	1.2×10^{-6}	$<$ 1 ms.	16	1.6	16.2
2M	2.4×10^{-6}	$<$ 1 ms.	20.8	0.9	21.3
3M	3.6×10^{-6}	0.02	376.2	4.8	28.8
4M	4.9×10^{-6}	0.002	26.2	1.6	33
5M	6×10^{-6}	0.003	42.4	2.4	33.8

Table 4.10: Incremental computation time (in sec.) as a function of the density of `RMAT-100-4000` using batch size 100.

Initial edges	Initial density	`IMCE`	`STIX`	`OV`	`MCMEI`
1K	0.2	0.005	1.4	0.01	0.05
2K	0.4	0.04	544	0.07	1
2.5K	0.5	0.2	$>$ 1 hour	0.3	10
3K	0.6	2.6	$>$ 1 hour	3	220.8
3.5K	0.7	146	$>$ 1 hour	129	$>$ 1 hour

ca-cit-HepTh. For this graph, we found that with the increase in batch size from 10 to 100, the number of subsumed clique candidates that are not actually subsumed, increases quite a lot. This affects the overall computation time. The possible reason is that the density of other graphs are smaller, the sizes of new cliques for those graphs are smaller, and the search for subsumed cliques is short, even for large batch size. From this observation, it seems, for dense graph, it is good to use small batch size to reduce the redundant computation in the subsumed clique computation such as in the case of ca-cit-HepTh graph.

Comparison with Baseline: We compare with algorithm Naive (NaiveD), which recomputes the set of maximal cliques each time there is an addition (deletion) of edges and show the results in Table 4.11 (Table 4.12). As expected, both IMCE and IMCED significantly outperform Naive and NaiveD when the number of edges inserted/deleted in a batch is small (less than 1 million for Naive and less than 10^5, for NaiveD, in our experiments). When the number of edges is larger than 1 million, Naive outperforms IMCE. This is not surprising, since as the number of edges in a batch increases, the size of the change also increases, and there is lesser benefit in using an incremental algorithm.

However, Naive and NaiveD have an additional problem related to memory consumption, since they have to store the set of maximal cliques in order to compute the symmetric difference. For instance, NaiveD cannot execute even for a single edge deletion for wikipedia-growth, due to insufficient memory. On flickr-growth and ca-cit-HepTh, NaiveD also comes to a near standstill since it tries to store the set of maximal cliques in memory. Since IMCE and IMCED do not store the set of all maximal cliques, they do not run into similar issues of memory consumption.

Comparison with prior works: We also compare the computation time of IMCE with prior works as shown in Table 4.7. Clearly, IMCE is many orders of magnitude (more than 1000) faster than prior algorithms for most of the input graphs except for ca-cit-HepTh and facebook-friendship because, (1) these graphs are of small sizes compared to the other graphs and (2) the number of maximal cliques at the initial states of these graphs are small compared to the other graphs. One reason why IMCE is so much faster than prior works is that IMCE systematically selects a local subgraph of the entire graph to search for new and subsumed maximal cliques. This reduces the computation effort considerably. OV tried to achieve such a local computation but OV is not provably change-sensitive for new maximal cliques, and its computation of subsumed cliques is expensive since the algorithm iterates over the entire set of maximal cliques for deriving subsumed cliques. A similar strategy of iterating over the entire set of maximal cliques for deriving maximal clique set of the updated graph as in MCMEI makes the algorithm less efficient. Next we compare IMCE with STIX, *ov*, and MCMEI upon changing (increasing) the density of the input graphs and we present the results in Table 4.9 and Table 4.10 and compare IMCED with STIXD and MCMED upon changing (decreasing) the density of the input graph and we present the results in Table 4.5 and Table 4.4. Note that we cannot compare on other larger graphs because all of the prior works require the set of maximal cliques of the initial graph to start the computation and the number of maximal cliques at different states of graph are so large that they cannot fit in the main memory. We observe that both IMCE and IMCED are magnitude of order faster than the prior works which is as expected except that for RMAT-100-4000 where the performance of OV is similar to that of IMCE. This is because the graph RMAT-100-4000 is small and the number of maximal cliques of this graph is also small compared to the other graphs.

Table 4.11: Comparison of incremental computation time (sec.) of IMCE and Naive for adding a single batch with different batch sizes starting from a graph with 1 million initial edges. ρ indicates the batch size.

DataSet	$\rho = 100$		$\rho = 1000$		$\rho = 10000$		$\rho = 100000$		$\rho = 1000000$	
	Naive	IMCE	Naive	IMCE	Naive	IMCE	Naive	IMCE	Naive	IMCE
dblp-coauthor	4.5	0.001	4.6	0.009	4.8	0.07	5	1.5	9.5	67.3
flickr-growth	10	0.002	10	0.03	9.6	0.4	11	3.4	50.9	127
wikipedia-growth	8.7	0.003	8.5	0.02	9.4	0.2	9.7	2.4	21.9	25.9

Summary of results: To summarize the results of our experiments, we note the following: (1) IMCE and IMCED are change-sensitive: the computation time to enumerate the change in the set of maximal cliques is proportional to the magnitude of the change in the set of maximal cliques.

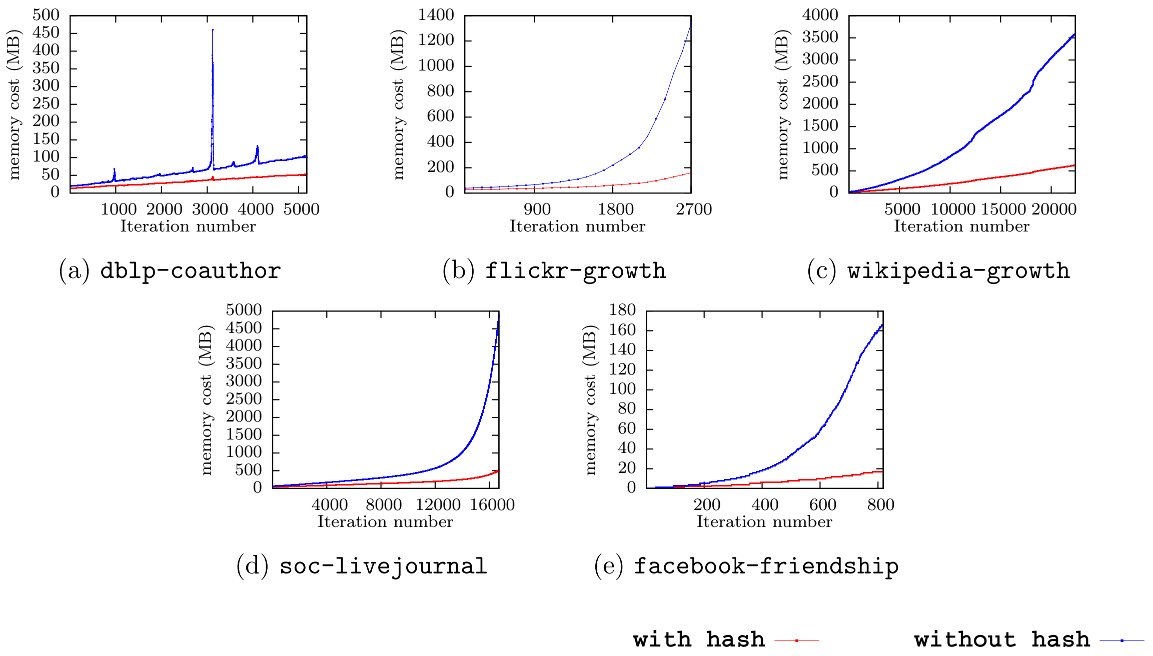


Figure 4.15: Memory cost of `IMCE` with and without using hash function ($\rho = 1000$).

Table 4.12: Comparison of decremental computation time (sec.) of `IMCED` and `NaiveD` for deleting a single batch with different batch sizes starting from the original graph. ρ indicates the batch size.

DataSet	$\rho = 100$		$\rho = 1000$		$\rho = 10000$		$\rho = 100000$	
	`NaiveD`	`IMCED`	`NaiveD`	`IMCED`	`NaiveD`	`IMCED`	`NaiveD`	`IMCED`
`dblp-coauthor`	22.6	0.02	22.3	0.2	22.6	6.4	25.3	> 720
`facebook-friendship`	19.4	0.3	19.5	0.6	19.2	4.8	14.9	192

(2) `IMCE` and `IMCED` are two to three orders of magnitude faster than prior algorithms. (3) The computation time for the maintenance increases when the density of the graph increases. (4) the use of hash signatures for storing maximal cliques greatly reduces the memory consumption.

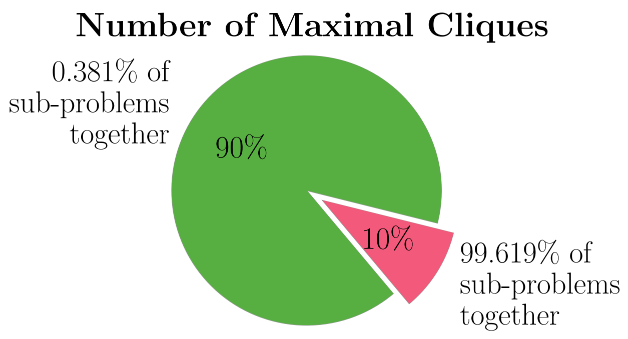
CHAPTER 5. PARALLEL MAXIMAL CLIQUE ENUMERATION ON STATIC AND DYNAMIC GRAPHS

5.1 Introduction

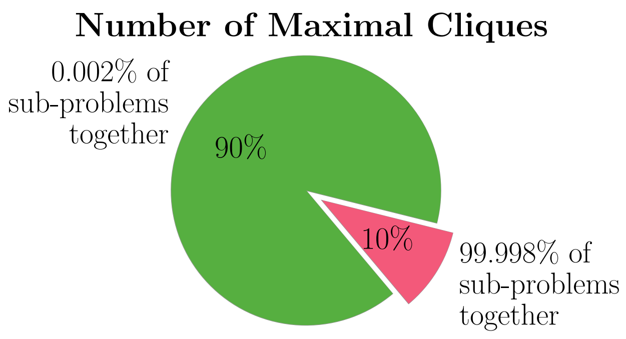
Sequential approaches to MCE can lead to high runtimes on large graphs. Based on our experiments, a real-world network `Orkut` with approximately 3 million vertices, 117 million edges requires approximately 8 hours to enumerate all maximal cliques using an efficient sequential algorithm due to Tomita et al. [143]. Graphs that are larger and/or more complex cannot be handled by sequential algorithms with a reasonable turnaround time, and the high computational complexity of MCE calls for parallel methods.

In this work, we consider shared-memory parallel methods for MCE. In the shared-memory model, the input graph can reside within globally shared memory, and multiple threads can work in parallel on enumerating maximal cliques. Shared-memory parallelism is attractive today since machines with tens to hundreds of cores and hundreds of Gigabytes of shared-memory are readily available. The advantage of using shared-memory approach over a distributed memory approach are: (1) Unlike distributed memory, it is not necessary to divide the graph into subgraphs and communicate the subgraphs among processors. In shared-memory, different threads can work concurrently on a single shared copy of the graph (2) Sub-problems generated during MCE are often highly imbalanced, and it is hard to predict which sub-problems are small and which are large, while initially dividing the problem into sub-problems. With a shared-memory method, it is possible to further subdivide sub-problems and process them in parallel. With a distributed memory method, handling such imbalances in sub-problem sizes requires greater coordination and is more complex.

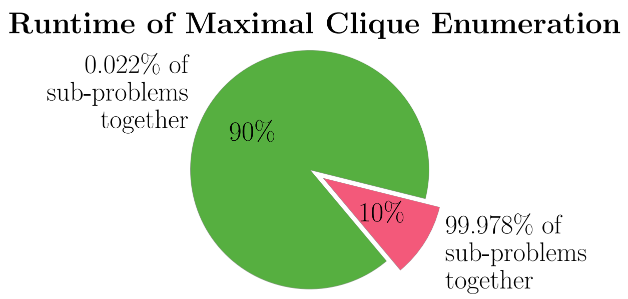
To show how imbalanced the sub-problems can be, in Fig. 5.1, we show data for two real-world networks `As-Skitter` and `Wiki-Talk`. These two networks have millions of edges and tens of millions



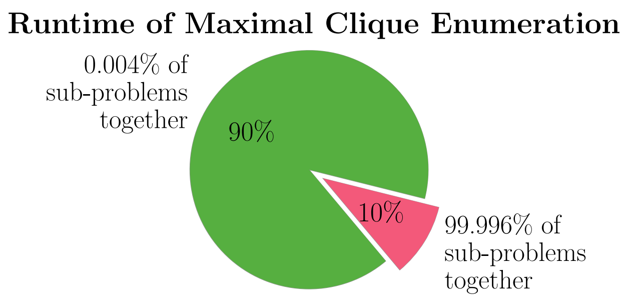
(a) `As-Skitter`



(b) `Wiki-Talk`



(c) `As-Skitter`



(d) `Wiki-Talk`

Figure 5.1: Imbalanced in sizes of sub-problems for MCE, where each sub-problem corresponds to the maximal cliques of a single vertex in the given graph. (a) `As-Skitter`: 0.3% of sub-problems form 90% of total number of maximal cliques. (b) `Wiki-Talk`: only 0.002% of sub-problems yield 90% of all maximal cliques. (c) `As-Skitter`: 0.02% of sub-problems take 90% of total runtime of MCE. (d) `Wiki-Talk`: only 0.004% of sub-problems take 90% of total runtime of MCE

of maximal cliques (for statistics on these networks, see Section 5.5). Consider a division of the MCE problem into per-vertex sub-problems, where each sub-problem corresponds to the set of all maximal cliques containing a single vertex in a network, and suppose these sub-problems were solved independently, while taking care to prune out search for the same maximal clique multiple times. For `As-Skitter`, we observed that 90% of total runtime required for MCE is taken by only 0.022% of the sub-problems, and less than 0.4% of all sub-problems yield 90% of all maximal cliques. Even larger skew in sub-problem sizes is observed in the `Wiki-Talk` graph. This data demonstrates that load balancing is a central issue for parallel MCE.

Prior works on parallel MCE have largely focused on distributed memory algorithms [151, 129, 95, 138]. There are a few works on shared-memory parallel algorithms [158, 45, 87]. However, these algorithms do not scale to larger graphs due to memory or computational bottlenecks – either the algorithms miss out significant pruning opportunities as in [45] or they need to generate a large number of non-maximal cliques as in [158, 87].

Roadmap. The rest of the sections are organized as follows. We present preliminaries in Section 5.2, followed by a description of algorithms for a static graph in Section 5.3, algorithms for a dynamic graph in Section 5.4, and experimental evaluation in Section 5.5.

5.2 Preliminaries

We consider a simple undirected graph without self loops or multiple edges. For graph G, let $V(G)$ denote the set of vertices in G and $E(G)$ denote the set of edges in G. Let n denote the size of $V(G)$, and m denote the size of $E(G)$. For vertex $u \in V(G)$, let $\Gamma_G(u)$ denote the set of vertices adjacent to u in G. When the graph G is clear from the context, we use $\Gamma(u)$ to mean $\Gamma_G(u)$. Let $\mathcal{C}(G)$ denote the set of all maximal cliques in G.

Parallel Cost Model: For analyzing our shared-memory parallel algorithms, we use the CRCW PRAM model [18], which is a model of shared parallel computation that assumes concurrent reads and concurrent writes. Our parallel algorithm can also work in other related models of shared-memory such as EREW PRAM (exclusive reads and exclusive writes), with a logarithmic factor increase in work as well as parallel depth. We measure the effectiveness of the parallel algorithm using the *work-depth* model [18]. Here, the "work" of a parallel algorithm is equal to the total number of operations of the parallel algorithm, and the "depth" (also called the "parallel time" or the "span") is the longest chain of dependent computations in the algorithm. A parallel algorithm is said to be *work-efficient* if its total work is of the same order as the work due to the best sequential algorithm[1]. We aim for work-efficient algorithms with a low depth, ideally poly-logarithmic in the

[1]Note that work-efficiency in the CRCW PRAM model does not imply work-efficiency in the EREW PRAM model

size of the input. Using Brent's theorem [18], it can be seen that a parallel algorithm on input size n with a depth of d can theoretically achieve $\Theta(p)$ speedup on p processors as long as $p = O(n/d)$.

We next restate a result on concurrent hash tables [130] that we use in proving the work and depth bounds of our parallel algorithms.

Theorem 5 (Theorem 3.15 [130])**.** *There is an implementation of a hash table, which, given a hash function with expected uniform distribution, performs n_1 insert, n_2 delete and n_3 find operations in parallel using $O(n_1 + n_2 + n_3)$ work and $O(1)$ depth on average.*

5.3 Parallel MCE Algorithms on a Static Graph

In this section, we present new shared-memory parallel algorithms for MCE. We first describe a parallel algorithm `ParTTT`, a parallelization of the sequential `TTT` algorithm and an analysis of its theoretical properties. Then, we discuss bottlenecks in `ParTTT` that arise in practice, leading us to another algorithm `ParMCE` with better practical performance. `ParMCE` uses `ParTTT` as a subroutine – it creates appropriate sub-problems that can be solved in parallel, and hands off the enumeration task to `ParTTT`.

5.3.1 Algorithm `ParTTT`

Our first algorithm `ParTTT` is a work-efficient parallelization of the sequential `TTT` algorithm. The two main components of `TTT` (Algorithm 1) are (1) Selection of the pivot element (Line 3) and (2) Sequential backtracking for extending candidate cliques until all maximal cliques are explored (Line 5 to Line 11). We discuss how to parallelize each of these steps.

Parallel Pivot Selection: Within a single recursive call of `ParTTT`, the pivot element is computed in parallel using two steps, as described in `ParPivot` (Algorithm 7). In the first step, the size of the intersection $\texttt{cand} \cap \Gamma(u)$ is computed in parallel for each vertex $u \in \texttt{cand} \cup \texttt{fini}$. In the second step, the vertex with the maximum intersection size is selected. The parallel algorithm for selecting a pivot is presented in Algorithm 7. The following lemma proves that the parallel pivot selection is work-efficient with logarithmic depth:

Lemma 16. *The total work of* `ParPivot` *is* $O(\sum_{w \in \texttt{cand} \cup \texttt{fini}}(\min\{|\texttt{cand}|, |\Gamma(w)|\}))$, *which is* $O(n^2)$, *and depth is* $O(\log n)$.

Proof. If the sets `cand` and $\Gamma(w)$ are stored as hashsets, then for vertex w the size $t_w = |\texttt{intersect}(\texttt{cand}, \Gamma(w))|$ can be computed sequentially in time $O(\min\{|\texttt{cand}|, |\Gamma(w)|\})$ – the intersection of two sets S_1 and S_2 can be found by considering the smaller set among the two, say S_2, and searching for its elements within the larger set, say S_1. It is possible to parallelize the computation of $\texttt{intersect}(S_1, S_2)$ by executing the searches elements in S_2 in parallel, followed by counting the number of elements that lie in the intersection, which can also be done in parallel in a work-efficient manner using $O(1)$ depth using Theorem 5. Since computing the maximum of a set of n numbers can be accomplished using work $O(n)$ and depth $O(\log n)$, for vertex w, t_w can be computed using work $O(\min\{|\texttt{cand}|, |\Gamma(w)|\})$ and depth $O(\log n)$. Once the different t_w are computed, $argmax(\{t_w : w \in \texttt{cand} \cup \texttt{fini}\})$ can be computed using additional work $|\texttt{cand} \cup \texttt{fini}|$ and depth $O(\log n)$. Hence, the total work of `ParPivot` is $O(\sum_{w \in \texttt{cand} \cup \texttt{fini}}(\min\{|\texttt{cand}|, |\Gamma(w)|\})$. Since the size of `cand`, `fini`, and $\Gamma(w)$ are bounded by n, this is $O(n^2)$, but typically much smaller. □

```
Algorithm 7: ParPivot($G, K, \texttt{cand}, \texttt{fini}$)
  Input: $G$: Input graph; $K$: A clique in $G$ that may be further extended; cand: A
         set of vertices that may extend $K$; fini: A set of vertices that have been
         used to extend $K$.
  Output: pivot vertex $v \in \texttt{cand} \cup \texttt{fini}$.
1 for $w \in \texttt{cand} \cup \texttt{fini}$ do in parallel
2     In parallel, compute $t_w \leftarrow |\texttt{intersect}(\texttt{cand}, \Gamma_G(w))|$
3 In parallel, find $v \leftarrow argmax(\{t_w : w \in \texttt{cand} \cup \texttt{fini}\})$
4 return $v$
```

Parallelization of Backtracking: We first note that there is a sequential dependency among the different iterations within a recursive call of `TTT`. In particular, the contents of the sets `cand` and `fini` in a given iteration are derived from the contents of `cand` and `fini` in the previous iteration. Such sequential dependence of updates of `cand` and `fini` restricts us from calling the recursive `TTT` for different vertices of `ext` in parallel. To remove this dependency, we adopt a different view of `TTT` which enables us to make the recursive calls in parallel. The elements of `ext`, the vertices to be considered for extending a maximal clique, are arranged in a predefined total order. Then, we unroll the loop and explicitly compute the parameters `cand` and `fini` for recursive calls.

Suppose $\langle v_1, v_2, ..., v_\kappa \rangle$ is the order of vertices in `ext` to be processed. Each vertex $v_i \in$ `ext`, once added to K, should be removed from further consideration from `cand`. To ensure this, in `ParTTT`, we explicitly remove vertices $v_1, v_2, ..., v_{i-1}$ from `cand` and add them to `fini`, before making the recursive calls. As a consequence, parameters of the ith iteration are computed independently of prior iterations.

Algorithm 8: `ParTTT`$(\mathcal{G}, K, \texttt{cand}, \texttt{fini})$

```
Input: $\mathcal{G}$ - The input graph
$K$ - a non-maximal clique to extend
cand - Set of vertices that may extend $K$
fini - vertices that have been used to extend $K$
Output: Set of all maximal cliques of $G$ containing $K$ and vertices from cand but
        not containing any vertex from fini
1  if (cand = $\emptyset$) & (fini = $\emptyset$) then
2      Output $K$ and return
3  pivot ← ParPivot($\mathcal{G}$, cand, fini)
4  ext[$1..\kappa$] ← cand − $\Gamma_{\mathcal{G}}$(pivot) // in parallel
5  for $i \in [1..\kappa]$ do in parallel
6      $q$ ← ext[$i$]
7      $K_q \leftarrow K \cup \{q\}$
8      cand$_q$ ← intersect(cand \ ext[$1..i-1$], $\Gamma_{\mathcal{G}}(q)$)
9      fini$_q$ ← intersect(fini ∪ ext[$1..i-1$], $\Gamma_{\mathcal{G}}(q)$)
10     ParTTT($\mathcal{G}$, $K_q$, cand$_q$, fini$_q$)
```

Now we prove the work efficiency and low depth of `ParTTT` in the following lemma:

Lemma 17. *Total work of* `ParTTT` *(Algorithm 8) is $O(3^{n/3})$ and depth is $O(M \log n)$ where n is the number of vertices in the graph and M is the size of a maximum clique in G.*

Proof. First, we analyze the total work. Note that the computational tasks in `ParTTT` is different from `TTT` at Line 8 and Line 9 of `ParTTT` where at an iteration i, we remove all vertices $\{v_1, v_2, ..., v_{i-1}\}$ from `cand` and add all these vertices to `fini` as opposed to the removal of a single vertex v_{i-1} from `cand` and addition of that vertex to `fini` as in `TTT` (Line 8 and Line 9 of Algorithm 1). Therefore, in `ParTTT`, additional $O(n)$ work is required due to independent computations of $\texttt{cand}_q$ and $\texttt{fini}_q$. The total work, excluding the call to `ParPivot` is $O(n^2)$. Adding up the work of `ParPivot`, which requires $O(n^2)$ work, requires $O(n^2)$ total work for each single call of `ParTTT` excluding further recursive calls (Algorithm 8, Line 10), which is same as in original sequential algorithm `TTT` (Section 4, [143]). Hence, using Lemma 2 and Theorem 3 of [143], we infer that the total work of `ParTTT` is the same as the sequential algorithm `TTT` and is bounded by $O(3^{n/3})$.

Next we analyze the depth of the algorithm. The depth of `ParTTT` consists of the (sum of the) following components: (1) Depth of `ParPivot`, (2) Depth of computation of `ext`, (3) Maximum depth of an iteration in the for loop from Line 5 to Line 10. According to Lemma 16, the depth of `ParPivot` is $O(\log n)$. The depth of computing `ext` is $O(\log n)$ because it takes $O(1)$ time to check whether an element in `cand` is in the neighborhood of `pivot` by doing a set membership check on the set of vertices that are adjacent to `pivot`. Similarly, the depth of computing $\texttt{cand}_q$ and $\texttt{fini}_q$ at Line 8 and Line 9 are $O(\log n)$ each. The remaining is the depth of the call of `ParTTT` at Line 10. Observe that the recursive call of `ParTTT` continues until there is no further vertex to add for expanding K, and this depth can be at most the size of the maximum clique which is M because, at each recursive call of `ParTTT` the size of K is increased by 1. Thus, the overall depth of `ParTTT` is $O(M \log n)$. □

Corollary 1. *Using P parallel processors that shared-memory,* `ParTTT` *(Algorithm 8) is a parallel algorithm for MCE, and can achieve a worst case parallel time of $O\left(\frac{3^{n/3}}{M \log n} + P\right)$ using P parallel processors. This is work-efficient as long as $P = O(\frac{3^{n/3}}{M \log n})$, and also work-optimal.*

Proof. The parallel time follows from using Brent's theorem [18], which states that the parallel time using P processors is $O(w/d+P)$, where w and d are the work and the depth of the algorithm respectively. If the number of processors $P = O\left(\frac{3^{n/3}}{M \log n}\right)$, then using Lemma 17 the parallel time is $O\left(\max\{\frac{3^{n/3}}{P}, M \log n\}\right) = O\left(\frac{3^{n/3}}{P}\right)$. The total work across all processors is $O(3^{n/3})$, which is worst-case optimal, since the size of the output can be as large as $3^{n/3}$ maximal cliques (Moon and Moser [104]). □

5.3.2 Algorithm ParMCE

While ParTTT is a theoretically work-efficient parallel algorithm, we note that its runtime can be further improved. While the worst case work complexity of ParPivot matches that of the pivoting routine in TTT, in practice, the work in ParTTT can be higher, since computation of $\texttt{cand}_q$ and $\texttt{fini}_q$ has additional overhead that increases as the size of the $\texttt{cand}_q$ and $\texttt{fini}_q$ lists increase. This can result in a lower speedup than the theoretically expected one.

We set out to improve on this to derive a more efficient parallel implementation through a more selective use of ParPivot in that the cost of pivoting can be reduced by carefully choosing many pivots in parallel instead of a single pivot element as in ParTTT at the beginning of the algorithm. We first note that the cost of ParPivot is the highest during the iteration when the parameter K (clique so far) is empty. During this iteration, the set of vertices still to be considered, $\texttt{cand} \cup \texttt{fini}$, can be high, as large as the number of vertices in the graph. To improve upon this, we can perform the first few steps of pivoting, when K is empty, using a sequential algorithm. Once the set K has at least one element in it, the number of the vertices in $\texttt{cand} \cup \texttt{fini}$ still to be considered, drops down to no more than the size of the intersection of neighborhoods of all vertices in K, which is typically a number much smaller than the number of vertices in the graph (it is smaller than the smallest degree of a vertex in K). Problem instances with K set to a single vertex can be seen as sub-problems and on each of these sub-problems, the overhead of ParPivot is much smaller since the number of vertices that have to be dealt with is also much smaller.

Based on this observation, we present a parallel algorithm `ParMCE` that works as follows. The algorithm can be viewed as considering for each vertex $v \in V(G)$, a subgraph G_v that is induced by the vertex v and its neighborhood $\Gamma_G(v)$. It enumerates all maximal cliques from each subgraph G_v in parallel using `ParTTT`. While processing sub-problem G_v, it is important to not enumerate maximal cliques that are being enumerated elsewhere, in other sub-problems. To handle this, the algorithm considers a specific ordering of all vertices in V such that v is the least ranked vertex in each maximal clique enumerated from G_v. The subgraphs G_v for each vertex v are handled in parallel – these subgraphs need not be processed in any particular order. However, the ordering allows us to populate the `cand` and `fini` sets accordingly, so that each maximal clique is enumerated in exactly one sub-problem. The order in which the vertices are considered is defined by a "rank" function **rank**, which indicates the position of a vertex in the total order. The specific ordering that is used influences the total work of the algorithm, as well as the load balance of the parallel implementation.

Load Balancing: Observe that the sizes of the subgraphs G_v may vary widely because of two reasons: (1) the subgraphs themselves may be of different sizes, depending on the vertex degrees, and (2) the number of maximal cliques and the sizes of the maximal cliques containing v can vary widely from one vertex to another. Clearly, the sub-problems that deal with a large number of maximal cliques or maximal cliques of a large size are more expensive than others.

In order to maintain the size of the sub-problems approximately balanced, we use an idea from PECO [138], where we choose the rank function on the vertices in such a way that for any two vertices v and w, $\textbf{rank}(v) > \textbf{rank}(w)$ if the complexity of enumerating maximal cliques from G_v is higher than the complexity of enumerating maximal cliques from G_w. By giving a higher rank to v than w, we are decreasing the complexity of the sub-problem G_v, since the sub-problem at G_v need not enumerate maximal cliques that involve any vertex whose rank is less than v. Hence, the higher the rank of vertex v, the lower is its "share" (of maximal cliques it belongs to) of maximal cliques in G_v. We use this idea for approximately balancing the workload across sub-problems. The additional enhancements in `ParMCE`, when compared with the idea from PECO are as follows: (1) In

PECO the algorithm is designed for distributed memory so that the subgraphs and sub-problems have to be explicitly copied across the network, and (2) In ParMCE, the vertex specific sub-problem, dealing with G_v is itself handled through a parallel algorithm, ParTTT. However, in PECO, the sub-problem for each vertex was handled through a sequential algorithm.

Note that it is computationally expensive to accurately count the number of maximal cliques within G_v, and hence it is not possible to compute the rank of each vertex exactly according to the complexity of handling G_v. Instead, we estimate the complexity of handling G_v using some easy-to-evaluate metrics on the subgraphs. In particular, we consider the following:

- **Degree Based Ranking:** For vertex v, define $\texttt{rank}(v) = (d(v), id(v))$ where $d(v)$ and $id(v)$ are degree and identifier of v respectively. For two vertices v and w, $\texttt{rank}(v) > \texttt{rank}(w)$ if $d(v) > d(w)$ or $d(v) = d(w)$ and $id(v) > id(w)$; $rank(v) < rank(w)$ otherwise.

- **Triangle Count Based Ranking:** For vertex v, define $\texttt{rank}(v) = (t(v), id(v))$ where $t(v)$ is the number of triangles containing vertex v. This is more expensive to compute than degree based ranking, but may yield a better estimate of the complexity of maximal cliques within G_v.

- **Degeneracy Based Ranking** [49]**:** For a vertex v, define $\texttt{rank}(v) = (degen(v), id(v))$ where $degen(v)$ is the degeneracy of a vertex v. A vertex v has degeneracy number k when it belongs to a k-core but no $(k+1)$-core where a k-core is a maximal induced subgraph with minimum degree of each vertex k in that subgraph. A computational overhead of using this ranking is due to computing the degeneracy of the vertices which takes $O(n+m)$ time where n is the number of vertices and m is the number of edges.

The different implementations of ParMCE using degree, triangle, and degeneracy rankings are called as ParMCEDegree, ParMCETri, ParMCEDegen respectively.

```
Algorithm 9: ParMCE($G$)
  Input: $G$: The input graph.
  Output: $\mathcal{C}(G)$: set of all maximal cliques of $G$.
1 for $v \in V(G)$ do in parallel
2     Create $G_v$, the subgraph of $G$ induced by $\Gamma_G(v) \cup \{v\}$
3     $K \leftarrow \{v\}$, cand $\leftarrow \phi$, fini $\leftarrow \phi$
4     for $w \in \Gamma_G(v)$ do in parallel
5         if $rank(w) > rank(v)$ then cand $\leftarrow$ cand $\cup \{w\}$
6         else fini $\leftarrow$ fini $\cup \{w\}$
7     ParTTT($G_v, K,$ cand, fini)
```

5.4 Parallel MCE Algorithm on a Dynamic Graph

When the graph changes over time due to addition of edges, the maximal cliques of the updated graph also changes. The update in the set of maximal cliques consists of (1) the set of new maximal cliques- the maximal cliques that are newly formed and (2) the set of subsumed cliques - maximal cliques of the original graph that are subsumed by the new maximal cliques. The combined set of new and subsumed maximal cliques is called the set of changes and the size of this set refers to the size of change in the set of maximal cliques.

It is important to update the set of maximal cliques in a dynamic graph when it serves as the building block in many important problems. For example, the work of Chateau et al. [27] on maintaining common intervals among genomes, the work of Duan et al. [46] on incremental k-clique clustering, the work of Hussain et al. [67] on maintaining the maximum range-sum query over a point stream use maximal clique as the building block in an appropriately defined graph.

Note that the size of change can be as small as $O(1)$ and can be as large as exponential in the size of the graph for addition of a single new edge. For example, consider a graph of size n which is missing a single edge from being a clique. The size of change is only 3 when that missing edge is added to the graph because there will be only one new maximal clique of size n and two

```
Algorithm 10: ParIMCENew($G, H$)
   Input: $G$ - input graph
   $H$ - Set of $\rho$ edges being added to $G$
   Output: Cliques in $\Lambda^{new} = \mathcal{C}(G+H) \setminus \mathcal{C}(G)$
 1 $G' \leftarrow G + H$
 2 Consider edges of $H$ in an arbitrary order $e_1, e_2, \ldots, e_\rho$
 3 for $i \leftarrow 1, 2, \ldots, \rho$ do in parallel
 4     $e \leftarrow e_i = (u, v)$
 5     $V_e \leftarrow \{u, v\} \cup \{\Gamma_{G'}(u) \cap \Gamma_{G'}(v)\}$
 6     $\mathcal{G} \leftarrow$ Graph induced by $V_e$ on $G'$
 7     $K \leftarrow \{u, v\}$
 8     cand $\leftarrow V_e \setminus \{u, v\}$ ; fini $\leftarrow \emptyset$
 9     $S \leftarrow$ ParTTTExcludeEdges($\mathcal{G}, K$, cand, fini, $\{e_1, e_2, ..., e_{i-1}\}$)
10     $\Lambda^{new} \leftarrow \Lambda^{new} \cup S$
```

subsumed cliques each of size $(n-1)$. On the other hand consider a Moon-Moser graph [104] of size n. Addition of a single edge to this graph makes the size of the changes in the order of $O(3^{n/3})$.

In a previous work, we presented a sequential algorithm IMCE [38] for solving the problem of updating the set of maximal cliques in a dynamic graph in an efficient manner in an incremental setting when new edges are added to the graph. IMCE consists of FastIMCENewClq for computing new maximal cliques and IMCESubClq for computing subsumed cliques. However, IMCE is still unable to update the set of maximal cliques when the size of change is large. For instance, it takes IMCE around 9.4 hours to add approximately 90000 edges to the Ca-Cit-HepTh graph (with original graph density 0.01) incrementally, starting from the empty graph. The high computational cost of IMCE calls for parallel methods.

In this section we present parallel algorithms for enumerating the set of new and subsumed cliques when an edge set $H = \{e_1, e_2, ..., e_\rho\}$ is added to a graph G. Our parallel algorithms are based on IMCE [38]. In this work, we focus on (1) processing new edges in parallel, (2) enumerating new maximal cliques using ParTTT, and (3) Parallelizing IMCESubClq [38]. First we describe an efficient parallel algorithm for generating new maximal cliques, i.e, the maximal cliques in $G + H$

Table 5.1: Brief description of the incremental algorithms in this work.

Objective	**Sequential Algorithm [38]**	**Parallel Algorithm (this work)**	**Overview of Parallel Algorithms**
Enumerating new maximal cliques	FastIMCENewClq	ParIMCENew	(1) Process new edges in parallel. (2) Enumerate maximal cliques using ParTTTExcludeEdges.
Enumerating subsumed cliques	IMCESubClq	ParIMCESub	(1) Generate candidates in parallel (executing inner for loop of IMCESubClq in parallel). (2) Process each candidate in parallel (executing candidate processing step of IMCESubClq in parallel).

that are not present in G and then an efficient parallel algorithm for generating subsumed maximal cliques, i.e, the cliques which are maximal in G but not maximal in $G + H$. We present a shared-memory parallel algorithm ParIMCE for the incremental maintenance of maximal cliques. ParIMCE consists of (1) algorithm ParIMCENew for enumerating new maximal cliques and (2) algorithm ParIMCESub for enumerating subsumed cliques. A brief description of the algorithms is discussed in Table 5.1.

5.4.1 Parallel Enumeration of New Maximal Cliques

Here, we present a parallel algorithm ParIMCENew for enumerating the set of new maximal cliques when a set of new edges is added. The idea is that we iterate over new edges in parallel and at each (parallel) iteration we construct a subgraph of the original graph and enumerate the set of all maximal cliques in that subgraph. We present an efficient parallel algorithm ParTTTExcludeEdges (Algorithm 11) for this enumeration. The description of ParIMCENew is presented in Algorithm 10.

Note that ParIMCENew is based upon an existing sequential algorithm FastIMCENewClq [38] for enumerating new maximal clique that uses TTTExcludeEdges (Algorithm 3) [38] that enumerates all new maximal cliques without any duplication. ParTTTExcludeEdges is based upon the duplicate avoidance technique similar to the technique used in TTTExcludeEdges and the parallelization technique similar to ParTTT. More specifically, ParTTTExcludeEdges follows a global ordering of the new edges to avoid redundancy in the enumeration process. Note that the correctness of ParIMCENew is followed by the correctness of the sequential algorithm FastIMCENewClq. Following lemma shows the work efficiency and depth of ParIMCENew:

Algorithm 11: `ParTTTExcludeEdges`$(G, K, \texttt{cand}, \texttt{fini}, \mathcal{E})$

```
Input: $G$ - The input graph
$K$ - Set of vertices forming a clique
cand - Set of vertices that may extend $K$
fini - vertices that have been used to extend $K$
$\mathcal{E}$ - set of edges to ignore
1  if (cand $= \emptyset$) & (fini $= \emptyset$) then
2      Output $K$ // K is a maximal clique
3      return
4  pivot $\leftarrow$ ParPivot($G$, cand, fini)
5  ext[$1..\kappa$] $\leftarrow$ cand $- \Gamma_G$(pivot) // in parallel
6  for $i \in [1..\kappa]$ do in parallel
7      $q \leftarrow$ ext[$i$]
8      $K_q \leftarrow K \cup \{q\}$
9      if $K_q \cap \mathcal{E} \neq \emptyset$ then
10         return
11     cand$_q \leftarrow$ intersect(cand $\setminus$ ext[$1..i-1$], $\Gamma_G(q)$)
12     fini$_q \leftarrow$ intersect(fini $\cup$ ext[$1..i-1$], $\Gamma_G(q)$)
13     ParTTTExcludeEdges($G, K_q$, cand$_q$, fini$_q$, $\mathcal{E}$)
```

Lemma 18. *Given a graph G and a new edge set H,* `ParIMCENew` *is work-efficient, i.e, the total work is of the same order of the time complexity of* `FastIMCENewClq`*. The depth of* `ParIMCENew` *is $O(\Delta^2 + M \log \Delta)$ where Δ is the maximum degree and M is the size of a maximum clique in $G + H$.*

Proof. First we prove the work efficiency of `ParIMCENew` followed by the depth of the algorithm. Note that, for proving the work-efficiency we will show that procedure at each line from Line 4 to Line 10 of `ParIMCENew` is work-efficient. Line 4 to Line 8 are work-efficient because, all these procedures are sequential in `ParIMCENew`. The parallel set operations at Line 5 and Line 10 is work-efficient using Theorem 5. Now we will show the work efficiency of `ParIMCENew` as follows. If we disregard Lines 7-10 of `TTTExcludeEdges` and Lines 9-10 of `ParTTTExcludeEdges` then the total work of `ParTTTExcludeEdges` is the same as the time complexity of `ParTTT` following the work efficiency of `ParTTT`. Next, we say that the time complexity of Lines 7-10 of `TTTExcludeEdges` is the same as the time complexity of Lines 9-10 of `ParTTTExcludeEdges` because in `TTTExcludeEdges` we use two global hashtables - one for maintaining the adjacent vertices of the currently processing vertex in the set of new edges and another for maintaining the indexes of the new edges that we define before the beginning of the enumeration of new maximal cliques. With these two hashtables, we can check the *if* condition at Line 9 of `ParTTTExcludeEdges` in parallel with total work $O(n)$ using Theorem 5 which is of the same order of the time complexity of performing *if* condition check at Line 7 of `TTTExcludeEdges`. This completes the proof of work efficiency of `ParTTTExcludeEdges`. For proving the depth of `ParIMCENew`, note that the depth is the sum of the depths of procedures at Line 5, 6, 9, 10 of `ParIMCENew` because the cost of all operations in other lines are $O(1)$ each. The depth of executing intersection in parallel at Line 5 is $O(1)$ using Theorem 5, the depth of the procedure for constructing the graph at Line 6 is $O(\Delta^2)$ as we construct the graph sequentially, the depth `ParTTTExcludeEdges` is $O(M \log \Delta)$ following the depth of `ParTTT`, and the depth of Line 10 is $O(1)$ because we can do this operation in parallel using Theorem 5. Thus, the overall depth of `ParIMCENew` follows. □

5.4.2 Parallel Enumeration of Subsumed Cliques

```
Algorithm 12: ParIMCESub($G, H, C, \Lambda^{new}$)
Input: $G$ - Input Graph
       $H$ - Edge set being added to $G$
       $C$ - Set of maximal cliques in $G$
       $\Lambda^{new}$ - set of new maximal cliques in $G + H$
Output: All cliques in $\Lambda^{del} = \mathcal{C}(G) \setminus \mathcal{C}(G + H)$
1  $\Lambda^{del} \leftarrow \emptyset$
2  for $c \in \Lambda^{new}$ do in parallel
3      $S \leftarrow \{c\}$
4      for $e = (u, v) \in E(c) \cap H$ do
5          $S' \leftarrow \phi$
6          for $c' \in S$ do in parallel
7              if $e \in E(c')$ then
8                  $c_1 = c' \setminus \{u\}$ ; $c_2 = c' \setminus \{v\}$
9                  $S' \leftarrow S' \cup c_1$ ; $S' \leftarrow S' \cup c_2$
10             else
11                 $S' \leftarrow S' \cup c'$
12         $S \leftarrow S'$
13     for $c' \in S$ do in parallel
14         if $c' \in C$ then
15             $\Lambda^{del} \leftarrow \Lambda^{del} \cup c'$
16             $C \leftarrow C \setminus c'$
```

In this section we present a parallel algorithm ParIMCESub based on the sequential algorithm IMCESubClq [38] for the enumeration of subsumed cliques. In ParIMCESub we perform parallelization in doing the following : (1) removing a single new edges from all the candidates in parallel and (2) checking for the candidacy of the subsumed cliques in parallel. We present ParIMCESub in Algorithm 12. In the following lemma we will show the work efficiency and depth of ParIMCESub:

Lemma 19. *Given a graph G and a new edge set H,* ParIMCESub *is work-efficient; the total work is of the same order of the time complexity of* IMCESubClq*. The depth of* ParIMCENew *is*

$O(min\{M^2, \rho\})$ for processing each new maximal clique where M is the size of a maximum clique in $G + H$ and ρ is the size of H.

Proof. First note that the procedure of `ParIMCESub` is exactly same as the procedure of `IMCESubClq` except for the parallel loops at Line 6 and Line 13 of `ParIMCESub` whereas these loops are sequential in `IMCESubClq`. As all the computations in `ParIMCESub` is exactly same as the computations in `IMCESubClq` except for the loop parallelization, `ParIMCESub` is work-efficient.

For proving the parallel depth of `ParIMCESub`, first note that all the elements of S at Line 6 of `ParIMCESub` are processed in parallel and the total cost of executing Lines 7 to 11 is $O(1)$. In addition, the depth of the operation at Line 12 of `ParIMCESub` is $O(1)$ using the concurrent hashtable. Therefore, the overall depth of the procedures from Line 4 to Line 12 is the number of new edges in a new maximal clique c considered at Line 2 of `ParIMCESub` which is $O(\min\{M^2, \rho\})$. Next, the depth of Line 14 to 16 is $O(1)$ because, it only takes $O(1)$ to execute Line 15 and 16 using Theorem 5. Therefore, for each new maximal clique, the depth of the procedure for enumerating all cliques subsumed by it is $O(\min\{M^2, \rho\})$. □

5.5 Evaluation

In this section, we experimentally evaluate the performance of our shared-memory parallel static (`ParTTT` and `ParMCE`) and dynamic (`ParIMCE`) algorithms for `MCE` on static and dynamic (real world and synthetic) graphs to show the parallel speedup and scalability of our algorithms over efficient sequential algorithms `TTT` and `IMCE` respectively. We also compare our algorithms with state-of-the-art parallel algorithms for MCE to show that the performance of our algorithm has substantially improved over the prior works. We run all the experiments on a computer configured with Intel Xeon (R) CPU E5-4620 running at 2.20GHz , with 32 physical cores (4 NUMA nodes each with 8 cores) and 1 TB RAM.

5.5.1 Datasets

We use eight different real-world static and dynamic networks from publicly available repositories KONECT [83], SNAP [86], and Network Repository [122] for doing the experiments. Dataset statistics are summarized in Table 5.2. For our experiments, we convert these networks to simple undirected graphs without self loops and without duplicate edges by removing self-loops, edges weights, parallel edges, and edge directions.

We consider `DBLP-Coauthor`, `As-Skitter`, `Wikipedia`, `Wiki-Talk`, and `Orkut` networks for the evaluation of the algorithms on static graphs and `DBLP-Coauthor`, `Flickr`, `Wikipedia`, `LiveJournal`, and `Ca-Cit-HepTh` networks for the evaluation of the algorithms on dynamic graphs.

`DBLP-Coauthor` shows the collaboration of authors of papers from DBLP computer science bibliography. In this graph, vertices represent authors, and there is an edge between two authors if they have published a paper [122]. `As-Skitter` is an Internet topology graph which represents the autonomous systems, connected to each other on the Internet [83]. `Wikipedia` is a network of English Wikipedia in 2013, where vertices represent pages in English Wikipedia, and there is an edge between two pages p and q if there is a hyperlink in page p to page q [86]. `Wiki-Talk` contains users of Wikipedia as vertices and each edge between two users in this graph indicates if one of the users has edited the page talk of the other user on Wikipedia [86]. `Orkut` is a social network where each vertex represents a user in the network, and there is an edge if there is a friendship relation between two users [83]. Similar to `Orkut`, `Flickr` and `LiveJournal` are also social networks where a vertex represents a user and an edge represents the friendship between two users. `Ca-Cit-HepTh` is a citation network of high energy physics theory where a vertex represents a paper and there is an edge between paper u and paper v if u cites v.

For the evaluation of `ParTTT` and `ParMCE`, we present the entire graph as input to the algorithm in the form of an edge list and for the evaluation of the algorithms on dynamic graphs, we start with the empty graph that contains all vertices but no edges and each time we add a set of edges in the increasing order of timestamps in a streaming manner for computing the changes in the set of maximal cliques over time. All the graphs we use for the evaluation of `ParIMCE` are real dynamic graphs with timestamp of creation is attached to every edge except for `LiveJournal` which is a static graph. We convert this graph to a dynamic graph by randomly permuting the edges and use that ordering for creating the stream of edges.

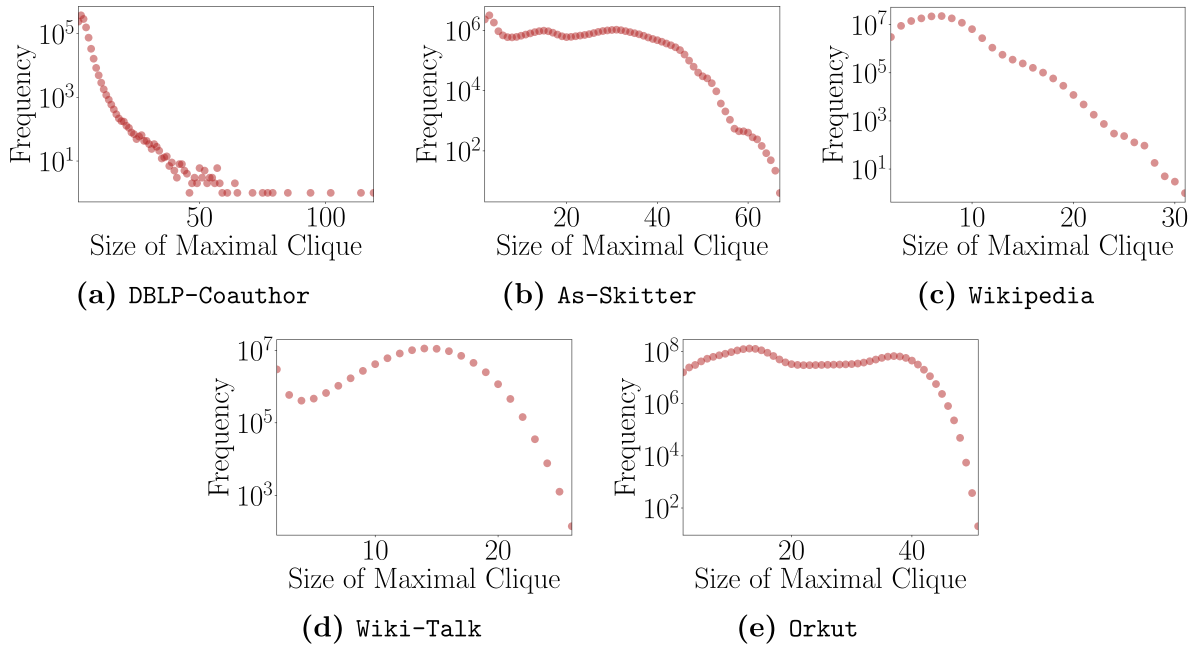


Figure 5.2: Frequency distribution of sizes of maximal cliques across different input graphs.

To understand the datasets better, we illustrated the frequency distribution of sizes of maximal cliques in Fig. 5.2. The size of maximal cliques in `DBLP-Coauthor` can be as large as 100 vertices. Although the number of such large size maximal cliques are small, the depth of the search space increases exponentially for enumerating the large maximal cliques. As shown in Fig. 5.2, most of the graphs contains more than tens of millions of maximal cliques. For example, `Orkut` contains more than two billion maximal cliques. Thus, the depth and breadth of the search tree makes MCE a challenging problem to solve.

5.5.2 Implementation of the Algorithms

In the implementations of `ParTTT`, `ParMCE`, and `ParIMCE`, we use `parallel_for` and `parallel_for_each` constructs provided by the Intel TBB parallel library [20] for implementing parallel for loop. We use concurrent_hash_map provided by TBB for atomic operations on hashtable as needed in our implementations. We use C++11 standard for the implementation of the algorithms and compile all the sources using Intel ICC compiler version 18.0.3 with optimization level '-O3'. We use the command 'numactl -i all' for balancing the memory in a NUMA machine. System level load balancing is performed using a dynamic work stealing scheduler [20] built inside TBB.

To compare with prior works on MCE, we implement some of them [143, 49, 45, 138, 158] in C++, and we use the executable of the C++ implementations for the rest (GreedyBB [123] and Hashing [87]) of the algorithms provided by the respective authors. See Subsection 5.5.4 for more details.

We compute the degeneracy number and triangle count for each vertex using sequential procedures. While the computation of per-vertex triangle counts and the degeneracy ordering could be potentially parallelized, implementing a parallel method to rank vertices based on their degeneracy number or triangle count is in itself a non-trivial task. We decided not to parallelize these routines since the degeneracy- and triangle-based ordering did not yield significant benefits when compared with degree-based ordering, where as degree-based ordering is trivially available, without any additional computation.

We assume that the entire graph is stored in available in shared global memory. The runtime of ParMCE consists of (1) the time required to rank vertices of the graph based on the ranking metric used in the algorithm, i.e. degree, degeneracy number, or triangle count of vertices and (2) the time required to enumerate all maximal cliques. For ParMCEDegen and ParMCETri algorithms, the runtime of ranking is also reported. Figures 5.3 and 5.4 show the parallel speedup (with respect to the runtime of TTT) and the runtime of ParMCE using different vertex ordering strategies, respectively. Table 5.4 shows the breakdown of the runtime into time for ordering and the time for clique enumeration.

The runtime of ParIMCE consists of (1) the computation time of ParIMCENew and (2) the computation time of ParIMCESub. In the implementation of ParIMCENew, we follow the design of ParMCE and instead of executing ParTTTExcludeEdges on the entire (sub)graph for an edge as in Line 9 of

ParIMCENew, we execute ParTTTExcludeEdges on per-vertex sub-problems in parallel. For dealing with load balance, we use degree based ordering of the vertices of $\mathcal{G}$ in creating the sub-problems. This choice of implementation of ParIMCENew comes from the improved performance of ParMCE using degree based vertex ordering over ParTTT in static case. For experiment on dynamic graphs, we use the batch size of 1000 edges for all the graphs except for an extremely dense graph Ca-Cit-HepTh (with original graph density 0.01) where we use batch size of 10 edges.

5.5.3 Discussion of the Results

Here we present and interpret the results of the empirical evaluation of the parallel algorithms that we design in this work. First we show the parallel speedup (with respect to the sequential algorithms) and scalability (with respect to the number of cores) of our parallel algorithms. Next we compare our works with the state-of-the-art sequential and parallel algorithms for MCE problem to show that our algorithms are substantially improved over the prior works.

5.5.3.1 Parallel MCE on Static Graphs

The total runtime of the parallel algorithms with 32 threads are shown in Table 5.3. We observe that ParTTT achieves a speedup of **5x-14x** over the sequential algorithm TTT. The three versions of ParMCE, ParMCEDegree, ParMCEDegen, ParMCETri achieve a speedup of **15x-21x** with 32 threads, when we consider only the runtime for maximal clique enumeration. The speedups are smaller for ParMCEDegen and ParMCETri when we add up the time taken by ranking strategies (See Figure 5.3).

The reason for the higher runtimes of ParTTT when compared with ParMCE is the greater cumulative overhead of computing the pivot and in processing the cand and fini sets in ParTTT. For example, for DBLP-Coauthor graph, in ParTTT, the cumulative overhead of computing pivot is 248 sec. and cumulative overhead of updating the cand and fini is 38 sec. whereas in ParMCE, these numbers are 156 sec. and 21 sec. respectively and these reduced cumulative times in ParMCE are reflected in the overall reduction in the parallel enumeration time of ParMCE over ParTTT by a factor of 2.

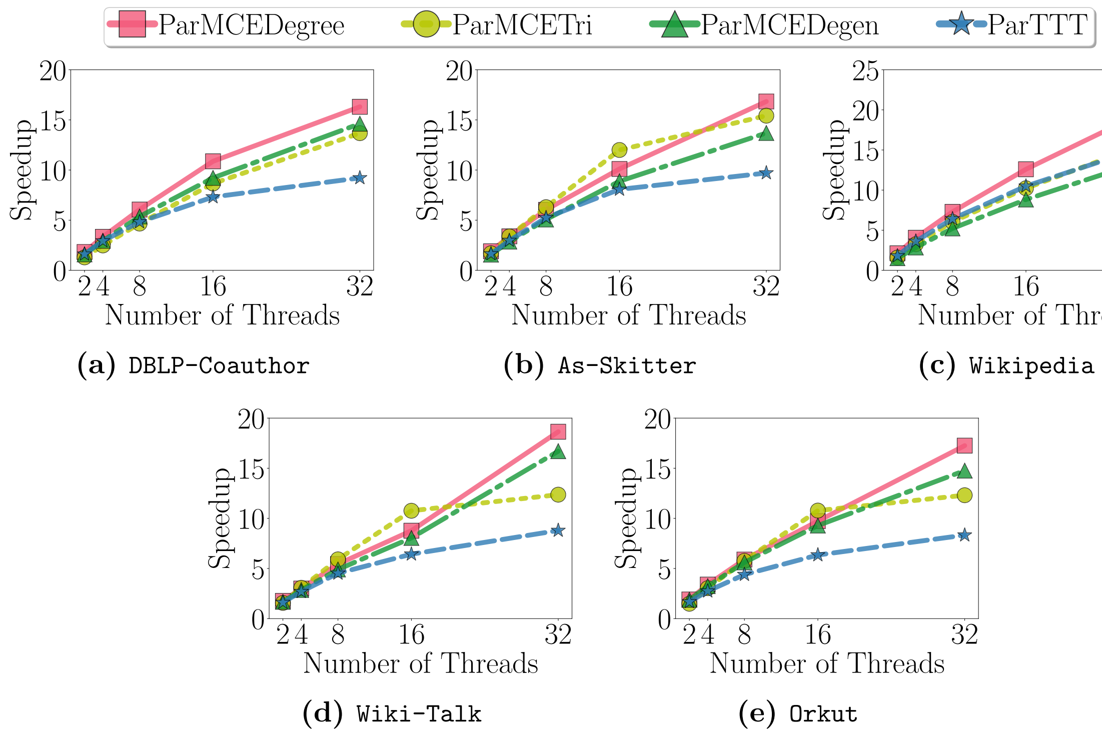


Figure 5.3: Parallel speedup when compared with `TTT` (sequential algo. due to Tomita et al. [143]) as a function of the number of threads.

Impact of vertex ordering on overall performance of `ParMCE`. Next we consider the influence of different vertex ordering strategies, degree, degeneracy, and triangle count, on the performance of `ParMCE`. The total computation time when using different vertex ordering strategies are presented in Table 5.4. Overall, we observe that degree based ordering (`ParMCEDegree`) usually achieves the smallest (or close to the smallest) runtime for clique enumeration, even when we don't take into account the time to compute the ordering. If we add in the time for computing the ordering, *degree based ordering is clearly better than triangle count or degeneracy based orderings*, since degree based ordering is available for free, while the degeneracy based ordering and triangle based ordering require additional computational overhead.

Scaling up with the degree of parallelism. As the number of threads (and the degree of

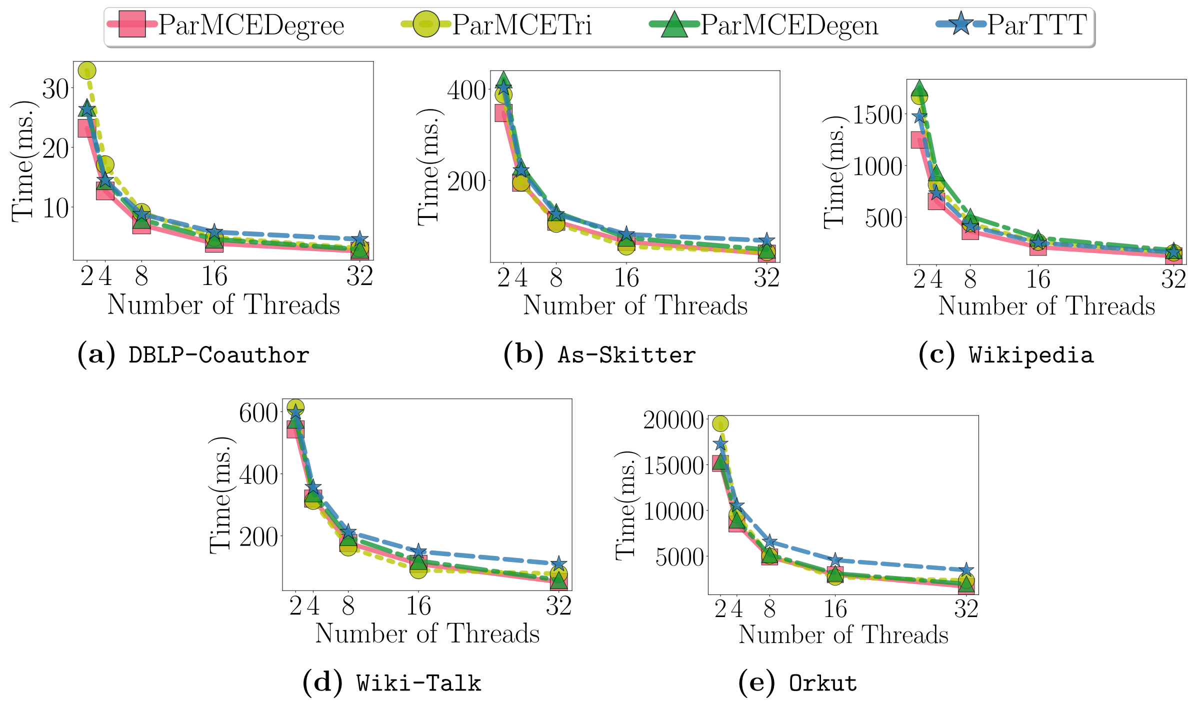
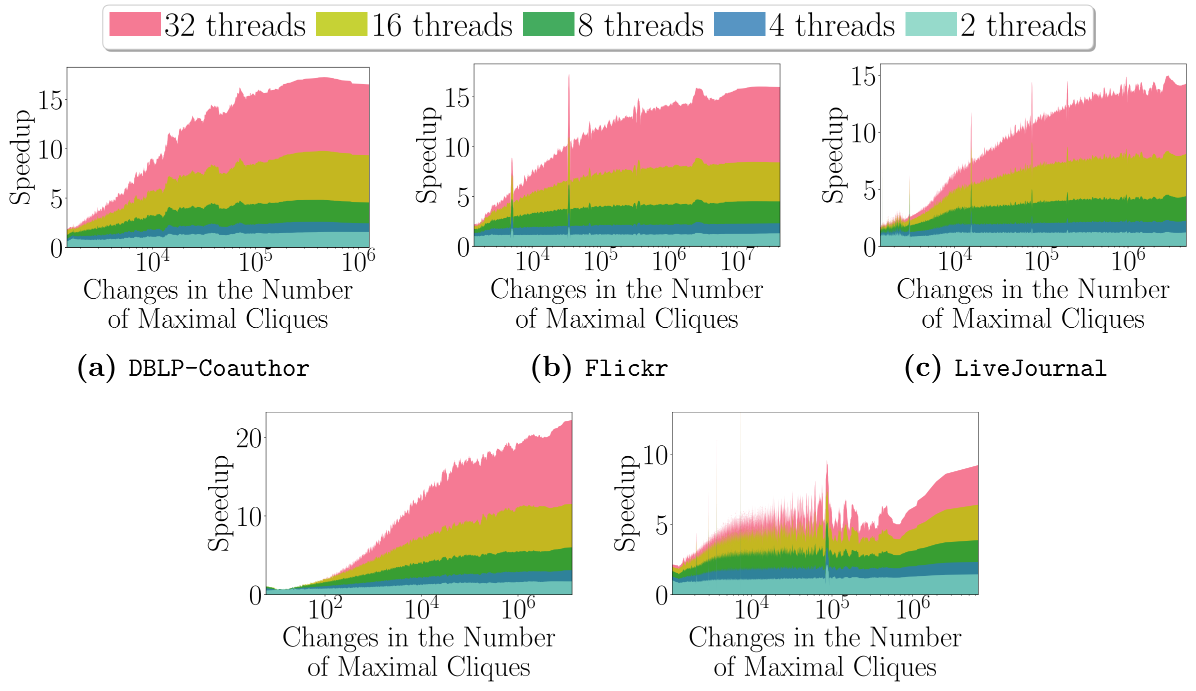


Figure 5.4: Runtime as a function of the number of threads.

parallelism) increases, the runtime of `ParMCE` and of `ParTTT` decreases, and the speedup as a function of the number of threads is shown in Figure 5.3 and the runtimes are shown in Figure 5.4. We see that `ParMCEDegree` achieves a speedup of more than **15x** on all graphs, using 32 threads.

5.5.3.2 Parallel MCE on Dynamic Graphs

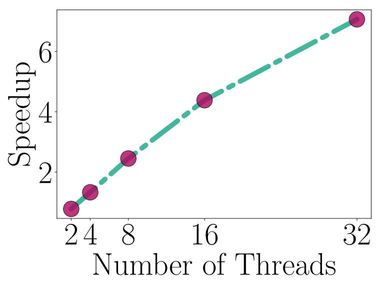
The cumulative runtime of `IMCE` and `ParIMCE` are presented in Table 5.5 which shows that the speedup achieved by `ParIMCE` is **3.6x-19.1x** over `IMCE`. This wide spectrum of speedups is mainly due to the variations in the size of the changes in the set of maximal cliques (number of new maximal cliques + number of subsumed maximal cliques) in the course of the incremental computation which can be observed in Figure 5.5. From this plot, we can see that the speedup increases with the increase in the size of the changes in the set of maximal cliques. This trend is as expected because the effect of parallelism will be prominent whenever the number of parallel tasks will become sufficiently large. This happens when the number of new and subsumed maximal



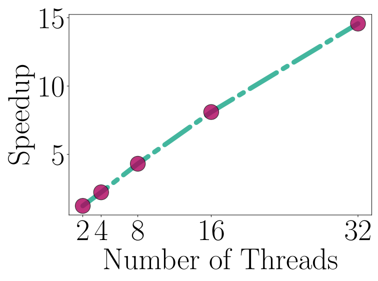
(a) DBLP-Coauthor (b) Flickr (c) LiveJournal

(d) Ca-Cit-HepTh (e) Wikipedia

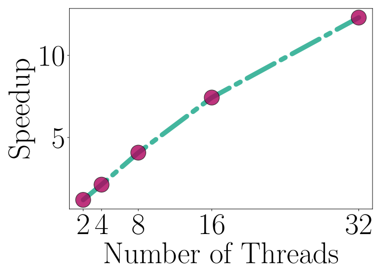
Figure 5.5: Parallel speedup of ParIMCE over IMCE as a function of the size of the change in the set of maximal cliques. The size of the change is measured by the total number of new maximal cliques and subsumed maximal cliques when a batch of edges is added to the graph.



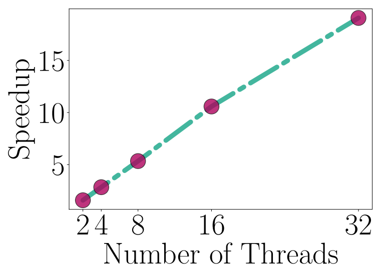
(a) `DBLP-Coauthor`



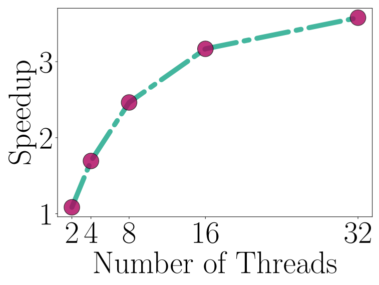
(b) `Flickr`



(c) `LiveJournal`



(d) `Ca-Cit-HepTh`



(e) `Wikipedia`

Figure 5.6: Parallel speedup of `ParIMCE` over `IMCE` as a function of number of threads, using the cumulative time of `ParIMCE` and of `IMCE` for processing all batches of edges.

cliques are large.

Scalability. The degree of parallelism increase with the increase in the number of threads. From Figure 5.6 we can see that the speedup increases linearly with the number of threads. This behavior shows the scalability of our parallel algorithm `ParIMCE`. When the size of the changes will become large, scalability will become prominent because otherwise, most of the processors will remain idle when there will not be large amount of parallel tasks to fully utilize all the available processors. This is observed in `Wikipedia` (Figure 5.6) where the cumulative size of change is relatively small.

5.5.4 Comparison with prior work

We compare the performance of `ParMCE` with prior sequential and parallel algorithms for MCE. We consider the following sequential algorithms: `GreedyBB` due to Segundo et al. [123], `TTT` due to Tomita et al. [143], and `BKDegeneracy` due to Eppstein et al. [49]. For the comparison with parallel algorithm, we consider algorithm `CliqueEnumerator` due to Zhang et al. [158], `Peamc` due to Du et al. [45], `PECO` due to Svendsen et al. [138], and most recent parallel algorithm `Hashing` due to Lessley et al. [87]. The parallel algorithms `CliqueEnumerator`, `Peamc`, and `Hashing` are designed for the shared-memory model, while `PECO` is designed for distributed memory. We modified `PECO` to work with shared-memory, by reusing the method for sub-problem construction, and eliminating the need to communicate subgraphs by storing a single copy of the graph in shared-memory. We considered three different ordering strategies for `PECO`, which we call `PECODegree`, `PECODegen`, and `PECOTri`. The comparison of performance of `ParMCE` with `PECO` is presented in Table 5.6. We note that `ParMCE` is significantly better than that of `PECO`, no matter which ordering strategy was considered.

The comparison of `ParMCE` with other shared-memory algorithms `Peamc`, `CliqueEnumerator`, and `Hashing` is shown in Table 5.7. The performance of `ParMCE` is seen to be much better than that of any of these prior shared-memory parallel algorithms. For the graph `DBLP-Coauthor`, `Peamc` did not finish within 5 hours, whereas `ParMCE` takes at most around 50 secs for enumerating 1.2 million maximal cliques. The poor running time of `Peamc` is due to two following reasons: (1) the algorithm does not apply efficient pruning techniques such as pivoting, used in `TTT`, and (2) the method to determine the maximality of a clique in the search space is not efficient. The `CliqueEnumerator` algorithm runs out of memory after a few minutes. The reason is that `CliqueEnumerator` maintains a bit vector for each vertex that is as large as the size of the input graph, and additionally, needs to store intermediate non-maximal cliques. For each such non-maximal clique, it is required to maintain a bit vector of length equal to the size of the vertex set of the original graph. Therefore, in `CliqueEnumerator` a memory issue is inevitable for a graph with millions of vertices.

A recent parallel algorithm in the literature, `Hashing` also has a significant memory overhead, and ran out of memory on the input graphs that we considered. The reason for its high memory requirement is that `Hashing` enumerates intermediate non-maximal cliques before finally outputting maximal cliques. The number of such intermediate non-maximal cliques may be very large, even for graphs with few number of maximal cliques. For example, a maximal clique of size c contains $2^c - 1$ non-maximal cliques.

Next, we compare the performance of `ParMCE` with that of sequential algorithms `BKDegeneracy` and a recent sequential algorithm `GreedyBB` – results are in Table 5.8. For large graphs, the performance of `BKDegeneracy` is almost similar to `TTT` whereas `GreedyBB` performs much worse than `TTT`. Since our `ParMCE` algorithm outperforms `TTT`, we can conclude that `ParMCE` is significantly faster than other sequential algorithms.

5.5.5 Summary of Experimental Results

We found that both `ParTTT` and `ParMCE` yield significant speedups over the sequential algorithm `TTT`nearly linear in the number of cores available. `ParMCE` using the degree-based vertex ranking always performs better than `ParTTT`. The runtime of `ParMCE` using degeneracy/triangle count based vertex ranking is sometimes worse than `ParTTT` due to the overhead of sequential computation of vertex ranking – note that this overhead is not needed in `ParTTT`. The parallel speedup of `ParMCE` is better when the input graph has many large sized maximal cliques. Overall, `ParMCE` consistently outperforms prior sequential and parallel algorithms for MCE. For a dynamic graph we found that `ParIMCE` consistently yields a substantial speedup over the efficient sequential algorithm `IMCE`. Further, the speedup of `ParIMCE` improves as the size of the change (to be enumerated) becomes larger.

Table 5.2: Static and Dynamic Networks, used for evaluation, and their properties. For some of the graphs used for evaluating the incremental algorithms (`Flickr` and `Ca-Cit-HepTh`), we could not report the information about maximal cliques as they did not finish within 8 hours, even using parallel algorithms.

Dataset	**#Vertices**	**#Edges**	**#Maximal Cliques**	**Average Size of Maximal Cliques**	**Size of Largest Clique**
`DBLP-Coauthor`	1,282,468	5,179,996	1,219,320	3	119
`Orkut`	3,072,441	117,184,899	2,270,456,447	20	51
`As-Skitter`	1,696,415	11,095,298	37,322,355	19	67
`Wiki-Talk`	2,394,385	4,659,565	86,333,306	13	26
`Wikipedia`	1,870,709	36,532,531	131,652,971	6	31
`Flickr`	2,302,925	22,838,276	-	-	-
`Ca-Cit-HepTh`	22,908	2,444,798	-	-	-

Table 5.3: Runtime (in sec.) of `TTT`, `ParTTT`, and `ParMCE` with different vertex orderings on 32 cores. The numbers exclude the time taken for vertex ordering. Note that the best algorithm, which uses degree based vertex ordering, has zero additional cost for computing the vertex ordering.

Dataset	`TTT`	`ParTTT`	`ParMCEDegree`	`ParMCEDegen`	`ParMCETri`
`DBLP-Coauthor`	42	4	2	3	3
`Orkut`	28923	3472	1676	2350	1959
`As-Skitter`	660	68	39	43	48
`Wiki-Talk`	961	109	52	78	58
`Wikipedia`	2646	160	123	155	179

Table 5.4: Total Runtime (in sec.) of `ParMCE` with different vertex orderings (using 32 threads). Total Runtime (TT) = Ranking Time (RT) + Enumeration Time (ET).

Dataset	`ParMCEDegree`	`ParMCEDegen`			`ParMCETri`		
		RT	**ET**	**TT**	**RT**	**ET**	**TT**
`DBLP-Coauthor`	3	25	3	28	42	3	45
`Orkut`	1676	928	2350	3278	2166	1959	4125
`As-Skitter`	39	41	43	84	122	48	170
`Wiki-Talk`	52	23	78	101	74	58	132
`Wikipedia`	123	244	155	399	950	179	1129

Table 5.5: Cumulative runtime (in sec.) over the incremental computation across all edges, with IMCE and ParIMCE using 32 threads. The total number of edges that are processed is also presented.

Dataset	**#Edges Processed**	IMCE	ParIMCE	**Parallel Speedup**
DBLP-Coauthor	5.1M	6608	933	**7x**
Flickr	4.1M	35238	2416	**14.6x**
Wikipedia	36.5M	9402	2614	**3.6x**
LiveJournal	19.2M	30810	2497	**12.3x**
Ca-Cit-HepTh	93.8K	33804	1767	**19.1x**

Table 5.6: Comparison of parallel runtime (excluding the time for computing vertex ranking) (in sec.) of ParMCE with a version of PECO that is modified to use shared-memory, using 32 threads. Three different variants are considered for each algorithm based on the vertex ordering strategy.

Dataset	PECODegree	ParMCEDegree	PECODegen	ParMCEDegen	PECOTri	ParMCETri
DBLP-Coauthor	6.4	2.6	6.9	3.1	6.8	2.9
Orkut	2050.7	1676.4	2183.4	2350	2361.9	1959.3
As-Skitter	261.5	39.2	331.8	42.8	260.9	48.2
Wiki-Talk	1729.7	51.6	1728.2	77.8	1720	57.6
Wikipedia	8982.5	123.3	9110.4	155.3	8938	178.8

Table 5.7: Comparison of runtimes (in sec.) of ParMCE with prior works on shared-memory algorithms for MCE (with 32 threads).

Dataset	ParMCEDegree	Hashing	CliqueEnumerator	Peamc
DBLP-Coauthor	2.6	Out of memory in 3 min.	Out of memory in 10 min.	Not complete in 5 hours.
Orkut	1676.4	Out of memory in 7 min.	Out of memory in 20 min.	Not complete in 5 hours.
As-Skitter	39.2	Out of memory in 5 min.	Out of memory in 10 min.	Not complete in 5 hours.
Wiki-Talk	51.6	Out of memory in 10 min.	Out of memory in 20 min.	Not complete in 5 hours.
Wikipedia	123.3	Out of memory in 10 min.	Out of memory in 20 min.	Not complete in 5 hours.

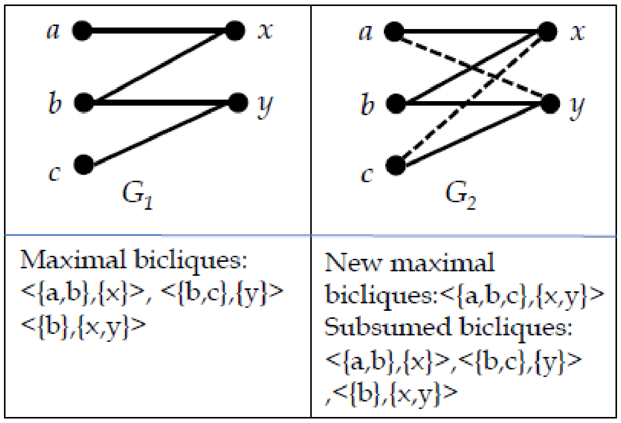
Table 5.8: Total runtime (sec.) of parallel algorithm ParMCE (with different vertex ranking, with 32 threads) and sequential algorithms BKDegeneracy and GreedyBB.

Dataset	BKDegeneracy	GreedyBB	ParMCEDegree	ParMCEDegen	ParMCETri
DBLP-Coauthor	53.6	Not finish in 30 min.	2.6	28.1	44.3
Orkut	29812.3	Out of memory in 5 min.	1676.4	3278	4125.3
As-Skitter	641.7	Out of memory in 10 min.	39.2	83.8	170.2
Wiki-Talk	1003.2	Out of memory in 10 min.	51.6	100.8	131.2
Wikipedia	2243.6	Out of memory in 10 min.	123.3	399	1128

CHAPTER 6. MAINTENANCE OF MAXIMAL BICLIQUES

6.1 Introduction

In this work we consider the incremental MBE problem, of maintaining the set of maximal bicliques in a bipartite graph that is evolving continuously over time due to the addition of stream of new edges. Let $G = (L, R, E)$ be a simple undirected bipartite graph with its vertex set partitioned into L, R, and edge set $E \subseteq L \times R$. Let $\mathcal{BC}(G)$ denote the set of all maximal bicliques in G.



Maximal bicliques: <{a,b},{x}>, <{b,c},{y}> <{b},{x,y}>

New maximal bicliques:<{a,b,c},{x,y}> Subsumed bicliques: <{a,b},{x}>,<{b,c},{y}> ,<{b},{x,y}>

Figure 6.1: Change in maximal bicliques when the graph changes from G_1 to G_2 due to the addition of edge set $H = \{\{a, y\}, \{c, x\}\}$. Each maximal biclique in G_1 is subsumed by a larger maximal biclique in G_2, and there is one new maximal biclique in G_2.

Suppose that starting from bipartite graph $G_1 = (L, R, E)$, the state of the graph changes to $G_2 = (L, R, E \cup H)$ due to the addition of a set of new edges H. Let $\Upsilon^{new}(G_1, G_2) = \mathcal{BC}(G_2) \setminus \mathcal{BC}(G_1)$ denote the set of new maximal bicliques that arise in G_2 that were not present in G_1 and $\Upsilon^{del}(G_1, G_2) = \mathcal{BC}(G_1) \setminus \mathcal{BC}(G_2)$ denote the set of maximal bicliques in G_1 that are no longer maximal bicliques in G_2 (henceforth called subsumed bicliques). See Fig. 6.1 for an example. Let $\Upsilon(G_1, G_2) = \Upsilon^{new}(G_1, G_2) \cup \Upsilon^{del}(G_1, G_2)$ denote the symmetric difference of $\mathcal{BC}(G_1)$ and $\mathcal{BC}(G_2)$. We consider the following questions:

(1) How large can be the size of $\Upsilon(G_1, G_2)$? In particular, can a small change in the set of edges cause a large change in the set of maximal bicliques in the graph?

(2) How can we compute $\Upsilon(G_1, G_2)$ efficiently? Can we quickly compute $\Upsilon(G_1, G_2)$ when $|\Upsilon(G_1, G_2)|$ is small? In short, can we design *change-sensitive algorithms* for enumerating elements of $\Upsilon(G_1, G_2)$, whose time complexity is proportional to the size of change, $|\Upsilon(G_1, G_2)|$?

Roadmap: The remaining sections are organized as follows. We present definitions and preliminaries in Section 6.2. Then we describe our algorithms in Section 6.3, results on the size of change in the set of maximal bicliques in Section 6.4, and experimental results in Section 6.5.

6.2 Preliminaries

Let $V(G)$ denote the set of vertices of G and $E(G)$ the set of edges in G. Let n and m denote the number of vertices and number of edges in G respectively. Let $\Gamma_G(u)$ denote the set of vertices adjacent to vertex u in G. If the graph G is clear from the context, we use $\Gamma(u)$ to mean $\Gamma_G(u)$. For an edge $e = (u, v) \in E(G)$, let $G - e$ denote the graph after deleting $e \in E(G)$ from G and $G + e$ denote the graph after adding $e \notin E(G)$ to G. For a set of edges H, let $G + H$ $(G - H)$ denote the graph obtained after adding (deleting) H to (from) $E(G)$. Similarly, for a vertex $v \notin V(G)$, let $G + v$ denote the graph after adding v to G and for a vertex $v \in V(G)$, let $G - v$ denote the graph after deleting v and all its adjacent edges from $E(G)$. Let $\Delta(G)$ denote the maximum degree of a vertex in G and $\delta(G)$ the minimum degree of a vertex in G.

Results for a static graph. In [117], Prisner presented the following result on the number of maximal bicliques in a bipartite graph with n vertices. Let $CP(k)$ denotes the *cocktail-party* graph which is a bipartite graph with k vertices in each partition where $V(CP(k)) = \{a_1, a_2, \ldots, a_k, b_1, b_2, \ldots, b_k\}$ and $E(CP(k)) = \{(a_i, b_p) : i \neq p\}$ [117]. See Figure 6.2 for an example.

Theorem 6 (Theorem 2.1 [117]). *Every bipartite graph with n vertices contains at most $2^{\frac{n}{2}} \approx 1.41^n$ maximal bicliques, and the only extremal (maximal) bipartite graphs are the graphs $CP(k)$.*

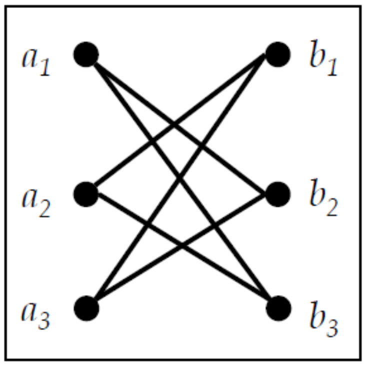


Figure 6.2: Cocktail-party graph on 6 vertices $CP(3)$

As a subroutine, we use an algorithm for enumerating maximal bicliques from a static undirected graph, whose runtime is proportional to the number of maximal bicliques. There are a few such algorithms [6, 92, 159]. We use the following result due to Liu et al. [92] as it provides the current best time complexity.

Theorem 7 (Liu et al., [92]). *For a graph G with n vertices, m edges, maximum degree Δ, and number of maximal bicliques μ, there is an algorithm* `MineLMBC` *for enumerating maximal bicliques in G with time complexity $O(n\Delta\mu)$ and space complexity $O(m+\Delta^2)$.*

`MineLMBC` is an algorithm for enumerating maximal bicliques of a static graph $G=(V,E)$ that is based on depth-first-search. It takes as input the graph G and the size threshold s. The algorithm enumerates all maximal bicliques of G with size of each partition at least s. Clearly, by setting $s=1$, the algorithm enumerates all maximal bicliques of G. Please see Section 7.2 for a more details discussion on `MineLMBC`.

6.3 Algorithms for Maximal Bicliques

For graph G and set of edges H, we use Υ^{new} to mean $\Upsilon^{new}(G,G+H)$, and Υ^{del} to mean $\Upsilon^{del}(G,G+H)$. Before presenting our change-sensitive algorithm for maximal bicliques, we first consider two baseline approaches for the problem.

6.3.1 Baseline Algorithms for Maximal Bicliques

First we consider a straightforward approach for maintaining maximal bicliques using a current state-of-the-art algorithm for static graphs. This algorithm, which we call as `BaselineBC`, works by enumerating $\mathcal{BC}(G+H)$, the set of all maximal bicliques in $(G+H)$ once G is updated with a set of new edges H. It then outputs the symmetric difference between $\mathcal{BC}(G)$ (maintained in memory) and $\mathcal{BC}(G+H)$.

We next present another baseline `BaselineBC`*, which is better than `BaselineBC`. The idea in `BaselineBC`* is to focus on the portion of the graph where changes occur. Let V_H denote the set of all vertices in G that are incident to at least one edge in H. For enumerating new maximal bicliques, we note that it is sufficient to consider the subgraph G_H of $G+H$ that is induced by V_H and the vertices in $\cup_{v \in V_H} \Gamma_{G+H}(v)$. `BaselineBC`* enumerates all maximal bicliques in G_H using a state-of-the-art algorithm for static graphs. Each biclique b thus generated is a new maximal biclique if b contains at least an edge from H. For enumerating subsumed bicliques, we note each subsumed maximal biclique b' in G is a subgraph of at least one new maximal biclique b, and must also be contained in $b-H$. Thus, subsumed maximal bicliques are enumerated by considering each new maximal b, and enumerating maximal bicliques in $b-H$. If a biclique thus enumerated is present in $\mathcal{BC}(G)$, it is output as a subsumed biclique.

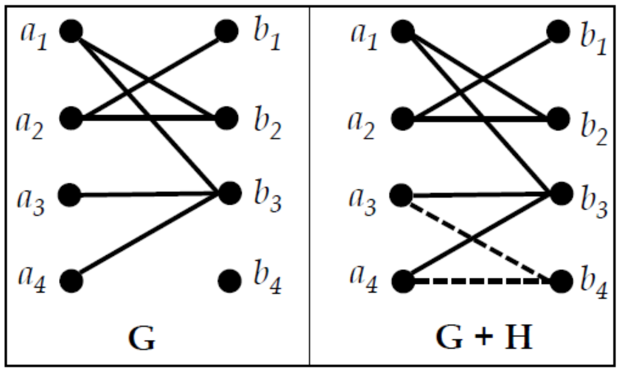


Figure 6.3: The original graph G has 4 maximal bicliques. When new edges in H (in dotted line) are added to G, all maximal bicliques in G remain maximal in $G+H$ and only one maximal biclique is newly formed ($< \{a_3, a_4\}, \{b_3, b_4\} >$).

We can expect `BaselineBC*` to do much better than `BaselineBC`. Still `BaselineBC*` it is not change-sensitive, because it may, in the process of enumerating new maximal bicliques, generate bicliques of G that remain maximal in $G+H$. For example, see Fig. 6.3. We next present algorithms that carefully avoid enumerating any maximal biclique of G that remains maximal in $G + H$.

6.3.2 Change-Sensitive Algorithm `DynamicBC`

Our change-sensitive algorithm, `DynamicBC`, has two parts: (1) Algorithm `NewBC` for enumerating new maximal bicliques, described in Section 6.3.3 and (2) Algorithm `SubBC` for enumerating subsumed bicliques, described in Section 6.3.4.

```
Algorithm 13: DynamicBC($G, H, \mathcal{BC}(G)$)
    Input: $G$ - Input bipartite graph, $H$ - Edges being added to $G$, $\mathcal{BC}(G)$ - set of
           maximal bicliques of $G$
    Output: $\Upsilon$ : the union of set of new maximal bicliques and subsumed bicliques
1   $\Upsilon^{new} \leftarrow$ NewBC($G, H$)
2   $\Upsilon^{del} \leftarrow$ SubBC($G, H, \mathcal{BC}(G), \Upsilon^{new}$)
3   $\Upsilon \leftarrow \Upsilon^{new} \cup \Upsilon^{del}$
```

The main result on the time complexity of `DynamicBC` is summarized in the following theorem.

Theorem 8. `DynamicBC` *enumerates the change in the set of maximal bicliques, with time complexity* $O(\Delta^2\rho|\Upsilon^{new}| + 2^\rho|\Upsilon^{new}|)$ *where* Δ *is the maximum degree of a vertex in* $G+H$ *and* ρ *is the size of* H*, the set of newly added edges.*

We note that if ρ is constant, the time complexity of enumerating the change is $O(\Delta^2|\Upsilon^{new}|)$. Thus we have the following observation.

Observation 3. `DynamicBC` *is a change-sensitive algorithm for MBE, when the number of edges added,* ρ *is a constant.*

6.3.3 Enumerating New Maximal Bicliques

In our algorithm, we require that each maximal biclique enumerated by `NewBC` to contain at least one edge from H, thus forcing it to be a new maximal biclique. Let G' denote the graph $G + H$. For each new edge $e \in H$, let $\mathcal{BC}'(e)$ denote the set of maximal bicliques in G' containing edge e.

Lemma 20. $\Upsilon^{new} = \cup_{e\in H}\mathcal{BC}'(e)$.

Proof. Each biclique in Υ^{new} must contain at least one edge from H. To see this, consider a biclique $b \in \Upsilon^{new}$. If b did not contain an edge from H, then b is also a maximal biclique in G, and hence cannot belong to Υ^{new}. Hence, $b \in \mathcal{BC}'(e)$ for some edge $e \in H$, and $b \in \cup_{e\in H}\mathcal{BC}'(e)$. This shows that $\Upsilon^{new} \subseteq \cup_{e\in H}\mathcal{BC}'(e)$.

Next consider a biclique $b \in \cup_{e\in H}\mathcal{BC}'(e)$. It must be the case that $b \in \mathcal{BC}'(h)$ for some h in H. Thus b is a maximal biclique in $G + H$. Since b contains edge $h \in H$, b cannot be a biclique in G. Thus $b \in \Upsilon^{new}$. This shows that $\cup_{e\in H}\mathcal{BC}'(e) \subseteq \Upsilon^{new}$. □

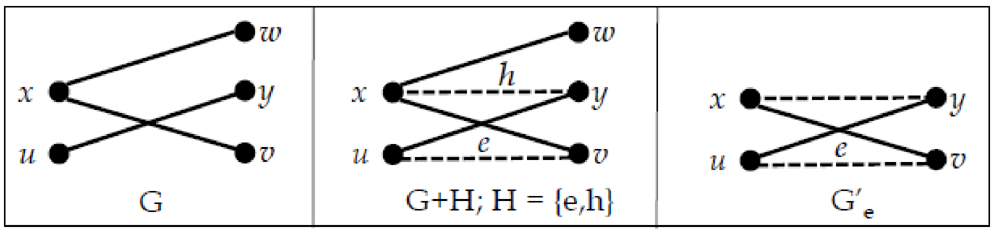


Figure 6.4: Construction of G'_e from $G' = G + H$ when a set of new edges $H = \{e, h\}$ is added to G. $A = \Gamma_{G'}(v) = \{u, x\}$ and $B = \Gamma_{G'}(u) = \{v, y\}$.

Next, for each edge $e = (u, v) \in H$, we present an efficient way to enumerate all bicliques in $\mathcal{BC}'(e)$ through enumerating maximal bicliques in a specific subgraph G'_e of G', constructed as follows. Let $A = \Gamma_{G'}(u)$ and $B = \Gamma_{G'}(v)$. Then $G'_e = (A, B, E')$ is a subgraph of G' induced by vertices in A and B, and all edges between these sets of vertices. See Fig. 6.4 for an example of the construction of G'_e.

Lemma 21. *For each* $e \in H$, $\mathcal{BC}'(e) = \mathcal{BC}(G'_e)$

Proof. First we show that $\mathcal{BC}'(e) \subseteq \mathcal{BC}(G'_e)$. Consider a biclique $b = (X, Y)$ in $\mathcal{BC}'(e)$. Let $e = (u, v)$. Here b contains both u and v. Suppose that $u \in X$ and $v \in Y$. According to the construction G'_e contains all the vertices adjacent to u and all the vertices adjacent to v. And in b, all the vertices in X are connected to all the vertices in Y. Hence, b is a biclique in G'_e. Also, b is a maximal biclique in G', and G'_e is an induced subgraph of G' which contains all the vertices of b. Hence, b is a maximal biclique in G'_e.

Next we show that $\mathcal{BC}(G'_e) \subseteq \mathcal{BC}'(e)$. Consider a biclique $b' = (X', Y')$ in $\mathcal{BC}(G'_e)$. Clearly, b' contains e as it contains both u and v and b' is a maximal biclique in G'_e. Hence, b' is also a biclique in G' that contains e. Now we prove that b' is also maximal in G'. Suppose not, that there is a vertex $w \in V(G')$ such that b' can be extended with w. Then, as per the construction of G'_e, $w \in V(G'_e)$ since w must be adjacent to either u or v. Then, b' is not maximal in G'_e. This is a contradiction. Hence, b' is also maximal in G'. Therefore, $b' \in \mathcal{BC}'(e)$. □

Based on the above observation, we present our change-sensitive algorithm `NewBC` (Algorithm 14). We use an output-sensitive algorithm for a static graph `MineLMBC` for enumerating maximal bicliques from G'_e. Note that typically, G'_e is much smaller than G' since it is localized to edge e, and hence enumerating all maximal bicliques from G'_e should be relatively inexpensive.

Theorem 9. `NewBC` *enumerates the set of all new bicliques arising from the addition of H in time $O(\Delta^2\rho|\Upsilon^{new}|)$ where Δ is the maximum degree of a vertex in G' and ρ is the size of H. The space complexity is $O(|E(G')| + \Delta^2)$.*

Proof. First we consider correctness of the algorithm. From Lemma 20 and Lemma 21, we know that Υ^{new} is enumerated by enumerating $\mathcal{BC}(G'_e)$ for every $e \in H$. Our algorithm does this exactly, and uses the `MineLMBC` algorithm for enumerating $\mathcal{BC}(G'_e)$. For the runtime, consider that the algorithm iterates over each edge e in H. In each iteration, it constructs a graph G'_e and runs `MineLMBC`(G'_e). Note that the number of vertices in G'_e is no more than 2Δ, since it is the size of the union of the edge neighborhoods of one of the ρ edges in G'. The set of maximal bicliques generated in each iteration is a subset of Υ^{new}, therefore the number of maximal bicliques generated

```
Algorithm 14: NewBC($G, H$)
   Input: $G$ - Input bipartite graph, $H$ - Edges being added to $G$
   Output: bicliques in $\Upsilon^{new}$, each biclique output once
1  Consider edges of $H$ in an arbitrary order $e_1, e_2, \ldots, e_\rho$
2  $G' \leftarrow G + H$
3  for $i = 1 \ldots \rho$ do
4      $e \leftarrow e_i = (u, v)$
5      $G'_e \leftarrow$ a subgraph of $G'$ induced by $\Gamma_{G'}(u) \cup \Gamma_{G'}(v)$
6      Generate bicliques of $G'_e$ using MineLMBC. Let $B$ denote the set of the generated
        bicliques.
7      for $b \in B$ do
8          if $b$ does not contain an edge $e_j$ for $j < i$ then
9              Add $b$ to $\Upsilon^{new}$
10 return $\Upsilon^{new}$
```

from each iteration is no more than $|\Upsilon^{new}|$. From Theorem 7, we have that the runtime of each iteration is $O(\Delta^2|\Upsilon^{new}|)$. Since there are ρ edges in H, the result on runtime follows. For the space complexity, we note that the algorithm does not store the set of new bicliques in memory at any point. The space required to construct G'_e is linear in the size of G'. From Theorem 7, the total space requirement is $O(|E(G')| + \Delta^2)$. □

6.3.4 Enumerating Subsumed Maximal Bicliques

We now consider enumerating $\mathcal{BC}(G) \setminus \mathcal{BC}(G')$ where $G' = G + H$. Suppose a new maximal biclique b of G' subsumed a maximal biclique b' of G. Note that b' is also a maximal biclique in $b - H$. One approach is to enumerate all maximal bicliques in $b - H$ and then check which among them is maximal in G. However, checking maximality of a biclique is a costly operation in itself, since we need to consider the neighborhood of every vertex in the biclique. Another idea is to store the bicliques of the graph explicitly and see which among the generated bicliques are contained in the set of maximal bicliques of G. This is not desirable either, since large amount of memory is required to store the set of all maximal bicliques of G.

We consider a more efficient approach, of storing the signatures of the maximal bicliques instead of storing the bicliques themselves. We then enumerate all maximal bicliques in $b - H$ and for each biclique thus generated, we compare the signature of the generated biclique with the signatures of the bicliques stored. An algorithm following this idea is presented in Algorithm 15. This reduces the memory requirement. We use a standard hash function (the 64 bit murmur hash [1]). For computing the signature of a biclique, first we represent the biclique in a canonical form (vertices in first partition represented in lexicographic order followed by vertices in another partition represented in lexicographic order). Then we convert the string into bytes, and apply the hash function to derive the signature. By storing hash signatures instead of maximal bicliques, we are able to quickly check whether a maximal biclique from $b - H$ is contained in the set of maximal bicliques of G by comparing their hash values. We also pay a lower memory cost, when compared with storing all bicliques.

With the approach of storing hash signatures of bicliques, there is a small probability of a hash collision, i.e. the case when two bicliques A and B are unequal, but their hash values are equal. The effect of a collision is a false positive – our algorithm may incorrectly conclude that a biclique is a subsumed biclique where as it is not. However, this is a very unlikely event with the use of 64 bit signatures, where the chances of two unequal strings having the same hash value is extremely small. In our experiments, the set of bicliques that were enumerated by our algorithm always matches the set of subsumed bicliques. Note that we can always double check each such biclique by explicitly checking if this is maximal in G, to avoid a chance of a false positive. We did not do this in our implementation since the chance of a false positive is so small.

Now we prove that Algorithm 15 enumerates all maximal bicliques of $b - H$.

Lemma 22. *In Algorithm 15, for each $b \in \Upsilon^{new}$, S after Line* 14 *contains all maximal bicliques in $b - H$.*

Proof. First observe that, removing H from b is equivalent to removing those edges in H which are present in b. Hence, computing maximal bicliques in $b - H$ reduces to computing maximal

[1] https://sites.google.com/site/murmurhash/

bicliques in $b - H_1$ where H_1 is the set of all edges in H which are present in b. We use induction on the number of edges k in H_1. Consider the base case, when $k = 1$. H_1 contains a single edge $e_1 = \{u, v\}$. Clearly, $b - H_1$ has two maximal bicliques $b \setminus \{u\}$ and $b \setminus \{v\}$. Suppose, that the set H_1 is of size k. Our inductive hypothesis is that all maximal bicliques in $b - H_1$ are enumerated. Consider $H_1' = \{e_1, e_2, ..., e_k, e_{k+1}\}$ with $k+1$ edges. Now each maximal biclique b' in $b - H_1$ either remains maximal within $b - H_1'$ (if at least one endpoint of e_{k+1} is not in b') or generates two maximal bicliques in $b - H_1'$ (if both endpoints of e_{k+1} are in b'). Thus, for each $b \in \Upsilon^{new}$, S after Line 14 contains all maximal bicliques within $b - H$. □

We now show that the above algorithm is a change-sensitive algorithm for enumerating all elements of Υ^{del} when the number of edges ρ in H is constant.

Theorem 10. *Algorithm 15 enumerates all bicliques in $\Upsilon^{del} = \mathcal{BC}(G) - \mathcal{BC}(G + H)$ using time $O(2^\rho|\Upsilon^{new}|)$ where ρ is the number of edges in H. The space complexity of the algorithm is $O(|E(G')| + |V(G')| + \Delta^2 + |\mathcal{BC}(G)|)$.*

Proof. We first show that every biclique b' enumerated by the algorithm is indeed a biclique in Υ^{del}. Note that b' is a maximal biclique in G, due to explicitly checking the condition. Further, b' is not a maximal biclique in $G + H$, since it is a proper subgraph of b, a maximal biclique in $G + H$. Next, we show that all bicliques in Υ^{del} are enumerated. Consider any subsumed biclique $b' \in \Upsilon^{del}$. It must be contained within $b \setminus H$, where b is a maximal biclique within Υ^{new}. Moreover, b' will be a maximal biclique within $b \setminus H$, and will be enumerated by the algorithm according to Lemma 22.

For the time complexity we show that for any $b \in \Upsilon^{new}$, the maximum number of maximal bicliques in $b - H$ is 2^ρ using induction on ρ. Suppose $\rho = 1$ so that H contains a single edge, say $e_1 = (u, v)$. Then, $b - H$ has two maximal bicliques, $b \setminus \{u\}$ and $b \setminus \{v\}$, proving the base case. Suppose that for any set H of size k, it was true that $b - H$ has no more than 2^k maximal bicliques. Consider a set $H'' = \{e_1, e_2, \ldots, e_{k+1}\}$ with $k+1$ edges. Let $H' = \{e_1, e_2, \ldots, e_k\}$. Subgraph $b - H''$ is obtained from $b - H'$ by deleting a single edge e_{k+1}. By induction, we have that $b - H'$ has no more than 2^k maximal bicliques. Each maximal biclique b' in $b - H'$ either remains a maximal

Algorithm 15: `SubBC`$(G, H, BC, \Upsilon^{new})$

```
Input: $G$ - Input bipartite graph
       $H$ - Edge set being added to $G$
       $BC$ - Set of maximal bicliques in $G$
       $\Upsilon^{new}$ - set of new maximal bicliques in $G + H$
Output: All bicliques in $\Upsilon^{del} = \mathcal{BC}(G) \setminus \mathcal{BC}(G + H)$
1  $\Upsilon^{del} \leftarrow \emptyset$
2  for $b \in \Upsilon^{new}$ do
3      $S \leftarrow \{b\}$
4      for $e = (u, v) \in E(b) \cap H$ do
5          $S' \leftarrow \phi$
6          for $b' \in S$ do
7              if $e \in E(b')$ then
8                  $b_1 = b' \setminus \{u\}$ ; $b_2 = b' \setminus \{v\}$
9                  $S' \leftarrow S' \cup \{b_1, b_2\}$
10             else
11                 $S' \leftarrow S' \cup b'$
12         /* $S'$ contains all the maximal bicliques in $b - \{e_1, e_2, ..., e_k\}$ where
              $\{e_1, e_2, ..., e_k\} \subseteq E(b) \cap H$ are considered so far.        */
13
14         $S \leftarrow S'$
15     for $b' \in S$ do
16         if $b' \in BC$ then
17             Add $b'$ to $\Upsilon^{del}$
18             $BC \leftarrow BC \setminus b'$
19 return $\Upsilon^{del}$
```

biclique within $b - H''$ (if at least one endpoint of e_{k+1} is not in b'), or leads to two maximal bicliques in $b - H''$(if endpoints of e_{k+1} are in different bipartition of b'). Hence, the number of maximal bicliques in $b - H''$ is no more than 2^{k+1}, completing the inductive step. Following this, for each biclique $b \in \Upsilon^{new}$, we need to check for maximality for no more than 2^{ρ} bicliques in G. This checking can be performed by checking whether each such generated biclique in contained in the set $\mathcal{BC}(G)$ and for each biclique, this can be done in constant time.

For the space bound, we first note that in Algorithm 15, enumerating maximal bicliques within $b - H$ consumes space $O(|E(G')| + \Delta^2)$, and checking for maximality can be done in space linear in size of G. However, for storing the maximal bicliques in G takes $O(|\mathcal{BC}(G)|)$ space. Hence, for these operations, the overall space-cost for each $b \in \Upsilon^{new}$ is $O(|E(G')| + |V(G')| + \Delta^2 + |\mathcal{BC}(G)|)$. The only remaining space cost is the size of Υ^{new}, which can be large. Note that, the algorithm only iterates through Υ^{new} in a single pass. If elements of Υ^{new} are provided as a stream from the output of an algorithm such as `NewBC`, then they do not need to be stored within a container, so that the memory cost of receiving Υ^{new} is reduced to the cost of storing a single maximal biclique within Υ^{new} at a time. □

```
Algorithm 16: Decremental(G,H)
  Input: $G$ - Input bipartite graph, $H$ - Edges being deleted from $G$
  Output: $\Upsilon^{new}(G, G-H) \cup \Upsilon^{del}(G, G-H)$
1 $\Upsilon^{new} \leftarrow \phi$; $\Upsilon^{del} \leftarrow \phi$; $G'' \leftarrow G - H$
2 $\Upsilon^{del} \leftarrow$ NewBC($G''$, $H$)
3 $\Upsilon^{new} \leftarrow$ SubBC($G''$, $H$, $\mathcal{BC}(G'')$, $\Upsilon^{del}$)
4 return $\Upsilon^{new} \cup \Upsilon^{del}$
```

6.3.5 Decremental and Fully Dynamic Cases

We now consider the maintenance of maximal bicliques in the decremental case, when edges are deleted from the graph. This case can be handled using a reduction to the incremental case. We show in Lemma 23 that the maintenance of maximal bicliques due to deletion of a set of edges

H from a bipartite graph G is equivalent to the maintenance of maximal bicliques due to addition of H to the bipartite graph $G - H$. An algorithm for the decremental case based on Lemma 23 is presented in Algorithm 16.

Lemma 23. $\Upsilon^{new}(G, G - H) = \Upsilon^{del}(G - H, G)$ *and* $\Upsilon^{del}(G, G - H) = \Upsilon^{new}(G - H, G)$

Proof. Note that $\Upsilon^{new}(G, G - H)$ is the set of all bicliques that are maximal in $G - H$, but not in G. By definition, this is equal to $\Upsilon^{del}(G - H, G)$. Similarly we have $\Upsilon^{del}(G, G - H) = \Upsilon^{new}(G - H, G)$. □

The fully dynamic case, where there is a set of edges added as well as a set of edges deleted, can be handled as follows. Suppose that edge set H was added and set H' was deleted. We first ensure that common $H \cap H' = \emptyset$. If the intersection is non-empty, then all common edges are removed from H and H', since they will not have an impact on the graph. We then use the incremental algorithm to handle the addition of H, followed by the decremental algorithm to handle deletion of H'. Using the outputs of these two algorithms, it is possible to enumerate the total change. We note that this algorithm is not provably change-sensitive, since the intermediate results that are output after the addition of H are not necessarily part of the final output.

6.4 Magnitude of change in Bicliques

We consider the maximum change in the set of maximal bicliques when a set of edges is added to the bipartite graph. Let $\lambda(n)$ denote the maximum size of $\Upsilon(G, G + H)$ taken over all n vertex bipartite graphs G and edge sets H. We derive the following upper bound on the maximum size of $\Upsilon(G, G + H)$ in the following Lemma:

Lemma 24. $\lambda(n) \leq 2g(n)$.

Proof. Note that, for any bipartite graph G with n vertices and for any new edge set H it must be true that $|\mathcal{BC}(G)| \leq g(n)$ and $|\mathcal{BC}(G + H)| \leq g(n)$. Since $|\Upsilon^{new}(G, G + H)| \leq |\mathcal{BC}(G + H)|$ and $|\Upsilon^{del}(G, G + H)| \leq |\mathcal{BC}(G)|$, it follows that $|\Upsilon(G, G + H)| \leq |\mathcal{BC}(G + H)| + |\mathcal{BC}(G)| \leq 2g(n)$. □

Next we analyze the upper bound of $|\Upsilon(G, G+e)|$ in the following when an edge $e \notin E(G)$ is added to G.

Theorem 11. *For an integer $n \geq 2$, a bipartite graph $G = (L, R, E)$ with n vertices, and any edge $e = (u, v) \notin E(G), u \in L, v \in R$, the maximum size of $\Upsilon(G, G+e)$ is $3g(n-2)$, and for each even n, there exists a bipartite graph that achieves this bound.*

We prove this theorem in the following two lemmas. In Lemma 25 we prove that the size of $\Upsilon(G, G+e)$ can be as large as $3g(n-2)$ and in Lemma 27 we prove that the size of $\Upsilon(G, G+e)$ is at most $3g(n-2)$.

Lemma 25. *For any even integer $n > 2$ there exists a bipartite graph G on n vertices and an edge $e = (u, v) \notin E(G)$ such that $|\Upsilon(G, G+e)| = 3g(n-2)$.*

Proof. We use proof by construction. Consider bipartite graph $G = (L, R, E)$ constructed on vertex set $L \cup R$ with n vertices such that $|L| = |R| = n/2$. Let $u \in L$ and $v \in R$ be two vertices and let $L' = L \setminus \{u\}$ and $R' = R \setminus \{v\}$. Let G'' denote the induced subgraph of G on vertex sets L' and R'. In our construction, G'' is $CP(\frac{n}{2} - 1)$. In graph G, in addition to the edges in G'', we add an edge from each vertex in R' to u and an edge from each vertex in L' to v. We add edge $e = (u, v)$ to G to get graph $G' = G + e$ (see Fig. 6.5 for construction). We claim that the size of $\Upsilon(G, G')$ is $3g(n-2)$.

First, we note that the total number of maximal bicliques in G is $2g(n-2)$. Each maximal biclique in G contains either vertex u or v, but not both. The number of maximal bicliques that contain vertex u is $g(n-2)$, since each maximal biclique in G'' leads to a maximal biclique in G by adding u. Similarly, the number of maximal bicliques in G that contains v is $g(n-2)$, leading to a total of $2g(n-2)$ maximal bicliques in G.

Next, we note that the total number of maximal bicliques in G' is $g(n-2)$. To see this, note that each maximal biclique in G' contains both vertices u and v. Further, for each maximal biclique in G'', we get a corresponding maximal biclique in G' by adding vertices u and v. Hence the number of maximal bicliques in G' equals the number of maximal bicliques in G'', which is $g(n-2)$.

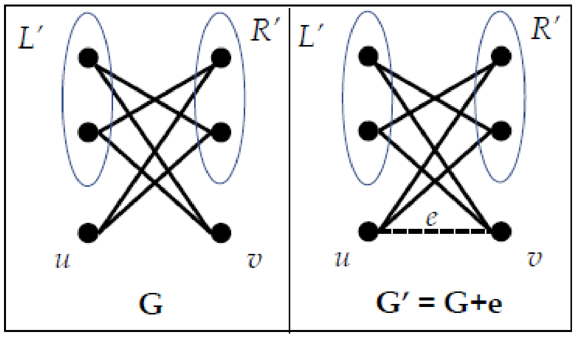


Figure 6.5: Construction showing the changes in the set of maximal bicliques when a new edge is added. G is in the left on $n = 6$ vertices. G'' consists of vertices in L' and R' and edges among them to make it a cocktail-party graph. G' in the right is obtained by adding edge $e = (u, v)$ to G.

No maximal biclique in $\mathcal{BC}(G)$ contains both u and v, while every maximal biclique in G' contains both u and v. Hence, $\mathcal{BC}(G)$ and $\mathcal{BC}(G')$ are disjoint sets, and $|\Upsilon(G, G')| = |\mathcal{BC}(G)| + |\mathcal{BC}(G')| = 3g(n-2)$. □

Now we will prove a few results that we will use in proving Lemma 27.

Lemma 26. *If $e = (u, v)$ is added to G, each biclique $b \in \mathcal{BC}(G) - \mathcal{BC}(G+e)$ contains either u or v.*

Proof. Proof by contradiction. Suppose there is maximal biclique $b = (b_1, b_2)$ in $\mathcal{BC}(G) - \mathcal{BC}(G+e)$ that contain neither u nor v. Then, b must be maximal biclique in G. Since b is not a maximal biclique in $G + e$, b is contained in another maximal biclique $b' = (b'_1, b'_2)$ in $G + e$. Note that b' must contain edge $e = (u, v)$, and hence, both vertices u and v. Since b' is a biclique, every vertex in b'_2 is connected to u in G'. Hence, every vertex in b_2 is connected to u even in G. Therefore, $b \cup \{u\}$ is a biclique in G, and b is not maximal in G, contradicting our assumption. □

Observation 4. *For a bipartite graph $G = (L, R, E)$ and a vertex $u \in V(G)$, the number of maximal bicliques that contains u is at most $g(n-1)$.*

Proof. Suppose, $u \in L$. Then each maximal biclique b in G that contains u, corresponds to a unique maximal biclique in $G - \{u\}$. Such maximal bicliques can be derived from b by deleting u from b.

As the maximum number of maximal bicliques in $G - \{u\}$ is $g(n-1)$, the maximum number of maximal bicliques in G can be no more than $g(n-1)$. □

Observation 5. *The number of maximal bicliques containing a specific edge* (u, v) *is at most* $g(n-2)$.

Proof. Consider an edge $(u, v) \in E(G)$. Let vertex set $V' = (\Gamma_G(u) \cup \Gamma_G(v)) - \{u, v\}$, and let G' be the subgraph of G induced by V'. Each maximal biclique b in G that contains edge (u, v) corresponds to a unique maximal biclique in G' by simply deleting vertices u and v from b. Also, each maximal biclique b' in G' corresponds to a unique maximal biclique in G that contains (u, v) by adding vertices u and v to b'. Thus, there is a bijection between the maximal bicliques in G' and the set of maximal bicliques in G that contains edge (u, v). The number of maximal bicliques in G' can be at most $g(n-2)$ since G' has no more than $(n-2)$ vertices, completing the proof. □

Lemma 27. *For a bipartite graph* $G = (L, R, E)$ *on* n *vertices and edge* $e = (u, v) \notin E(G)$, *the size of* $\Upsilon(G, G + e)$ *can be no larger than* $3g(n-2)$.

Proof. Proof by contradiction. Suppose there exists a bipartite graph $G = (L, R, E)$ and edge $e \notin E(G)$ such that $|\Upsilon(G, G + e)| > 3g(n-2)$. Then either $|\mathcal{BC}(G + e) - \mathcal{BC}(G)| > g(n-2)$ or $|\mathcal{BC}(G) - \mathcal{BC}(G + e)| > 2g(n-2)$.

Case 1: $|\mathcal{BC}(G + e) - \mathcal{BC}(G)| > g(n-2)$: This means that total number of new maximal bicliques formed due to addition of edge e is larger than $g(n-2)$. Note that each new maximal biclique formed due to addition of e must contain e. From Observation 5, the total number of maximal bicliques in an n vertex bipartite graph containing a specific edge can be at most $g(n-2)$. Thus, the number of new maximal bicliques after adding edge e is at most $g(n-2)$, contradicting our assumption.

Case 2: $|\mathcal{BC}(G) - \mathcal{BC}(G + e)| > 2g(n-2)$: Using Lemma 26, each maximal biclique $b \in \mathcal{BC}(G) - \mathcal{BC}(G + e)$ must contain either u or v, but not both. Suppose that b contains u but not v. Then, b must be a maximal biclique in $G - v$. Using Observation 4, we see that the number of maximal bicliques in $G - v$ that contains a specific vertex u is no more than $g(n-2)$. In a similar way, the number of possible maximal bicliques that contain v is at most $g(n-2)$. Therefore, the total number of maximal bicliques in $\mathcal{BC}(G) - \mathcal{BC}(G+e)$ is at most $2g(n-2)$, contradicting our assumption. □

Combining Lemma 24, Theorem 11 and using the fact that $3g(n-2) = 1.5g(n)$ for even n, we obtain the following when n is even:

Theorem 12. $1.5g(n) \leq \lambda(n) \leq 2g(n)$

6.5 Experimental Evaluation

In this section, we present results of an experimental evaluation of our algorithms.

6.5.1 Datasets

Table 6.1: Summary of the input graphs.

Dataset	Nodes	Edges (start)	Edges (stop)	Edges (original graph)
`epinions-rating-init`	$876,252$	0	$1,210,000$	$13,668,320$
`lastfm-song-init`	$1,085,612$	0	$361,500$	$4,413,834$
`movielens-10M-init`	$80,555$	0	$8,500$	$10,000,054$
`wiktionary-init`	$2,123,868$	0	$235,000$	$5,573,038$

We consider the following real-world bipartite graphs in our experiments. A summary of the datasets is presented in Table 6.1. We collect all the datasets from KONECT - The Koblenz Network Collection [2]. In the `epinions-rating` [1] graph, vertices consist of users in one partition and products in another partition. There is an edge between a user and a product if the user rated that product. In the `lastfm-song` [2] graph, vertices consist of users in one partition and the songs in another partition. When a user listens to a song an edge connect the user with that song. In the `movielens-10M` [3] graph, vertices consist of users in one partition and movies in another partition. There is an edge between a user and a movie if the user rated that movie. In the `wiktionary` [4] graph, vertices consist of users and pages from English Wiktionary. There is an edge between a user

[2]http://konect.uni-koblenz.de/

and a page if that user edited the page. Each bipartite graph has timestamps on their edges. We converted each bipartite graph into a simple undirected bipartite graph by ignoring edge directions, and considered the earliest creation time of an edge as the timestamp, if there are multiple edges in the original graph.

We create the initial graphs (`epinions-rating-init`, `lastfm-song-init`, `movielens-10M-init`, and `wiktionary-init`) by removing all the edges from the original graphs and present the edge stream in the increasing order of their time-stamps. In Table 6.1, column `Edges (start)` represents the number of edges in the initial graph and column `Edges (stop)` represents the total number of edges inserted until we stop the experiment. For each input graph we run each algorithm upto 2 hours.

6.5.2 Experimental Setup and Implementation Details

We implemented all algorithms in Java on a 64-bit Intel(R) Xeon(R) CPU clocked at 3.10 Ghz and 8G DDR3 RAM with 6G heap memory space. Unless otherwise specified, each batch consists of 100 edges and size threshold $s = 1$, where the size threshold refers to the minimum size of each partition of a maximal biclique. We report the median of 3 runs for each input graph.

Metrics: We evaluate our algorithms using the following metrics: (1) computation time for new maximal bicliques and subsumed bicliques when a set of edges are added, (2) memory consumption, that is the main memory used by the algorithm for storing the graph, and other data structures used by the algorithm, (3) cumulative time, that is the total computation time from the initial graph till we stop the experiment with different batch sizes, and (4) change-sensitiveness, the relation of the total computation time to the size of change. We measure the size of change as the sum of the total number of edges in the new maximal bicliques and the subsumed bicliques (change-in-edges) as well as the sum of the total number of nodes (change-in-nodes) and

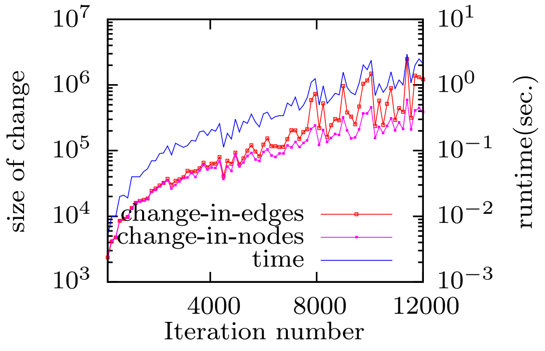
Table 6.2: For each algorithm, the number shows the cumulative computation time for the number (in the parenthesis) of batch additions incrementally.

Initial-graph	DynamicBC	BaselineBC	BaselineBC*
epinions-rating-init (424)	2 sec.	7, 200 sec.	17 sec.
lastfm-song-init (625)	93 sec.	7, 920 sec.	1, 740 sec.
movielens-10M-init (58)	7 min.	out of memory after 23 min.	10.8 min.
wiktionary-init (494)	8 min.	out of memory after 96 min.	149.15 min.

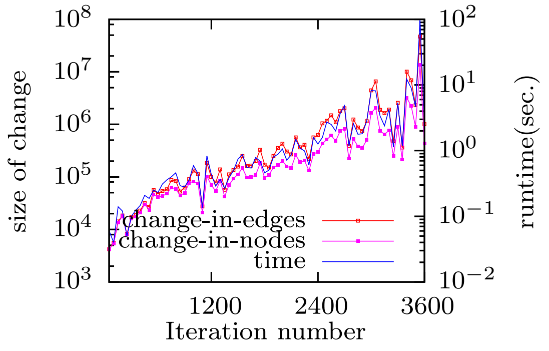
6.5.3 Discussion of Results

Comparison with Baseline Algorithms: We compare the performance of DynamicBC with baseline algorithms BaselineBC and BaselineBC*. We use MineLMBC [92] for enumerating bicliques from a static graph. Table 6.2 shows a comparison of the runtimes of DynamicBC with BaselineBC and BaselineBC*. From the table, it is clear that DynamicBC is orders of magnitude faster than BaselineBC and many times faster than BaselineBC*. For instance, for adding 625 batches of edges starting from lastfm-song-init, DynamicBC takes about 93 sec., BaselineBC about 7, 920 sec., and BaselineBC* about 1, 740 sec.

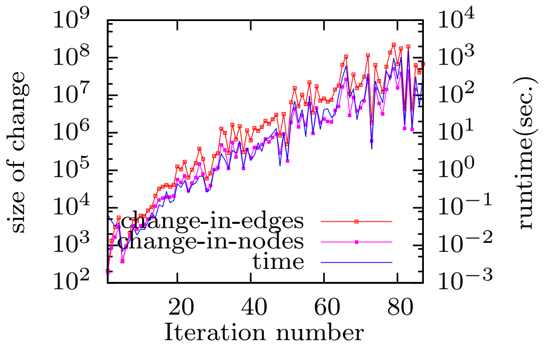
Computation Time per Batch of Edges: Fig. 6.6 shows the computation time (per batch) versus iteration number where one batch is added in each iteration. From the plots, we observe an increasing trend in computation time with the iteration number. There are two reasons for this. One is that with more iterations, the graph becomes denser, and the average degree increases. This contributes to the runtime of computing new maximal bicliques, as is predicted by theory (Theorem 8). Another reason is that the size of change in the maximal bicliques typically increases as more edges are added to the graph, as can be seen from the figure. Whenever the size of change in maximal bicliques drops, the computation time also drops. Fig. 6.7 shows the breakdown of computation time of DynamicBC into time taken for enumerating new bicliques (NewBC) and time taken for enumerating subsumed bicliques (SubBC). Observe that the average computation time (where the average is taken over a range of iterations) increases for both new maximal bicliques and subsumed bicliques as more batches are added, for the same reasons as above.



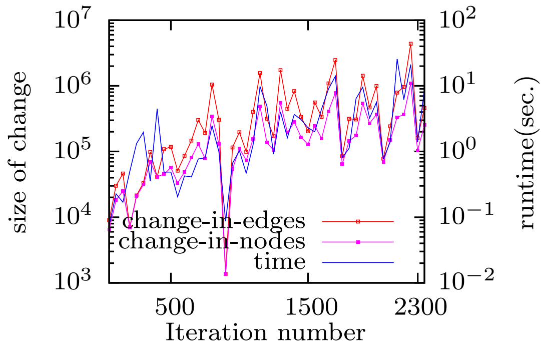
(a) `epinions-rating-init`



(b) `lastfm-song-init`

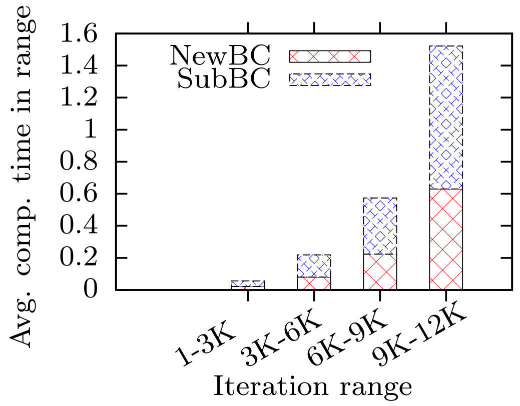


(c) `movielens-10M-init`

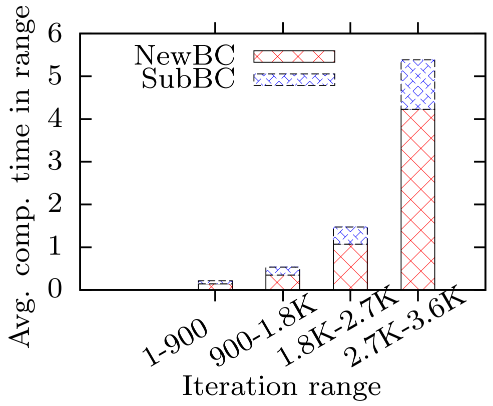


(d) `wiktionary-init`

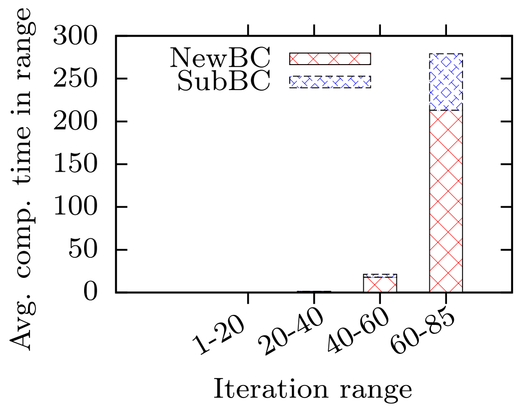
Figure 6.6: Computation time (in sec.) for total change vs. size of total change. The left y-axis shows the change and the right y-axis shows the computation time.



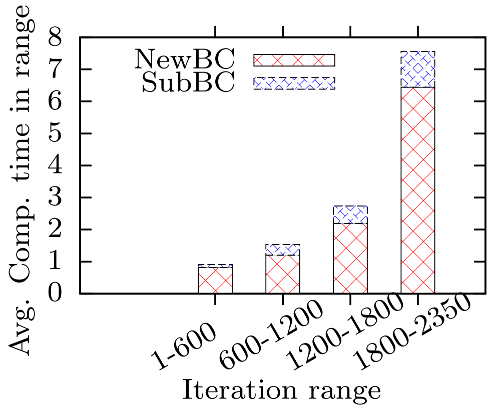
(a) `epinions-rating-init`



(b) `lastfm-song-init`



(c) `movielens-10M-init`



(d) `wiktionary-init`

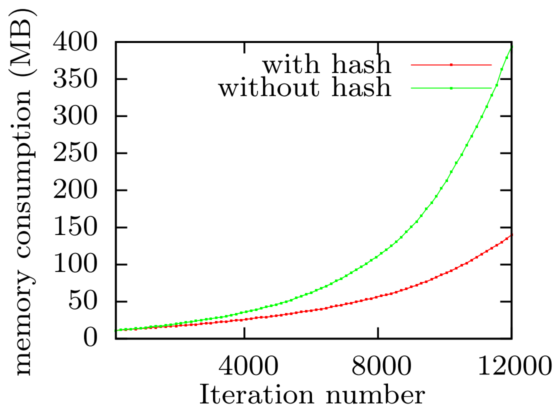
Figure 6.7: Computation time (in sec.) broken down into time for new and subsumed bicliques.

Change-Sensitiveness: Fig. 6.6 shows the computation time and the size of change, as measured in terms of the number of nodes as well as number of edges. We observe that the computation time increases as the size of change increases and decreases as the size of change decreases. But the relationship is not exactly linear. This is because the computation time depends also on the degree of vertices of the graph, which increases as more edges are inserted.

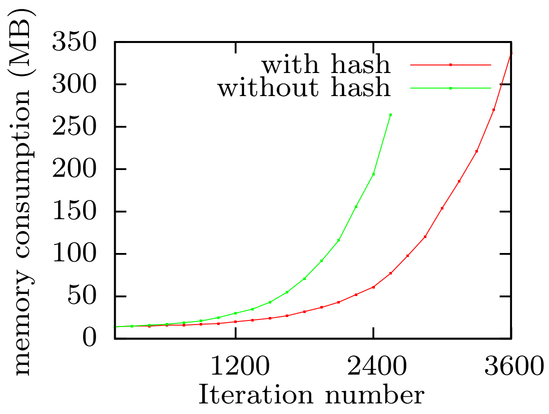
Memory Consumption: Fig. 6.8 shows the memory consumption of `DynamicBC`. Since `SubBC` needs to maintain the maximal bicliques in memory for computing subsumed bicliques, we report the memory consumption in two cases: (1) when the maximal bicliques are stored in memory, (2) when hash signatures of maximal bicliques are stored in memory. Storing signatures consumes less memory than storing actual bicliques (by storing the node sets) as the signatures have fixed size (64 bits) no matter the size of the bicliques. This difference in memory is also clear in the plots. The difference in memory consumption is not prominent during the initial iterations because the sizes of maximal bicliques are much smaller during initial iterations.

Effect of Batch Size on Cumulative Computation Time: Table 6.3 shows the cumulative computation time for different graphs when we use different batch sizes. We observe that overall the total computation time increases when the batch size increases. The reason is the computation time for subsumed bicliques, which increases with increasing batch size, while the computation time for the new maximal bicliques remains almost the same across different batch sizes except `movielens-10M-init`. In case of `movielens-10M-init`, we observed that the time for computing new maximal bicliques also increased with the batch size. The reason is that the algorithm for new maximal bicliques can enumerate the same (new) maximal biclique multiple times, for considering new edges. Such duplicates are suppressed before emitting, but contribute to additional runtime. For this graph, the number of duplicates increased considerably for a batch size of say, 100.

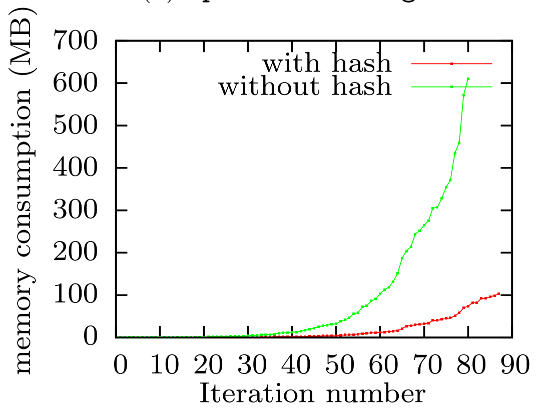
The time complexity for `SubBC` has (in the worst case) an exponential dependence on the batch size. Therefore, the computation time for subsumed bicliques tends to increase with an increase in the batch size. However, with a very small batch size (such as 1 or 10), the cost of enumerating subsumed bicliques was mostly dominated by the cost of enumerating new maximal bicliques.



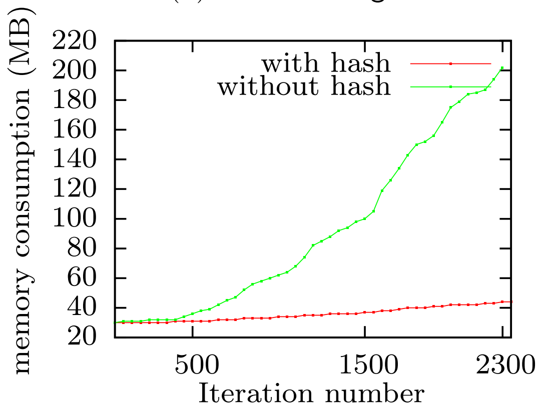
(a) `epinions-rating-init`



(b) `lastfm-song-init`



(c) `movielens-10M-init`



(d) `wiktionary-init`

Figure 6.8: Memory consumption (in MB) with and without using hash function.

Table 6.3: Total computation time in hours for different batch sizes. The total time is split into two numbers. The first number is the time for new maximal bicliques and the second number is the time for subsumed maximal bicliques.

Initial-graph	**batch-size**-1	**batch-size**-10	**batch-size**-100
`epinions-rating-init`	1 (0.9 + 0.1)	1 (0.8 + 0.2)	2 (0.8 + 1.2)
`lastfm-song-init`	1.5 (1.45 + 0.09)	1.8 (1.5 + 0.3)	1.9 (1.5 + 0.4)
`movielens-10M-init`	0.4 (0.36+0.04)	0.6 (0.5 + 0.1)	2.1 (1.6 + 0.5)
`wiktionary-init`	1.8 (1.7 + 0.1)	1.8 (1.7 + 0.1)	2 (1.7 + 0.3)

Table 6.4: Total computation time in hour by varying the threshold size s.

Initial-graph	$s = 2$	$s = 4$	$s = 6$	$s = 8$	$s = 10$	$s = 12$
`epinions-rating-init`	1.2	0.9	0.7	0.4	0.3	0.3
`lastfm-song-init`	1.6	1.3	0.7	0.4	0.3	0.3
`movielens-10M-init`	2.1	1.9	1.6	0.9	0.3	0.1
`wiktionary-init`	1.5	1	0.7	0.5	0.4	0.4

Effect of Size Threshold on Computation Time: We also consider maintaining maximal bicliques with specified size threshold s, where it is required that each bipartition has size at least s. Table 6.4 shows the cumulative computation time by varying the threshold s. As expected, the cumulative computation time decreases as the size threshold s increases, since there is more pruning possible during the depth-first-search performed by Algorithm `MineLMBC`.

CHAPTER 7. PARALLEL MAXIMAL BICLIQUE ENUMERATION ON STATIC BIPARTITE GRAPH

7.1 Introduction

The runtime of sequential algorithms for MBE can be high on large and complex graphs. For example, `MineLMBC` takes approximately 11 hours to enumerate 5.2 million maximal bicliques from a bipartite `IMDB` network with 1.2 million vertices and 3.8 million edges in an 8 core Intel E5 2650 processor. The runtimes will naturally be even higher for larger and denser graphs, and this can be a bottleneck for analysis of bicliques in large bipartite graphs. The natural way to improve the turnaround time is to use parallel computing.

In this work we develop shared-memory parallel algorithms for MBE, which can use the power of a multicore machine to speedup enumeration of maximal bicliques from a graph. Shared memory machines are now commonplace, and machines with tens to hundreds of cores and hundreds of gigabytes are readily available. The shared-memory parallel platform offers certain advantages over a distributed memory platform for problems on graph analysis. On a shared-memory machine, the graph does not have to be partitioned across processors, like in a distributed memory system. Graph partitioning, which is itself a complex task, can thus be avoided. Further, there is no need to communicate using messages between nodes. Communication among tasks can be achieved using reading and writing from shared memory, which is much cheaper than message passing.

Roadmap: The rest of the sections are organized as follows. We present preliminaries in Section 7.2, followed by parallel algorithms Section 7.3, and experimental evaluation in Section 7.4.

7.2 Preliminaries

We consider simple undirected bipartite graph $G = (L, R, E)$ where L and R are two partitions and $E \subseteq L \times R$. The set of vertices adjacent to a vertex v is denoted by $\Gamma(v)$ and the set of vertices common to all the vertices in the set X is denoted by $\Gamma(X)$. Mathematically, $\Gamma(v) = \{u|(u,v) \in E\}$ and $\Gamma(X) = \{u|\forall x \in X, (u,x) \in E\}$. We denote by $\Gamma_2(v)$ all the vertices reachable in 2 hop from v and by $\deg(v)$ the number of vertices adjacent to v. Let d denote the maximum degree of the graph G, M denote the number of maximal biclique in G, and M_v denote the number of maximal bicliques in G containing a particular vertex v.

Sequential Algorithm `MineLMBC`: The algorithm `MineLMBC` enumerates all maximal bicliques of a simple undirected graph G by exploring the graph in a depth-first manner. Each node in the search tree generates a maximal biclique and spawns child nodes by adding the vertices to the current set X (which a bipartition of the current maximal biclique) one at a time from the set $\texttt{tail}(X)$ which is a set of candidate vertices for generating maximal bicliques from X often called tail vertices of X. For generating all maximal bicliques when G is a bipartite graph, $\texttt{tail}(X)$ is initialized with the smaller bipartition, $\Gamma(X)$ with the larger bipartition, X with an empty set, and minimum size $ms = 1$. `MineLMBC` is formally described in Algorithm 17.

```
Algorithm 17: MineLMBC($X, \Gamma(X), \texttt{tail}(X), ms$)
Input: $X$ - vertex set, $\Gamma(X)$ - adjacency list of $X$
$\texttt{tail}(X)$ - tail vertices of $X$
$ms$ - minimum size threshold.
Output: $B$ - Set of all maximal bicliques containing $X$.
1  for $v \in \texttt{tail}(X)$ do
2      if $(|\Gamma(X \cup \{v\})| < ms)$ then
3          $\texttt{tail}(X) \leftarrow \texttt{tail}(X) \setminus \{v\}$
4  if $|X| + |\texttt{tail}(X)| < ms$ then
5      return
6  sort vertices of $\texttt{tail}(X)$ into ascending order of $|\Gamma(X \cup \{v\})|$
7  for $v \in \texttt{tail}(X)$ do
8      $\texttt{tail}(X) \leftarrow \texttt{tail}(X) \setminus \{v\}$
9      if $|X \cup \{v\}| + |\texttt{tail}(X)| > ms$ then
10         $Y \leftarrow \Gamma(\Gamma(X \cup \{v\}))$
11         if $Y \setminus (X \cup \{v\}) \subseteq \texttt{tail}(X)$ then
12             if $|Y| \geq ms$ then
13                 $B \leftarrow B \cup < Y, \Gamma(X \cup \{v\}) >$
14             MineLMBC($Y, \Gamma(X \cup \{v\}), \texttt{tail}(X) \setminus Y, ms$)
```

The time complexity of generating all maximal bicliques in a bipartite graph G using `MineLMBC` is $O(ndM)$ where n is the size of the smaller bipartition of G and other notations carry their usual meaning.

In the analysis of our parallel algorithms, we will use parallel cost model as described in Section 5.2.

7.3 Parallel MBE Algorithms

In this section, we design two shared memory parallel algorithms for MBE. The first algorithm `ParLMBC` is inspired by the state-of-the-art output sensitive algorithm `MineLMBC` and the second algorithm `ParMBE` is inspired by the state-of-the-art distributed algorithm `CDFS` and our parallel algorithm `ParLMBC`. Although `ParLMBC` is a theoretically work-efficient parallel algorithm, algorithm `ParMBE` is practically much faster than `ParLMBC` because it subdivide the problem into multiple tasks per vertex and reduces the overall time by significantly reducing the size of the `tail` set per task basis as opposed to the larger `tail` set in the parallel recursive calls in `ParLMBC`. We will discuss about `ParMBE` followed by the discussion on `ParLMBC` which is the subroutine for enumerating maximal bicliques in tasks per vertex in the algorithm `ParMBE`.

7.3.1 Algorithm `ParLMBC`

`ParLMBC` is an work-efficient parallelization of the sequential algorithm `MineLMBC` (Algorithm 17) that consists of three main components: (1) pruning of the `tail` set (Lines 1-3), (2) sorting the vertices in the tail set (Line 6), and (3) recursive exploration of the search space in depth-first order (Lines 7-14) where each iteration corresponds to the exploration of a sub search space. Now we explain how we parallelize each of these components:

Parallel pruning of the `tail` set: Within a single call to `MineLMBC`, pruning on the `tail` vertex set is performed in parallel in a straight forward manner: iterate over the vertices in the `tail` set in parallel and remove those that fails to satisfy the threshold size criteria as in Line 2 of Algorithm 17.

The total work of this step is $O(nd)$ following the analysis in the sequential algorithm description and the depth is $O(1)$ following the depth of intersection of two sets as discussed in Chapter 5 (Lemma 16).

Parallel sorting the vertices in the `tail` set: Sorting the vertices in a set using a parallel sorting algorithm is difficult to achieve because parallel sorting algorithm works on list data structure such as array or vector where the elements are indexed in the data structure which is not the case when the elements are in an unordered set such as the vertices in the `tail` set in our situation. We overcome this difficulty by putting the elements of `tail` set in an array (assume the array is A) in parallel with identity mapping meaning that $A[v] = v$ only if v is in `tail` set and $A[v] = 0$ otherwise. Next we apply parallel filter operation on the array to compact it (an array containing only the elements in the pruned `tail` set) and assume that the resulting array is A'. Next we apply parallel sorting algorithm to sort the elements in A' with comparison on $\Gamma(\cdot)$ instead of the absolute values of the vertices in the tail set. Finally we generate the sorted array consisting of the vertices in pruned `tail` set. The total work of this step is $O(n \log n)$ which is a combination of array A' construction step with total work $O(n)$ and sorting step with total work $O(n \log n)$ and the depth is $O(\log n)$ which consists of the $O(\log n)$ depth for the construction of A' and $O(\log n)$ depth for parallel sorting.

Parallel unrolling the iterative recursion: We first observe that there is a sequential dependency in the iterations because of the update process of the `tail` vertex set as in Line 8 of Algorithm 17. Therefore, it is not straightforward to execute on each vertex (at Line 7 of Algorithm 17) in parallel. Also, we observed that we can run the iterations in parallel if the dependency of `tail` set can be removed. We exactly do this by creating a local `tail` set for each iteration and initializing it with the vertices from index $i + 1$ till κ where κ is the size of `tail` set before the iterations begin and i is the current iteration number if presented sequentially. The total work of this step is $O(n + d^2)$ which is a combination of (1) updating the `tail` set with total work $O(n)$,

(2) constructing the Y with total work $O(d^2)$, and (2) subset check as in Line 11 of Algorithm 17 with total work $O(n)$. The depth of this step is $O(d^2)$ which is a combination of the depths of these 3 aforementioned components: $O(1)$ for updating `tail` set, $O(d^2)$ for computing Y, and $O(1)$ for subset check.

Parallel computation of Y: We compute $\Gamma(\Gamma(X \cup \{v\}))$ by iterating on each vertex in $\Gamma(X \cup \{v\})$ in parallel. For each vertex u adjacent to some vertex in $\Gamma(X \cup \{v\})$ we use an `atomic` counter and increment by one for each vertex w in $\Gamma(X \cup \{v\})$ adjacent to u. For doing this, we use a hashmap with vertices as `key` and its counter as the `value`. Finally, we consider those vertices from the map in the set Y with counter value $|\Gamma(X \cup \{v\})|$. More details are in Section 7.4.2.

```
Algorithm 18: ParLMBC($X, \Gamma(X)$, tail($X$), $ms$)
   Input: $X$ - vertex set, $\Gamma(X)$ - adjacency list of $X$
   tail($X$) - tail vertices of $X$, $ms$ - minimum size threshold.
   Output: $B$ - Set of all maximal bicliques containing $X$.
1  for $v \in$ tail($X$) do in parallel
2      if $(|\Gamma(X \cup \{v\})| < ms)$ then
3          tail($X$) $\leftarrow$ tail($X$) $\setminus \{v\}$
4  if $|X| + |$tail($X$)$| < ms$ then
5      return
6  parallel sort vertices of tail($X$) into ascending order of $|\Gamma(X \cup \{v\})|$
7  Let the elements of sorted tail($X$) are presented in the order $0..\kappa$
8  for $i \in [0..\kappa]$ do in parallel
9      ntail($X$) $\leftarrow$ $tail[i + 1..\kappa]$
10     if $|X \cup \{v\}| + |$ntail($X$)$| > ms$ then
11         $Y \leftarrow \Gamma(\Gamma(X \cup \{v\}))$ in parallel
12         if $Y \setminus (X \cup \{v\}) \subseteq$ ntail($X$) then
13             if $|Y| \geq ms$ then
14                 $B \leftarrow B \cup < Y, \Gamma(X \cup \{v\}) >$
15             ParLMBC($Y, \Gamma(X \cup \{v\})$, ntail($X$) $\setminus Y, ms$)
```

The formal description of the parallel techniques is presented in Algorithm 18 and we present the work and depth analysis of `ParLMBC` in the following theorem:

Theorem 1. *Given a bipartite graph $G = (L, R, E)$ with $n = |L| \leq |R|$, the number of maximal bicliques M, and the maximum degree d, the total work of* `ParLMBC` *is $O(ndM)$ and the depth of the algorithm is $O(d(d^2 + \log n))$.*

Proof. From the discussion on the parallel steps, it is easy to see that the total work of `ParLMBC` is $O(ndM)$.

To show the depth of the algorithm, note that the overall depth is the depth of a single recursive call multiplied by the depth of the search tree. From the previous discussion of the depth of the individual parallel steps, it is clear to see that the depth of a single recursive call is $O(d^2 + \log n)$. Now the depth of the search tree is the maximum degree of the graph d. This is because, the size of X increases by at least 1 when the depth of the search tree is increased by 1 (Line 11 of Algorithm 18). For contradiction assume that the depth of the search tree is $d+1$. Then there will be an X at depth $d+1$ with at least $d+1$ vertices. This is a contradiction because the maximum degree of the graph d. Thus, the depth of the algorithm follows. □

7.3.2 Algorithm `ParMBE`

While `ParLMBC` is an work-efficient (with respect to `MineLMBC`) parallel algorithm, we design another efficient parallel algorithm inspired by the distributed algorithm `CDFS`. The high level idea is to reduce the size of the `tail` set in each of the recursive calls in `ParLMBC`. We do this by dividing the entire problem into multiple subproblems with one problem per vertex where the goal is to enumerate all maximal bicliques where that vertex is the least among all the vertices in that partition.

Based on this high level idea we present a parallel algorithm `ParMBE` that works in the following way: For each vertex $v \in V(G)$, we create a subgraph G_v consisting of the vertices in the set $\Gamma_2(v)$ and enumerate all maximal bicliques from G_v using our parallel algorithm `ParLMBC`. While working on the subproblems, it is important not to enumerate a maximal bicliques more than once. We ensure this by assuming a total ordering of the vertices and initializing the `tail` set for each subproblem with the vertices that comes after v in that ordering and belongs to the partition of

v. We create this ordering by defining a `rank` function based on which we order the vertices. In this work our `rank` function is based on the degree of the vertices where for any two vertices u and v, $\texttt{rank}(u) > \texttt{rank}(v)$ if $\deg(u) > \deg(v)$ and when $\deg(u) = \deg(v)$, $\texttt{rank}(u) > \texttt{rank}(v)$ if the absolute value of u is greater than the absolute value of v. `ParMBE` is presented formally in Algorithm 19

```
Algorithm 19: ParMBE($G, ms$)
  Input: $G = (L, R, E)$ - input graph, $ms$ - minimum size threshold.
  Output: $B$ - Set of all maximal bicliques containing $X$.
1 for $v \in L$ do in parallel
2     $X \leftarrow \{v\}$
3     $\Gamma(X) \leftarrow \Gamma(v)$
4     tail($X$) $\leftarrow \emptyset$
5     for $w \in \Gamma(v)$ do in parallel
6         for $y \in \Gamma(w)$ do in parallel
7             if rank($y$) > rank($v$) then
8                 tail($X$) $\leftarrow$ tail($X$) $\cup \{y\}$
9     ParLMBC($X, \Gamma(X)$, tail($X$), $ms$)
```

7.4 Experiments

we empirically evaluate our parallel algorithms `ParLMBC` and `ParMBE` and compare with the state-of-the-art sequential and parallel algorithms `MineLMBC` and `CDFS` respectively on real world bipartite networks to show the parallel speedup and scalability of our parallel algorithms. We show that our practical efficient parallel algorithm `ParMBE` provides magnitude of order speedup compared with `MineLMBC` and better runtime compared with distributed parallel algorithm `CDFS`. We evaluate all the experiments on a Intel 16 core machine where each core is a Xeon(R) CPU E5 2650@2.0GHz processor with 128 GB main memory in the entire system.

7.4.1 Datasets

We use 11 real world static bipartite networks from publicly available repository KONECT [83] for the experiments. The summary of the dataset is presented in Table 7.1.

Table 7.1: Static Bipartite Networks used for evaluation, and their properties.

Dataset	#Vertices	#Edges	#Maximal Bicliques
`Writers`	135,568	144,340	57,222
`Actors(DBpedia)`	157,183	281,396	119,868
`DBpedia locations`	225,486	293,697	75,360
`Record labels`	186,689	233,286	42,393
`Marvel`	19,428	96,662	206,135
`CiteSeer`	286,748	512,267	171,354
`YouTube`	124,325	293,360	1,826,587
`Occupations`	229,301	250,945	104,974
`IMDB`	1,199,919	3,782,463	5,160,061
`Stack Overflow`	641,873	1,301,942	3,320,824
`BookCrossing`	445,801	1,149,739	54,458,953

7.4.2 Implementation of the algorithms

In this work we use Intel TBB [20] for all the parallel implementations which extends C++ for writing parallel implementations in an efficient manner by including algorithms, highly concurrent containers, locks and atomic operations, dynamic work stealing scheduler, and a scalable memory allocator and providing easy to use APIs for using those in the parallel implementations. In this work we use some of these that we explain below. We use `parallel_for` and `parallel_for_each` APIs for the implementation of the `parallel for` loop. For the atomic operations on hashtable we use `concurrent_hash_map`, for the atomic operations on unordered set we use `concurrent_unordered_set`, and for atomic operations on the dynamic array we use `concurrent_vector`. We implement $\Gamma(\Gamma(X \cup \{v\}))$ in parallel in `ParLMBC`. For doing this, we use a `concurrent_hash_map` from vertex as the `key` and the the number of vertices in the set $\Gamma(X \cup \{v\})$ it is adjacent to as the `value`. Then we iterate on the vertices of $\Gamma(X \cup \{v\})$ in parallel and update the frequency of the vertices adjacent to each vertex in $\Gamma(X \cup \{v\})$. Finally, we generate the set Y with the vertices in the `concurrent_hash_map` whose frequency is $|\Gamma(X \cup \{v\})|$. For the parallel sort we use `parallel_sort`. All of these are provided by TBB. We use C++11 for the implementation of the algorithms and compile the sources using Intel ICC compiler version 18.0.3 with optimization level '-O3'. System level load balancing is performed using a dynamic work stealing scheduler [20] built inside TBB.

For the comparison with prior works, we implement state of the art MapReduce algorithm `CDFS` in shared memory setting that we call `MCoreCDFS`. We also implement a more recent sequential algorithm `iMBEA` [159].

7.4.3 Discussion of the Results

Now we present and interpret the results of the empirical evaluations of our parallel algorithms. First we will show the parallel speedup (with respect to `MineLMBC`) and scalability of `ParLMBC` and `ParMBE` to show that the performance of the parallel algorithms improve when the number of core is increased. Next we compare our parallel algorithms with `MCoreCDFS` to show that `ParMBE` performs better than `MCoreCDFS` on all the input graph and then we compare with the sequential algorithm `iMBEA` to show that `ParLMBC` and `ParMBE` are magnitude order faster than `iMBEA`. This is as expected because the performance of `iMBEA` is much worse than the performance of `MineLMBC`.

Parallel Speedup: We show the parallel speedup of `ParLMBC` and `ParMBE` in Table 7.2. The result clearly shows the substantially better performance of `ParMBE` over `ParLMBC` and magnitude of order parallel speedup of `ParMBE` compared with `MineLMBC`. However, more than $16\times$ speedup of `ParMBE` in a 16 core machine clearly indicates that the sequential algorithm `MineLMBC` is not the most efficient one.

Scalability: We show the scalability of our parallel algorithms `ParLMBC` and `ParMBE` in Table 7.3. The result shows the algorithms scale up almost linearly as we increase the degree of parallelism by increasing the number of threads. The x axis is the number of the threads used and the y axis is the parallel speedup which is a function of the number of threads.

Table 7.2: Runtime (in sec.) of MineLMBC, ParLMBC, and ParMBE on 16 cores. Numbers in the parenthesis indicates the parallel speedup.

Dataset	MineLMBC	ParLMBC	ParMBE
Writers	405	91.2 (**4.4x**)	0.12 (**3375x**)
Actors(DBpedia)	1464	295.5 (**5x**)	2.1 (**697x**)
DBpedia locations	568.4	135.2 (**4.2x**)	0.22 (**2583.6x**)
Record labels	53.03	12.9 (**4.11x**)	0.21 (**252.5x**)
Marvel	15.45	4.5 (**3.4x**)	2.1 (**7.3x**)
CiteSeer	2554.4	544.8 (**4.7x**)	0.65 (**3929.8x**)
YouTube	446.97	85.9 (**5.2x**)	45.4 (**9.8x**)
Occupations	1891.46	482.5 (**3.9x**)	0.22 (**8597.5x**)
IMDB	40775.6	7904 (**5.16x**)	621.5 (**65.6x**)
Stack Overflow	34635	4634.9 (**7.5x**)	4296.2 (**8.06x**)
BookCrossing	29229	3894.4 (**7.5x**)	3481 (**8.4x**)

Table 7.3: Scalability of ParMBE with respect to MineLMBC by varying the number of threads.

Dataset	16T	8T	4T	2T	1T
Writers	3375x	2531.2x	1760.9x	1038.5x	675x
Actors(DBpedia)	697x	366x	203.3x	105.3x	54.2x
DBpedia locations	2583.6x	1960x	1291.8x	789.4x	474x
Record labels	252.5x	204x	139.6x	81.6x	48x
Marvel	7.3x	4.8x	3.2x	1.7x	0.9x
CiteSeer	3929.8x	2240.7x	1351.5x	709.6x	393x
YouTube	9.8x	5.3x	2.9x	1.5x	0.8x
Occupations	8597.5x	5910.8x	3860.1x	2125.2x	1261x
IMDB	65.6x	35x	20.3x	10.7	6.3x
Stack Overflow	8x	4.2x	2.4x	1.2x	0.6x
BookCrossing	8.4x	4.6x	2.6x	1.3x	0.6x

Comparison with prior works: We compare our parallel algorithms `ParLMBC` and `ParMBE` with `MCoreCDFS`. From Table 7.4 we see that `ParMBE` is upto **2x** faster than `MCoreCDFS`. This speedup is due to the use of parallel algorithm `ParLMBC` in `ParMBE` for enumerating the maximal biclique from the subproblems instead of `MineLMBC` as in `MCoreCDFS`. We also evaluate a prior sequential algorithm `iMBEA` and it appears that `MineLMBC` is significantly faster over `iMBEA`. For instance, `MineLMBC` takes around 15 seconds to enumerate around 206K maximal bicliques where as `iMBEA` takes more than 30 minutes to enumerate those maximal bicliques. In another example, `MineLMBC` takes around 560 seconds to enumerate around 75K maximal bicliques where as `iMBEA` takes more than an hour for doing exactly the same job.

Table 7.4: Comparison of runtime (in sec.) of `ParMBE` with `CDFS` on 16 cores.

Dataset	CDFS	ParMBE
Writers	0.13	0.12
Actors(DBpedia)	5.8	**2.1**
DBpedia locations	0.25	0.22
Record labels	0.23	0.21
Marvel	2.7	2.1
CiteSeer	1.2	**0.65**
YouTube	82.3	**45.4**
Occupations	0.3	0.22
IMDB	889.3	**621.5**
Stack Overflow	8866.8	**4296.2**
BookCrossing	6767.8	**3481**

7.4.4 New Sequential Algorithm `FMBE`

Based on the observation from Table 7.2 the parallel speedup of `ParMBE` compared with `MineLMBC` is magnitude of order better than the number of cores (16) in the multicore machine where we execute all the parallel algorithms. Clearly `MineLMBC` is not the optimized sequential algorithm and it indicates a gap between the possibility of a better sequential algorithm and the algorithm `MineLMBC`. We fill the gap by designing a new sequential algorithm `FMBE` where we execute all the procedures in `ParMBE` sequentially. Surprisingly, we find that the time complexity of `FMBE` is better than the time complexity of `MineLMBC` in both the theory and in practice. We formally present `FMBE` in Algorithm 20.

Typically the size of the `tail` set for each subproblem becomes significantly smaller than the size of the `tail` set for the entire graph. Intuitively the significant reduction in the runtime is related to the reduction in the size of the `tail` set in the argument of `MineLMBC` in algorithm `FMBE`. The following lemma shows a better time complexity of `FMBE` than `MineLMBC` when the maximum degree d of the graph is much smaller than the number of vertices n.

Lemma 28. *Given a bipartite graph $G = (L, R, E)$ with $n = |L| \leq |R|$, the number of maximal bicliques M, and the maximum degree d, the time complexity of* `FMBE` *is $O(d^4 M)$.*

Proof. First we show that if $b = (b_L, b_R)$ is a maximal biclique of G where $b_L \subseteq L$ and $b_R \subseteq R$, it will be enumerated from G_v only where v is the least ranked vertex among all the vertices in b_L. Suppose this is not the case, and assume that b is enumerated from another subgraph G_w for some $w \in L$. Then, v is not the least ranked vertex among the vertices of b_L based on the construction of G_w. This is a contradiction.

Now for each vertex $v \in L$, the the size of G_v is $O(d^2)$ because, in constructing G_v, we consider the vertex v, the vertices in $\Gamma(v)$ and the vertices adjacent to each of the vertex in $\Gamma(v)$. The time complexity of `MineLMBC` on the instance of G_v is $O(d^3 M_v)$ where M_v is the number of maximal bicliques in G_v and d is the maximum degree of G_v which is same as the maximum degree of the original graph G. Thus the overall time complexity is $\sum_{v \in L} O(d^3 M_v)$ which is $O(d^4 M)$ because each maximal cliques will be enumerated at most d times by d different subproblems. This completes the proof. □

In Table 7.5 we show that the runtime of `FMBE` is magnitude of order lower than the runtime of `MineLMBC` in almost all the input graphs. This shows that `FMBE` is a significantly better sequential algorithm than `MineLMBC`. In this table we also show that runtime of `ParMBE` on 16 threads to show the consistent parallel speedup compared with `FMBE`.

Algorithm 20: FMBE(G)

```
Input: $G = (L, R, E)$ - input graph
Output: $B$ - Set of all maximal bicliques containing $X$.
1 for $v \in L$ do
2     $X \leftarrow \{v\}$
3     $\Gamma(X) \leftarrow \Gamma(v)$
4     tail$(X) \leftarrow \emptyset$
5     for $w \in \Gamma(v)$ do
6         for $y \in \Gamma(w)$ do
7             if rank$(y) >$ rank$(v)$ then
8                 tail$(X) \leftarrow$ tail$(X) \cup \{y\}$
9     MineLMBC$(X, \Gamma(X),$ tail$(X), 1)$
```

Table 7.5: Comparison of runtime (in sec.) of MineLMBC, FMBE, and ParMBE (on 16 threads). The speedup of ParMBE in the parenthesis is with respect to new sequential algorithm FMBE.

Dataset	MineLMBC	FMBE	ParMBE
Writers	405	0.72	0.12 (**6x**)
Actors(DBpedia)	1464	25.9	2.1 (**12x**)
DBpedia locations	568.4	1.4	0.22 (**7x**)
Record labels	53.03	1.32	0.21 (**6.3x**)
Marvel	15.45	15.76	2.1 (**7.5x**)
CiteSeer	2554.4	6.6	0.65 (**10x**)
YouTube	446.97	426.07	45.4 (**9.4x**)
Occupations	1891.5	1.9	0.22 (**8.6**)
IMDB	40775.6	2654.6	621.5 (**4.3x**)
Stack Overflow	34,635	24,858	4296.2 (**5.8x**)
BookCrossing	29,229	25,718	3481(**7.5x**)

CHAPTER 8. CONCLUSION AND FUTURE WORK

In this work we develop incremental and parallel algorithms for efficient mining of dense structures such as maximal cliques and maximal bicliques that can efficiently process the stream of incoming edges and can utilize the power of multiple cores in a multicore computing system. We theoretically prove the efficiency of our algorithms and design practical algorithms considering the shortcomings of the theoretical designs. Next we implement all the algorithms using efficient data structures and library tools to show that our algorithms are efficient in practice. We also demonstrate through experimental studies that our incremental and parallel algorithms substantially outperforms the state of the art algorithms.

One possible future direction is to extend the incremental algorithmic techniques developed in this work to the other types of dense structures such as quasi-cliques, quasi-bicliques etc. In regards of parallel computations, possible future direction is to design efficient parallel algorithm with provable work-efficiency such as the parallel algorithms developed in this work.

Another direction of further research is in the direction of software development. There are many softwares that are developed either for efficient stream computation or for efficient parallel computation. So, it will be interesting to develop software that can combine both. It will be interesting to design incremental and parallel algorithms such as one developed in this work for other dense structures such as k-core, quasi-cliques etc. which will help in developing robust software for incremental data analysis.

One more direction is to realize massive parallelism. There can be many possibilities such as (1) combining distributed parallel computation and shared memory parallel computations: distribute the workload in multiple nodes and perform shared memory parallel computation in each of the node; (2) combining GPU computation and shared memory parallel computation: combine the power of GPU as well as the shared memory multicore system to efficiently perform the computation utilizing the power of data parallelism as in GPU and the power of instruction level parallelism as in multicore (nested fork-join model).

BIBLIOGRAPHY

[1] Epinions product ratings network dataset – KONECT. http://konect.uni-koblenz.de/networks/epinions-rating, Apr. 2017.

[2] Last.fm song network dataset – KONECT. http://konect.uni-koblenz.de/networks/lastfm_song, Apr. 2017.

[3] Movielens 10m network dataset – KONECT. http://konect.uni-koblenz.de/networks/movielens-10m_rating, Apr. 2017.

[4] Wiktionary (en) network dataset – KONECT. http://konect.uni-koblenz.de/networks/edit-enwiktionary, Apr. 2017.

[5] J. Abello, M. G. Resende, and S. Sudarsky. Massive quasi-clique detection. In *Latin American symposium on theoretical informatics*, pages 598–612. Springer, 2002.

[6] G. Alexe, S. Alexe, Y. Crama, S. Foldes, P. L. Hammer, and B. Simeone. Consensus algorithms for the generation of all maximal bicliques. *Discrete Applied Mathematics*, 145(1):11–21, 2004.

[7] J. I. Alvarez-Hamelin, L. Dall'Asta, A. Barrat, and A. Vespignani. Large scale networks fingerprinting and visualization using the k-core decomposition. In *Advances in neural information processing systems*, pages 41–50, 2006.

[8] R. Andersen. A local algorithm for finding dense subgraphs. *ACM Transactions on Algorithms (TALG)*, 6(4):60, 2010.

[9] R. Andersen and K. Chellapilla. Finding dense subgraphs with size bounds. In *International Workshop on Algorithms and Models for the Web-Graph*, pages 25–37. Springer, 2009.

[10] A. Angel, N. Koudas, N. Sarkas, D. Srivastava, M. Svendsen, and S. Tirthapura. Dense subgraph maintenance under streaming edge weight updates for real-time story identification. *The VLDB Journal*, pages 1–25, 2013.

[11] D. Avis and K. Fukuda. Reverse search for enumeration. *Discrete Applied Mathematics*, 65:21–46, 1993.

[12] B. Bahmani, R. Kumar, and S. Vassilvitskii. Densest subgraph in streaming and mapreduce. *VLDB*, 5(5):454–465, 2012.

[13] S. Barman. Approximating nash equilibria and dense bipartite subgraphs via an approximate version of caratheodory's theorem. In *Proceedings of the forty-seventh annual ACM symposium on Theory of computing*, pages 361–369. ACM, 2015.

[14] V. Batagelj and M. Zaversnik. An o (m) algorithm for cores decomposition of networks. *arXiv preprint cs/0310049*, 2003.

[15] A. Benshahar, V. Chalifa-Caspi, D. Hermelin, and M. Ziv-Ukelson. A biclique approach to reference-anchored gene blocks and its applications to genomic islands. *Journal of Computational Biology*, 25(2):214–235, 2018.

[16] S. Bhattacharya, M. Henzinger, D. Nanongkai, and C. Tsourakakis. Space-and time-efficient algorithm for maintaining dense subgraphs on one-pass dynamic streams. In *Proceedings of the forty-seventh annual ACM symposium on Theory of computing*, pages 173–182. ACM, 2015.

[17] M. Bhattacharyya and S. Bandyopadhyay. Mining the largest quasi-clique in human protein interactome. In *Adaptive and Intelligent Systems, 2009. ICAIS'09. International Conference on*, pages 194–199. IEEE, 2009.

[18] G. E. Blelloch and B. M. Maggs. Parallel algorithms. In *Algorithms and theory of computation handbook*, pages 25–25, 2010.

[19] R. D. Blumofe. *Executing multithreaded programs efficiently*. PhD thesis, Massachusetts Institute of Technology, 1995.

[20] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM (JACM)*, 46(5):720–748, 1999.

[21] P. Boldi and S. Vigna. The webgraph framework i: compression techniques. In *Proceedings of the 13th international conference on World Wide Web*, pages 595–602. ACM, 2004.

[22] F. Braun, O. Caelen, E. N. Smirnov, S. Kelk, and B. Lebichot. Improving card fraud detection through suspicious pattern discovery. In *International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems*, pages 181–190. Springer, 2017.

[23] C. Bron and J. Kerbosch. Algorithm 457: finding all cliques of an undirected graph. *Commun. ACM*, 16(9):575–577, 1973.

[24] M. Brunato, H. H. Hoos, and R. Battiti. On effectively finding maximal quasi-cliques in graphs. In *International conference on learning and intelligent optimization*, pages 41–55. Springer, 2007.

[25] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-mat: A recursive model for graph mining. In *Proceedings of the 2004 SIAM International Conference on Data Mining*, pages 442–446. SIAM, 2004.

[26] M. Charikar. Greedy approximation algorithms for finding dense components in a graph. In *International Workshop on Approximation Algorithms for Combinatorial Optimization*, pages 84–95. Springer, 2000.

[27] A. Chateau, P. Riou, and E. Rivals. Approximate common intervals in multiple genome comparison. In *Bioinformatics and Biomedicine (BIBM), 2011 IEEE International Conference on*, pages 131–134. IEEE, 2011.

[28] Y. Chen and G. M. Crippen. A novel approach to structural alignment using realistic structural and environmental information. *Protein science*, 14(12):2935–2946, 2005.

[29] J. Cheng, Y. Ke, S. Chu, and M. T. Özsu. Efficient core decomposition in massive networks. In *2011 IEEE 27th International Conference on Data Engineering*, pages 51–62. IEEE, 2011.

[30] J. Cheng, Y. Ke, A. W.-C. Fu, J. X. Yu, and L. Zhu. Finding maximal cliques in massive networks. *TODS*, 36(4):21, 2011.

[31] S.-T. Cheng, Y.-C. Chen, and M.-S. Tsai. Using k-core decomposition to find cluster centers for k-means algorithm in graphx on spark. *CLOUD COMPUTING 2017*, page 103, 2017.

[32] N. Chiba and T. Nishizeki. Arboricity and subgraph listing algorithms. *SIAM J. Comput.*, 14:210–223, 1985.

[33] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick. Reachability and distance queries via 2-hop labels. *SIAM Journal on Computing*, 32(5):1338–1355, 2003.

[34] J. Cohen. Trusses: Cohesive subgraphs for social network analysis. *National Security Agency Technical Report*, 16, 2008.

[35] J. Cohen. Graph twiddling in a mapreduce world. *Computing in Science & Engineering*, 11(4):29, 2009.

[36] P. Damaschke. Enumerating maximal bicliques in bipartite graphs with favorable degree sequences. *Information Processing Letters*, 114(6):317–321, 2014.

[37] A. Das, S.-V. Sanei-Mehri, and S. Tirthapura. Shared-memory parallel maximal clique enumeration. *arXiv preprint arXiv:1807.09417*, 2018.

[38] A. Das, M. Svendsen, and S. Tirthapura. Change-sensitive algorithms for maintaining maximal cliques in a dynamic graph. *CoRR*, abs/1601.06311, 2016.

[39] N. S. Dasari, R. Desh, and M. Zubair. Park: An efficient algorithm for k-core decomposition on multicore processors. In *Big Data (Big Data), 2014 IEEE International Conference on*, pages 9–16. IEEE, 2014.

[40] V. M. Dias, C. M. De Figueiredo, and J. L. Szwarcfiter. Generating bicliques of a graph in lexicographic order. *Theoretical Computer Science*, 337(1):240–248, 2005.

[41] V. M. Dias, C. M. de Figueiredo, and J. L. Szwarcfiter. On the generation of bicliques of a graph. *Discrete Applied Mathematics*, 155(14):1826–1832, 2007.

[42] S. N. Dorogovtsev, A. V. Goltsev, and J. F. F. Mendes. K-core organization of complex networks. *Physical review letters*, 96(4):040601, 2006.

[43] A. C. Driskell, C. Ané, J. G. Burleigh, M. M. McMahon, B. C. O'Meara, and M. J. Sanderson. Prospects for building the tree of life from large sequence databases. *Science*, 306(5699):1172–1174, 2004.

[44] N. Du, B. Wu, L. Xu, B. Wang, and X. Pei. A parallel algorithm for enumerating all maximal cliques in complex network. In *ICDM Workshop*, pages 320–324. IEEE, 2006.

[45] N. Du, B. Wu, L. Xu, B. Wang, and P. Xin. Parallel algorithm for enumerating maximal cliques in complex network. In *Mining Complex Data*, pages 207–221. Springer, 2009.

[46] D. Duan, Y. Li, R. Li, and Z. Lu. Incremental k-clique clustering in dynamic social networks. *Artificial Intelligence Review*, pages 1–19, 2012.

[47] A. Epasto, S. Lattanzi, and M. Sozio. Efficient densest subgraph computation in evolving graphs. In *Proceedings of the 24th International Conference on World Wide Web*, pages 300–310. International World Wide Web Conferences Steering Committee, 2015.

[48] D. Eppstein. Arboricity and bipartite subgraph listing algorithms. *Information processing letters*, 51(4):207–211, 1994.

[49] D. Eppstein, M. Löffler, and D. Strash. Listing all maximal cliques in sparse graphs in near-optimal time. In *ISAAC*, pages 403–414, 2010.

[50] D. Eppstein and D. Strash. Listing all maximal cliques in large sparse real-world graphs. In P. Pardalos and S. Rebennack, editors, *Experimental Algorithms*, volume 6630 of *LNCS*, pages 364–375. 2011.

[51] P. Erds and A. Rényi. On the evolution of random graphs. *Publ. Math. Inst. Hung. Acad. Sci*, 5:17–61, 1960.

[52] H. Esfandiari, S. Lattanzi, and V. Mirrokni. Parallel and streaming algorithms for k-core decomposition. *arXiv preprint arXiv:1808.02546*, 2018.

[53] W. Fan, C. Hu, and C. Tian. Incremental graph computations: Doable and undoable. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 155–169. ACM, 2017.

[54] E. Galbrun, A. Gionis, and N. Tatti. Overlapping community detection in labeled graphs. *Data Mining and Knowledge Discovery*, 28(5-6):1586–1610, 2014.

[55] A. Gély, L. Nourine, and B. Sadi. Enumeration aspects of maximal cliques and bicliques. *Discrete applied mathematics*, 157(7):1447–1459, 2009.

[56] D. Gibson, R. Kumar, and A. Tomkins. Discovering large dense subgraphs in massive graphs. In *VLDB*, pages 721–732, 2005.

[57] H. Gmati, A. Mouakher, A. Gonzalez-Pardo, and D. Camacho. A new algorithm for communities detection in social networks with node attributes. *Journal of Ambient Intelligence and Humanized Computing*, pages 1–13, 2019.

[58] A. V. Goldberg. *Finding a maximum density subgraph.* University of California Berkeley, CA, 1984.

[59] A. V. Goltsev, S. N. Dorogovtsev, and J. F. F. Mendes. k-core (bootstrap) percolation on complex networks: Critical phenomena and nonlocal effects. *Physical Review E*, 73(5):056101, 2006.

[60] H. M. Grindley, P. J. Artymiuk, D. W. Rice, and P. Willett. Identification of tertiary structure resemblance in proteins using a maximal common subgraph isomorphism algorithm. *J. Mol. Biol.*, 229(3):707–721, 1993.

[61] R. A. Hanneman and M. Riddle. Introduction to social network methods. `http://faculty.ucr.edu/~hanneman/nettext/`. Textbook on the web.

[62] E. Harley, A. Bonner, and N. Goodman. Uniform integration of genome mapping data using intersection graphs. *Bioinformatics*, 17(6):487–494, 2001.

[63] J. A. Hartigan and M. A. Wong. Algorithm as 136: A k-means clustering algorithm. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 28(1):100–108, 1979.

[64] M. Hattori, Y. Okuno, S. Goto, and M. Kanehisa. Development of a chemical structure comparison method for integrated analysis of chemical and genomic information in the metabolic pathways. *J. Am. Chem. Soc.*, 125(39):11853–11865, 2003.

[65] X. Huang, H. Cheng, L. Qin, W. Tian, and J. X. Yu. Querying k-truss community in large and dynamic graphs. In *SIGMOD*, pages 1311–1322, 2014.

[66] S.-C. Hung, M. Araujo, and C. Faloutsos. Distributed community detection on edge-labeled graphs using spark. In *12th International Workshop on Mining and Learning with Graphs (MLG)*, volume 113, 2016.

[67] M. M.-u. Hussain, A. Wang, and G. Trajcevski. Co-maxrs: Continuous maximizing range-sum query. *Sciences*, 305:110–129, 2015.

[68] A. Java, X. Song, T. Finin, and B. L. Tseng. Why we twitter: An analysis of a microblogging community. In *WebKDD/SNA-KDD*, pages 118–138, 2007.

[69] H. Jin, N. Wang, D. Yu, Q.-S. Hua, X. Shi, and X. Xie. Core maintenance in dynamic graphs: A parallel approach based on matching. *IEEE Transactions on Parallel and Distributed Systems*, 29(11):2416–2428, 2018.

[70] R. Jin, Y. Xiang, N. Ruan, and D. Fuhry. 3-hop: a high-compression indexing scheme for reachability query. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 813–826. ACM, 2009.

[71] D. S. Johnson, M. Yannakakis, and C. H. Papadimitriou. On generating all maximal independent sets. *Information Processing Letters*, 27(3):119–123, 1988.

[72] P. F. Jonsson and P. A. Bates. Global topological features of cancer proteins in the human interactome. *Bioinformatics*, 22(18):2291–2297, 2006.

[73] H. Kabir and K. Madduri. Parallel k-core decomposition on multicore platforms. In *Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2017 IEEE International*, pages 1482–1491. IEEE, 2017.

[74] H. Kabir and K. Madduri. Parallel k-truss decomposition on multicore systems. In *High Performance Extreme Computing Conference (HPEC), 2017 IEEE*, pages 1–7. IEEE, 2017.

[75] W. Khaouid, M. Barsky, V. Srinivasan, and A. Thomo. K-core decomposition of large networks on a single pc. *Proceedings of the VLDB Endowment*, 9(1):13–23, 2015.

[76] A. Khosraviani and M. Sharifi. A distributed algorithm for γ-quasi-clique extractions in massive graphs. In *Innovative Computing Technology*, pages 422–431. Springer, 2011.

[77] S. Khuller and B. Saha. On finding dense subgraphs. In *International Colloquium on Automata, Languages, and Programming*, pages 597–608. Springer, 2009.

[78] I. Koch. Enumerating all connected maximal common subgraphs in two graphs. *Theoretical Computer Science*, 250(1):1–30, 2001.

[79] S. Koichi, M. Arisaka, H. Koshino, A. Aoki, S. Iwata, T. Uno, and H. Satoh. Chemical structure elucidation from 13c nmr chemical shifts: Efficient data processing using bipartite matching and maximal clique algorithms. *JCIM*, 54(4):1027–1035, 2014.

[80] D. Koller and N. Friedman. *Probabilistic Graphical Models: Principles and Techniques*. MIT Press, 2009.

[81] F. Kose, W. Weckwerth, T. Linke, and O. Fiehn. Visualizing plant metabolomic correlation networks using clique-metabolite matrices. *Bioinformatics*, 17(12):1198–1208, 2001.

[82] R. Kumar, P. Raghavan, S. Rajagopalan, and A. Tomkins. Trawling the web for emerging cyber-communities. *Computer networks*, 31(11):1481–1493, 1999.

[83] J. Kunegis. Konect: the koblenz network collection. In *WWW*, pages 1343–1350. ACM, 2013.

[84] A. Kyrola, G. Blelloch, and C. Guestrin. Graphchi: Large-scale graph computation on just a {PC}. In *Presented as part of the 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*, pages 31–46, 2012.

[85] S. Lehmann, M. Schwartz, and L. K. Hansen. Biclique communities. *Phys. Rev. E*, 78:016108, Jul 2008.

[86] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, 2014.

[87] B. Lessley, T. Perciano, M. Mathai, H. Childs, and E. W. Bethel. Maximal clique enumeration with data-parallel primitives. In *2017 IEEE 7th Symposium on Large Data Analysis and Visualization (LDAV)*, pages 16–25. IEEE, 2017.

[88] J. Li, H. Li, D. Soh, and L. Wong. A correspondence between maximal complete bipartite subgraphs and closed patterns. In *European Conference on Principles of Data Mining and Knowledge Discovery*, pages 146–156. Springer, 2005.

[89] J. Li, K. Sim, G. Liu, and L. Wong. Maximal quasi-bicliques with balanced noise tolerance: Concepts and co-clustering applications. In *Proceedings of the 2008 SIAM International Conference on Data Mining*, pages 72–83. SIAM, 2008.

[90] R. Li, J. X. Yu, and R. Mao. Efficient core maintenance in large dynamic graphs. *TKDE*, 26(10):2453–2465, 2014.

[91] C. Liang, Y. Li, and J. Luo. A novel method to detect functional microrna regulatory modules by bicliques merging. *IEEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB)*, 13(3):549–556, 2016.

[92] G. Liu, K. Sim, and J. Li. Efficient mining of large maximal bicliques. In *Data warehousing and knowledge discovery*, pages 437–448. Springer, 2006.

[93] G. Liu and L. Wong. Effective pruning techniques for mining quasi-cliques. In *Joint European conference on machine learning and knowledge discovery in databases*, pages 33–49. Springer, 2008.

[94] D. Lo, D. Surian, K. Zhang, and E.-P. Lim. Mining direct antagonistic communities in explicit trust networks. In *CIKM*, pages 1013–1018, 2011.

[95] L. Lu, Y. Gu, and R. Grossman. dmaximalcliques: A distributed algorithm for enumerating all maximal cliques and maximal clique distribution. In *Data Mining Workshops (ICDMW), 2010 IEEE International Conference on*, pages 1320–1327. IEEE, 2010.

[96] L. Lü, T. Zhou, Q.-M. Zhang, and H. E. Stanley. The h-index of a network node and its relation to degree and coreness. *Nature communications*, 7:10168, 2016.

[97] K. Makino and T. Uno. New algorithms for enumerating all maximal cliques. In *SWAT*, pages 260–272. 2004.

[98] A. McGregor, D. Tench, S. Vorotnikova, and H. T. Vu. Densest subgraph in dynamic graph streams. In *MFCS*, pages 472–482, 2015.

[99] D. Miorandi and F. De Pellegrini. K-shell decomposition for dynamic complex networks. In *8th International Symposium on Modeling and Optimization in Mobile, Ad Hoc, and Wireless Networks*, pages 488–496. IEEE, 2010.

[100] N. Mishra, D. Ron, and R. Swaminathan. A new conceptual clustering framework. *Machine Learning*, 56(1-3):115–151, 2004.

[101] M. Mitzenmacher, J. Pachocki, R. Peng, C. Tsourakakis, and S. C. Xu. Scalable large near-clique detection in large-scale networks via sampling. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 815–824. ACM, 2015.

[102] S. Mohseni-Zadeh, P. Brézellec, and J.-L. Risler. Cluster-c, an algorithm for the large-scale clustering of protein sequences based on the extraction of maximal cliques. *Comp. Biol. Chem.*, 28(3):211–218, 2004.

[103] A. Montresor, F. De Pellegrini, and D. Miorandi. Distributed k-core decomposition. *IEEE Transactions on parallel and distributed systems*, 24(2):288–300, 2013.

[104] J. W. Moon and L. Moser. On cliques in graphs. *Israel J. Math.*, 3(1):23–28, 1965.

[105] A. P. Mukherjee and S. Tirthapura. Enumerating maximal bicliques from a large graph using mapreduce. In *IEEE BigData Congress*, pages 707–716, 2014.

[106] A. P. Mukherjee and S. Tirthapura. Enumerating maximal bicliques from a large graph using mapreduce. *IEEE Trans. Services Computing*, 10(5):771–784, 2017.

[107] A. P. Mukherjee, P. Xu, and S. Tirthapura. Enumeration of maximal cliques from an uncertain graph. *IEEE Trans. Knowl. Data Eng.*, 29(3):543–555, 2017.

[108] T. Murata. Discovery of user communities from web audience measurement data. In *Web Intelligence, 2004. WI 2004. Proceedings. IEEE/WIC/ACM International Conference on*, pages 673–676. IEEE, 2004.

[109] R. A. Mushlin, A. Kershenbaum, S. T. Gallagher, and T. R. Rebbeck. A graph-theoretical approach for pattern discovery in epidemiological research. *IBM systems journal*, 46(1):135–149, 2007.

[110] M. A. U. Nasir, A. Gionis, G. D. F. Morales, and S. Girdzijauskas. Fully dynamic algorithm for top-k densest subgraphs. *algorithms*, 9(15):24, 2017.

[111] R. Nataraj and S. Selvan. Parallel mining of large maximal bicliques using order preserving generators. *International Journal of Computing*, 8(3):105–113, 2014.

[112] T. J. Ottosen and J. Vomlel. Honour thy neighbour: clique maintenance in dynamic graphs. In *PGM*, pages 201–208, 2010.

[113] G. Palla, I. Derényi, I. Farkas, and T. Vicsek. Uncovering the overlapping community structure of complex networks in nature and society. *Nature*, 435(7043):814–818, 2005.

[114] N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal. Efficient mining of association rules using closed itemset lattices. *Information systems*, 24(1):25–46, 1999.

[115] J. Pattillo, A. Veremyev, S. Butenko, and V. Boginski. On the maximum quasi-clique problem. *Discrete Applied Mathematics*, 161(1-2):244–257, 2013.

[116] J. Pei, D. Jiang, and A. Zhang. On mining cross-graph quasi-cliques. In *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*, pages 228–238. ACM, 2005.

[117] E. Prisner. Bicliques in graphs i: Bounds on their number. *Combinatorica*, 20(1):109–117, 2000.

[118] L. Quick, P. Wilkinson, and D. Hardcastle. Using pregel-like large scale graph processing frameworks for social network analysis. In *2012 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining*, pages 457–463. IEEE, 2012.

[119] O. Rokhlenko, Y. Wexler, and Z. Yakhini. Similarities and differences of gene expression in yeast stress conditions. *Bioinformatics*, 23(2):e184–e190, 2007.

[120] J. E. Rome and R. M. Haralick. Towards a formal concept analysis approach to exploring communities on the world wide web. In *Formal Concept Analysis*, volume 3403 of *LNCS*, pages 33–48. 2005.

[121] R. A. Rossi. Fast triangle core decomposition for mining large graphs. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 310–322. Springer, 2014.

[122] R. A. Rossi and N. K. Ahmed. The network data repository with interactive graph analytics and visualization. In *AAAI*, pages 4292–4293, 2015.

[123] P. San Segundo, J. Artieda, and D. Strash. Efficiently enumerating all maximal cliques with bit-parallelism. *Computers & Operations Research*, 92:37–46, 2018.

[124] M. J. Sanderson, A. C. Driskell, R. H. Ree, O. Eulenstein, and S. Langley. Obtaining maximal concatenated phylogenetic data sets from large sequence databases. *Mol. Biol. Evol.*, 20(7):1036–1042, 2003.

[125] A. E. Sariyüce, B. Gedik, G. Jacques-Silva, K. Wu, and Ü. V. Çatalyürek. Streaming algorithms for k-core decomposition. *PVLDB*, 6(6):433–444, 2013.

[126] A. E. Sarıyüce, B. Gedik, G. Jacques-Silva, K.-L. Wu, and Ü. V. Çatalyürek. Incremental k-core decomposition: algorithms and evaluation. *The VLDB Journal*, 25(3):425–447, 2016.

[127] A. E. Sariyüce, C. Seshadhri, and A. Pinar. Local algorithms for hierarchical dense subgraph discovery. *Proceedings of the VLDB Endowment*, 12(1):43–56, 2018.

[128] R. Schenkel, A. Theobald, and G. Weikum. Hopi: An efficient connection index for complex xml document collections. In *International Conference on Extending Database Technology*, pages 237–255. Springer, 2004.

[129] M. C. Schmidt, N. F. Samatova, K. Thomas, and B.-H. Park. A scalable, parallel algorithm for maximal clique enumeration. *Journal of Parallel and Distributed Computing*, 69(4):417–428, 2009.

[130] O. Shalev and N. Shavit. Split-ordered lists: Lock-free extensible hash tables. *Journal of the ACM (JACM)*, 53(3):379–405, 2006.

[131] Y. Shao, L. Chen, and B. Cui. Efficient cohesive subgraphs detection in parallel. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, pages 613–624. ACM, 2014.

[132] K. Sim, J. Li, V. Gopalkrishnan, and G. Liu. Mining maximal quasi-bicliques to co-cluster stocks and financial ratios for value investment. In *Data Mining, 2006. ICDM'06. Sixth International Conference on*, pages 1059–1063. IEEE, 2006.

[133] K. Sim, J. Li, V. Gopalkrishnan, and G. Liu. Mining maximal quasi-bicliques: Novel algorithm and applications in the stock market and protein networks. *Statistical Analysis and Data Mining: The ASA Data Science Journal*, 2(4):255–273, 2009.

[134] N. Simsiri, K. Tangwongsan, S. Tirthapura, and K.-L. Wu. Work-efficient parallel union-find with applications to incremental graph connectivity. In *European Conference on Parallel Processing*, pages 561–573. Springer, 2016.

[135] V. Stix. Finding all maximal cliques in dynamic graphs. *Comput. Optim. Appl.*, 27(2):173–186, 2004.

[136] S. Sun, Y. Wang, W. Liao, and W. Wang. Mining maximal cliques on dynamic graphs efficiently by local strategies. In *Data Engineering (ICDE), 2017 IEEE 33rd International Conference on*, pages 115–118. IEEE, 2017.

[137] M. Svendsen, A. P. Mukherjee, and S. Tirthapura. Mining maximal cliques from a large graph using mapreduce: Tackling highly uneven subproblem sizes. *J. Parallel Distrib. Comput.*, 79-80:104–114, 2015.

[138] M. Svendsen, A. P. Mukherjee, and S. Tirthapura. Mining maximal cliques from a large graph using mapreduce: Tackling highly uneven subproblem sizes. *JPDC*, 79:104–114, 2015.

[139] N. Tatti and A. Gionis. Density-friendly graph decomposition. In *Proceedings of the 24th International Conference on World Wide Web*, pages 1089–1099. International World Wide Web Conferences Steering Committee, 2015.

[140] M. Thorup. Decremental dynamic connectivity. *Journal of Algorithms*, 33(2):229–243, 1999.

[141] E. Tomita and T. Kameda. An efficient branch-and-bound algorithm for finding a maximum clique with computational experiments. *J. Global Optimization*, 44(2):311, 2009.

[142] E. Tomita, Y. Sutani, T. Higashi, S. Takahashi, and M. Wakatsuki. A simple and faster branch-and-bound algorithm for finding a maximum clique. In *WALCOM: Algorithms and Computation, 4th International Workshop, WALCOM 2010, Dhaka, Bangladesh, February 10-12, 2010. Proceedings*, pages 191–203, 2010.

[143] E. Tomita, A. Tanaka, and H. Takahashi. The worst-case time complexity for generating all maximal cliques and computational experiments. *Theoretical Computer Science*, 363(1):28–42, 2006.

[144] E. Tomita, K. Yoshida, T. Hatta, A. Nagao, H. Ito, and M. Wakatsuki. A much faster branch-and-bound algorithm for finding a maximum clique. In *Frontiers in Algorithmics, 10th International Workshop, FAW 2016, Qingdao, China, June 30- July 2, 2016, Proceedings*, pages 215–226, 2016.

[145] C. Tsourakakis. The k-clique densest subgraph problem. In *Proceedings of the 24th international conference on world wide web*, pages 1122–1132. International World Wide Web Conferences Steering Committee, 2015.

[146] C. Tsourakakis, F. Bonchi, A. Gionis, F. Gullo, and M. Tsiarli. Denser than the densest subgraph: extracting optimal quasi-cliques with quality guarantees. In *Proceedings of the*

19th ACM SIGKDD international conference on Knowledge discovery and data mining, pages 104–112. ACM, 2013.

[147] C. E. Tsourakakis. A novel approach to finding near-cliques: The triangle-densest subgraph problem. *arXiv preprint arXiv:1405.1477*, 2014.

[148] S. Tsukiyama, M. Ide, H. Ariyoshi, and I. Shirakawa. A new algorithm for generating all the maximal independent sets. *SIAM J. Comput.*, 6(3):505–517, 1977.

[149] T. Uno. An efficient algorithm for solving pseudo clique enumeration problem. *Algorithmica*, 56(1):3–16, 2010.

[150] J. Wang and J. Cheng. Truss decomposition in massive networks. *Proceedings of the VLDB Endowment*, 5(9):812–823, 2012.

[151] B. Wu, S. Yang, H. Zhao, and B. Wang. A distributed algorithm to enumerate all maximal cliques in mapreduce. In *Frontier of Computer Science and Technology, 2009. FCST'09. Fourth International Conference on*, pages 45–51. IEEE, 2009.

[152] C. Wulff-Nilsen. Faster deterministic fully-dynamic graph connectivity. In *Proceedings of the twenty-fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1757–1769. SIAM, 2013.

[153] Y. Xu, J. Cheng, A. W.-C. Fu, and Y. Bu. Distributed maximal clique computation. In *IEEE BigData Congress*, pages 160–167, 2014.

[154] C. Yan, J. G. Burleigh, and O. Eulenstein. Identifying optimal incomplete phylogenetic data sets from sequence databases. *Mol. Phylogenet. Evol.*, 35(3):528–535, 2005.

[155] M. J. Zaki and C.-J. Hsiao. Charm: An efficient algorithm for closed itemset mining. In *Proceedings of the 2002 SIAM international conference on data mining*, pages 457–473. SIAM, 2002.

[156] Z. Zeng, J. Wang, L. Zhou, and G. Karypis. Coherent closed quasi-clique discovery from large dense graph databases. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 797–802. ACM, 2006.

[157] B. Zhang, B.-H. Park, T. Karpinets, and N. F. Samatova. From pull-down data to protein interaction networks and complexes with biological relevance. *Bioinformatics*, 24(7):979–986, 2008.

[158] Y. Zhang, F. N. Abu-Khzam, N. E. Baldwin, E. J. Chesler, M. A. Langston, and N. F. Samatova. Genome-scale computational approaches to memory-intensive applications in systems biology. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 12. IEEE Computer Society, 2005.

[159] Y. Zhang, C. A. Phillips, G. L. Rogers, E. J. Baker, E. J. Chesler, and M. A. Langston. On finding bicliques in bipartite graphs: a novel algorithm and its application to the integration of diverse biological data types. *BMC bioinformatics*, 15(1):1, 2014.

[160] Y. Zhang, J. Wang, Z. Zeng, and L. Zhou. Parallel mining of closed quasi-cliques. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–10. IEEE, 2008.

[161] Y. Zhang, J. X. Yu, Y. Zhang, and L. Qin. A fast order-based approach for core maintenance. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, pages 337–348. IEEE, 2017.

[162] R. Zhou, C. Liu, J. X. Yu, W. Liang, and Y. Zhang. Efficient truss maintenance in evolving networks. *arXiv preprint arXiv:1402.2807*, 2014.