ABSTRACT

HENDRIX, WILLIAM THOMAS. On Graph Perturbation Theory and Algorithms for Scalable Mining of Noisy and Uncertain Graph Data with Knowledge Priors. (Under the direction of Nagiza F. Samatova.)

The solutions to many interesting questions lie in the ability to extract sets of tightly connected or correlated elements from a mass of relational-type data. For example, a researcher may formulate various questions about social communities within a social network, correlated stocks in market data, or proteins in a protein interaction network that form functional modules. However, such questions are often difficult to answer, as the data may exhibit noise or uncertainty, objects may belong to multiple groups, the data to be analyzed may be very large in scale, and the results of a naïve algorithm may not be relevant to the user's interests. In this thesis, we make several novel contributions to solving this problem, including applying the *knowledge priors* of a domain expert to the algorithmic search process, as well as introducing the concept of Graph Perturbation Theory in order to cope with uncertainty.

In our first component, we tackle the problems of noisy and uncertain data by introducing the concept of Graph Perturbation Theory, which consists of calculating graph properties as the graph undergoes some *perturbation*, or change. This perturbation may represent some new knowledge, reinterpretation of old data, or exploratory "what if"–type scenario. We develop and apply Graph Perturbation Theory to the problem of enumerating the fully connected subgraphs of a graph. The advantage in modeling our correlated groups as *maximal cliques* is that cliques can produce overlapping groups, and their fully connected nature ensures a strong positive relationship despite uncertainty in the individual edges present. By enumerating the set of all such maximal cliques of the graph, we can exhaustively describe the correlations present in the graph and avoid relying on heuristic methods. Our application of Graph Perturbation Theory to this problem resulted in speedups of up to 120 times over enumerating the maximal cliques of the perturbed algorithm using a state-of-the-art clique enumeration algorithm.

In our second component, we harness the knowledge priors of domain experts by allowing a user to specify the "items of interest" that the correlated items should be enriched with respect to, allowing us to constrain the search space of the algorithm as well as improve the chance of discovering sets of identifying correlated items relevant to the user's interests. We tackle the problem of incorporating experts' *knowledge priors* into the search for overlapping groups in noisy and uncertain data by proposing a novel subgraph structure, the μ, γ-quasi-clique. Our definition, which incorporates the information contained the user's query, is a relaxation of the fully connected clique property, and enables us to better compensate for missing, or *false negative*, relationships in our network, which would fracture a single large group into multiple,

highly overlapping cliques. We evaluate the effectiveness of our approach by demonstrating its scalability to larger graphs and reproduce biological results that establish the practical value of our work.

In our third component, we address the large scale nature of modern datasets by developing parallel strategies for Graph Perturbation Theory algorithms. As Graph Perturbation Theory algorithms utilize various indices to facilitate calculating graph properties of the perturbed graph, we discuss techniques for dealing with the highly data-intensive nature of such algorithms. To establish the effectiveness of our proposed techniques, we implement a parallel version of the perturbed clique enumeration algorithm presented in the first component, developing additional theory that allows us to decouple the action of this algorithm from its previous computation. We present detailed empirical results demonstrating the scalability of our algorithm as well as the effectiveness of the various techniques we propose.

On Graph Perturbation Theory and Algorithms for Scalable Mining of Noisy and Uncertain Graph Data with Knowledge Priors

by
William Thomas Hendrix

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Science

Raleigh, North Carolina

2010

APPROVED BY:

Rada Chirkova	Jon Doyle
Anatoli Melechko	Nagiza F. Samatova Chair of Advisory Committee

BIOGRAPHY

William Hendrix earned his Bachelor's of Science in Computer Science and Mathematics from Furman University in Spring 2007, graduating Magna cum Laude. He was admitted to North Carolina State University in Fall 2007 with a Dean's Fellowship and a Research Assistantship with Dr. Nagiza Samatova. Under Dr. Samatova's guidance, he was able to submit and pass the Written Qualifier in Spring 2008. In Fall 2009, he served as a course instructor for the undergraduate course in automata theory and was nominated by Dr. David Thuente, the Director of Graduate Programs, for an Outstanding Teaching Assistant award in Spring 2010. He also acted as a Teaching Assistant for the graduate and undergraduate courses in automata theory in Spring 2009 and 2010 and was nominated in Spring 2009 for another Outstanding Teaching Assistant award. In Spring 2010, he completed the Oral Examination, and he defended his dissertation in Fall 2010.

ACKNOWLEDGEMENTS

I would first like to thank my advisor, Prof. Nagiza Samatova, for her tireless support and invaluable guidance over the course of my graduate career. This work would not have been possible without her support. I would also like to thank my committee members, Profs. Rada Chirkova, Jon Doyle, and Anatoli Melechko, for their helpful suggestions and encouragement.

I would also like to extend thanks to the other members of Dr. Samatova's research group for their moral support, as well as the collaborations we have shared. I would especially like to thank Matt Schmidt, Tatdow (Ann) Pansombut, Andrea Rocha, Wenbin Chen, Paul Breimyer, and Zhengzhang Chen for the insights they have shared on our various projects.

I would also like to thank my family, whose unwavering support inspired me to persevere through all of the challenges I have faced in my graduate career.

My work has been supported by the "Exploratory Data Intensive Computing for Complex Biological Systems" and "Ultrascale Computational Modeling of Phenotype-Specific Metabolic Processes in Microbial Communities" projects from the U.S. Department of Energy (Office of Advanced Scientific Computing Research and Office of Biological and Environmental Research, Office of Science). My work has also been sponsored by the Dean of the Department of Computer Science at North Carolina State University.

TABLE OF CONTENTS

List of Tables . . . . **vi**

List of Figures . . . . **viii**

Chapter 1 Introduction **1**
- 1.1 Graph Perturbation Theory for Noisy and Uncertain Data 3
- 1.2 Dense, Enriched Subgraph Enumeration 4
- 1.3 Enabling the Parallelization of Perturbation-Type Algorithms 5
- 1.4 Publications 5

Chapter 2 Graph Perturbation Theory for Noisy and Uncertain Data **7**
- 2.1 Introduction 7
- 2.2 Background 9
- 2.3 Helper Lemmas 10
- 2.4 Adding and removing a vertex 11
  - 2.4.1 Adding a vertex 11
  - 2.4.2 Removing a vertex 11
- 2.5 Adding and removing a single edge 14
  - 2.5.1 Adding an edge 14
  - 2.5.2 Removing an edge 16
- 2.6 Multiple edges 17
  - 2.6.1 Adding multiple edges 17
  - 2.6.2 Adding multiple edges to an isolated vertex 20
  - 2.6.3 Removing multiple edges 22
- 2.7 Algorithm 23
  - 2.7.1 Indices 28
- 2.8 Results 28
- 2.9 Conclusion 34

Chapter 3 Dense, Enriched Subgraph Enumeration **37**
- 3.1 Introduction 37
- 3.2 Theory 39
- 3.3 Algorithm 42
- 3.4 Results 46
- 3.5 Biological application 48
  - 3.5.1 Motivation and Background 48
  - 3.5.2 Results 50
- 3.6 Conclusion 53

Chapter 4 Enabling the Parallelization of Perturbation-Type Algorithms . . . **54**
4.1 Introduction . . . 54
4.2 Parallel edge removal algorithm . . . 54
4.2.1 Serial algorithm . . . 55
4.2.2 Parallelization strategy . . . 56
4.2.3 Primary work division . . . 56
4.2.4 Parallelizing index accesses . . . 57
4.2.5 Pruning duplicate subgraphs . . . 59
4.2.6 Workload distribution . . . 62
4.2.7 Scalability results . . . 64
4.3 Parallel edge addition algorithm . . . 65
4.3.1 Serial algorithm . . . 65
4.3.2 Parallel algorithm . . . 66
4.3.3 Scalability results . . . 67
4.4 Conclusion . . . 69

Chapter 5 Conclusion and Future Work . . . **70**

References . . . **73**

LIST OF TABLES

Table 2.1 Runtime results for six perturbations on synthetic graphs of increasing size 32
Table 2.2 Runtime results for 6 perturbations on synthetic graphs of increasing edge counts . . . 33
Table 2.3 Runtime Results for Edge Removal on the *E. coli* network . . . 34
Table 2.4 Runtime Results for Edge Addition on the *E. coli* network . . . 35

Table 3.1 Values by which d_v is decreased when a vertex $u \in C$ is moved to S or removed from C . . . 45
Table 3.2 Values by which e is decreased when a vertex $u \in C$ is moved to S or removed from C . . . 45
Table 3.3 Table summarizing values by which g_v decreases when a vertex $u \in C$ is moved to S (column "M") or removed from C (column "R") . . . 45
Table 3.4 Table summarizing values by which m_v decreases when a vertex $u \in C$ is moved to S (column "M") or removed from C (column "R") . . . 46
Table 3.5 Parameter settings for the various types of dense, enriched subgraphs we use to test our algorithm . . . 46
Table 3.6 Graph size and number of maximal quasi-cliques for graphs generated using R-MAT . . . 47
Table 3.7 Identified proteins . . . 51

Table 4.1 Effects of duplicate pruning on algorithm performance. Here, we report the number of (potentially duplicate) cliques of $\mathcal{C}_+$ identified by the algorithm as well as timing results using the in-memory index access and producer-consumer strategies on a single processor. As the duplicate pruning results are only relevant during the recursive removal phase of the algorithm, we include here only the time spent in the *Enumerate* phase of the algorithm, which encompasses clique retrieval, recursive removal, and load balancing. The amount of time spent during the other phases of the algorithm was consistent between the two runs—see Figure 4.1 for a comparison of the time spent in the various stages of the edge removal algorithm. . . . 62

Table 4.2 Scalability results for edge weight–induced perturbation on the Medline graph. All times are given in seconds and represent the longest duration that a single processor spent on the given task. The timing results were broken down as follows. *Init* time represents the amount of time taken by the algorithm in allocating memory for the algorithm data structures and reading the graph and indices into memory. *Root time* represents the amount of time taken by the algorithm in generating the initial set of *candidate list* structures for the modified BK algorithm. *Enumerate time* represents the combined amount of time required to run the BK algorithm to detect the set of cliques in $\mathcal{C}_+$ as well as performing the recursive removal procedure, index lookups required to calculate $\mathcal{C}_-$, and load balancing. *Idle time* represents the time spent by a processor that has finished its workload and has no other work to steal. 68

LIST OF FIGURES

Figure 2.1 Example of a graph in the case of vertex removal. Edges with dotted lines are removed when vertex u is removed. Vertices in the white circle belong to the maximal clique C in the original graph. Vertices in the vertical striped circle belong to $N_G(u) \cup \{u\}$. Vertices in the gray circle belong to $\bigcup_{v \in N_G(u)} N_G(v)$. 13

Figure 2.2 Example of a graph in the case of edge addition. The edge being added is the dotted line between vertices u and v. Vertices in the circle with slashes (/) belong to the clique C in the original graph, G. Vertices in the circle filled with backslashes (\) belong to the clique D in G. Vertices in the area filled with the $\times$'s belong to $C \cap D$. Vertices in the white circle belong to $N_G(u)$ and vertices in the dark gray circle belong to $N_G(v)$. Vertices in the light gray area belong to $N_G(u) \cap N_G(v)$. 15

Figure 2.3 Percentage improvement in runtime for file index–based implementation over MySQL–based implementation of edge removal algorithm for three example networks 29

Figure 2.4 Speedup relative to BK for randomly generated perturbations on sparse graphs as graph size increases. The left graph shows the speedup for edge addition perturbations, and the right graph shows the speedup for edge removal perturbations. The number of edges in all graphs is approximately 8 times the number of vertices. 30

Figure 2.5 Speedup relative to BK for randomly generated perturbations on sparse graphs as graph density increases. The left graph shows the speedup for edge addition perturbations, and the right graph shows the speedup for edge removal perturbations. The number of vertices in all graphs is approximately 8,192. 31

Figure 2.6 Number and percentage of predicted protein interactions having a given confidence value. 31

Figure 2.7 Speedup factor of the PGCE algorithm with respect to the BK algorithm for edge removal. The networks are, from left to right, those of *E. coli*, *M. tuberculosis*, *S. typhimurium*, and *Synechocystis*. 36

Figure 2.8 Speedup factor of the PGCE algorithm with respect to the BK algorithm for edge addition. The networks are, from left to right, those of *E. coli*, *M. tuberculosis*, *S. typhimurium*, and *Synechocystis*. 36

Figure 3.1 Timing results for μ, γ-quasi-clique enumeration algorithm. Time is reported in milliseconds per quasi-clique. Descriptions of the various quasi-cliques can be found in Table 3.5, and descriptions of the graphs used can be found in Table 3.6. 48

Figure 3.2 Speedup results for using hierarchical bitmap index in μ, γ-quasi-clique enumeration algorithm. Speedup is reported in percentage; i.e., a value of 100% indicates that using the hierarchical bitmap index was twice as fast as the implementation with the flat index, and a value of -100% indicates that using the flat bitmap index was twice as fast as the implementation with the hierarchical index. 49
Figure 3.3 The five clusters identified by our algorithm 50

Figure 4.1 Timing results for disk-based (4.1a) and in-memory (4.1b) index accesses. All times are given in seconds and represent the longest duration that a single processor spent on the given task. The timing results were broken down as follows. *Init time* represents the amount of time required to read and broadcast the graph and index data. *Root time* represents the amount of time required to read in the set of clique IDs that contain an edge being removed. *Enumerate time* represents the amount of time required to retrieve the clique vertices, perform the recursive removal procedure, and load balance. *Idle time* represents the amount of time spent by a processor that has no work and is unable to find work to steal. 59
Figure 4.2 A graph in which subgraph $\{B, C, E\}$ is common to three different cliques: $\{A, B, C, E\}$, $\{B, C, D, E\}$, and $\{B, C, E, F\}$. 60
Figure 4.3 Timing breakdown for work distribution strategy using in-memory index accesses with no load balancing (4.3a) and work stealing (4.3b). All times are given in seconds and represent the longest duration that a single processor spent on the given task. The timing results were broken down as follows. *Init time* represents the amount of time required to read and broadcast the graph and index data. *Root time* represents the amount of time required to read in the set of clique IDs that contain an edge being removed. *Enumerate time* represents the amount of time required to retrieve the clique vertices, perform the recursive removal procedure, and load balance. *Idle time* represents the amount of time spent by a processor that has no work and is unable to find work to steal. 63
Figure 4.4 Evaluation of speedup of enumeration time for producer-consumer (4.4a) and workload distribution (4.4b) models. The *Fetch* time, or amount of time spent stealing work, is also plotted in Figure 4.4b. Speedup was calculated for the time spent in the *Enumerate* phase of the algorithm, in which the algorithm retrieves the vertices of a given clique, performs the recursive removal procedure, and balances the processor workloads. The speedup for the *Enumerate* phase of the algorithm is plotted against ideal (linear) speedup. In Figure 4.4b, *Fetch* time is plotted on the secondary axis, in seconds. 65

Figure 4.5 Normalized speedup versus number of processors. Speedups were calculated as $(t_1 * n_c)/t_{c,p}$, where t_1 is the *Enumerate* time for calculating the perturbation on one copy of the graph using a single processor, n_c is the number of copies of the graph, and $t_{c,p}$ is the *Enumerate* time for calculating the perturbation on c copies using p processors. *Enumerate time* represents the combined amount of time required to run the BK algorithm to detect the set of cliques in $\mathcal{C}_+$ as well as performing the recursive removal procedure, index lookups required to calculate $\mathcal{C}_-$, and load balancing. 69

Chapter 1

Introduction

An emerging field of research is in discovering and analyzing the structure, dynamics, and function of complex networks across a variety of different domains, including social networks, ecological systems, the Internet, and others [6, 7, 9, 13, 52, 53, 54, 60, 70, 83]. Mining such networks for relationships allows scientists to develop *data-driven* insights into fruitful directions in their fields, complementing the typical *hypothesis-driven* scientific methodology.

In the context of biology, examples of relational data include protein interaction networks, gene regulatory networks, and metabolic pathways, and phylogenetic trees [86, 92]. Additionally, climatologists have used graph data to model spatial and temporal dynamics of climate systems [22, 23, 29, 75, 76, 77, 78, 79, 80, 87, 88]. Another application area and an emerging field of research is in the area of social networks [62, 15, 20, 84], in which the relationships between individuals and groups of people can be studied. Such types of relational data are often represented in the form of a graph, as this is the mathematical abstraction for a network of relationships.

Oftentimes, we wish to uncover relationships and correlations within these relational data. In the context of social networks, such correlations take the form of communities [20, 84]. In a biological context, collections of proteins that are strongly related may form protein complexes or perform related functions within a cell [91].

In the past, strong correlations in relational data have been discovered using clustering or graph theoretical methods. Clustering, which has long been a fruitful area of research, seeks to divide a set of data into groups in which the group elements are more similar to others in the same group than the elements of other groups. Traditional "hard" clustering, like the well-known k-means algorithm, seeks to partition the data set into disjoint groups, so that no element belongs to multiple groups simultaneously. However, in some domains, the items of interest may belong to multiple groups at once, rendering such methods inappropriate. For example, in the context of marketing data, a single person might be male, over the age of 55, and someone who prefers comedies to drama. In this context, several approaches to "soft" clustering, including fuzzy logic-based approaches [45] (also, see [38]), probabilistic approaches [33, 68], and consensus clustering [21] have been developed to produce overlapping clusters.

An alternative to using clustering to uncover these strong correlations among data elements is to approach the data from a graph-theoretic perspective. By defining these relationships mathematically, we can avoid the imprecision of heuristic clustering-based approaches, and we can enumerate the complete set of groups that fulfill our defined criteria.

As the correlated sets of items that we are searching for are allowed to overlap, we can search for such clusters by enumerating the *cliques* of a graph. A *clique* is a fully connected subgraph, that is, a set of vertices in which all edges are joined by vertices. Due to its fully connected nature, a clique larger than size two or three is very likely to represent some nontrivial relationship, even in the presence of uncertainty in the underlying data.

In spite of the advantages offered by graph theoretical approaches to discovering strongly correlated items in relational data, there are several challenges that need to be overcome. First, many real-world datasets exhibit uncertainty. In the biological domain, high-throughput experiments provide clues to the relationships between large numbers of proteins and genes, but this level of throughput comes at the cost of higher numbers of false positive and negative inferred relationships [44]. Additionally, traditional methods would require reevaluating the relationships as new data were uncovered and verified. Second, existing methods often fail to take advantage of researchers' domain-specific knowledge. Finally, the datasets that scientists are interested in mining are very large [63, 64, 65, 73, 55, 66, 85, 89], so a solid approach to evaluating such data must be able to scale.

In order to tackle these challenges, we propose three components for this thesis. In the first component, we address the problems of uncertain and noisy data by proposing a novel approach to the problem with Graph Perturbation Theory, which identifies the dense components of a graph (modeled as maximal cliques) as the structure of the graph changes. We derive relevant theoretical results for our approach, develop an algorithm implementation, and verify the effectiveness of our algorithm and theoretical results empirically. In the second component, we address the lack of support for incorporating domain knowledge by developing a novel graph theoretic structure, the μ, γ-quasi-clique, for detecting dense components that incorporates experts' domain knowledge into the search procedure. We develop theoretical results, implement an algorithm based on this theory, quantitatively evaluate the ability of our implementation to enumerate the μ, γ-quasi-cliques of large graphs, and present a biological case study in which our approach was applied to the problem of detecting genes related to biological hydrogen (biohydrogen) production. In the third component, we address the need for scalability by developing parallelization strategies for Graph Perturbation Theory algorithms. We develop a parallel implementation of our perturbed clique enumeration and empirically evaluate its scalability to multiple processors.

1.1 Graph Perturbation Theory for Noisy and Uncertain Data

Problem. We first approach the problem of uncovering correlations in uncertain and noisy data by modeling the uncertainty in the data as a change, or *perturbation.* Such perturbations may represent new data, reinterpretation of previous results, or even exploratory "what-if" scenarios.

We use solution to the maximal clique enumerate (MCE) problem to identify the tightly knit, potentially overlapping correlations for which we are searching. As mentioned earlier, cliques are fully connected subgraphs, and as such, provide strong evidence that the items involved are related, and by enumerating only the *maximal* cliques of a graph (those cliques not contained by another clique), we avoid enumerating the numerous non-maximal cliques that describe the relationships already implied by the larger maximal cliques.

Calculating the maximal clique enumeration of a graph has already been successfully applied by researchers across a number of different domains. Example applications include identifying protein-protein interaction complexes within protein interaction data [93], aligning 3-dimensional protein structures [17], integrating genome mapping data [32], identifying co-expressed genes [61], identifying common secondary structure elements of proteins [30], finding clusters of orthologous genes [56], clustering similar mass spectrometry spectra [72], and detecting social hierarchy from email communications [62].

The problem of enumerating the maximal cliques of a graph is a computationally expensive problem with exponential worst-case complexity [50]. Calculating the maximal clique enumeration (MCE) of a single graph can provide valuable insights into the community structure of the graph; however, when dealing with noisy or uncertain data, we may need to consider the maximal cliques of several closely related graphs, and the computational cost of doing so quickly becomes prohibitively expensive.

Approach. As efficient algorithms have already been developed for the problem of enumerating the cliques of a static graph [10, 5, 48, 81, 41, 18, 49, 74, 69], we focus on how to "update" the clique enumeration of a graph undergoing some sort of perturbation.

While others have considered the problem of updating graph properties as the graph changes (for ex., [59], [37], [47]), these approaches have primarily focused on graph properties that can be calculated in polynomial time. Coping with a problem that requires potentially exponential time calls for a different approach, and to this end, we introduce the concept of Graph Perturbation Theory to handle such computationally challenging problems.

The primary inspiration behind Graph Perturbation Theory is that, by indexing the graph data for a previously calculated problem instance, we can use these various indices to accelerate the recalculation of our graph property in question. By combining our Graph Perturbation Theory approach with the advantages inherent in calculating only the changes to the maximal clique enumeration, we are able to achieve speedups of up to 120 over traditional state-of-the-art maximal clique enumeration approaches. The results of this work have been published at the *ACM SIGKDD* conference [35] and in *Theoretical Computer Science* [36]. Our approach and results are detailed in Chapter 2 of this thesis.

1.2 Dense, Enriched Subgraph Enumeration

Problem. The second issue we address in this thesis is the problem of integrating researchers' domain knowledge into the discovery process. As a motivating example, consider the case where researchers apply clustering or other data mining techniques and then analyze the results by hand to decide whether the results conform to their expectations. This manual curation process is very time-consuming, and if the results being analyzed do not reflect known facts, the researcher will usually need to perform additional analysis on the data, perhaps with different parameters. By incorporating the researchers' domain knowledge into the search procedure, we can reduce the time wasted on parameter tuning and sanity checking.

Approach. Our second approach to finding overlapping clusters in graph data complements and expands on our previous work. To help integrate researchers' prior knowledge into the graph mining process, we introduce a novel graph structure, the μ, γ-quasi-clique, which represents a densely connected subgraph that is "enriched" relative to some subset of "interesting" vertices specified by the user. By allowing researchers to specify a set of "objects of interest," we hope to reduce time spent on searching for and analyzing dubious or unimportant relationships in the data.

Additionally, we accommodate uncertainty in the data by generalizing the notion of a "dense" subgraph from a maximal clique to a subgraph in which each vertex is adjacent to "most" of the others (at a threshold specified by the user), rather than focusing on enabling rapid enumeration of slightly perturbed graphs. This generalization is especially useful in the case of incomplete data, where a single group of highly related vertices can be divided into a large number of fully connected subgraphs with relatively few missing edges. As with enumerating cliques, we focus on enumerating only the maximal dense, enriched subgraphs, but the lack of the fully connected property of cliques complicates the question of determining when a subgraph is maximal.

Further details on our approach, including an empirical evaluation demonstrating its effectiveness on large sparse graphs and a discussion of an application of our algorithm to a biological problem of practical interest, appear in Chapter 3. Results from this work have also been published at the BIOCOMP 2010 conference [34].

1.3 Enabling the Parallelization of Perturbation-Type Algorithms

Problem. While real-world graphs are often sparse and exhibit other properties that allow dense or complete subgraph enumeration to be completed in a tractable amount of time [6, 54], these graphs can also be quite large and require long periods of time to process, costing valuable computing resources. A scalable parallel algorithm for dense subgraph discovery in uncertain data would allow large, uncertain graph to be analyzed in a reasonable amount of time.

Approach. As a third component of our work, we develop a scalable parallel implementation of our algorithm for maximal clique enumeration in perturbed graphs. We propose solutions to the challenges of how to break down the actions of the perturbed clique enumeration algorithm into parallelizable chunks, how to organize the available processors so that the workload is evenly and efficiently distributed among them, and how to parallelize the process of checking a subgraph for maximality. We propose novel theory to address the problem of calculating maximality, and we empirically evaluate a number of strategies to improve the scalability of our Perturbed Clique Enumeration algorithm. Based on our proposed techniques, we were able to demonstrate good scaling for our approach on up to 64 processors. Further details on our approach and scalability studies appear in Chapter 4.

1.4 Publications

The following is a list of publications related to the work in this thesis. Papers 1, 2, and 4 in the list below are related to the work discussed in Chapter 2, and paper 3 is related to the work discussed in Chapter 3. A paper describing the work in Chapter 4 is in preparation.

1. W. Hendrix, M.C. Schmidt, P. Breimyer, and N.F. Samatova, "Theoretical Underpinnings for Maximal Clique Enumeration on Perturbed Graphs," *Theoretical Computer Science*, vol. 411, no. 26–28, pp. 2520–2536, 2010.

2. W. Hendrix, M.C. Schmidt, P. Breimyer, N.F. Samatova, "On Perturbation Theory and an Algorithm for Maximal Clique Enumeration in Uncertain and Noisy Graphs," *ACM SIGKDD* conference, in *U '09: Proceedings of the 1st ACM SIGKDD Workshop on Knowledge Discovery from Uncertain Data*, pp. 48–56, 2009.

3. W. Hendrix, A.M. Rocha, M. Elmore, J. Trien, and N.F. Samatova, "Discovery of Enriched Biological Motifs Using Knowledge Priors with Application to Biohydrogen Production," *BIOCOMP 2010*, vol. 1, pp. 17–23, 2010.

4. N.F. Samatova, M.C. Schmidt, W. Hendrix, P. Breimyer, K. Thomas, and B.-H. Park, "Coupling graph perturbation theory with scalable parallel algorithms for large-scale enumeration of maximal cliques in biological graphs," *SciDAC 2008: Scientific Discovery through Advanced Computing*, vol. 125, pp. 12053, 2008.

The author of this thesis has participated in several other projects resulting in publications. The results from these projects are not included in this thesis. A list of these publications (as of the writing of this document) appears below.

- P. Breimyer, W. Hendrix, G. Kora, and N.F. Samatova, "pR: Lightweight, Easy-to-Use Middleware to Plugin Parallel Analytical Computing with R," *IKE*, pp. 667–673, 2009.
- P. Breimyer, G. Kora, W. Hendrix, N. Shah, and N.F. Samatova, "pR: Automatic Parallelization of Data-Parallel Statistical Computing Codes for R in Hybrid Multi-Node and Multi-Core Environments," *IADIS AC 2009*, pp. 28–32, 2009.
- N.F. Samatova, P. Breimyer, W. Hendrix and M.C. Schmidt, and T.-M. Rhyne, "An Outlook into Ultra-Scale Visualization of Large-Scale Biological Data," *Supercomputing* conference, *Ultra Vis: 2008 Workshop on Ultrascale Visualization*, pp. 29–39, 2008.
- Z. Chen, K. Wilson, Y. Jin, W. Hendrix, and N.F. Samatova, "Detecting and Tracking Community Dynamics in Evolutionary Networks," *IEEE International Conference on Data Mining* (ICDM 2010), in *Workshop on Social Interactions and Service Providers (SIASP) 2010*, 2010. (accepted)
- W. Chen, W. Hendrix, A.M. Rocha, M.C. Schmidt, and N.F. Samatova, "The Multiple Alignment Algorithm for Metabolic Pathways without Abstraction," *IEEE International Conference on Data Mining* (ICDM 2010), in *Workshop on Biological Data Mining and its Applications in Healthcare*, 2010. (accepted)

Chapter 2

Graph Perturbation Theory for Noisy and Uncertain Data

2.1 Introduction

A graph is an abstraction for representing a network of relationships and is useful in a variety of diverse settings. We loosely define a graph here as a set of vertices, which represent some "objects of interest," along with a set of edges, which connect two vertices and correspond to some relationship between the associated objects.

One feature of graphs, such as those built from protein interaction data, is the large amount of noise and uncertainty in the underlying data. Since the data underlying the graph cannot be relied upon absolutely, potential edges are assigned weights based on the likelihood of their existence. Since some analytical graph algorithms require unweighted graph edges, there is often a need to determine an arbitrary likelihood threshold above which an edge is included in the graph. Different threshold values result in different levels of sensitivity and specificity in the results found by the graph algorithm. Hence, several different threshold values may need to be explored, each corresponding to a different graph to be examined.

To best handle such threshold manipulations, as well as any other type of change that might be applied to a graph, we introduce the concept of Graph Perturbation Theory. Graph Perturbation Theory refers to the practice of efficiently recalculating the properties of the graph as it undergoes some change or *perturbation*. We achieve efficiency through the use of various indices (described further in Section 2.7.1) to access the properties of the original or *baseline* graph and by delineating only the changes induced by the perturbation.

In the case of creating perturbations by applying various threshold to a weighted graph, our baseline graph would be the initial graph formed by applying some threshold value to the edge weights, and the perturbed graph graphs are formed by modifying the initial treshold value. These perturbed graphs will have additional edges over the baseline graph when the threshold is lowered and removed edges relative to the baseline graph when the threshold is raised. Perturbed graphs may also be formed as the knowledge base on which the graph was built is refined by new experimental data. It should also be noted that graph perturbations that result in both the addition and removal of edges can be decomposed into two perturbations, one consisting only of edge removals and the other consisting only of edge additions, which can be iteratively applied. The focus of this paper is the establishment of necessary and sufficient conditions for evaluating a graph property—the set of maximal cliques of the graph—for a perturbed graph.

In order to discuss maximal cliques in perturbed graphs, we first set out definitions for a few basic terms from graph theory. A vertex and an edge are *incident* if the edge joins that vertex to another, and two vertices are *adjacent* if an edge joins them. A *neighborhood* of a vertex is the set of all vertices that are adjacent to it. A *clique* is a set of vertices which are all pairwise adjacent, and a *maximal* clique is a clique which is not contained within some larger clique of the graph. Additionally, in the context of this paper we assume that all graphs are simple—that is, no edge joins a vertex to itself and at most one edge can join any pair of distinct vertices. This assumption is made because neither loops nor multiple edges contribute meaningfully to the set of maximal cliques of a graph.

The maximal clique enumeration (MCE) problem consists of enumerating all maximal cliques in a graph. The utility of solving this problem is in discovering tightly knit, possibly overlapping sets within some overarching network of relationships. Potential applications of finding maximal cliques in graphs include identifying protein-protein interaction complexes within protein interaction data [93], aligning 3-dimensional protein structures [17], integrating genome mapping data [32], identifying co-expressed genes [61], identifying common secondary structure elements of proteins [30], finding clusters of orthologous genes [56], clustering similar mass spectrometry spectra [72], and detecting social hierarchy from email communications [62].

Enumerating all of the maximal cliques of a graph is computationally expensive. Moon and Moser demonstrated that a graph of n vertices may have as many as $3^{n/3}$ maximal cliques [50], providing a strict lower bound on the worst-case complexity of any enumeration algorithm. Since the number of maximal cliques in a graph may increase exponentially with the size of a graph, the efficiency of MCE algorithms is often measured with respect to the computational cost per maximal clique.

We aim to improve the efficiency of defining the set of maximal cliques of a perturbed graph by providing necessary and sufficient conditions to enumerate the difference set of the sets of maximal cliques of the baseline graph and the perturbed graph (sets $\mathcal{C}$ and $\mathcal{C}_{new}$ respectively). The difference set is the set of maximal cliques that are either unique to the baseline graph ($\mathcal{C}_{-}$) or unique to the perturbed graph ($\mathcal{C}_{+}$). If both the set of maximal cliques of the baseline graph and the difference set are known, then the set of maximal cliques of the perturbed graph can be defined ($\mathcal{C}_{new} = (\mathcal{C} \cup \mathcal{C}_{+}) \setminus \mathcal{C}_{-}$). This approach differs from the standard approach, which defines $\mathcal{C}_{new}$ by using an MCE algorithm to enumerate each member of $\mathcal{C}_{new}$. While both approaches would enumerate the maximal cliques unique to the perturbed graph, our approach would also have to enumerate the maximal cliques unique to the baseline graph, while the standard approach would also have to enumerate the maximal cliques that are common to both graphs. The benefit to our approach is that when the baseline and perturbed graphs are similar, the set of maximal cliques unique to the baseline graph will be much smaller than the set of maximal cliques that are common to both graphs. Thus, if both approaches have a similar computational cost per maximal clique enumerated, then the computational cost of our approach will be much less than the standard approach.

The rest of the paper is organized as follows. Section 2.2 will review some of the existing algorithms for MCE as well as previous work on graph properties in changing graphs. Section 2.3 will introduce some helper lemmas that will be used throughout the paper. The necessary and sufficient conditions for defining the difference set in the case of creating a perturbed graph by adding or removing a vertex, adding or removing a single edge, or adding or removing multiple edges will be introduced in Sections 2.4, 2.5, and 2.6, respectively. Section 2.7 will describe an algorithm that utilizes the conditions presented in Sections 2.4, 2.5, and 2.6 to efficiently enumerate the difference set. Section 2.7.1 details the various indices that we employed in our algorithm, and Section 2.8 discusses the results of our comparison of the performance of our perturbed graph MCE algorithm with that of well-known, efficient MCE algorithms.

2.2 Background

Previous researchers have produced several different algorithms for enumerating the maximal cliques of a graph. BK, one of the most widely used and enduring MCE algorithms, was developed by Bron and Kerbosch [10]. It operates by recursively expanding and shrinking a vertex set while keeping track of additional information to avoid enumerating duplicate maximal cliques and to limit unnecessary searching. A survey by Johnston [42] indicated that BK performed favorably against a set of related MCE algorithms. Another backtracking algorithm, developed by Akkoyunlu [5], recursively divides the search space such that every maximal clique in the original search space was contained in exactly one of the new, smaller search spaces. Later, the backtracking algorithm of Loukakis and Tsouros [48] was shown to be more efficient than BK for small graphs (up to 220 vertices). An efficient MCE algorithm was presented by Tomita et al. [74] in 2006. Based heavily on BK, it achieves its performance with the use of an abbreviated output format.

Though backtracking search is the common basis of the previously mentioned MCE algorithms, other search methods have been used. Several algorithms, including those by Tsukiyama et al. [81], Chiba and Nishizeki [18], Johnson et al. [41], and Makino and Uno [49], are based on the the concept of reverse search, in which a “parent” relationship is used to iterate through a tree containing the maximal cliques of a graph. Another approach, explored by Stix [69], is to calculate the maximal clique enumeration starting from either a fully connected or an edgeless graph and forming the desired graph by adding or removing edges, adjusting the enumeration as the desired graph emerges.

Several previous researchers have also considered the problem of how graph properties are altered by changes in the underlying graph. Many researchers, such as Rauch [59] and Henzinger and King [37], have focused on the connectivity of a graph, though other properties, such as matching [47], have also been considered. Though much of previous work in the field uses the term *dynamic* to describe graphs undergoing changes, this term suggests that the graph is undergoing continuous change and the graph properties are being adjusted in real time. However, since the set of maximal cliques of a graph may be exponential in the size of that graph, calculating the enumeration may still carry a substantial computational cost even when only producing an update to the set. Thus, we introduce the term *perturbed* to indicate that maximal clique enumeration on the altered graph occurs in a discrete, update-like fashion.

2.3 Helper Lemmas

We consider the necessary and sufficient conditions for determining the altered clique enumeration, $\mathcal{C}_{new}$, of the perturbed graph G_{new} using information from the graph prior to the perturbation, G, along with its clique enumeration, $\mathcal{C}$. In particular, we will often consider the neighborhood of a vertex v in G, notated by $N_G(v)$, as well as the neighborhood of a vertex v in G_{new}, notated $N_{G_{new}}(v)$.

As graphs consist of edge and vertex sets, a graph may be transformed into any other through repeated application of four basic operations: adding a vertex, removing a vertex, adding an edge and removing an edge. These four shall be the first perturbation types that we consider. However, we first introduce two lemmas which will be invaluable in proving our results on the perturbed clique enumeration. The first of these lemmas is simply the observation that a clique of G will be a clique in G_{new} if no edge in the clique is removed in the perturbation.

Lemma 2.3.1. *Suppose the edges $(u_1, v_1), (u_2, v_2), \ldots, (u_j, v_j)$ are being removed from G and the edges $(x_1, y_1), (x_2, y_2), \ldots, (x_k, y_k)$ are being added to G to form G_{new}. For a vertex set C in G, if C is a clique in G and there is no i such that $u_i \in C$ and $v_i \in C$, then C is a clique in G_{new}.*

Proof. If no such i exists, no edge of C is being removed, so the vertices of C will be pairwise adjacent in G_{new}. $\square$

The second lemma guarantees that a maximal clique of G will be maximal in G_{new} if no edge incident to the clique is added in the perturbation.

Lemma 2.3.2. *Suppose the edges $(u_1, v_1), (u_2, v_2), \ldots, (u_j, v_j)$ are being added to G and the edges $(x_1, y_1), (x_2, y_2), \ldots, (x_k, y_k)$ are being removed from G to form G_{new}. For a vertex set $C \in \mathcal{C}$, if C is a clique in G_{new} and C contains no vertex x_i or y_i for any i, then $C \in \mathcal{C}_{new}$.*

Proof. We prove the claim by contradiction. Suppose $C \in \mathcal{C}$ is a clique in G_{new} and C contains no x_i or y_i, but $C \notin \mathcal{C}_{new}$. Since C is a clique in G_{new} and $C \notin \mathcal{C}_{new}$, there must be some clique $S \in \mathcal{C}_{new}$ such that $C \subset S$. Let $a \in S \setminus C$. Since S is a clique in G_{new}, S must contain no edge being removed from G. Also, since C contains no x_i or y_i, a must be adjacent to all vertices of C in G. However, this implies $C \cup \{a\}$ is a clique in G, contradicting our supposition that $C \in \mathcal{C}$. $\square$

2.4 Adding and removing a vertex

The first perturbational cases we consider are the addition and removal of a single vertex from a graph. The nature of adding or removing a vertex allows for simple, intuitive results due to guarantees about the adjacency information of the vertex involved.

2.4.1 Adding a vertex

The first, and simplest, of the four basic perturbations is the process of adding a vertex to a graph. In this section, we consider the perturbed graph G_{new} formed by adding a single vertex u to a graph G which does not contain u. In the following theorem, we see that the maximal clique enumeration of G_{new}, $\mathcal{C}_{new}$, differs from the maximal clique enumeration of G, $\mathcal{C}$, by the clique containing only the vertex u.

Theorem 2.4.1. $\mathcal{C}_{new} = \mathcal{C} \cup \{u\}$.

Proof. As u is not connected to any vertex of G, no cliques spanning the two exist, so the cliques of G_{new} will be the disjoint union of the cliques of G (the set $\mathcal{C}$) with the cliques of u ($\{u\}$). $\square$

2.4.2 Removing a vertex

The second of the four basic perturbations is the process of removing a vertex from a graph. In this section, we consider the perturbed graph G_{new} formed by removing a single vertex u from a graph G containing u. However, unlike the process of adding a vertex, there are two possible cases to consider. In the first, the vertex u is isolated in G, which is the inverse of adding the vertex to the graph. In the following theorem, we demonstrate that $\mathcal{C}_{new}$ can be formed from $\mathcal{C}$ by removing the clique containing only u.

Theorem 2.4.2. *If u is isolated, $\mathcal{C}_{new} = \mathcal{C} \setminus \{u\}$.*

Proof. G can be constructed by adding the vertex u to G_{new}, so $\mathcal{C} = \mathcal{C}_{new} \cup \{u\}$ by Theorem 2.4.1. Since $u \notin G_{new}$, $\{u\} \notin \mathcal{C}_{new}$, so $\mathcal{C}_{new} = \mathcal{C} \setminus \{u\}$. □

However, when the vertex u is not isolated, the process for ascertaining $\mathcal{C}_{new}$ becomes more difficult. Rather than having a single rule which defines $\mathcal{C}_{new}$ directly, we shall examine all the cliques of $\mathcal{C}$ and place theoretical conditions on their behavior as the vertex u is removed. (Later, when presenting our algorithmic results, we will use these conditions, along with indexing techniques, to avoid systematically considering every clique of $\mathcal{C}$.) When removing u from G, it is sufficient to divide the cliques of $\mathcal{C}$ based on whether or not each contains the vertex u. In the following theorem, we see that those cliques which do not contain u are retained from $\mathcal{C}$.

Theorem 2.4.3. *If $C \in \mathcal{C}$ and $u \notin C$, $C \in \mathcal{C}_{new}$.*

Proof. If $u \notin C$, C is a clique in G_{new} by Lemma 2.3.1. As no edges are being added to G, $C \in \mathcal{C}_{new}$ by Lemma 2.3.2. □

In the next theorem, we show that the remaining maximal cliques of $\mathcal{C}$, i.e., those cliques which contain u, form cliques in G_{new} once u has been removed. Further, we identify a maximality set for the new cliques—if no vertex of this set is adjacent to every vertex of the newly formed clique, the clique must be maximal in G_{new}.

Theorem 2.4.4. *If $C \in \mathcal{C}$ and $u \in C$, $C \setminus \{u\} \in \mathcal{C}_{new}$ if there exists no $x \in (\bigcup_{v \in N_G(u)} N_G(v)) \setminus (\{u\} \cup N_G(u))$ such that x is adjacent to every vertex of $C \setminus \{u\}$.*

Proof. As $N_G(u)$ is assumed to be nonempty, $C \neq \{u\}$, and since u has no neighbors in G_{new}, C (White area in Figure 2.1) will not be a clique in G_{new}. Consider $C \setminus \{u\}$. Since $C \setminus \{u\} \subseteq C$, $C \setminus \{u\}$ is a clique in G. Also, as $u \notin C \setminus \{u\}$, $C \setminus \{u\}$ is a clique in G_{new} by Lemma 2.3.1. Suppose $C \setminus \{u\}$ is not maximal in G_{new}. So, there exists some $S \in \mathcal{C}_{new}$ such that $C \setminus \{u\} \subset S$. Let $x \in S \setminus C$. Clearly, $x \notin C \setminus \{u\}$, and since S is a clique in G_{new}, x is adjacent to every vertex of $C \setminus \{u\}$ in G_{new}. Note, $x \neq u$, as u is is not adjacent to any vertex of $C \setminus \{u\}$ in G_{new}. Since $u \notin C \setminus \{u\}$ and $x \neq u$, x must be adjacent to every vertex of $C \setminus \{u\}$ in G. If $x \in N_G(u)$ (Vertical striped area in Figure 2.1), x is adjacent to u in G in addition to all vertices of $C \setminus \{u\}$, making $C \cup \{x\}$ a clique in G. However, this contradicts $C \in \mathcal{C}$, so $x \notin N_G(u)$. As all vertices of $C \setminus \{u\}$ are adjacent to u in G, $C \setminus \{u\} \subseteq N_G(u)$, and since $C \setminus \{u\}$ is nonempty, x is adjacent to at least one vertex of $N_G(u)$, so $x \in (\bigcup_{v \in N_G(u)} N_G(v)) \setminus (\{u\} \cup N_G(u))$ (Vertical striped gray area in Figure 2.1). □

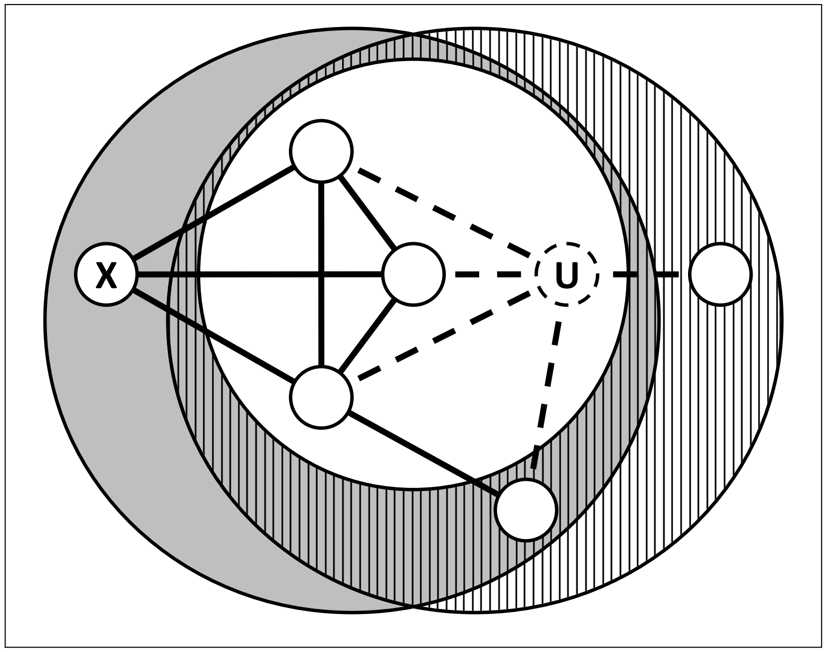


Figure 2.1: Example of a graph in the case of vertex removal. Edges with dotted lines are removed when vertex u is removed. Vertices in the white circle belong to the maximal clique C in the original graph. Vertices in the vertical striped circle belong to $N_G(u) \cup \{u\}$. Vertices in the gray circle belong to $\bigcup_{v \in N_G(u)} N_G(v)$.

Finally, we show that the process defined by the previous two theorems is sufficient for determining the set $\mathcal{C}_{new}$ from $\mathcal{C}$. The proof of Theorem 2.4.5 will demonstrate that the previous two results are sufficient to define the set $\mathcal{C}_{new}$.

Theorem 2.4.5. *All elements of $\mathcal{C}_{new}$ are considered by Theorems 2.4.3–2.4.4.*

Proof. Let S be an arbitrary member of $\mathcal{C}_{new}$. $u \notin S$, as u has been removed from G. Also, as no edges are removed from G_{new} to G, S is a clique in G by Lemma 2.3.1.

If $S \cap N_G(u) = \emptyset$, S is maximal in G by Lemma 2.3.2. As such, $S \in \mathcal{C}$. Since $u \notin S$, $S \in \mathcal{C}_{new}$ by Theorem 2.4.3.

Suppose $S \cap N_G(u) \neq \emptyset$, and let $v \in S \cap N_G(u)$. If $S \nsubseteq N_G(u)$, $S \setminus N_G(u) \neq \emptyset$. As no $v \in S \setminus N_G(u)$ is adjacent to u in G, u must not contradict the maximality of S in G. Since no other vertex could contradict the maximality of S in G without contradicting its (assumed) maximality in G_{new}, $S \in \mathcal{C}$. As $u \notin S$, $S \in \mathcal{C}_{new}$ by Theorem 2.4.3.

Otherwise, $S \subseteq N_G(u)$. Every vertex of S is adjacent to u in G, so $S \cup \{u\}$ is a clique in G. Suppose $S \cup \{u\}$ is not maximal in G. So, there exists some $C \in \mathcal{C}$ such that $(S \cup \{u\}) \subset C$. Let $x \in C \setminus S$. Clearly, $x \neq u$ and x is adjacent to all vertices of S in G. However, this implies x is adjacent to all vertices of S in G_{new}, contradicting $S \in \mathcal{C}_{new}$. Thus, $S \cup \{u\}$ must be maximal in G, so $S \cup \{u\} \in \mathcal{C}$. Since $S \in \mathcal{C}_{new}$, there is no a vertex adjacent to all vertices of $S = (S \cup \{u\}) \setminus \{u\}$ in G, so $(S \cup \{u\}) \setminus \{u\} = S \in \mathcal{C}_{new}$ by Theorem 2.4.4. □

2.5 Adding and removing a single edge

We now consider the other two basic perturbations, adding or removing an edge. Without specific guarantees on the neighbors of a vertex (as in the previous cases), the resulting necessary and sufficient conditions will be more complex, as in Subsection 2.4.2. From here on, all perturbational cases will be divided depending on which endpoints of perturbed edges are contained in the clique.

2.5.1 Adding an edge

The third, and most complex, of the four basic perturbations is the process of adding a single edge to a graph. In this section, we consider the perturbed graph G_{new} formed by adding edge (u, v) to the graph G, where u and v are nonadjacent vertices of G. As before, we split the cases being considered before identifying corresponding cliques in $\mathcal{C}_{new}$. In Theorem 2.5.1, we see that any maximal clique in $\mathcal{C}$ which contains neither u nor v is retained in $\mathcal{C}_{new}$.

Theorem 2.5.1. *If $C \in \mathcal{C}$ and $u, v \notin C$, $C \in \mathcal{C}_{new}$.*

Proof. Since no edge is being removed from G, C is a clique in G_{new} by Lemma 2.3.1, and since $u, v \notin C$, $C \in \mathcal{C}_{new}$ by Lemma 2.3.2. □

In Theorem 2.5.2, we see that cliques of $\mathcal{C}$ which contain exactly one of u or v are maximal cliques of G_{new} so long as the other vertex does not contradict their maximality.

Theorem 2.5.2. *If $C \in \mathcal{C}$ and $u \in C$, $C \in \mathcal{C}_{new}$ unless v is adjacent to every vertex of $C \setminus \{u\}$. Similarly, if $v \in C$, $C \in \mathcal{C}_{new}$ unless u is adjacent to every vertex of $C \setminus \{v\}$.*

Proof. Since no edges are being removed from G, C is a clique in G_{new} by Lemma 2.3.1, so $C \in \mathcal{C}_{new}$ if C is maximal in G_{new}. Suppose it is not, and let $S \in \mathcal{C}_{new}$ such that $C \subset S$. Let $x \in S \setminus C$. If $x \neq v$, x is adjacent to all vertices of C in G, contradicting our assumption that $C \in \mathcal{C}$. Thus, $x = v$, so v is adjacent to every vertex of C, proving the contrapositive.

The proof for when $v \in C$ follows similarly. □

In Theorem 2.5.3, we identify those cliques of $\mathcal{C}_{new}$ which are not contained in $\mathcal{C}$, that is, the novel cliques introduced by the perturbation. One caveat is that this procedure considers pairs of cliques from $\mathcal{C}$ and may prove costly when there are many such pairings.

Theorem 2.5.3. *If $C, D \in \mathcal{C}$, $u \in C$ and $v \in D$, then $E = (C \cap D) \cup \{u, v\} \in \mathcal{C}_{new}$ if there is no $x \in N_G(u) \cap N_G(v)$ such that $x \notin C \cup D$, but x is adjacent to every vertex of $C \cap D$.*

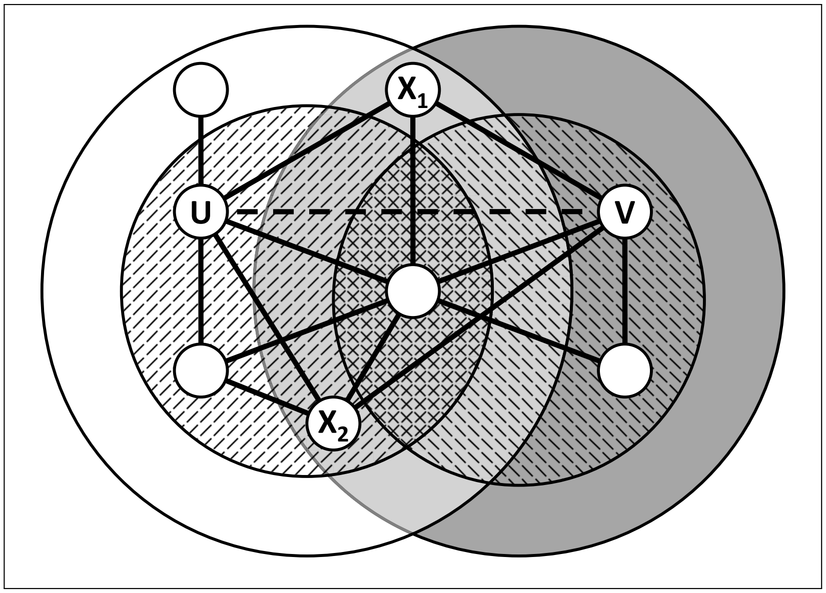


Figure 2.2: Example of a graph in the case of edge addition. The edge being added is the dotted line between vertices u and v. Vertices in the circle with slashes (/) belong to the clique C in the original graph, G. Vertices in the circle filled with backslashes (\) belong to the clique D in G. Vertices in the area filled with the $\times$'s belong to $C \cap D$. Vertices in the white circle belong to $N_G(u)$ and vertices in the dark gray circle belong to $N_G(v)$. Vertices in the light gray area belong to $N_G(u) \cap N_G(v)$.

Proof. First, we show E is a clique in G_{new}. Since $C \cap D \subset C$, all vertices of $C \cap D$ ($\times$ filled area in Figure 2.2) are pairwise adjacent and are adjacent to u. Since $C \cap D \subset D$, all vertices of $C \cap D$ are adjacent to v, and as edge (u, v) is being added to G, u and v are adjacent in G_{new}. As such, all vertices of E are pairwise adjacent.

Suppose E is not maximal in G_{new}. Thus, there exists some clique S such that $E \subset S$. Let $x \in S \setminus E$. Since $u, v \in E$, $u, v \in S$, so x is adjacent to u and v in G_{new}. Since no edge is being removed from G, this implies $x \in N_G(u) \cap N_G(v)$ (Light gray area in Figure 2.2). Also, as each vertex of $C \cap D$ is contained in S, x is also adjacent to every vertex of $C \cap D$ in G, proving the contrapositive. In Figure 2.2, both x_1 and x_2 could serve as x to prove that $E = (C \cap D) \cup \{u, v\}$ is not a maximal clique in the new graph. $\square$

Lastly, we conclude by demonstrating that the previous three necessary conditions are also sufficient for constructing $\mathcal{C}_{new}$ so as to show that $\mathcal{C}_{new}$ is completely defined by Theorems 2.5.1–2.5.3.

Theorem 2.5.4. *All elements of $\mathcal{C}_{new}$ are considered by Theorems 2.5.1–2.5.3.*

Proof. Let S be an arbitrary member of $\mathcal{C}_{new}$. S must contain neither u nor v, only u, only v

or both u and v. We consider the four cases individually.

G can be formed by removing edge (u, v) from G_{new}, so $S \in \mathcal{C}$ unless $u, v \in S$ by Lemma 2.3.1. If $u, v \notin S$, $S \in \mathcal{C}$ and was added to $\mathcal{C}_{new}$ by Theorem 2.5.1. If $u \in S$ but $v \notin S$, $S \in \mathcal{C}$, so there is no vertex in G adjacent to all vertices of S, so S was added to $\mathcal{C}_{new}$ by Theorem 2.5.2. Similarly, if $v \in S$ but $u \notin S$, S was added to $\mathcal{C}_{new}$ by Theorem 2.5.2. Finally, if $u, v \in S$, $S \setminus \{u\}$ and $S \setminus \{v\}$ are cliques in G, so will be contained in some $C \in \mathcal{C}$ and $D \in \mathcal{C}$ such that $u \in C$ and $v \in D$. Since $S \setminus \{u, v\}$ is a subset of both C and D, $S \subseteq (C \cap D) \cup \{u, v\}$. Since S is maximal, this implies $S = (C \cap D) \cup \{u, v\}$ and there is no vertex not in S adjacent to all vertices of S, so S was added to $\mathcal{C}_{new}$ by 2.5.3. □

2.5.2 Removing an edge

We now examine the fourth and final basic perturbation, removing a single edge from a graph. For this section, we will consider the perturbed graph G_{new} formed by removing the edge (u, v) from the graph G, where u and v are adjacent vertices in G. We distinguish the cases of cliques that contain both endpoints of the removed edge and those that do not. Theorem 2.5.5 demonstrates that those which do not contain both endpoints also form maximal cliques in the perturbed graph.

Theorem 2.5.5. *If $C \in \mathcal{C}$ and either $u \notin C$ or $v \notin C$, $C \in \mathcal{C}_{new}$.*

Proof. If $u \notin C$ or $v \notin C$, C is a clique in G_{new} by Lemma 2.3.1. Since no edges are being added to G to form G_{new}, C is maximal in G_{new} by Lemma 2.3.2, so $C \in \mathcal{C}_{new}$. □

Theorem 2.5.6 outlines two candidate cliques which may be formed from a clique of $\mathcal{C}$ containing both u and v, along with the vertices which may contradict their maximality in G_{new}. Unlike in previous perturbations, this implies a single clique in the graph may well correspond to two cliques in the perturbed graph, potentially increasing the number of cliques exponentially.

Theorem 2.5.6. *If $C \in \mathcal{C}$ and $u, v \in C$, $C \setminus \{u\} \in \mathcal{C}_{new}$ if there exists no $x \in N_G(v) \setminus N_G(u)$ such that x is adjacent to every vertex of $C \setminus \{u, v\}$. Similarly, $C \setminus \{v\} \in \mathcal{C}_{new}$ if there exists no $x \in N_G(u) \setminus N_G(v)$ such that x is adjacent to every vertex of $C \setminus \{u, v\}$.*

Proof. $C \setminus \{u\} \subset C$, so $C \setminus \{u\}$ is a clique in G, and since $u \notin C \setminus \{u\}$, $C \setminus \{u\}$ is a clique in G_{new} by Lemma 2.3.1. Suppose $C \setminus \{u\}$ is not maximal in G_{new}. So, there is some clique S such that $C \setminus \{u\} \subset S$. Let $x \in S \setminus (C \setminus \{u\})$. Since $v \in C$, x is adjacent to v in G_{new}, and x is adjacent to every vertex of $C \setminus \{u, v\}$. Since no edges are removed from G_{new} to G, $x \in N_G(v)$. If x is adjacent to u in G_{new}, x would be adjacent to u in G. However, this would make $C \cup \{x\}$ a clique in G, contradicting our assumption that $C \in \mathcal{C}_t$. Thus, x is adjacent to every vertex of $C \setminus \{u, v\}$ and $x \in N_G(v) \setminus N_G(u)$, proving the contrapositive of the claim.

The proof for $C \setminus \{v\}$ follows similarly. $\square$

Finally, we establish in Theorem 2.5.7 that, much as in Theorems 2.4.5 and 2.5.4, the previous conditions are sufficient to construct $\mathcal{C}_{new}$ from $\mathcal{C}$.

Theorem 2.5.7. *All elements of $\mathcal{C}_{new}$ are considered by Theorems 2.5.5–2.5.6.*

Proof. Let $C \in \mathcal{C}_{new}$. Since no edge is removed from G_{new} to G, C is a clique in G by Lemma 2.3.1. Since u and v are not adjacent in G_{new}, $u \in C$, $v \in C$ or neither u nor v is in C. If $u, v \notin C$, $C \in \mathcal{C}$ by Lemma 2.3.2, so $C \in \mathcal{C}_{new}$ by Theorem 2.5.5.

Suppose that $u \in C$ but $v \notin C$. If C is maximal in G, $C \in \mathcal{C}$ by Lemma 2.3.2, so $C \in \mathcal{C}_{new}$ by Theorem 2.5.5. Otherwise, C is not maximal in G_{new}. Since C is a clique in G, there exists some $S \in \mathcal{C}$ such that $C \subset S$. Let $x \in S \setminus C$. x is adjacent to all vertices of C in G, so unless $x = v$, x is adjacent to all vertices of C in G_{new}, contradicting $C \in \mathcal{C}_{new}$. Thus, $S \setminus C = \{v\}$, so $C = S \setminus \{v\}$. Since $u \in C \subset S$, $u \in S$ and $v = x \in S$, S contains both u and v. Since $C = S \setminus \{u\} \in \mathcal{C}_{new}$, there is no vertex adjacent to every vertex of $S \setminus \{u\}$, so $C = S \setminus \{u\} \in \mathcal{C}_{new}$ by Theorem 2.5.6.

Likewise, if $v \in C$, $C \in \mathcal{C}_{new}$ by Theorem 2.5.5 if C is maximal in G, and $C \setminus \{v\} \in \mathcal{C}_{new}$ by Theorem 2.5.6 if C is not maximal in G. $\square$

2.6 Multiple edges

While the theorems outlined in sections 2.4.1–2.5.2 are certainly sufficient to account for any perturbation, calculating the maximal clique enumeration of a graph by repeatedly adding or removing a single edge is rather inefficient. Many of the cliques generated by these intermediate perturbations may be removed as a result of a later perturbation; ideally, we would only wish to enumerate cliques which are maximal in the final state of the graph. Moreover, removing an edge from a graph may potentially double the number of cliques in a graph, so the potential for wasted effort is quite high.

To remedy this inefficiency, we consider perturbations in which multiple edges are added or removed simultaneously. Rather than analyzing the complex general case, we restrict our attention to perturbations in which the edges being added or removed share a common vertex.

2.6.1 Adding multiple edges

In this section, we consider the perturbed graph G_{new} formed by adding the edges $(u, v_1), (u, v_2), \dots, (u, v_k)$ to the graph G, where $u, v_1, \dots, v_k$ are vertices of G and the common vertex, u, is not adjacent to any vertex v_i. We also define the set $V = \{v_1, \dots, v_k\}$, for convenience.

As when adding a single edge, those maximal cliques of G which do not contain an endpoint of an edge being added are maximal cliques in the perturbed graph, which we prove in Theorem 2.6.1.

Theorem 2.6.1. *If $C \in \mathcal{C}$, $u \notin C$ and $C \cap V = \emptyset$, then $C \in C_{new}$.*

Proof. As no edges are being removed from G, C is a clique in G_{new} by Lemma 2.3.1, and as no edge incident to C is being added to G, $C \in \mathcal{C}_{new}$ by Lemma 2.3.2. $\square$

We demonstrate in Theorem 2.6.2 that a maximal clique of G containing the common endpoint u is a maximal clique of G_{new} so long as no vertex of V contradicts its maximality.

Theorem 2.6.2. *If $C \in \mathcal{C}$ and $u \in C$, $C \in \mathcal{C}_{new}$ if there is no $x \in V$ such that x is adjacent to every vertex of $C \setminus \{u\}$ in G.*

Proof. Suppose $C \notin \mathcal{C}_{new}$. Since no edges are being removed from G, C is a clique in G_{new} by Lemma 2.3.1, so C must not be maximal in G_{new}. Thus, there exists some $S \in \mathcal{C}_{new}$ such that $C \subset S$. Let $x \in S \setminus C$. x is adjacent to every vertex of C in G_{new}. $u \in C$, so $x \neq u$. Thus, if $x \notin V$, x will be adjacent to every vertex of C in G. This implies $C \cup \{x\}$ is a clique in G, contradicting our assumption that $C \in \mathcal{C}$. Note that since $x \in V$, such an x would be adjacent to every vertex of $C \setminus \{u\}$ in G.

Thus, $C \in \mathcal{C}_{new}$ so long as no such x exists. $\square$

In Theorem 2.6.3, we see that, much as when only a single edge was being added, only u can contradict the maximality of a clique of $\mathcal{C}$ containing some vertex of V.

Theorem 2.6.3. *If $C \in \mathcal{C}$ and $C \cap V \neq \emptyset$, $C \in \mathcal{C}_{new}$ if there is some vertex of $C \setminus V$ that is not adjacent to u in G.*

Proof. Suppose $C \notin \mathcal{C}_{new}$. C is a clique in G_{new} by Lemma 2.3.1, so C must not be maximal in G_{new}. So, there exists some $S \in \mathcal{C}_{new}$ such that $C \subset S$. If $u \notin S \setminus C$, S is a clique in G by Lemma 2.3.1, but since $C \subset S$, this would contradict $C \in \mathcal{C}$. Thus, $u \in S \setminus C$. Since $u \in S \setminus C$ and S is a clique in G_{new}, u must be adjacent to all vertices of C in G_{new} and so must be adjacent to all vertices of $C \setminus V$ in G.

Thus, so long as some vertex of $C \setminus V$ is not adjacent to u, this must not be the case and $C \in \mathcal{C}_{new}$, by the contrapositive. $\square$

We now construct the cliques which have been introduced by the perturbation from pairs of maximal cliques in $\mathcal{C}$, and isolate those vertices of G which may contradict the maximality of the constructed cliques.

Theorem 2.6.4. *If $C, D \in \mathcal{C}$, $u \in C$ and $D \cap V \neq \emptyset$, then $E = ((C \cap D) \cup \{u\} \cup (D \cap V)) \in \mathcal{C}_{new}$ if there is no $x \in (N_G(u) \cup V) \cap (\bigcup_{v \in V} N_G(v))$ such that x is adjacent to every vertex of $(C \cap D) \cup (D \cap V)$ in G.*

Proof. First, we show that E is a clique in G_{new}. Note, as both C and D are cliques in G and no edges are being removed from G, vertices of $C \cap D$ are pairwise adjacent in G_{new}, and vertices of $D \cap V$ are also pairwise adjacent in G_{new}. As $u \in C \setminus (C \cap D)$ and C is a clique in G, u will be adjacent to every vertex of $C \cap D$ in G_{new}. As no edges are being removed and $C \cap D$ and $D \cap V$ are subsets of the clique D, all vertices of these two sets will be pairwise adjacent in G_{new}. Finally, as edges of the form (u, v_i) for all $v_i \in V$ are being added to G and $D \cap V \subset V$, all vertices of $D \cap V$ will be adjacent to u in G_{new}. Thus, all vertices of E will be pairwise adjacent in G_{new}, so E is a clique in G_{new}.

Suppose $E \notin \mathcal{C}_{new}$. Since E is a clique in G_{new}, E must not be maximal in G_{new}. So, there exists some $S \in \mathcal{C}_{new}$ such that $E \subset S$. Let $x \in S \setminus E$. Note, as $u \in E$, $x \neq u$. As $D \cap V \neq \emptyset$, there is some $v \in D \cap V$. $v \in D \cap V \subset E \subset S$. Since $v \in S$ and $x \notin E$, x and v are adjacent in G_{new}, so $u \in \displaystyle\bigcup_{v \in V} N_{G_{new}}(v)$. Since $x \neq u$, this implies $x \in \displaystyle\bigcup_{v \in V} N_G(v)$. As $u \in E$, $u \in S$, so x is adjacent to u in G_{new}. As such, $x \in N_G(u) \cup V$. Thus, $x \in (N_G(u) \cup V) \cap (\bigcup_{v \in V} N_G(v))$. Since $x \in S \setminus E$, $x \notin E$, and as $E \subset S$, x is adjacent to all vertices of E in G_{new}. So, since $x \neq u$, x is adjacent to every vertex of $(C \cap D) \cup (D \cap V)$ in G.

Thus, if no such x exists, $E \in \mathcal{C}_{new}$. □

Again, we demonstrate that the constructive theorems define all of $\mathcal{C}_{new}$.

Theorem 2.6.5. *All elements of $\mathcal{C}_{new}$ are considered by Theorems 2.6.1–2.6.4.*

Proof. Let $S \in \mathcal{C}_{new}$. If $u \notin S$ or $S \cap V = \emptyset$, S is a clique in G by Lemma 2.3.1. As no edge is being added from G_{new} to G, $S \in \mathcal{C}$ by Lemma 2.3.2. If $u \notin S$ and $S \cap V = \emptyset$, $S \in \mathcal{C}_{new}$ by Theorem 2.6.1.

Suppose $u \in S$ but $S \cap V = \emptyset$. If some $v \in V$ is adjacent to every vertex of $S \setminus \{u\}$, $S \cup \{v\}$ is a clique in G_{new}, contradicting $S \in \mathcal{C}_{new}$. Thus, no such v exists, so $S \in \mathcal{C}_{new}$ by Theorem 2.6.2.

Suppose $S \cap V \neq \emptyset$ but $u \notin S$. So, there is some $v \in S \cap V$. If all vertices of $S \setminus V$ are adjacent to u, $S \cup \{u\}$ is a clique in G_{new}, contradicting $S \in \mathcal{C}_{new}$. Thus, some vertex of $S \setminus V$ must not be adjacent to u in G, so $S \in \mathcal{C}_{new}$ by Theorem 2.6.3.

If the three previous cases are false, $u \in S$ and $S \cap V \neq \emptyset$. In this case, $S \setminus \{u\}$ and $S \setminus V$ are cliques in G. Let $C, D \in \mathcal{C}$ be such that $S \setminus V \subseteq C$ and $S \setminus \{u\} \subseteq D$. Note, $u \in C$ and $D \cap V \neq \emptyset$. $C \cap D = (S \setminus \{u\}) \cap (S \setminus V) = (S \setminus \{u\}) \setminus V$. $D \cap V = (S \setminus \{u\}) \cap V = S \cap V$, as $u \notin V$. As such, $(C \cap D) \cup \{u\} \cup (D \cap V) = ((S \setminus \{u\}) \setminus V) \cup \{u\} \cup (S \cap V) = S$. If some vertex x not in S is adjacent to every vertex of S in G, $S \cup \{x\}$ is a clique in G_{new}, contradicting

$S \in \mathcal{C}_{new}$. Thus, no such x can exist, so $S = (C \cap D) \cup \{u\} \cup (D \cap V) \in \mathcal{C}_{new}$ by Theorem 2.6.4. $\square$

2.6.2 Adding multiple edges to an isolated vertex

As a variant to the previous case, we now consider a perturbation where multiple edges incident to an isolated vertex are being added. Essentially the inverse process of removing a vertex which is not isolated, this perturbation offers significant advantages over the previous case in terms of computational complexity. In particular, we no longer need to consider pairs of cliques when constructing the novel cliques introduced by the perturbation.

As before, we let u be the common vertex and let $V = \{v_1, v_2, \ldots, v_k\}$ be the set of "outer" vertices. In the following theorem, we show that those cliques of $\mathcal{C}$ not containing an endpoint of one of the added edges are contained in $\mathcal{C}_{new}$.

Theorem 2.6.6. *If $C \in \mathcal{C}$, $u \notin C$ and $C \cap V = \emptyset$, then $C \in \mathcal{C}_{new}$.*

Proof. As no edges are being removed from G, C is a clique in G_{new} by Lemma 2.3.1, and as no edge incident to C is being added to G, $C \in \mathcal{C}_{new}$ by Lemma 2.3.2. $\square$

We prove, in Theorem 2.6.7, that the maximal clique consisting of just u is not a maximal clique in the perturbed graph. Theorem 2.6.7 is not, strictly speaking, necessary for constructing the set $\mathcal{C}_{new}$—no clique is indicated to be contained in $\mathcal{C}_{new}$—but it is included to complete the accounting of cliques from $\mathcal{C}$.

Theorem 2.6.7. *If $C \in \mathcal{C}$ and $u \in C$, $C = \{u\}$ and $C \notin \mathcal{C}_{new}$.*

Proof. Since u has no neighbors in G and $C \in \mathcal{C}$, C must equal $\{u\}$. As V is assumed to be nonempty, (u, v_1) is a clique in G_{new} containing $C = \{u\}$, so $C \notin \mathcal{C}_{new}$. $\square$

In Theorem 2.6.8, we establish that a clique containing some vertex of V is a maximal clique in the perturbed graph so long as it is not a subset of the set V.

Theorem 2.6.8. *If $C \in \mathcal{C}$ and $C \cap V \neq \emptyset$, $C \in \mathcal{C}_{new}$ unless $C \subseteq V$.*

Proof. Suppose $C \subseteq V$. As u is adjacent to every vertex of V in G_{new}, u is adjacent to every vertex of C in G_{new}, making $C \cup \{u\}$ a clique G_{new}, so $C \notin \mathcal{C}_{new}$.

Suppose $C \nsubseteq V$ and $C \notin \mathcal{C}_{new}$. Since no edge is being removed from G, C is a clique in G_{new} by Lemma 2.3.1, so C must not be maximal in G_{new}. Thus, there exists some $S \in \mathcal{C}_{new}$ such that $C \subset S$. Let $x \in S \setminus C$. If $x \neq u$, x would be adjacent to every vertex of C in G, but this would contradict our assumption that $C \in \mathcal{C}$. As such, $S = C \cup \{u\}$. However, since $C \nsubseteq V$, there exists some $y \in C$ such that $y \notin V$, and since $N_{G_{new}}(u) = v$, y is not adjacent to u in G_{new}. As this contradicts our supposition of $S \in \mathcal{C}_{new}$, $C \in \mathcal{C}_{new}$ for any C such that $C \nsubseteq V$. $\square$

We now construct the novel cliques introduced by the perturbation. Unlike previous perturbations, this construction does not require us to consider pairs of cliques, resulting in significantly reduced computational complexity.

Theorem 2.6.9. *If $C \in \mathcal{C}$ and $C \cap V \neq \emptyset$, $(C \cap V) \cup \{u\} \in \mathcal{C}_{new}$ if there is no $x \in V \setminus C$ such that x is adjacent to all vertices of $C \cap V$.*

Proof. Since $C \cap V \neq \emptyset$, $v \in C$ for some $v \in V$. As $C \cap V \subseteq C$, all vertices of $C \cap V$ will be pairwise adjacent, and as $C \cap V \subseteq V$ and every vertex of V is adjacent to u in G_{new}, $(C \cap V) \cup \{u\}$ is a clique in G_{new}. Let $D = (C \cap V) \cup \{u\}$, and suppose D is not maximal in G_{new}. So, there exists some $S \in \mathcal{C}_{new}$ such that $D \subset S$. Let $x \in S \setminus D$. Since $u \in D \subset S$, x is adjacent to u in G_{new}, and since $N_{G_{new}}(u) = V$, $x \in V$. As $x \notin D$, $x \notin C \cap V$, so $x \in V \setminus C$, since $x \in V$. Also, since $x \in S \setminus D$, x is adjacent to every vertex of D in G_{new}, so x is adjacent to every vertex of $D \setminus \{u\} = C \cap V$ in G. Thus, such an $x \in C \setminus V$ must exist if D is not maximal, proving the contrapositive. □

Finally, we demonstrate that the previous theorems fully account for the members of $\mathcal{C}_{new}$.

Theorem 2.6.10. *All elements of C_{new} are considered by Theorems 2.6.6–2.6.9.*

Proof. Let S be an arbitrary member of $\mathcal{C}_{new}$. As no edges are added from G_{new} to G, $S \in \mathcal{C}$ so long as S is a clique in G by Lemma 2.3.2. Since u is adjacent to only the vertices of V in G_{new}, $S \cap V \neq \emptyset$ if $u \in S$. We consider the three remaining possibilities for S: $u \notin S$ and $S \cap V = \emptyset$, $u \notin S$ and $S \cap V \neq \emptyset$, and $u \in S$ and $S \cap V \neq \emptyset$.

If $u \notin S$ or $S \cap V = \emptyset$, S is a clique in G by Lemma 2.3.1, so $S \in \mathcal{C}$. If $u \notin S$ and $S \cap V = \emptyset$, $S \in \mathcal{C}_{new}$ by 2.6.6.

Suppose $S \cap V \neq \emptyset$ but $u \notin S$. By Lemma 2.3.1, S is a clique in G, so $S \in \mathcal{C}$. If $S \subseteq V$, $S \cup \{u\}$ would be a clique in G_{new}, contradicting $S \in \mathcal{C}_{new}$. As such, $S \nsubseteq V$, so $S \in \mathcal{C}_{new}$ by 2.6.8.

In the final case, $u \in S$ and $S \cap V \neq \emptyset$. Consider $S_u = S \setminus \{u\}$. $S_u \subset S$, so S_u is clique in G_{new}, and since $u \notin S_u$, S_u is a clique in G by Lemma 2.3.1. Since $S \setminus \{u\}$ is a clique in G by Lemma 2.3.1, there is some clique $C \in \mathcal{C}$ such that $S_u \subseteq C$. Since u is adjacent only to vertices of V in G_{new}, $S_u \subseteq V$. So, since $S_u \subseteq V$ and $S_u \subseteq C$, $S_u \subseteq C \cap V$. Suppose $S_u \neq C \cap V$, and let $x \in (C \cap V) \setminus S_u$. Since $x \in (C \cap V) \setminus S_u$, $x \in V$. So, x is adjacent to u in G_{new}, and since x is adjacent to all vertices of S_u in G_{new}, x must be adjacent to all vertices of S in G_{new}. However, this contradicts our assumption that $S \in \mathcal{C}_{new}$. As such, $S_u = S \setminus \{u\} = C \cap V$, so $S = (C \cap V) \cup \{u\}$. As S is maximal in G_{new}, there is no vertex of G that is adjacent to all vertices of S, so $S \in \mathcal{C}_{new}$ by 2.6.9. □

2.6.3 Removing multiple edges

Finally, we consider the reverse case of adding multiple edges incident to a common vertex—removing multiple edges incident to a single vertex. Echoing our previous notation, we consider the perturbed graph G_{new} formed by removing the edges $(u, v_1), (u, v_2), \ldots, (u, v_k)$ from G, where $u, v_1, \ldots, v_k$ are vertices of G with u adjacent to all vertices of $V = \{v_1, \ldots, v_k\}$.

We demonstrate, in Theorem 2.6.11, that those cliques of $\mathcal{C}$ which do not contain any edge being removed are maximal cliques of the perturbed graph.

Theorem 2.6.11. *If $C \in \mathcal{C}$ and either $u \notin C$ or $C \cap V = \emptyset$, $C \in \mathcal{C}_{new}$.*

Proof. If $u \notin C$ or $C \cap V = \emptyset$, C is a clique in G_{new} by Lemma 2.3.1. Since no edges are being added to G to form G_{new}, C is maximal in G_{new} by Lemma 2.3.2, so $C \in \mathcal{C}_{new}$. □

In Theorems 2.6.12 and 2.6.13, we show that, much as when removing a single edge, each maximal clique of G containing an edge to be removed may potentially correspond to two maximal cliques in the perturbed graph. We also identify the set of vertices which can establish this maximality.

Theorem 2.6.12. *If $C \in \mathcal{C}$, $u \in C$ and $C \cap V \neq \emptyset$, then $C \setminus \{u\} \in \mathcal{C}_{new}$ if there exists no $x \in (\bigcup_{v \in V} N_G(v)) \setminus (\{u\} \cup N_G(u))$ such that x is adjacent to every vertex of $C \setminus \{u\}$.*

Proof. Consider $C \setminus \{u\}$. Since $C \setminus \{u\} \subset C$, $C \setminus \{u\}$ is a clique in G, and since $u \notin C \setminus \{u\}$, $C \setminus \{u\}$ is a clique in G_{new} by Lemma 2.3.1.

Suppose $C \notin \mathcal{C}_{new}$. By the contrapositive of Lemma 2.3.2, this implies $C \setminus \{u\}$ is not maximal in G_{new}. So, there is some clique $S \in \mathcal{C}_{new}$ such that $C \setminus \{u\} \subset S$. Let $x \in S \setminus (C \setminus \{u\})$. x must be adjacent to every vertex of $C \setminus \{u\}$, and since $C \cap V \neq \emptyset$, x is adjacent to some $v \in V$ in G_{new}. Since no edges are removed from G_{new} to G, $x \in N_G(v)$, so $x \in (\bigcup_{v \in V} N_G(v))$. Also, as u is not adjacent to any vertex of V in G_{new}, $x \neq u$. If x is adjacent to u in G_{new}, x would be adjacent to u in G. However, this would make $C \cup \{x\}$ a clique in G, contradicting our assumption that $C \in \mathcal{C}$. Thus, x is adjacent to every vertex of $C \setminus \{u\}$ and $x \in (\bigcup_{v \in V} N_G(v)) \setminus (\{u\} \cup N_G(u))$, proving the contrapositive of the claim. □

Theorem 2.6.13. *If $C \in \mathcal{C}$, $u \in C$ and $C \cap V \neq \emptyset$, then $C \setminus V \in \mathcal{C}_{new}$ if there exists no $x \in N_G(u) \setminus V$ such that x is adjacent to every vertex of $C \setminus (\{u\} \cup V)$.*

Proof. Consider $C \setminus V$. Since $C \setminus V \subset C$, $C \setminus V$ is a clique in G, and since $(C \setminus V) \cap V = \emptyset$, $C \setminus V$ is a clique in G_{new} by Lemma 2.3.1.

Suppose $C \setminus V \notin \mathcal{C}_{new}$. By the contrapositive of Lemma 2.3.2, this implies $C \setminus V$ is not maximal in G_{new}. So, there is some clique S such that $C \setminus V \subset S$. Let $x \in S \setminus (C \setminus V)$. Since $u \in C \setminus V$, x is adjacent to u in G_{new}. Since no edges are removed from G_{new} to G, $x \in N_G(u)$, and x must be adjacent to every vertex of $C \setminus (\{u\} \cup V)$ in G. Since x is adjacent to u in G_{new}, $x \notin V$. Thus, x is adjacent to every vertex of $C \setminus (\{u\} \cup V)$ and $x \in N_G(u) \setminus V$, proving the contrapositive of the claim. □

Lastly, we prove that the previous conditions are sufficient to account for all maximal cliques of the perturbed graph.

Theorem 2.6.14. *All elements of $\mathcal{C}_{new}$ are considered by Theorems 2.6.11–2.6.13.*

Proof. Let $C \in \mathcal{C}_{new}$. Since no edge is removed from G_{new} to G, C is a clique in G by Lemma 2.3.1. Since u is not adjacent to any vertex of V in G_{new}, it cannot be the case that $u \in C$ and $C \cap V \neq \emptyset$. Thus, $u \in C$ and $C \cap V = \emptyset$, $u \notin C$ and $C \cap V \neq \emptyset$, or $u \notin C$ and $C \cap V = \emptyset$.

If $u \notin C$ and $C \cap V = \emptyset$, $C \in \mathcal{C}$ by Lemma 2.3.2, so $C \in \mathcal{C}_{new}$ by Theorem 2.6.11.

Suppose that $u \in C$ and $C \cap V = \emptyset$. If C is maximal in G, $C \in \mathcal{C}$ by Lemma 2.3.2, in which case $C \in \mathcal{C}_{new}$ by Theorem 2.6.11. So, suppose C is not maximal in G. Since C is a clique in G, there exists some $S \in \mathcal{C}$ such that $C \subset S$. Let $x \in S \setminus C$. x is adjacent to all vertices of C in G, so unless $x \in V$, x is adjacent to all vertices of C in G_{new}, contradicting $C \in \mathcal{C}_{new}$. Thus, $S \setminus C \subseteq V$, so $C \supseteq S \setminus V$. Since $C \cap V = \emptyset$, $C \subseteq S \setminus V$, so $C = S \setminus V$. $x \in S \cap V$, so $S \cap V \neq \emptyset$. Since $C = S \setminus V \in \mathcal{C}_{new}$, there can be no vertex of $N_G(u) \setminus V$ adjacent to every vertex of $S \setminus V$, so $C = S \setminus V \in \mathcal{C}_{new}$ by Theorem 2.6.13.

Suppose that $u \notin C$ and $C \cap V \neq \emptyset$. If C is maximal in G, $C \in \mathcal{C}$ by Lemma 2.3.2, in which case $C \in \mathcal{C}_{new}$ by Theorem 2.6.11. So, suppose C is not maximal in G. Since C is a clique in G, there exists some $S \in \mathcal{C}$ such that $C \subset S$. Let $x \in S \setminus C$. x is adjacent to all vertices of C in G, so unless $x = u$, x is adjacent to all vertices of C in G_{new}, contradicting $C \in \mathcal{C}_{new}$. Thus, $S \setminus C = \{u\}$, so $C = S \setminus \{u\}$. Since $C \cap V \subset S \cap V$, $S \cap V \neq \emptyset$. As $C = S \setminus \{u\} \in \mathcal{C}_{new}$, there also must be no vertex adjacent to every vertex of $S \setminus \{u\}$, so $C = S \setminus \{u\} \in \mathcal{C}_{new}$ by Theorem 2.5.6. □

2.7 Algorithm

As the theoretical results for adding and removing vertices are straightforward, we present two algorithms for calculating the clique set of a perturbed graph that utilize the theory and theorems presented in Sections 2.4–2.6. Algorithm 1 is designed for the case in which edges are removed from a graph, and Algorithm 5 is designed for when edges are added.

In the case of edge removal, due to the theory outlined in Theorems 2.6.11 and 2.6.14, the set $\mathcal{C}_-$ will consist of exactly those cliques of $\mathcal{C}$ that contain at least one edge being removed. Theorems 2.6.12, 2.6.13, and 2.6.14 imply that $\mathcal{C}_+$ can be calculated by taking the cliques C of $\mathcal{C}_-$ and iteratively considering the subcliques $C \setminus \{u\}$ and $C \setminus V$ for each set of removed edges sharing a common endpoint u and checking these subcliques for maximality.

Two observations are employed in the design of our algorithm to improve performance. The first is that identifying cliques containing a removed edge would normally require scanning every clique of $\mathcal{C}$ for the removed edge—this is particularly inefficient if the graph G undergoes multiple perturbations. To remedy this inefficiency, we index the set of cliques by the edges they contain so that the set $\mathcal{C}_-$ may be "retrieved" much more quickly.

The second observation is that in the process of removing edges, many vertices that were adjacent to the clique (or formerly in the clique) may be checked multiple times to see if they are adjacent to all vertices of the clique. Thus, rather than repeatedly checking adjacency with vertices of the clique, we analyze the connectivity between the clique and its neighboring vertices, store this information in a data structure (lines 4–7 of Algorithm 1), and update this information as vertices are removed from the clique (in the function `remove_from_clique`). In this fashion, nonmaximal cliques can be detected when a vertex in the neighborhood of the clique has its last nonadjacent vertex of the clique removed. Pseudocode for enumerating the difference sets $\mathcal{C}_-$ and $\mathcal{C}_+$ when edges are being removed is detailed in Algorithms 1 and 2 and the `remove_from_clique` and `return_to_clique` functions.

Algorithm 1: Pseudocode outline of the perturbation algorithm

Input: G : graph from which edges are being removed
Input: E : set of edges being removed from G to form G_{new}
Input: $\mathcal{C}$: set of maximal cliques of G
Output: $\mathcal{C}_-$: set of maximal cliques of G that are no longer maximal in G_{new}
Output: $\mathcal{C}_+$: set of maximal cliques of G_{new} that were not maximal in G

1 **Algorithm:** Perturbed_MCE_Removal
2 $\mathcal{C}_- \leftarrow$ cliques of G containing an edge of E
3 **foreach** clique *clique* in $\mathcal{C}_-$ **do**
4   *counter* $\leftarrow$ set of vertices of G adjacent to some vertex of *clique*
5   *nonadj* $\leftarrow$ associative map between vertices of *clique* and the set of *clique* vertices to which they are nonadjacent
6   *counter_nonadj* $\leftarrow$ associative map between vertices of *clique* and the set of *counter* vertices to which they are nonadjacent
7   *num_nonadj* : associative map between vertices of *counter* and the number of *clique* vertices to which they are nonadjacent
8   **call** `recursive_removal` (Algorithm 2)
9 **end**

Algorithm 2: Pseudocode of the recursive algorithm for finding subsets of a clique that form a maximal clique in either G or G_{new}—functions `remove_from_clique` and `return_to_clique` are defined separately

```
Algorithm: recursive_removal
if all edges of E have been removed from clique then
    Add clique to C+
else
    v ← a vertex of clique with nonadj[v] ≠ {}
    if remove_from_clique(v) returns success then
        call recursive_removal
    call return_to_clique(v)
    foreach x in nonadj[v] do
        call remove_from_clique(x)
    end
    if no remove_from_clique call returned failure then
        call recursive_removal
    foreach x in nonadj[v] do
        call return_to_clique(x)
    end
end
```

Algorithm 3: Function for removing a vertex from a clique

```
remove_from_clique(v):
Move v from clique to counter
num_nonadj[v] ← |nonadj[v]|
foreach x in nonadj[v] do
    Move v from nonadj[x] to counter_nonadj[x]
end
foreach x in counter_nonadj[v] do
    num_nonadj[x] ← num_nonadj[x] - 1
end
if some num_nonadj value was reduced to zero then
    return failure
else
    return success
```

Algorithm 4: Function for adding a vertex back to a clique

`return_to_clique`(v):
Move v from *counter* to *clique*
foreach x in *nonadj*$[v]$ **do**
    Move v from *counter_nonadj*$[x]$ to *nonadj*$[v]$
end
foreach x in *counter_nonadj*$[v]$ **do**
    $num\_nonadj[x] \leftarrow num\_nonadj[x] + 1$
end

Though our theoretical results for edge additions rely on considering pairs of cliques, we choose to instead base the implementation of the algorithm for handling edge additions by applying our results for edge removal to the inverse perturbation. That is, we consider the perturbation that would transform G_{new} to G by removing from the set of edges being added to G, and we enumerate the difference sets $\mathcal{C}_+$ and $\mathcal{C}_-$ in the same fashion as we would $\mathcal{C}_+$ and $\mathcal{C}_-$, respectively, in the removal case. By our results for edge removal, the cliques of $\mathcal{C}_+$ will be exactly those maximal cliques of G_{new} containing an edge being added, and we can apply our edge removal algorithm to the cliques of C_+ to calculate the set $\mathcal{C}_-$ (lines 18–23 of Algorithm 5). However, the edge removal algorithm may overestimate the set $\mathcal{C}_-$ by including cliques that are not maximal in G, which we resolve by removing any resulting clique that was not a maximal clique of G (lines 25–26). By repeated application of Theorems 2.5.1–2.5.4, we see that the cliques of $\mathcal{C}_-$ will all be subsets of cliques from $\mathcal{C}_+$, so this process is guaranteed to enumerate all of $\mathcal{C}_-$. For our implementation, we used a modification of the BK algorithm [10] to calculate the cliques of G_{new} containing an edge that was added to G. As BK operates by expanding the vertex set *compsub* to find maximal cliques, we need only to "seed" *compsub* with our edges of G_{new} that were added (lines 4–16 of Algorithm 5) in order to find all of the maximal cliques that contain these edges in G_{new}.

Algorithm 5: Pseudocode algorithm for discovering the cliques of G containing an edge of E in the case of edge addition; *compsub*, *candidate* and *not* are arrays used during the operation of BK

Input: G : graph from which edges are being removed
Input: E : set of edges being removed from G to form G_{new}
Input: $\mathcal{C}$: set of maximal cliques of G
Output: $\mathcal{C}_-$: set of maximal cliques of G that are no longer maximal in G_{new}
Output: $\mathcal{C}_+$: set of maximal cliques of G_{new} that were not maximal in G

Algorithm: Perturbed_MCE_Addition
$\mathcal{C}_+ \leftarrow \{\}$
$V \leftarrow$ the set of vertices incident to an edge in E
not $\leftarrow \{\}$
foreach vertex v in V **do**
    $U \leftarrow$ the set of vertices joined to v by an edge of E
    Remove vertices of *not* from U
    foreach u in U **do**
        compsub $\leftarrow$ {u, v}
        candidate $\leftarrow$ the set of all common neighbors of u and v not in *not*
        Execute BK using *compsub*, *candidate* and *not*
        Add resulting maximal cliques to $\mathcal{C}_+$
        Add u to *not*
    end
    Remove vertices of U from *not*
    Add v to *not*
end
foreach clique *clique* in $\mathcal{C}_+$ **do**
    counter $\leftarrow$ set of vertices of G adjacent to some vertex of *clique*
    nonadj $\leftarrow$ associative map between vertices of *clique* and the set of *clique* vertices to which they are nonadjacent
    counter_nonadj $\leftarrow$ associative map between vertices of *clique* and the set of *counter* vertices to which they are nonadjacent
    num_nonadj : associative map between vertices of *counter* and the number of *clique* vertices to which they are nonadjacent
    call `recursive_removal` (Algorithm 2)
end
foreach clique *clique* in $\mathcal{C}_-$ **do**
    if *clique* $\notin \mathcal{C}$ **then**
        Remove *clique* from $\mathcal{C}_-$
end

2.7.1 Indices

The algorithm employs four indices on the maximal cliques. These take the form of dense indices, in which a pointer associated with each value in the domain (the *key* or indexed attribute) points to the location of first value associated with that key in a sorted data file. Thus, to find all of the values associated with key value k_i, we look up where the values associated with keys k_i and k_{i+1} begin and perform a sequential read on the sorted data file between these two offsets. Note that this approach, while efficient and easy to implement, stores a pointer even for keys that have no associated values, such as edge IDs that do not correspond to edges of the original graph and thus have no associated cliques.

The first index maps a clique identification value, or clique ID, to the IDs of the vertices contained in the clique. The other three indices, described below, leverage this index in order to look up whether a candidate clique was contained in $\mathcal{C}$, which cliques of $\mathcal{C}$ contain a specific edge, and what vertices of G are adjacent to some vertex of a particular clique.

The second index maps a hash value to each clique ID, so that when a candidate clique is discovered by the algorithm, it can be compared to every maximal clique of $\mathcal{C}$ that maps to the same hash value. As the algorithm does not maintain a strict ordering of candidate cliques, the hash value is calculated by hashing each vertex of the clique individually and combining these using the commutative exclusive or (XOR) operation.

The third index maps each edge of G to the clique IDs of cliques in $\mathcal{C}$ containing the particular edge, which can then be combined with the previous index relating the clique ID to the set of vertices in the clique. Thus, the algorithm can enumerate the set $\mathcal{C}_-$ in the case of edge removal by simply iterating through the set of edges being removed and accumulating the clique IDs in a hash table to avoid duplicate cliques.

The final index maps clique IDs to the vertices that are adjacent to some vertex of the clique. These vertex sets are then retrieved, in the case of edge removal, alongside the vertices of a clique of $\mathcal{C}_-$ and used as an initial value for the counter set.

2.8 Results

To measure the effectiveness of an algorithm based on the theorems presented in this paper, we implemented the algorithm in Section 2.7 and compared its performance when enumerating the maximal cliques of perturbed graphs to that of a standard maximal clique enumeration (MCE) algorithm. We refer to the implementation of the algorithm as PGCE (Perturbed Graph Clique Enumerator). We measure the PGCE algorithm's performance in comparison to the well-known, efficient MCE algorithm of Bron and Kerbosch [10] (BK). Our implementation of the BK algorithm comes from the authors of [67], which use the version of BK described in

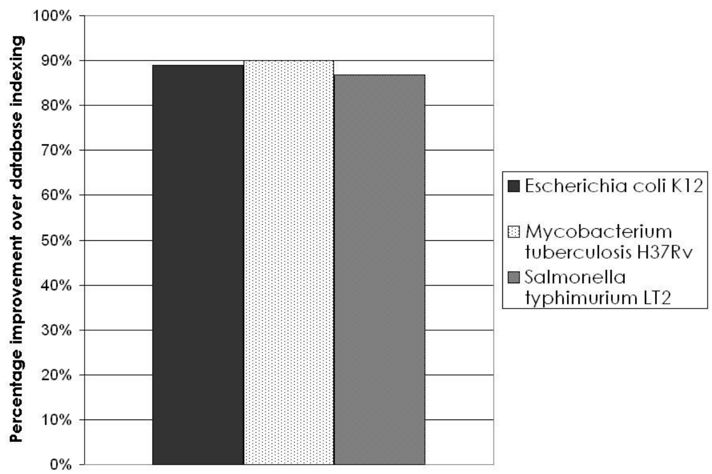


Figure 2.3: Percentage improvement in runtime for file index–based implementation over MySQL–based implementation of edge removal algorithm for three example networks

[10]. This implementation made use of a number of data structures to allow the BK algorithm to perform comparably if not better to more recent MCE algorithms such as those of Makino and Uno [49] and Tomita et al. [74]. Though the BK implementation in [67] allows for parallel execution, we only compared the PGCE algorithm to serial (single processor) execution of the algorithm.

Initially, the algorithm presented in Section 2.7 was implemented in the C programming language, and a MySQL database was used to store the indices. However, it soon became apparent that the overhead introduced by the database, particularly with regard to the database-intensive retrieval of the set $\mathcal{C}_{-}$, largely negated the benefits of calculating the maximal clique enumeration by the perturbed algorithm. Consequently, a file-based indexing scheme was adopted in place of MySQL. As shown in Figure 2.3, file-based indexing achieved a speedup of 85–90% over the MySQL implementation on three example networks.

We performed two further tests in this section. In the first, we tested the scalability of the algorithm relative to BK by running it on several synthetic graphs, varying the number of vertices and edges. In the second, we tested PGCE against BK on real-world biological networks, in which perturbations were induced by varying a threshold value for the weighted edges.

For the first test, we evaluated the algorithm's scalability to larger and more dense graphs relative to BK. For this purpose, we generated several graphs with various vertex and edge counts using the graph generator R-MAT [16]. To test the scalability to larger sparse graphs, we generated two series of sparse graphs, one with 8 times as many edges as vertices (for an

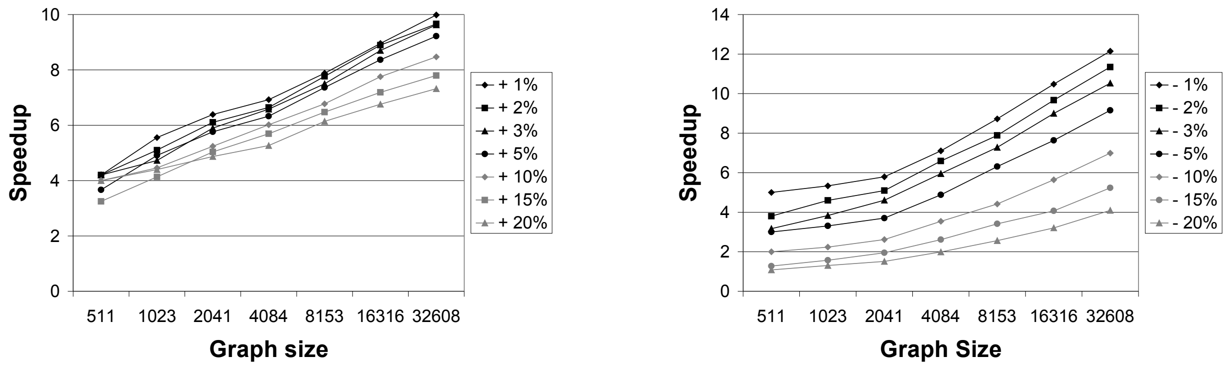


Figure 2.4: Speedup relative to BK for randomly generated perturbations on sparse graphs as graph size increases. The left graph shows the speedup for edge addition perturbations, and the right graph shows the speedup for edge removal perturbations. The number of edges in all graphs is approximately 8 times the number of vertices.

average degree of 16) where the number of vertices ranged from 512 to 32,576 in powers of two, and another series with a fixed 8,192 vertices where the number of edges ranged from 3 to 15 times as many edges as vertices. All of the graphs were produced using the default R-MAT parameters of $a = 0.45, b = c = 0.15$, and $d = 0.25$.

We preprocessed the resulting graphs to remove isolated vertices and loops and generated random perturbations from 20% of the total number of edges removed to 20% of the number of edges being added, where each edge or non-edge has an equal probability of being perturbed. (This methodology affects more cliques when removing edges than when adding edges.) We used BK and PGCE to enumerate the cliques of the perturbed graphs of size at least three and compared the median runtimes of three runs on each of the two algorithms. The results appear in Figures 2.4 and 2.5. All runs were performed on a 2.33 GHz Intel Xeon server with 16 GB of RAM running RedHat Enterprise Linux 5.

As can be seen in Figure 2.4, PGCE achieves speedups over BK of 1.08–12.1 when adding edges to graphs of increasing size and speedups of 3.25–9.98 when removing edges. Figure 2.5 shows speedups of 1.93–20.4 for adding edges and 0.78–23.1 for removing edges from graphs where the number of edges is varied. PGCE performs better relative to BK on larger and more dense graphs, which cause BK to enumerate more cliques, and with smaller perturbations, which cause PGCE to enumerate fewer cliques. Also, in both cases, the performance of the PGCE algorithm was much less sensitive to changes in graph size and density for perturbations where edges were added. A more detailed cross-section of the experiments is presented in Tables 2.1

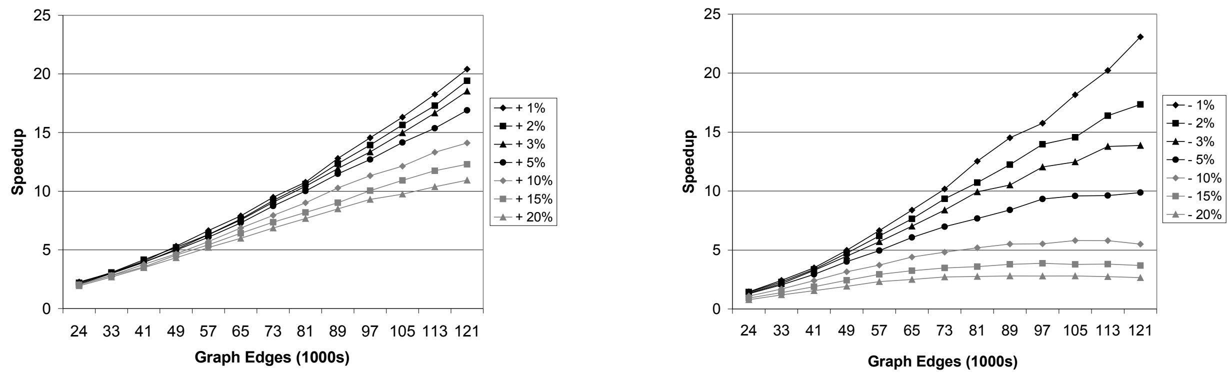


Figure 2.5: Speedup relative to BK for randomly generated perturbations on sparse graphs as graph density increases. The left graph shows the speedup for edge addition perturbations, and the right graph shows the speedup for edge removal perturbations. The number of vertices in all graphs is approximately 8,192.

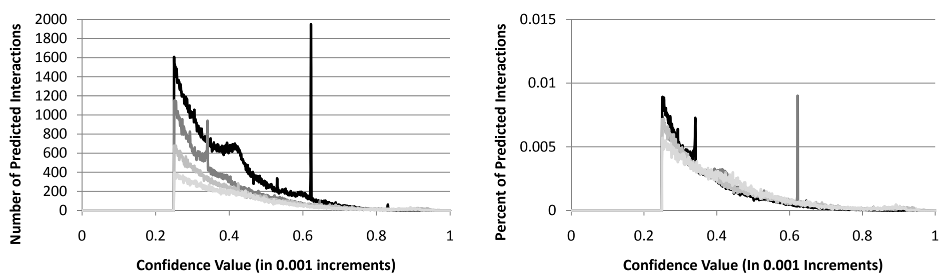


Figure 2.6: Number and percentage of predicted protein interactions having a given confidence value.

and 2.2.

For the second test, we evaluated the PGCE algorithm on perturbed graphs generated by using threshold manipulation on predicted protein interaction networks. The networks, made available through the Graemlin project [26] at Stanford[1], use experimental data to predict protein interactions in the following microbial species: *E. coli*, *M. tuberculosis*, *S. typhimurium*, and *Synechocystis*. Each of these predicted protein interactions is assigned a confidence value between 0 and 1 with 3 significant digits. To create the networks, we first created a weighted graph whose edges connected proteins with a predicted interaction and whose edge weights were the confidence value associated with that prediction. Then we applied a threshold by only

[1]Download at ftp://ftp-networks.stanford.edu/pub/graemlin_nets

Table 2.1: Runtime results for six perturbations on synthetic graphs of increasing size

Number of Vertices				
Baseline	4048	8153	16316	32608
Number of Edges				
Baseline	32199	64874	130251	251120
Number of Cliques of Size at Least Three				
Baseline	4408	5838	7855	9735
$\lvert\mathcal{C}_+\rvert$ / $\lvert\mathcal{C}_-\rvert$				
+ **5% Edges**	170 / 4	141 / 2	168 / 0	367 / 0
+ **10% Edges**	321 / 13	339 / 3	347 / 4	689 / 2
+ **20% Edges**	705 / 10	747 / 14	807 / 3	1534 / 2
− **5% Edges**	731 / 672	983 / 844	1433 / 1161	2027 / 1460
− **10% Edges**	1392 / 1274	1917 / 1647	2656 / 2128	3459 / 2530
− **20% Edges**	2143 / 2172	3201 / 2892	4677 / 3892	6027 / 4689
Median BK Runtime / Median PGCE Runtime (ms)				
+ **5% Edges**	291 / 46	692 / 94	1715 / 205	4195 /455
+ **10% Edges**	313 / 52	745 / 110	1846 / 238	4583 / 541
+ **20% Edges**	358 / 68	803 / 124	2116 / 313	5467 / 747
− **5% Edges**	239 / 49	568 / 90	1405 / 184	3388 / 370
− **10% Edges**	209 / 59	490 / 111	1230 / 218	2970 / 425
− **20% Edges**	157 / 79	369 / 144	920 / 287	2231 / 545
Number of Cliques per Millisecond (BK / PGCE)				
+ **5% Edges**	15.7 / 3.78	8.64 / 1.52	4.68 / 0.820	2.41 / 0.807
+ **10% Edges**	15.1 / 6.42	8.29 / 3.11	4.44 / 1.47	2.27 / 1.28
+ **20% Edges**	14.3 / 10.5	7.59 / 5.40	4.09 / 2.59	2.06 / 2.06
− **5% Edges**	16.0 / 28.6	8.84 / 20.3	4.79 / 14.1	2.46 / 9.42
− **10% Edges**	15.4 / 45.2	8.71 / 32.1	4.70 / 21.9	2.44 / 14.1
− **20% Edges**	15.2 / 54.6	8.29 / 42.3	4.42 / 29.9	2.30 / 19.7

including those edges whose edge weight was greater than the threshold value. In order to test the case of perturbing the graph through edge removal we increased the threshold value and for edge addition we lowered the threshold.

The experimental results for this second test were generated by adding or removing only a small percentage of the edges of the graph. This corresponds to the case when the threshold value is being fine tuned amongst a small range of possible values. As Figure 2.6 shows, there are a large number of predicted interactions with a small confidence value, but for high confidence values the number of predicted interactions is much smaller. This is a phenomenon we also observed in other types of data such as the functional association data stored in the STRING database [40]. In this case, it is likely that a large range of threshold values can be discounted based on the distribution, and the fine tuning of the threshold value will result in small graph

Table 2.2: Runtime results for 6 perturbations on synthetic graphs of increasing edge counts

Number of Vertices				
Baseline	8065	8132	8168	8179
Number of Edges				
Baseline	40697	56836	72870	88816
Number of Cliques of Size at Least Three				
Baseline	1566	4034	8104	13928
$\lvert\mathcal{C}_+\rvert$ / $\lvert\mathcal{C}_-\rvert$				
+ **5% Edges**	30 / 0	108 / 4	248 / 4	399 / 8
+ **10% Edges**	30 / 0	241 / 0	517 / 8	874 / 14
+ **20% Edges**	199 / 0	747 / 14	1102 / 12	1915 / 45
− **5% Edges**	345 / 220	759 / 626	2027 / 1460	2049 / 2071
− **10% Edges**	544 / 376	1401 / 1124	3459 / 2530	3704 / 3821
− **20% Edges**	1087 / 778	2311 / 1948	6027 / 4689	5277 / 5361
Median BK Runtime / Median PGCE Runtime (ms)				
+ **5% Edges**	221 / 56	498 / 82	935 / 107	1561 / 136
+ **10% Edges**	237 / 63	745 / 110	1009 / 127	1686 / 164
+ **20% Edges**	268 / 77	803 / 124	1167 / 170	1953 / 230
− **5% Edges**	182 / 62	406 / 82	761 / 109	1268 / 151
− **10% Edges**	161 / 67	357 / 96	660 / 137	1103 / 200
− **20% Edges**	124 / 80	274 / 118	495 / 182	818 / 292
Number of Cliques per Millisecond (BK / PGCE)				
+ **5% Edges**	7.23 / 0.536	8.31 / 1.37	8.93 / 2.36	9.17 / 2.99
+ **10% Edges**	7.00 / 1.46	7.96 / 2.56	8.54 / 4.13	8.77 / 5.41
+ **20% Edges**	6.59 / 2.10	7.37 / 4.54	7.88 / 6.55	8.09 / 8.52
− **5% Edges**	8.00 / 12.4	8.45 / 16.9	9.26 / 23.5	9.55 / 27.3
− **10% Edges**	8.01 / 19.0	8.31 / 26.3	9.04 / 34.9	9.50 / 37.5
− **20% Edges**	7.12 / 29.4	7.83 / 36.1	8.85 / 43.8	9.06 / 46.6

perturbations.

To compare the PGCE and BK algorithms we constructed a perturbed graph by either increasing or decreasing the threshold value by 0.001. We compared the runtimes of the two algorithms by comparing the time the BK algorithm took to enumerate the perturbed graph with the time the PGCE algorithm took to enumerate the $\mathcal{C}_-$ and $\mathcal{C}_+$ sets. We did not include the preprocessing and postprocessing steps in the runtime of the two algorithms because their portion of the overall runtime decreases significantly as the size of the graphs increases. All runs were again performed on a 2.33 GHz Intel Xeon server with 16 GB of RAM running RedHat Enterprise Linux 5.

The speedup results for the 0.001 perturbations are shown in Tables 2.3 and 2.4 and Figures 2.7 and 2.8. Table 2.3 shows the results of a set of runs on the *E. coli* network as the threshold

Table 2.3: Runtime Results for Edge Removal on the *E. coli* network

Threshold Value								
Baseline	0.500	0.501	0.502	0.503	0.504	0.505	0.506	0.507
Perturbed	0.501	0.502	0.503	0.504	0.505	0.506	0.507	0.508
Number of Edges								
Baseline	35133	34867	34635	34345	34080	33825	33586	33316
Perturbed	34867	34635	34345	34080	33825	33586	33316	33076
Removed	266	232	290	265	255	239	270	240
Number of Cliques								
Baseline	27831	27603	26923	26299	25459	24496	23857	23407
Perturbed	27603	26923	26299	25459	24496	23857	23407	22720
$\mathcal{C}_+$	1969	1449	2101	1514	1969	1583	1348	1308
$\mathcal{C}_-$	2197	2129	2725	2354	2932	2222	1798	1995
Algorithm Runtime (ms)								
BK	1184	1174	1152	1117	1079	1063	1014	1023
PGCE	260	247	332	294	353	274	203	223
Speedup	4.55	4.75	3.47	3.80	3.06	3.88	5.00	4.59
Number of Cliques per Millisecond								
BK	23.31	22.93	22.83	22.79	22.70	22.44	23.08	22.21
PGCE	16.02	14.49	14.54	13.16	13.88	13.89	15.50	14.81

value is increased and edges are removed from the graph. In this case, the PGCE algorithm outperforms the BK algorithm for the experimental perturbations because of the smaller sets which it has to enumerate ($\mathcal{C}_-$, $\mathcal{C}_+$). Table 2.4 shows the results when the threshold value is increased and edges are removed from the graph. For the addition case, it appears that the performance of the PGCE algorithm is due not only to the smaller enumerated set, but also to the increase in the speed in which each individual cliques is enumerated. Figures 2.7 and 2.8 show the speedup of the PGCE algorithm with respect to the BK algorithm for four different networks. These networks are, from left to right, those of *E. coli*, *M. tuberculosis*, *S. typhimurium*, and *Synechocystis*.

2.9 Conclusion

In summary, we have applied our concept of Graph Perturbation Theory to the problem of enumerating the maximal cliques of a perturbed graph by presenting necessary and sufficient conditions for enumerating these cliques in six different types of perturbation. Using these results, we described algorithms for calculating the maximal cliques induced and destroyed by two types of perturbations, and presented empirical results as to the efficacy of these algorithms over re-enumeration. Our experimental results on synthetic data showed that the proposed

Table 2.4: Runtime Results for Edge Addition on the *E. coli* network

Threshold Value								
Baseline	0.501	0.502	0.503	0.504	0.505	0.506	0.507	0.508
Perturbed	0.500	0.501	0.502	0.503	0.504	0.505	0.506	0.507
Algorithm Runtime (ms)								
BK	1232	1184	1174	1152	1117	1079	1063	1014
PGCE	21	16	21	21	22	21	13	13
Speedup	58.67	74	55.90	54.86	50.77	51.38	81.77	78
Number of Cliques per Millisecond								
BK	22.59	23.31	22.93	22.83	22.79	22.70	22.44	23.08
PGCE	198.38	223.63	229.81	184.19	222.77	181.19	242.00	254.08

algorithm achieves good performance relative to existing algorithm on small perturbations of graphs with many cliques. Our results for perturbations resulting from threshold manipulations on predicted protein interaction data showed speedups of between 3–6 in the case of edge removal and 20–120 in the case of edge addition. Our algorithms, or some refinement thereof, could prove a powerful tool for evaluating various “what-if” scenarios in graphs based on noisy or uncertain data.

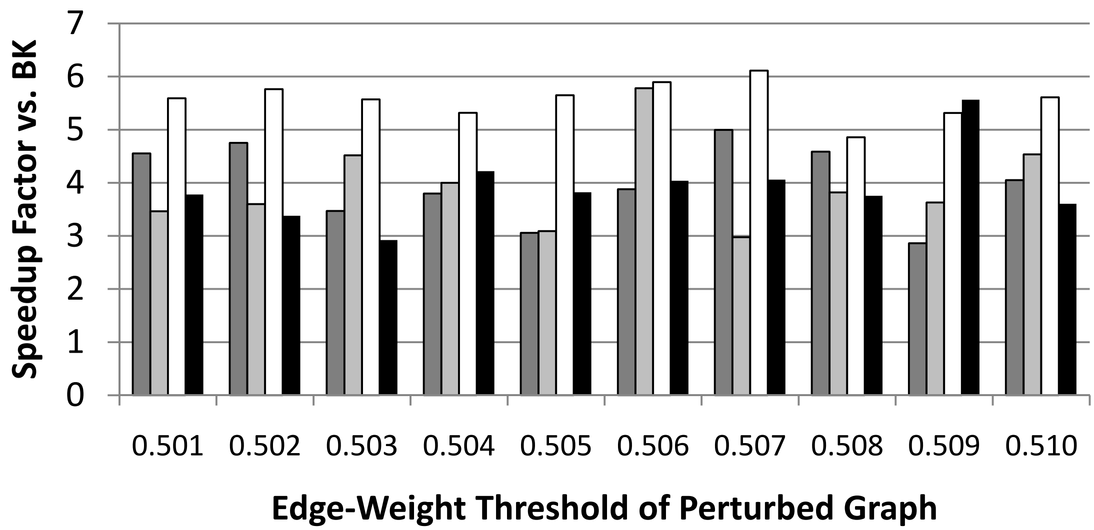


Figure 2.7: Speedup factor of the PGCE algorithm with respect to the BK algorithm for edge removal. The networks are, from left to right, those of *E. coli*, *M. tuberculosis*, *S. typhimurium*, and *Synechocystis*.

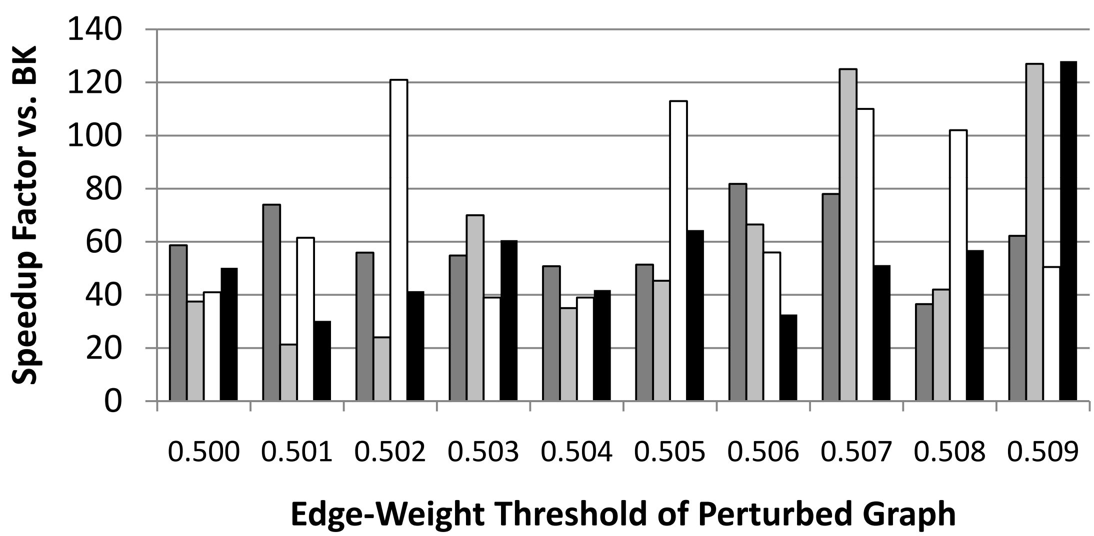


Figure 2.8: Speedup factor of the PGCE algorithm with respect to the BK algorithm for edge addition. The networks are, from left to right, those of *E. coli*, *M. tuberculosis*, *S. typhimurium*, and *Synechocystis*.

Chapter 3

Dense, Enriched Subgraph Enumeration

3.1 Introduction

In this component, we attempt to outline theoretical conditions on dense subgraphs of a network that are "enriched" with respect to some target set of vertices. An algorithm based on this theory would be able to answer "fuzzy queries" on graph data, identifying dense, possibly overlapping subgraphs in which the "query set" of vertices is overrepresented. By finding these dense, enriched "fuzzy clusters," or enriched quasi-cliques, we hope to achieve superior precision and coverage over conventional hard clustering techniques, which heuristically partition graphs into non-overlapping subgraphs. Further, by limiting the focus to discovering those "quasi-cliques" in which the query labels are overrepresented, the search space for identifying these quasi-cliques may be limited, which has the potential to improve execution time significantly over full quasi-clique enumeration.

As a motivating example of this work, in the context of a social network, people who belong to a dense cluster with a high concentration of "suspicious persons" are reasonably likely to be "suspicious" themselves—an intelligence analyst could then use this information to direct some of their information-gathering efforts. Alternatively, a biologist might wish to search a protein interaction network for dense clusters with large numbers of proteins associated with cellular respiration. Those proteins with unknown functions in the resulting subgraphs would likely have a function related to cellular respiration and may be appropriate for experiments and further inquiry.

Clustering, the unsupervised formation of data points into related groups, has long been a fruitful area of research (see survey in [39]). Traditional, "hard" clustering results in a partitioning of the data into clusters, but in many domains, including marketing, Web documents, and bioinformatics, the data elements under consideration may belong to multiple categories. For instance, a consumer may belong to multiple groups that affect his or her purchasing patterns, as with someone who is male, over the age of 55, and who enjoys comedies. In such cases, an approach that allows for overlapping clusters is more appropriate than one that partitions the data.

Previous approaches to finding overlapping clusters have included fuzzy logic-based approaches [45] (also, see [38]), probabilistic approaches [33, 68], stochastic approaches [25], and consensus clustering [21]. The discovery of dense non-clique subgraphs has recently been explored by a number of other researchers [3] [11] [14, 27, 57, 90], and a number of different formulations for what it means for a subgraph to be "dense" have emerged. In this work, we use the following definition for a "dense" subgraph:

Definition 3.1.1. *Given a labeled graph G and a real value $\gamma \in [0.5, 1]$, a subgraph S of G is a **γ-dense quasi-clique** if and only if every vertex of S is adjacent to at least $\gamma(|S| - 1)$ of the other vertices of S. If $\gamma(|S| - 1)$ is not a natural number, every vertex would need to be adjacent to $\lceil \gamma(|S| - 1) \rceil$ of the other vertices of S.*

The advantage of this definition is twofold. First, it corresponds nicely with the typical use of the term "density" in that it forces a certain fraction of the possible edges in the subgraph to exist. The second advantage is that by framing the definition as a condition that each vertex must satisfy, we force the resulting subgraphs to be "uniformly" dense. As an illustration, a graph consisting of an isolated vertex and a subgraph in which every pair of vertices is connected may contain a high overall percentage of the possible edges, but it is unlikely anyone would consider the isolated vertex to be related to the others in any significant sense.

Our definition for what it means for a dense subgraph to be enriched appears in Definition 3.1.2. The primary advantage of this definition is its similarity to the edge density criterion.

Definition 3.1.2. *Given a labeled graph G, a "query" set of vertices Q, a real value $\gamma \in (0.5, 1]$, and a real value $\mu \in (0, 1]$, a γ-dense quasi-clique S is **μ-enriched** with respect to Q if and only if at least $\mu|S|$ vertices of S are contained in Q.*

In the interest of brevity, μ-enriched γ-quasi-cliques will hereafter be refered to as μ, γ-quasi-cliques, and the "query" set of vertices will be denoted as Q. The rest of the chapter is organized as follows: in Section 3.2, we will cover several theoretical conditions that μ, γ-quasi-cliques must satisfy; in Section 3.3, we apply the results of these theorems to outline an algorithm for finding μ, γ-quasi-cliques; in Section 3.5, we present the results from an application of our algorithm to the problem of identifying genes related to biological hydrogen production; and in Section 3.6, we conclude the chapter.

3.2 Theory

As several previous dense subgraph enumeration algorithms have successfully employed an agglomerative (bottom-up) approach, we focus our efforts in this section on establishing a number of theoretical conditions for limiting the set of vertices that could be added to a subgraph to form a maximal μ, γ-quasi-clique. Our first result, originally presented in [57] as a special case of Theorem 1, places a limit on the *diameter*, or the maximum length of the shortest path between two vertices, of a μ, γ-quasi-clique. As this result has already been presented, the proof is omitted here.

Theorem 3.2.1. *Let S be a γ-dense quasi-clique where $\gamma \geq 0.5$. The diameter of S is at most 2.*

Corollary 3.2.1. *Let S be a subgraph of G. Any γ-dense quasi-clique that is a supergraph of S will consist of vertices at most distance 2 from every vertex of S. We denote this set as $N^2(S)$.*

For the rest of the results presented in this section, we will adopt the convention that S represents the current subgraph and C represents the set of "candidate" vertices that could be added to S to form a μ, γ-quasi-clique. By the result of Corollary 3.2.1, we can see that the set $N^2(S) \setminus S$ contains every vertex that could be added to S to form a μ, γ-quasi-clique, and as such, $C = N^2(S) \setminus S$ represents a good starting point for the candidate set.

Several of the following theorems are designed to reduce the size of C, eliminating unfruitful branches of the search tree as soon as possible. Theorems 3.2.2 and 3.2.3 use the γ density requirement to place bounds on how "connected" a vertex must be to be contained in a μ, γ-quasi-clique. Intuitively, the greatest possible percentage of vertices in S adjacent to a vertex v would occur if we moved to S the vertices adjacent to v in C, and if the resulting subgraph doesn't satisfy the γ density requirement on v, no supergraph of S will do so. These results are drawn from Lemmas 5.4–5.6 in [90], but a proof is presented here for clarity.

Theorem 3.2.2. *Let S be a subgraph of G, and let C be the set of all possible vertices that may be contained in some supergraph of S that is a γ-dense quasi-clique. Let v be any vertex of S, and let s_a and c_a be the number of vertices of S and C, respectively, that are adjacent to v. If $s_a + c_a < \gamma(|S| - 1 + c_a)$, no supergraph of S can be a γ-dense quasi-clique.*

Proof. Suppose the negation: $s_a + c_a < \gamma(|S| - 1 + c_a)$ and there exists a supergraph H of S that is a γ-dense quasi-clique.

Let h_a be the number of vertices of $H \setminus S$ adjacent to v. By the definition of the set C, $h_a \leq c_a$. Subtracting $\gamma(c_a - h_a)$ from both sides of the previous inequality, we see that $s_a + c_a - \gamma(c_a - h_a) < \gamma(|S| - 1 + h_a)$. As $\gamma > 0$ and $c_a - h_a \geq 0$, $c_a - h_a \geq \gamma(c_a - h_a)$, so

$$
\begin{aligned}
s_a + h_a &= s_a + c_a - (c_a - h_a) \\
&\leq s_a + c_a - \gamma(c_a - h_a) \\
&< \gamma(|S| - 1 + h_a)
\end{aligned}
$$

Since H must have at least $|S| + h_a$ vertices, $\gamma(|S| - 1 + h_a) \leq \gamma(|H| - 1)$, so $s_a + h_a < \gamma(|H|-1)$. However, H has only s_a+h_a vertices adjacent to v, implying that H is not a γ-dense quasi-clique. This is a contradiction; therefore, the claim must be true. □

Theorem 3.2.3. *Let S be a subgraph of G, and let v be a vertex in C. Let s_a be the number of vertices of S adjacent to v, and let c_a be the number of vertices of C adjacent to v. If $s_a + c_a \leq \gamma(|S| + c_a)$, neither $S \cup \{v\}$ nor any supergraph of it can be a γ-dense quasi-clique.*

Proof. The proof is similar to the proof for Theorem 3.2.2. □

Much as the previous two results establish bounds based on the density requirement for μ, γ-quasi-cliques, Theorem 3.2.4 allows us to prune the set of "candidate" vertices based on the enrichment requirement. The intuition behind the theorem is that S would not satisfy the μ enrichment criterion after moving all the vertices in C that are also in Q to S, no supergraph of S will be able to satisfy the μ enrichment criterion.

Theorem 3.2.4. *Let c_q be the number of vertices in $C \cap Q$. If there are fewer than $\mu|S| - (1-\mu)c_q$ vertices in $S \cap Q$, then neither S nor any supergraph will be μ-enriched.*

Proof. Suppose not, and let H be a supergraph of S that is a μ-enriched. Let h_q be the number of vertices of $H \setminus S$ that are in Q. Since C contains c_q vertices in Q, $h_q \leq c_q$. As S contains less than $\mu|S| - (1-\mu)c_q$ vertices in Q, H contains less than $\mu|S| - (1-\mu)c_q + h_q \leq \mu|S| - (1-\mu)h_q + h_q = \mu(|S| + h_q)$ vertices in Q. However, as $|H| \geq |S| + h_q$, this implies that H contains less than $\mu|H|$ vertices in Q, contradicting our assumption that H is μ-enriched. □

Finally, Theorems 3.2.5–3.2.8 establish restrictions on building μ, γ-quasi-cliques through a combination of the density and enrichment requirements. The intuitions here are that after adding all of the candidate vertices that are in Q and are adjacent to a vertex, we are limited in the number of other (non-query) adjacent vertices that we can add by the enrichment condition, and the density condition limits the number of other (non-adjacent) query vertices we can add (Theorems 3.2.5 and 3.2.6).

Lemma 3.2.1. *Let H be a μ, γ-quasi-clique, let S be a subgraph of H, and let v be a vertex of S. Let s_a be the number of vertices of S adjacent to v, $s_{\overline{q}}$ be the number of vertices in S that* *are not in Q, c_{aq} be the number of vertices in $H \setminus S$ that are in Q and adjacent to v, and $c_{a\overline{q}}$ be the number of vertices in $H \setminus S$ that are adjacent to v but not in Q.*

$$(1-\mu)(s_a + c_{aq} + c_{a\overline{q}} + \gamma) - \gamma(s_{\overline{q}} + c_{a\overline{q}}) \geq 0$$

Proof. H must contain at least $\mu|H|$ vertices in Q, so it contains at most $|H| - \mu|H| = (1-\mu)|H|$ vertices not in Q. By our definitions, $s_{\overline{q}}$ and $c_{a\overline{q}}$ both represent vertices in H that are not in Q, so this fact implies that

$$s_{\overline{q}} + c_{a\overline{q}} \leq (1-\mu)|H|. \tag{3.1}$$

Similarly, H must contain at least $\gamma(|H|-1)$ vertices adjacent to v, so

$$\gamma(|H|-1) \leq s_a + c_{aq} + c_{a\overline{q}}, \text{ or} \tag{3.2}$$

$$\gamma|H| \leq s_a + c_{aq} + c_{a\overline{q}} + \gamma. \tag{3.3}$$

Multiplying equation 3.1 by γ and equation 3.3 by $1-\mu$ (both of which must be positive), we see that

$$\gamma(s_{\overline{q}} + c_{a\overline{q}}) \leq \gamma(1-\mu)|H| \text{ and} \tag{3.4}$$

$$\gamma(1-\mu)|H| \leq (1-\mu)(s_a + c_{aq} + c_{a\overline{q}} + \gamma). \tag{3.5}$$

Thus, $\gamma(s_{\overline{q}} + c_{a\overline{q}}) \leq (1-\mu)(s_a + c_{aq} + c_{a\overline{q}} + \gamma)$, proving the claim. □

Theorem 3.2.5. *Let v be a vertex in S, and let s_a, $s_{\overline{q}}$, c_{aq}, and $c_{a\overline{q}}$ be as in Lemma 3.2.1, except with c_{aq} and $c_{a\overline{q}}$ being vertices in C rather than $H \setminus S$. If $\gamma < 1-\mu$ and $(1-\mu)(s_a + c_{aq} + c_{a\overline{q}} + \gamma) - \gamma(s_{\overline{q}} + c_{a\overline{q}}) < 0$, then neither S nor any supergraph of S will be a μ,γ-quasi-clique.*

Proof. Consider an arbitrary μ,γ-quasi-clique H such that S is a subgraph of H. If we let h_{aq} and $h_{a\overline{q}}$ represent the number of vertices in $H \setminus S$ that are adjacent to v and are in Q and not in Q, respectively, then h_{aq} and $h_{a\overline{q}}$ must satisfy the inequality $(1-\mu)(s_a + h_{aq} + h_{a\overline{q}} + \gamma) - \gamma(s_{\overline{q}} + h_{a\overline{q}}) \geq 0$ by Lemma 3.2.1. As H must be a subgraph of $S \cup C$, h_{aq} and $h_{a\overline{q}}$ must satisfy $0 \leq h_{aq} \leq c_{aq}$ and $0 \leq h_{a\overline{q}} \leq c_{a\overline{q}}$. As $\gamma < 1-\mu$, $(1-\mu)(s_a + h_{aq} + h_{a\overline{q}} + \gamma) - \gamma(s_{\overline{q}} + h_{a\overline{q}})$ is maximized at $h_{aq} = c_{aq}$ and $h_{a\overline{q}} = c_{a\overline{q}}$, so if $(1-\mu)(s_a + c_{aq} + c_{a\overline{q}} + \gamma) - \gamma(s_{\overline{q}} + c_{a\overline{q}}) < 0$, no subgraph of $S \cup C$ containing v may be a μ,γ-quasi-clique. □

Theorem 3.2.6. *Let v be a vertex in S, and let s_a, $s_{\overline{q}}$, and c_{aq} be as in Lemma 3.2.1, except with c_{aq} being vertices in C rather than $H \setminus S$. If $\gamma \geq 1-\mu$ and $(1-\mu)(s_a + c_{aq} + \gamma) - \gamma s_{\overline{q}} < 0$, then neither S nor any supergraph of S will be a μ,γ-quasi-clique.*

Proof. The proof for Theorem 3.2.6 is similar to the proof for Theorem 3.2.5, with the exception that $(1-\mu)(s_a + h_{aq} + h_{a\overline{q}} + \gamma) - \gamma(s_{\overline{q}} + h_{a\overline{q}})$ is maximized at $h_{aq} = c_{aq}$ and $h_{a\overline{q}} = 0$. □

Lemma 3.2.2. *Let H be a μ, γ-quasi-clique, let S be a subgraph of H, and let v be a vertex of S. Let s_q be the number of vertices of S in Q, $s_{\overline{a}}$ be the number of vertices in $S \setminus \{v\}$ not adjacent to v, c_{aq} be the number of vertices in $H \setminus S$ that are adjacent to v and in Q, and $c_{\overline{a}q}$ be the number of vertices in $H \setminus S$ that are in Q but not adjacent to v.*

$$(1-\gamma)(s_q + c_{aq} + c_{\overline{a}q}) - \mu(s_{\overline{a}} + c_{\overline{a}q} + 1 - \gamma) \geq 0$$

Proof. This lemma follows from the inequalities $(1-\gamma)(|H|-1) \leq s_{\overline{a}} + c_{\overline{a}q}$ and $\mu|H| \leq s_q + c_{aq} + c_{\overline{a}q}$, similar to Lemma 3.2.1. □

Theorem 3.2.7. *If $\gamma < 1-\mu$ and $(1-\gamma)(s_q + c_{aq} + c_{\overline{a}q}) - \mu(s_{\overline{a}} + c_{\overline{a}q} + 1 - \gamma) < 0$, then neither S nor any supergraph of S will be a μ, γ-quasi-clique.*

Theorem 3.2.8. *If $\gamma \geq 1-\mu$ and $(1-\gamma)(s_q + c_{aq}) - \mu(s_{\overline{a}} + 1 - \gamma) < 0$, then neither S nor any supergraph of S will be a μ, γ-quasi-clique.*

Proof. Similar to Theorems 3.2.5 and 3.2.6, Theorems 3.2.7 and 3.2.8 follow from maximizing the expression $(1-\gamma)(s_q + h_{aq} + h_{\overline{a}q}) - \mu(s_{\overline{a}} + c_{\overline{a}q} + 1 - \gamma)$ at $h_{aq} = c_{aq}$ and $h_{\overline{a}q} = c_{\overline{a}q}$ or 0, respectively. (Note that $\mu < 1-\gamma$ iff $\gamma < 1-\mu$.) □

3.3 Algorithm

In this section, we will apply the results established in Section 3.2 to design an algorithm to efficiently enumerate μ, γ-quasi-cliques. The basic premise of the algorithm is that we will build the μ, γ-quasi-cliques one vertex at a time, starting with a single query vertex v_0 and backtracking as we find maximal μ, γ-quasi-cliques or subgraphs that cannot be contained in a μ, γ-quasi-clique. For this section, we continue the convention that S represents the current subgraph under consideration, and C represents the set of vertices that could extend S to produce a μ, γ-quasi-clique. A pseudocode outline of the algorithm appears in Algorithms 6 and 7; details on how the theoretical results derived in Section 3.2 have been applied, how maximality of a μ, γ-quasi-clique is checked, and which vertex of C is chosen to add to S follow in the rest of the section.

From Corollary 3.2.1, we can see that $N^2(S)$ can serve as an appropriate starting point for our set C. However, rather than recalculate this intersection of sets every time a vertex is added to the set S, we first define C as the set of all vertices within distance 2 of the inital vertex, $N^2(v_0)$, and intersect C with $N^2(v)$ for each vertex v we add to S. As these $N^2(v_0)$

Algorithm 6: Pseudocode outline of the μ, γ-quasi-clique algorithm

foreach $v_0 \in Q$ **do**
    $S \leftarrow \{v_0\}$
    $C \leftarrow N^2(v_0)$
    Calculate e
    Calculate d_v for all $v \in S \cup C$
    Calculate g_v for all $v \in S \cup C$
    Calculate m_v for all $v \in S \cup C$
    Remove all unpromising vertices of C
    if $S \cup C$ is maximal **then**
        call `Enumerate`
end

sets can be precomputed and stored in a matrix, this should make for a much more efficient way to apply Corollary 3.2.1.

By Theorem 3.2.3, we know that for any vertex $v \in C$, if s_a represents the number of vertices of S adjacent to v, c_a represents the number of vertices of C adjacent to v, and $s_a + c_a \leq \gamma(|S| + c_a)$, then neither $S \cup \{v\}$ nor any supergraph can be a μ, γ-quasi-clique. Rather than recomputing this inequality every time we add or remove a vertex from S, we calculate and maintain the value of $s_a + c_a - \gamma(|S| + c_a)$ as d_v—when this value becomes zero or negative, v may be removed from C. Thus, at the outset of the algorithm, we calculate the values of $s_a + (1-\gamma)c_a$ for each $v \in C$. When a vertex $u \in C$ is added to S, d_v will remain unchanged if u is adjacent to v, but d_v will decrease by γ if u is not adjacent to v. Similarly, d_v increases by γ when a vertex nonadjacent to v is removed from S. When a vertex $u \in C$ is removed from C, d_v will decrease by $1-\gamma$ if u is adjacent to v, but it will be unaffected if u is nonadjacent to v. Similarly, d_v will increase by $1-\gamma$ when u is returned to C.

By Theorem 3.2.2, we know that for vertices $v \in S$, $s_a + c_a \leq \gamma(|S| - 1 + c_a)$, where s_a and c_a are defined as before. As such, we maintain a d_v value for each $v \in S$ just as we maintain d_v values for vertices in C (see Table 3.1). Finally, to calculate d_v when v is moved from C to S, we use the same value of d_v as when v was in the set C. (The value of d_v is unimportant if v is removed from C.) Table 3.1 summarizes the values by which d_v is updated.

When the value of d_u becomes negative for a vertex $u \in C$, we can remove u from C by the result of Theorem 3.2.3. Additionally, when d_v decreases below γ for a vertex $v \in S$, we can remove all vertices of C that are nonadjacent to v, as adding such vertices to S would violate Theorem 3.2.2.

In a similar fashion, we calculate the initial values for e and each g_v and m_v value and update these values as the algorithm progresses. We can then remove vertices from C whose

Algorithm 7: Pseudocode for `Enumerate` function

$T \leftarrow C$
while some vertices of C are marked **do**
> Remove all marked vertices from C
> **if** S violates one of the theoretical constraints **then**
> > Restore all vertices of $T \setminus C$ to C
> > **return**;
>
> **end**
> Update e and all d_v, g_v, and m_v values as appropriate
> **if** $S \cup C$ is nonmaximal **then**
> > Backtrack until some vertex of C is restored

end
while $C \neq \emptyset$ **do**
> Choose v in C according to some heuristic and move v to S
> Update e and all d_v, g_v, and m_v values as appropriate
> **if** $g_v < 0$ or $m_v < 0$ for some $v \in S$ **then**
> > Restore vertices of $T \setminus C$ to C
> > Update e, d_v, g_v, and m_v values appropriately
> > **return**
>
> **end**
> Mark all vertices of C to be removed
> **if** S does not violate any of the theoretical constraints **then**
> > **call** `Enumerate`
>
> Remove v from S
> Update e and all d_v, g_v, and m_v values as appropriate
> **if** S violates one of the theoretical constraints **then**
> > Restore vertices of $T \setminus C$ to C
> > Update e and all d_v, g_v, and m_v values
> > **return**
>
> **end**
> Iteratively remove unpromising vertices of C
> Update e and all d_v, g_v, and m_v values as appropriate
> **if** $S \cup C$ is nonmaximal **then**
> > Backtrack until some vertex of C is restored

end
if no recursive call of `Enumerate`() found a μ, γ-quasi-clique **then**
> Output S
> Update the maximality index for each vertex in S

end
Restore vertices of $T \setminus C$ to C
Update e, d_v, g_v, and m_v values appropriately
return

Table 3.1: Values by which d_v is decreased when a vertex $u \in C$ is moved to S or removed from C

	$u \in N(v)$	$u \notin \overline{N(v)} \cup \{v\}$	$u = v$
u moved to S	0	γ	0
u removed from C	$1-\gamma$	0	–

Table 3.2: Values by which e is decreased when a vertex $u \in C$ is moved to S or removed from C

	$u \in Q$	$u \notin Q$
u moved to S	0	μ
u removed from C	$1-\mu$	0

addition to C would violate Theorems 3.2.4, 3.2.5, 3.2.6, 3.2.7, or 3.2.8. Tables 3.2, 3.3, and 3.4 summarize the amount by which the e, g_v, and m_v values are updated as vertices are moved from C to S or removed from C.

In order to decide when a μ, γ-quasi-clique is maximal, we propose a maintaining a bitmap index of the μ, γ-quasi-cliques that contain each vertex. As the algorithm identifies μ, γ-quasi-cliques, it assigns numbers to them sequentially and adds these values to indices for the vertices contained in the μ, γ-quasi-cliques. Then, as we add and remove vertices from set C, we check these bitmap indices to see if there is an already-discovered μ, γ-quasi-clique that contains all vertices of $S \cup C$ by performing a bitwise `and` of the indices associated with the vertices of $S \cup C$. If there is an already-discovered μ, γ-quasi-clique that is a superset of $S \cup C$, we may safely backtrack, as no further extensions of S will be maximal.

One drawback of using a bitmap index, however, is that as more μ, γ-quasi-cliques are identified, the size of the index will increase. In an effort to avoid checking the entire index for each vertex (in the case where $S \cup C$ is maximal), we propose using a hierarchical bitmap index, in which each byte of the index is summarized by a single bit in a higher level index. As we are checking for the existence of a bit that is set in all of the indices related to the vertices of $S \cup C$, we do not need to examine any bytes that have no bits set. As such, we summarize

Table 3.3: Table summarizing values by which g_v decreases when a vertex $u \in C$ is moved to S (column "M") or removed from C (column "R")

u	$N(v), Q$		$N(v), \neg Q$		$\neg N(v), Q$		$\neg N(v), \neg Q$	
	M	R	M	R	M	R	M	R
$\gamma \geq 1-\mu$	0	$1-\mu$	$\gamma-(1-\mu)$	0	0	0	γ	0
$\gamma < 1-\mu$	0	$1-\mu$	0	$1-\mu-\gamma$	0	0	γ	0

Table 3.4: Table summarizing values by which m_v decreases when a vertex $u \in C$ is moved to S (column "M") or removed from C (column "R")

u	$N(v), Q$		$N(v), \neg Q$		$\neg N(v), Q$		$\neg N(v), \neg Q$	
	M	R	M	R	M	R	M	R
$\gamma \geq 1-\mu$	0	$1-\gamma$	0	0	$\mu-(1-\gamma)$	0	μ	0
$\gamma < 1-\mu$	0	$1-\gamma$	0	0	0	$1-\gamma-\mu$	μ	0

zero bytes in the "base level" index with a 0 and nonzero bytes with a 1. As the size of the index grows, we can add more levels, summarizing each byte in the "first level" index with a bit in a "second level" index, each byte in the "second level" index with a bit in the third, and so on. In this way, we can use higher level indices to reduce the number of bytes we need to check on the "base level" index.

3.4 Results

In this section, we present several empirical results to demonstrate the effectiveness of our algorithm at efficiently detecting dense and enriched subgraphs in large, sparse graphs. For these experiments, we ran our algorithm three times in order to detect different types of μ,γ-quasi-cliques. The three types of quasi-cliques we detect are: high density, low enrichment ("clique") subgraphs where Q contains every vertex of the graph; high enrichment, low density ("enriched") subgraphs with a small query set (every 10th vertex of $V(G)$); and moderate enrichment and density ("dense") subgraphs with a medium-sized query set (every 6th vertex of $V(G)$). These settings were chosen to test the algorithm (and various candidate vertex constraints) under a wide variety of conditions. The parameter settings for these three types of subgraphs appear in Table 3.5.

Table 3.5: Parameter settings for the various types of dense, enriched subgraphs we use to test our algorithm

Description	γ	μ	$\|Q\|$
clique	0.999	0.001	$\|V(G)\|$
enriched	0.5	0.90	$\|V(G)\|/10$
dense	0.85	0.85	$\|V(G)\|/6$

For these experiments, we used the R-MAT random graph generator [16] to generate sparse graphs of increasing size. The graphs were generated to have vertices equal to a power of two, with an average vertex degree of 14 ($|E(G)| = 7|V(G)|$). The graphs were then processed to remove isolated vertices, which do not contribute to our search for dense, enriched subgraphs. All graphs were generated using the default R-MAT parameters of $a = 0.45, b = 0.15, c = 0.15$, and $d = 0.25$. More details on the generated graphs can be found in Table 3.6.

Table 3.6: Graph size and number of maximal quasi-cliques for graphs generated using R-MAT

$\lvert V(G)\rvert$	$\lvert E(G)\rvert$	Quasi-cliques		
		clique	enriched	dense
127	889	569	23	14
255	1785	1199	64	21
510	3570	2593	104	72
1022	7154	5563	270	257
2039	14273	11831	485	432
4079	28553	24930	943	659
8132	56924	52025	1915	1774
16285	113995	106973	3991	4031

We implemented the algorithm described in Section 3.3. For our implementation, we select the candidate vertex to add to the subgraph using a trivial heuristic: the candidate that appears first in the array is chosen. We tested our algorithm on the R-MAT graphs described in Table 3.6 using all three of the parameter settings in Table 3.5, and we calculated the rate at which the μ, γ-quasi-cliques were produced. The results appear in Figure 3.1.

From Figure 3.1, we can see that the "clique" subgraphs were generated much more quickly than the "dense" or "enriched" quasi-cliques, likely due to the extremity of the density requirement for the "clique" subgraphs, which ensures that the resulting quasi-cliques are fully connected. Also notable is that the time required per quasi-clique appears to increase linearly on the log plot, implying that the time per quasi-clique increases polynomially with the size of the graph. Using a best fit curve, we see that the time per "clique" quasi-clique increases at a rate of approximately $O(n^{0.25})$, where n is the number of vertices in the graph, and the time per "dense" and "enriched" quasi-clique increases at a rate of approximately $O(n^{0.65})$. While this scaling is obviously dependent on the graphs being analyzed, this result does suggest that our algorithm would be able to efficiently calculate dense and enriched subgraphs on large, sparse graphs with a power-law degree distribution.

As a second experiment, we wished to evaluate the effectiveness of using the hierarchical bitmap index described in Section 3.3. For the purposes of this test, we implemented a second version of the algorithm that used only a flat (non-hierarchical) bitmap index, and we compared

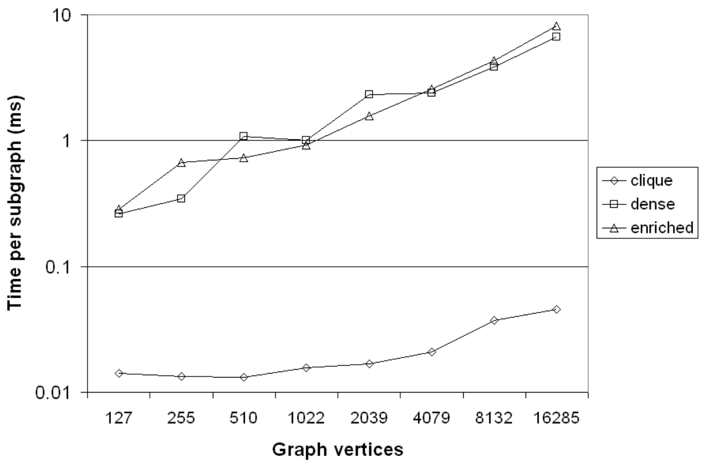


Figure 3.1: Timing results for μ, γ-quasi-clique enumeration algorithm. Time is reported in milliseconds per quasi-clique. Descriptions of the various quasi-cliques can be found in Table 3.5, and descriptions of the graphs used can be found in Table 3.6.

the time per quasi-clique for both implementations. The results appear in Figure 3.2.

From Figure 3.2, we can see that as the size of the graph increases, the hierarchical bitmap index provides a significant speedup in the rate of identifying "clique" subgraphs. When calculating "dense" and "enriched" subgraphs, the flat index offers a moderate improvement over the hierarchical index (as much as 53%), though this advantage disappears on graphs larger than 2048 vertices. These results are likely due to the fact that the graphs in question have significantly more "clique" subgraphs than "dense" or "enriched" subgraphs—as the size of the index grows, so does the potential advantage in using a hierarchical index. As such, we conclude that the hierarchical index is successful at improving the algorithmic runtime as the size of the index grows.

3.5 Biological application

In collaboration with a biologist, we were able to apply the algorithm described in this chapter to the problem of identifying clusters of proteins related to biohydrogen production. These results were published at BIOCOMP 2010 [34], but we reproduce them here for completeness.

3.5.1 Motivation and Background

Applying clustering algorithms to protein interaction networks allows us to find groups of proteins that interact with one another and likely have related functions. We can utilize data on protein-protein interactions to evaluate specific protein complexes, such as those associated with

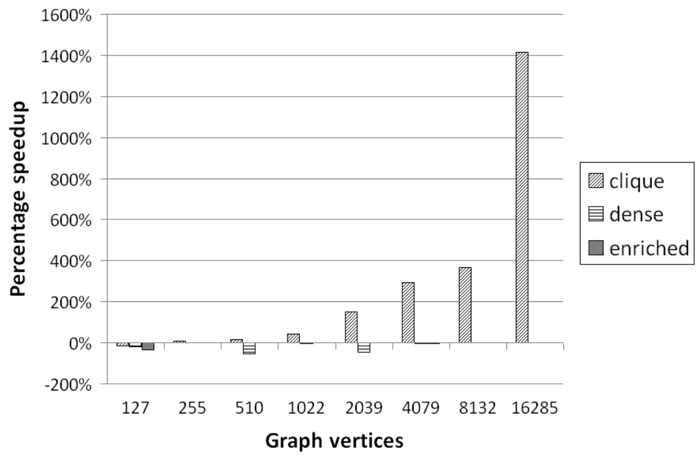


Figure 3.2: Speedup results for using hierarchical bitmap index in μ, γ-quasi-clique enumeration algorithm. Speedup is reported in percentage; i.e., a value of 100% indicates that using the hierarchical bitmap index was twice as fast as the implementation with the flat index, and a value of -100% indicates that using the flat bitmap index was twice as fast as the implementation with the hierarchical index.

hydrogenase, potentially identifying proteins essential for either the expression or regulation of key enzymes. Understanding these interactions is particularly important when engineering metabolic networks in microorganisms, and by to allowing us to incorporate knowledge about the proteins we are interested in, our algorithm can identify clusters that relate to exactly the protein complex(es) we are interested in investigating.

In hydrogen-producing organisms, the presence and expression of [NiFe]-hydrogenase and [FeFe]-hydrogenase enzymes play important roles in either hydrogen production or uptake by catalyzing the reaction $2H^+ + 2e^- \leftrightarrow H_2$ [82, 19]. Investigations of biosynthesis and localization of hydrogenases has revealed that protein functions vary from organism to organism and that the locality of the enzyme(s) within the cell is dependent on the type of hydrogenase present [82]. In addition, a number of essential proteins are required to synthesize the active sites of hydrogenase enzymes, such as [FeFe]-hydrogenase [82, 19, 43]. Although many biochemical and molecular studies are underway to characterize proteins and putative genes associated with hydrogenases, a full understanding of interactions between conserved proteins and their regulation is lacking.

Due to the importance of hydrogenase in evolution and consumption of hydrogen, there is an emphasis on applying metabolic engineering technologies to regulate expression of hydrogenases associated with H_2 production [51, 4]. In order to understand relationships between hydrogenase enzymes and the proteins required for synthesis of hydrogenases, we applied the algorithm described in Section 3.3 to investigate protein interactions for three hydrogenase entities present in the hydrogen-producing bacterium *Clostridium acetobutylicum*.

The functional relationship network was drawn from the STRING database [40], only treating interactions having scores above 0.750 as valid. We analyzed the resulting graph for dense subgraphs enriched with respect to the known hydrogenases HydA1 (CAC0028), HydA2 (CAC3230), and HydS,L (CAP0141 and CAP0142). We used parameters $\gamma = 0.75$ and $\mu = 0.001$, which correspond to searching for dense but not necessarily complete subgraphs that contain at least one of our known hydrogenase proteins.

3.5.2 Results

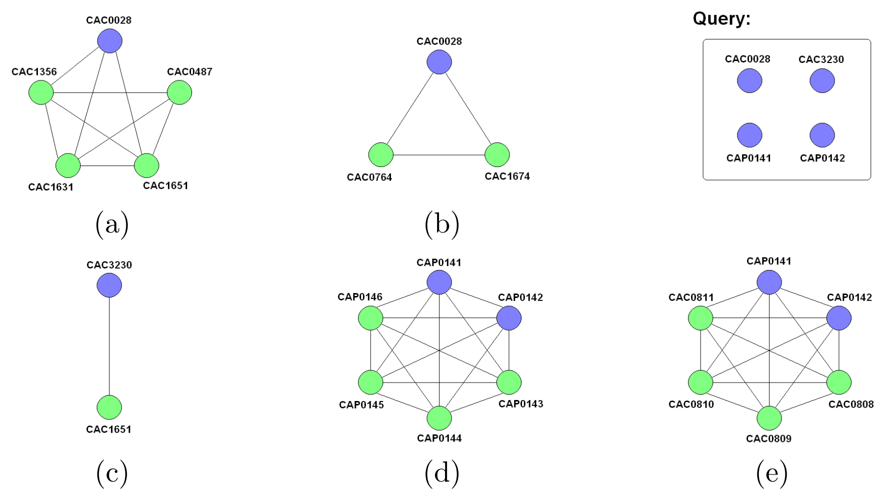


Figure 3.3: The five clusters identified by our algorithm

Our algorithm identified five different clusters, representing functionally associated groups of proteins with the query proteins—HydA1, HydA2, and HupS/L (Figure 3.3). In each cluster, the algorithm was able to identify at least one protein-protein functional association, with the highest number of associations occurring in clusters linked with the HydA1 and HupS/L hydrogenase enzymes. Lack of protein interactions by HydA2 (Figure 3.3c) may be attributed to low activity levels often associated with HydA2 compared to the other two hydrogenase enzymes [31]. Due to low protein-protein functional associations between HydA2 and accessory proteins, we will focus our analysis on proteins we identified as being associated with HydA1 (FeFe-Hydrogenase) and HupS/L (NiFe-hydrogenase).

Table 3.7: Identified proteins

Protein ID	Protein name	Function
CAC0028	HydA1	Hydrogenase I (Hydrogene dehydrogenase)
CAC3230	HydA2	Hydrogenase subunit (Ferredoxin)
CAP0141	HupS	Periplasmic hydrogenase small subunit, dehydrogenase
CAP0142	HupL	Periplasmic hydrogenase large subunit, dehydrogenase
CAC0487	–	Uncharacterized protein, ortholog of Thermotoga maritima
CAC1631	–	Biotin synthase family enzyme
CAC1356	–	Thiamine biosynthesis enzyme
CAC1651	–	Predicted GTPase with uncharacterized domain, ortholog of T. maritima
CAC1674	GltB	Small subunit of NADPH-dependent glutamate synthase
CAC0764	–	NADPH-dependent glutamate synthase beta chain
CAP0143	HupD	Hydrogenase maturation protease delta subunit, HyaD-like
CAP0144	HypQ3	Possible steroid binding protein
CAP0145	HypQ4	Putative uncharacterized protein
CAP0146	–	HypF homolog
CAC0808	HybG	Hydrogenase expression factor
CAC0809	HypE	Hydrogenase formation factor
CAC0810	–	Hydrogenase maturation factor
CAC0811	–	Hydrogenase expression-formation factor

[FeFe]-hydrogenase and associated proteins

The protein clusters we generated in Figs. 3.3a–b indicate that HydA1 functionally relates to at least five different proteins, including conserved accessory proteins CAC1356 (HydG), CAC 1631 (HydE), and CAC1651 (HydF), as well as glutamate synthase protein β-chain and small subunit CAC0764 and CAC1674. Strong interactions between the proteins HydA1, HydE, HydF, and HydG (Fig.3.3a) are consistent with previous findings by King et al. [43] and Posewitz et al. [58], which demonstrate the necessity for co-expression of HydA (or HydA1) with maturation proteins HydE, HydF, and HydG in order to synthesize the active site in *C. acetobutylicum*. Together, the three maturation proteins are functionally important for expression the [FeFe]-hydrogenase and deletion of one of the proteins may lead to inactivation of HydA1 [4].

In addition to these known interactions, the algorithm was able to identify functional associations between the query protein HydA1 and proteins not well defined in *C. acetobutylicum*. In Cluster A (Fig. 3.3a), a second set of interaction proteins occurs between the maturation proteins and protein CAC0487. According to STRING database, CAC0487 is an uncharacterized protein. Due to high interconnectedness with the maturation proteins, CAC0487 appears to be essential for development or co-occurrence of HydE, HydF, and HydG. The exact role of this accessory protein is unclear from the clustering diagrams, but it does not appear that the protein directly interacts with the [FeFe]-hydrogenase (HydA1). To fully understand the functional role of CAC0487, extensive characterization studies are necessary.

The interaction between HydA1, CAC0764 (glutamate synthase β-chain), and CAC1674 (glutamate synthase subunit) is another relationship not clearly defined by previous studies (Fig. 3.3b). Glutamate synthase genes are often associated with uptake and assimilation of ammonia. Although it seems unlikely that there should be an interaction between ammonia uptake proteins and hydrogenase, small unit glutamate synthase genes in *C. acetobutylicum* have been located near the gene *hydA*, which encodes [FeFe]-hydrogenase [71]. The glutamate synthase gene gltX was previously identified by Stutz and Reid [71] in a hydrogen cluster associated with hydrogenase. The exact role of gltX is still unclear; however, it is thought that the gene is functionally important for the production of solvents such as butanol and ethanol by *C. acetobutylicum* [71]. The presence of CAC0764 and CAC1674 in Cluster B and their interaction with HydA1 (Fig. 3.3b) has not been previously described in *C. acetobutylicum* to our knowledge, but we may hypothesize that these two glutamate synthase proteins are located close to *hydA* within the hydrogen cluster based on the evidence provided by Stutz and Reid.

[NiFe]-hydrogenase and associated proteins

The last two identified clusters (Figs. 3.3d–e) contain the [NiFe]-hydrogenase enzyme (HupS/L). In Cluster D (Fig. 3.3d), HupS and HupL are shown as fully interacting with CAP0143 (HupD), CAP0144 (possible steroid binding protein), CAP0145 (putative uncharacterized protein), and CAP0146 (HypF). According to [82], [NiFe]-hydrogenases, which are called *hup* genes and include *hupS* or *hupD*, generally play a role in maturation of hydrogenase enzymes and H-centers. In this study, HupD interacts directly with the large subunit HupL and two other unknown proteins. Also, in *Azotobacter chroococcum*, analysis of amino acid sequences in hydrogenase show that HupD is present in close proximity to HupL [24].

Cluster E (Fig. 3.3e) shows HupS and HupL functionally relates with accessory proteins CAC0811 (HypD), CAC0809 (HypE), CAC0810 (HypF), and CAC0808 (HybG). In general, *hyp* genes are often responsible for insertion of nickel and iron in [NiFe]-hydrogenase clusters [82] and are thus essential components in the biosynthesis of hydrogenase. Butland et al. [12] demonstrate that HybG is another accessory protein essential for synthesis of hydrogenase 2. In Fig. 3.3e, HybG interacts with both HypD and HypE, which is consistent with pull-down data for *Escheridchia coli*, where HybG is shown to form a complex with both HypD and HypE [12].

3.6 Conclusion

In Sections 3.2 and 3.3, we have identified a number of theoretical conditions that vertices of a μ, γ-quasi-clique must satisfy and provided a pseudocode outline of an algorithm for μ, γ-quasi-clique enumeration. In Section 3.4, we evaluated this algorithm on synthetic graphs and found that the algorithm runtime scales well to successively larger sparse graphs. We also tested our proposal for using a hierarchical bitmap index, and empirical results show that graphs with many μ, γ-quasi-cliques may benefit greatly from this technique, though the overhead will slow down algorithm runtime on graphs with fewer μ, γ-quasi-cliques. Finally, in Section 3.5, we presented an application of our algorithm to the problem of identifying genes related to biological hydrogen production, establishing the practical value of our contribution.

Chapter 4

Enabling the Parallelization of Perturbation-Type Algorithms

4.1 Introduction

In this chapter, we propose a scheme for parallelizing Graph Perturbation Theory algorithms, like the ones presented in Chapter 2. A primary motivating factor for this work is the large size of scientific and web-based datasets [63, 64, 65, 73, 55, 66, 85, 89]. By enabling these algorithms to scale to machines with multiple processors using shared and distributed memory, we expect to greatly extend the horizon of problems that are feasible to solve.

The structure of the chapter is as follows. In Section 4.2, we analyze the various aspects of the problem of parallelizing the edge removal case of the Perturbed Graph Clique Enumeration algorithm described in Chapter 2 and present empirical evidence that demonstrates the scalability of our algorithm. In particular, we focus on how to divide the work among the processors, how to access the various indices in a scalable fashion, how to distribute the initial workload, how to eliminate dependencies between the units of work, and how to balance the workloads among the processors. In Section 4.3, we consider the parallelization of the edge addition case of the algorithm and discuss aspects of the parallelization that are exclusive to the edge addition case. Finally, in Section 4.4, we present our conclusions and future work directions.

4.2 Parallel edge removal algorithm

In this section, we focus on the challenges inherent in parallelizing Graph Perturbation Theory algorithms, in general, and the edge removal case of the Perturbed Clique Enumeration algorithm, in particular. In Section 4.2.1, we provide a brief description of the edge removal algorithm.

4.2.1 Serial algorithm

In this section, we will briefly recapitulate the relevant theoretical results from Chapter 2 for updating the set of maximal cliques of a graph when some number of edges are being removed from the graph. As before, we refer to the graph before the perturbation as G, and we use G_{new} to denote the graph after the perturbation. Similarly, we use $\mathcal{C}$ and $\mathcal{C}_{new}$ to denote the set of maximal cliques of G and G_{new}, respectively. The objective of the perturbation algorithm is to enumerate the "difference sets" $\mathcal{C}_+ = \mathcal{C}_{new} \setminus \mathcal{C}$ and $\mathcal{C}_- = \mathcal{C} \setminus \mathcal{C}_{new}$ so that $\mathcal{C}_{new}$ may be determined from $\mathcal{C}$.

In the case where no edges are being added to the graph, we note that any maximal clique of G that is also a clique in G_{new} must be maximal in G_{new}. From this observation, we can see that the cliques of $\mathcal{C}_-$ will be exactly the cliques of G that do not contain an edge being removed. Conversely, the cliques of $\mathcal{C}_+$ will be the subgraphs of the cliques in $\mathcal{C}_-$ that form maximal cliques in G_{new}.

Theorem 4.2.1. *If a set of edges E_- is being removed from G, then*
$\mathcal{C}_- = \{S \in \mathcal{C} \mid S \text{ contains an edge being removed}\}$ *and*
$\mathcal{C}_+ = \{T \mid T \text{ is a complete subgraph of some } S \in \mathcal{C}_- \text{ that is maximal in } G_{new}\}$.

To enumerate the cliques of $\mathcal{C}_-$, we simply need to retrieve the set of maximal cliques of G that contain an edge being removed. In order to retrieve this set efficiently, we pre-calculate and index the cliques of $\mathcal{C}$ that contain each edge of G. We retrieve the set of cliques associated with each edge being removed and combine these sets, eliminating the "duplicate" cliques that contain more than one edge being removed.

To enumerate the cliques of $\mathcal{C}_+$, we employ a recursive procedure to divide each formerly maximal clique of G, as these subgraphs are likely to be missing relatively few edges. At each step of the procedure, we choose a single vertex, v, that is incident to at least one edge being removed and form two subgraphs that may be further divided. The first subgraph is formed by removing v from the current subgraph, and the other is formed by removing the vertices not adjacent to v in G_{new}. Effectively, these two subgraphs serve to eliminate all of the "non-edges" that are incident on v at each iteration, eventually resulting in fully connected subgraphs.

To ensure that the subgraphs we generate for $\mathcal{C}_+$ are maximal, we keep track of the vertices outside of the current subgraph that are adjacent to some vertex of the subgraph, as well as the number of subgraph vertices to which each of these *counter* vertices are not adjacent in G_{new}. We maintain this information because a clique is maximal if and only if there is no vertex of the graph adjacent to every vertex of the clique—if a *counter* vertex becomes adjacent to all of the vertices of our subgraph, we may stop the recursive division, as no further subgraph will be a maximal clique. We continue the division procedure until we have found all complete subgraphs of cliques in $\mathcal{C}_-$ that are maximal in G_{new}.

4.2.2 Parallelization strategy

In this section, we will analyze and evaluate several challenges to producing a scalable parallel version of the edge removal algorithm described in the previous section. Three major issues we consider when parallelizing the algorithm are:

- How to represent the major unit of work:
  - We choose a lightweight, coarse-grained unit of work by using clique IDs. (Further details in Section 4.2.3)
- How to limit or eliminate explicit or implicit dependencies in the serial algorithm:
  - We present two work distribution models capable of eliminating duplicate clique IDs without communicating between processors. (Further details in Section 4.2.6)
  - We introduce novel theory so that we can detect duplicate subgraphs in an *embarrassingly parallel* fashion. (Further details in Section 4.2.5)
- How to alleviate bottlenecks or serialization points in the parallel algorithm:
  - We read indices into memory so that we are less disk-bound. (Further details in Section 4.2.4)
  - We consider various load balancing strategies in Section 4.2.6.

As a result of our parallelization choices, we have divided the work of the algorithm into four "phases:"

- the *Init* phase, where the graph and index data are read from disk and broadcast between the processors;
- the *Root* phase, in which the initial workloads are calculated;
- the *Enumerate* or "main work" phase, in which the workloads are processed and workload balancing is performed; and
- the *Idle* phase, in which a processor has no work to either perform or acquire.

4.2.3 Primary work division

One challenge associated with parallelization is identifying an appropriate data structure for the main work unit. This data structure needs to be capable of maintaining all of the information relevant to performing the subgraph division process, but be as lightweight as possible so that the data structures may be passed among the processors.

For the recursive vertex removal process, we propose to parallelize execution along the retrieval and processing of the cliques of $\mathcal{C}_-$, using the individual clique ID as the unit of work. That is, there are two main "work" stages of the algorithm: in the *Root* phase, indexed data is accessed to retrieve the IDs of cliques that contain edges being removed from the graph, and in the *Enumerate* phase, the clique vertices are retrieved and the clique is subdivided as in the serial algorithm.

The advantage of using a clique ID as the major unit of work is twofold. The main advantage is that, in addition to being sufficient to fully describe the unit of work it represents, a clique ID is an extremely lightweight data structure and can be passed between threads or processors at minimal cost. Another advantage is that, by parallelizing the algorithm along the processing of individual cliques, we can retain the recursive structure of the original clique processing, without needing to alter the basic structure of the recursive division algorithm.

The primary disadvantage to using clique IDs as the basic unit of work is that this results in a coarse level of parallelism: each clique ID corresponds to a variable amount of work, and while clique IDs likely correspond to similar, relatively small workloads, it is possible that a single clique ID could generate a huge search space for the recursive removal procedure to explore, resulting in a large chunk of work that cannot be split across processors.

4.2.4 Parallelizing index accesses

One of the defining attributes of Graph Perturbation Theory algorithms is the utilization of various indices to improve the rate at which properties of the original (unperturbed) graph are calculated. However, in a typical large-scale parallel system, disk accesses are relatively expensive and unlikely to scale, as all of the users on the system may share a single distributed file system. As such, disk accesses need to be minimized and replaced with memory accesses where possible, since memory is local to each processor and is able to scale. Thus, we adopt a strategy of reading in the entire index when possible, or a large segment of the index when the index is too large to fit into memory. In extreme cases where the index is very large and reading in the indexed data becomes a bottleneck for the algorithm, compressing the indices can reduce the amount of data read from the disk, and the reduced I/O load may outweigh the cost of decompressing the index chunks.

Because we adopt the strategy of reading the indices into memory, there is a premium placed on being able to fit the entire index into memory. To this end, we chose to use a more compact representation for the index that associates edges with the cliques that contain them. In our results for Chapter 2, we used a *dense index* to store this data; that is, we listed the clique IDs for all possible edges between vertices of the original graph, regardless of whether there were any cliques (or even an edge in the graph) associated with those two vertices. For graphs with hundreds of thousands of vertices, though, this strategy can result in index files that are hundreds of gigabytes (GB) large, well beyond current memory capacities. However, if we modify the index so that only edges that exist in the graph are included in the index, we can reduce the size of the index from $O(|V|^2)$ to $O(|E|)$. As the graph itself requires at least $O(|E|)$ space to store in memory, we may assume that this reduced index is small enough to read directly into memory. The only downside to reducing the size of the index is that we incur a small overhead in finding what position in the reduced index corresponds to a given edge. Algorithmically, we implement this lookup as a binary search on the list of edges in the graph.

Index access results

To illustrate the efficacy of this approach, we implemented two versions of the edge removal algorithm. One of these implementations performs the various seek and read functions on the edge index data necessary to retrieve the initial set of clique IDs from the disk, whereas the second implementation reads the edge index data into memory and retrieves the relevant clique IDs by accessing the memory-resident index.

We ran our algorithm on a protein-protein interaction network derived by Zhang *et al.* [91]. This network was constructed by applying a threshold of 1.5 to the Purification Enrichment scores [1] calculated using experimental data published by Gavin *et al.* [28]. The network contains 2,436 vertices, which represent various proteins in the yeast *Saccharomyces cerevisiae*, and 15,795 edges, which represent likely interactions between the proteins. The graph contains 19,243 maximal cliques of size three or larger, which represent potential protein complexes. Unless otherwise stated, we discard maximal cliques of size one or two, as an isolated vertex or edge is unlikely to represent a significant relationship in a network.

For this experiment, we generated a 20% removal perturbation in which 3,159 edges of the graph were randomly selected to be removed, with an equal probability for each edge to be selected. This perturbation removes 18,779 maximal cliques from the 19,243 cliques of original graph and adds 33,941, for a total of 34,405 maximal cliques in the perturbed graph. Note that, as our generated perturbation is rather large, this particular edge removal is not a good example of the advantages inherent in employing the Graph Perturbation approach—most of the cliques in the graph are affected by the perturbation, and a full maximal clique enumeration of the resulting graph is faster than the perturbed algorithm. However, it is a good test case for the purposes of evaluating the performance and scalability of our approach as the runtime is large relative to many of the other perturbations we tested, and the graph and index files are relatively small.

For the purposes of these results, our implementation uses a simple producer-consumer model in which one processor is responsible for generating the clique IDs and all other processors request and process these clique IDs. The advantages and disadvantages of this model will be

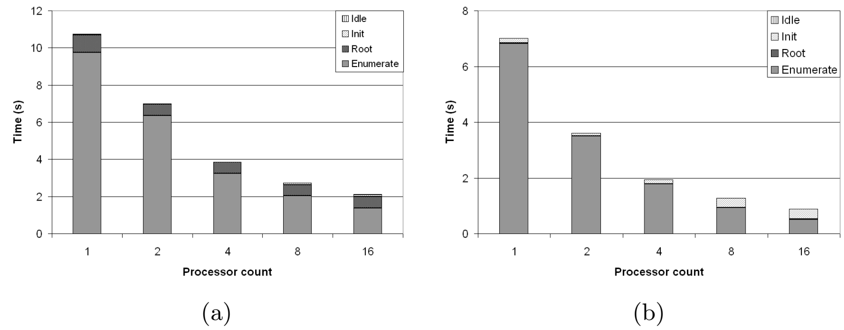


Figure 4.1: Timing results for disk-based (4.1a) and in-memory (4.1b) index accesses. All times are given in seconds and represent the longest duration that a single processor spent on the given task. The timing results were broken down as follows. *Init time* represents the amount of time required to read and broadcast the graph and index data. *Root time* represents the amount of time required to read in the set of clique IDs that contain an edge being removed. *Enumerate time* represents the amount of time required to retrieve the clique vertices, perform the recursive removal procedure, and load balance. *Idle time* represents the amount of time spent by a processor that has no work and is unable to find work to steal.

more fully discussed in Section 4.2.6.

This experiment and all of the other results in this chapter are run on the Jaguar system at Oak Ridge National Lab (ORNL). Jaguar consists of 7,832 quad-core Opteron compute nodes, each of which runs at 2.1 GHz and provides 8 GB of DDR2-800 memory. All of the code being tested was written in C and compiled using the `pgi/10.3.0` and `PrgEnv-pgi/2.1.1.41A` modules provided by ORNL.

As can be seen in Figure 4.1, reading the index data into memory reduced the overall single-processor algorithm time by a third. Further, when tested on 16 processors, the implementation using in-memory index accesses outperforms the other implementation by a factor of two, indicating that this approach is more scalable, in addition to being faster. Moreover, as the file system on large HPC clusters is typically shared among all users, the in-memory code should also perform more consistently, as its performance is less affected by the I/O needs of the other users on the system.

4.2.5 Pruning duplicate subgraphs

After using the indexed data to retrieve the set of cliques containing an edge being removed, the edge removal algorithm recursively subdivides the retrieved cliques in order to find the novel maximal cliques introduced by the edge removal. In order to improve the efficiency of this division procedure, we wish to avoid applying the recursive procedure multiple times to

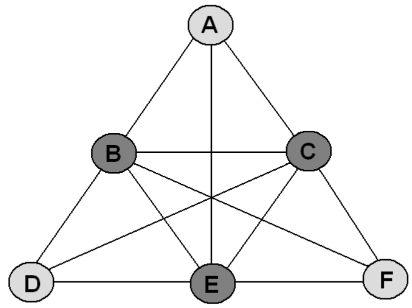


Figure 4.2: A graph in which subgraph $\{B, C, E\}$ is common to three different cliques: $\{A, B, C, E\}$, $\{B, C, D, E\}$, and $\{B, C, E, F\}$.

subgraphs that are contained in more than one clique of $\mathcal{C}$. For example, in Figure 4.2, the subgraph $\{B, C, E\}$ is contained in three maximal cliques, but if $\{B, C, E\}$ (or some subgraph thereof) were a maximal clique of G_{new}, we would only want to apply the division procedure to this subgraph once. In Chapter 2, we eliminated this type of work duplication by adding each subgraph considered by the removal algorithm into a hash table and ensuring that each subgraph produced by the recursive procedure was novel.

However, using a hash table is unsuitable for parallel processing, as accesses to this hash table would need to be synchronized among the various processors. In order to remedy this difficulty, we propose a technique that will allow us to detect duplicated subgraphs without requiring information about the prior computation. To do so, we allow a subgraph to be produced only by the clique of $\mathcal{C}_-$ that is lexicographically first among those cliques of $\mathcal{C}_-$ that are supergraphs.

Definition 4.2.1. We define some lexicographic ordering among the vertices, i.e., $V(G) = \{v_1, v_2, \ldots, v_n\}$. A subgraph S *lexicographically precedes* subgraph T iff there exists some $v_i \in S \setminus T$ such that $i < j$ for all $v_j \in T \setminus S$. We denote this relationship as $S \overset{\mathcal{L}}{<} T$.

Using this definition, we would only allow the subgraph $\{B, C, E\}$ in Figure 4.2 to be produced from the clique $\{A, B, C, E\}$. Note that this definition differs from the usual lexicographic ordering in that a supergraph of a graph would precede the graph it contains; however, as the perturbation algorithm does not compare subgraphs that contain one another, this definition is sufficient for our purposes. We now present a technical result that will allow us to recognize whether a subgraph is being produced from its lexicographically first supergraph in $\mathcal{C}_-$.

Theorem 4.2.2. *Let C be a clique of $\mathcal{C}_-$, and let $S \subset C$ be a maximal subgraph of C produced by the recursive procedure. Let $R = C \setminus S$, and let v_i be the lexicographically first counter vertex* *that is adjacent to every vertex of C in G (but not in G_{new}). C is the lexicographically first clique of $\mathcal{C}_{-}$ that contains subgraph S if and only if some vertex of $R_i = \{v_k \in R \mid k < v_i\}$ is nonadjacent to v_i in G.*

Proof. We prove the negation of the claim. Suppose that every vertex of R_i is adjacent to v_i. Consider the subgraph $X = S \cup R_i \cup \{v_i\}$. $S \cup R_i$ is a subset of C, so all vertices in R_i and S must be adjacent in G, and v_i is adjacent to all of S and R_i in G by supposition. Thus, X must form a clique in G. Further, X precedes C lexicographically, as every vertex of $C \setminus X$ is lexicographically after v_i by the construction of R_i. Thus, some supergraph of X forms a maximal clique in $\mathcal{C}_{-}$ that lexicographically precedes C.

Suppose that there is some clique $C' \in \mathcal{C}_{-}$ such that $C' \supset S$ and $C' \overset{\mathcal{L}}{<} C$. Let C' be the lexicographically first such clique. By Definition 4.2.1, this means that there is some $v_k \in C' \setminus C$ such that $k < j$ for all $v_j \in C \setminus C'$. Note that, as C' is a clique in G, v_k is adjacent to every vertex of S in G. Also, k must be the lexicographically first such vertex—if there were some v_j adjacent to all of S in G such that $j < k$, $S \cup \{v_j\}$ would be a clique of G that contains S and lexicographically precedes C' (as would the clique of $\mathcal{C}_{-}$ that contains $S \cup \{v_j\}$). Since $C' \overset{\mathcal{L}}{<} C$, no vertex of $C \setminus C'$ may precede v_k. As such, every vertex of R_i must be contained in C' and thus be adjacent to v_i in G. □

To make use of this result in the algorithm, we maintain R, the set of vertices removed from the original clique of $\mathcal{C}_{-}$. In addition, for each *counter* vertex, we maintain not only the number of subgraph vertices that are nonadjacent in G_{new}, but also the number of vertices nonadjacent in G. When a *counter* vertex v_i becomes adjacent to all vertices of the subgraph in G, we check whether v_i lexicographically precedes the lexicographically first vertex of R. If so, we iterate through R in lexicographic order until we find a vertex lexicographically later than v_i (indicating that the original clique is the lexicographically first) or a vertex nonadjacent to v_i, at which point we would backtrack.

Duplicate subgraph results

To illustrate the effectiveness of our theoretical results on duplicate subgraph pruning, we evaluate the edge removal code with and without the code that detects duplicate subgraphs enabled. This experiment is performed on the same 20% perturbation of the Gavin *et al.* data as in Section 4.2.6 and is also performed on the Jaguar system. For this experiment, we used a single processor, as well as our in-memory index accessing strategy. The results of this experiment, in Table 4.1, show that duplicate cliques can potentially represent a majority of the algorithm output if not properly pruned. Unsurprisingly, algorithm runtime was improved dramatically by the application of this theory. Moreover, the algorithm results without duplicate pruning would require further processing to produce an enumeration of maximal cliques without duplicates.

Table 4.1: Effects of duplicate pruning on algorithm performance. Here, we report the number of (potentially duplicate) cliques of $\mathcal{C}_+$ identified by the algorithm as well as timing results using the in-memory index access and producer-consumer strategies on a single processor. As the duplicate pruning results are only relevant during the recursive removal phase of the algorithm, we include here only the time spent in the *Enumerate* phase of the algorithm, which encompasses clique retrieval, recursive removal, and load balancing. The amount of time spent during the other phases of the algorithm was consistent between the two runs—see Figure 4.1 for a comparison of the time spent in the various stages of the edge removal algorithm.

Duplicate pruning?	$\|\mathcal{C}_+\|$	Enumerate time
Without	228373	25.681
With	33941	6.830

4.2.6 Workload distribution

In this section, we propose and evaluate two different strategies for distributing the algorithm workload among the processors. We briefly mentioned the first model, producer-consumer, in the previous section. In the producer-consumer model, a single processor is responsible for accessing the edge index to retrieve the set of clique IDs for cliques that contain an edge being removed from the graph. (We refer to this step of the edge removal algorithm as the *Root* phase.) After the producer has retrieved these clique IDs and placed them on a queue, it begins distributing these clique IDs to the consumers, or processing clique IDs if all of the consumers already have work.

All processors other than the producer are referred to as consumers. Consumers iteratively request a block of work from the producer and then process the clique IDs it receives. For our implementation, we chose to distribute work in blocks of 32 clique IDs. Larger blocks of work result in fewer transfers (and thus have less transfer overhead) and help occupy a consumer while the producer is distributing work to other consumers. On the other hand, overly large block sizes may result in a load imbalance and slow down the initial round of work distribution—our choice of block size strikes a balance between these extremes. A consumer continues to request work until the producer responds that it has no work left, at which point the consumer stops.

The primary advantage of the producer-consumer model is its simplicity and similarity to the serial code. In the serial algorithm, we used a bitmap to eliminate "duplicate" clique IDs that result from cliques containing multiple edges being removed; if we were to divide the

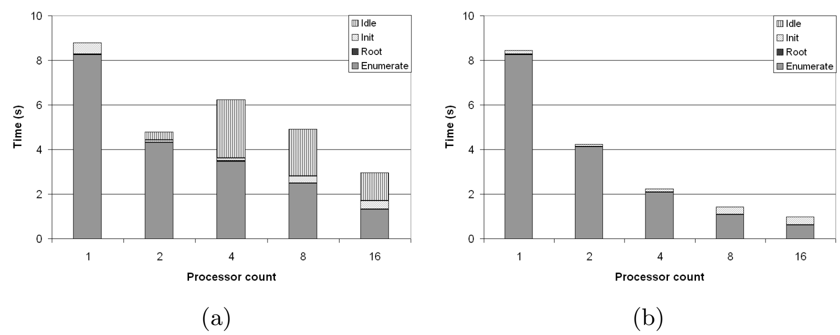


Figure 4.3: Timing breakdown for work distribution strategy using in-memory index accesses with no load balancing (4.3a) and work stealing (4.3b). All times are given in seconds and represent the longest duration that a single processor spent on the given task. The timing results were broken down as follows. *Init time* represents the amount of time required to read and broadcast the graph and index data. *Root time* represents the amount of time required to read in the set of clique IDs that contain an edge being removed. *Enumerate time* represents the amount of time required to retrieve the clique vertices, perform the recursive removal procedure, and load balance. *Idle time* represents the amount of time spent by a processor that has no work and is unable to find work to steal.

set of removed edges among the processors, we would need some communication or algorithmic strategy to eliminate these duplicate clique IDs before they are processed. A major disadvantage of this approach is that it serializes the *Root* phase of the algorithm and requires that most of the clique IDs be passed from the producer to a consumer before being processed. From our previous experiment (see Figure 4.1), we can see that that reading the index data into memory reduces the time spent on the *Root* phase drastically (to less than 0.01 seconds), so this strategy may be quite viable unless we are dealing with very large perturbations.

The second strategy we propose is to distribute the work of calculating the initial workload (the *Root* phase) among the processors and balancing the processors' workloads as the work queues empty. As mentioned earlier, though, the serial algorithm relies on a bitmap to eliminate retrieving cliques that contain more than one edge being removed, so dividing the set of removed edges among the processors would require some type of communication or algorithmic strategy to eliminate these duplicate clique IDs. As our random perturbation on the Gavin *et al.* data results in retrieving 488,972 clique IDs (of which only 18,779 are unique), eliminating such duplicates as efficiently as possible is an important priority.

To sidestep this challenge, we choose to divide the set of clique IDs among the processors, so that each processor retrieves the clique IDs associated with every edge being removed but ignores clique IDs that fall outside of its assigned IDs. Even though this strategy requires all of the processors to retrieve the full set of clique IDs, each processor can use a bitmap to eliminate duplicate clique IDs in its range without needing to communicate with other processors or retrieve the clique vertices before recognizing that a particular clique ID is a duplicate.

A potential difficulty with this workload division strategy is that the initial workload distribution is unlikely to be evenly distributed among the processors, leading to issues of load balancing among processors. To overcome this difficulty, we opt to use a dynamic load balancing strategy in which processors request additional work from others as their work queues run empty, a strategy known as "work stealing" [46].

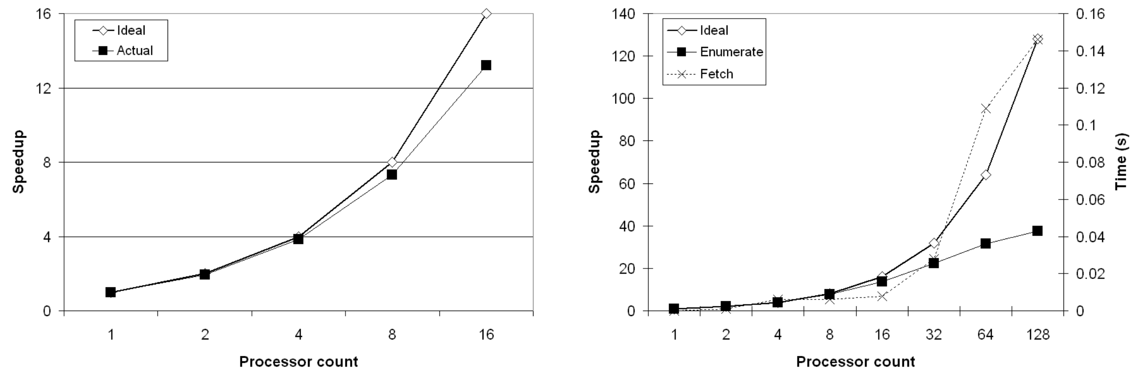
To evaluate the effectiveness of our workload distribution and "work stealing" strategy, we provide a comparison between the unbalanced workload, in which each processor is assigned a block of the clique data index and processes only those clique IDs that appear in this block, and the load balanced algorithm, in which these processors that finish their initial workloads early "steal" work from other processors. For this experiment, we use the same graph data and perturbation as in Section 4.2.6, and we run our code on Jaguar. The results of this experiment appear in Figure 4.3.

From Figures 4.3, it is clear that our strategy of work stealing is very effective in balancing the workloads among the processors, reducing the *Idle* below 0.01 seconds across all processors. However, if we compare the timing results in Figure 4.3 with our results for the producer-consumer model (Figure 4.1), we see that the work distribution/load balancing strategy introduces an overhead of 20.9% over the producer-consumer model on a single processor. This advantage decreases to 17.6% at 16 processors, though, suggesting that workload distribution strategy may be more scalable than the producer-consumer model at extreme scale.

4.2.7 Scalability results

To test the scalability of our algorithm, we used the same perturbation of the Gavin *et al.* data as described in Section 4.2.6, and we ran our algorithm on the ORNL Jaguar system. We evaluated the time spent by our algorithm in the *Enumerate* (main work) phase using both the producer-consumer and workload distribution models on up to 64 processors. The results appear in Figure 4.4.

In Figure 4.4, we can see that the speedup for the *Enumerate* phase of the algorithm tapers off quite a bit for the workload distribution strategy as the number of processors increases beyond 32. This loss of scalability coincides with (and is due to) a sharp increase in the amount of time spent by the algorithm in load balancing. As these events occur at the point where the *Enumerate* phase of the algorithm takes less than one second total, though, the effect may just be a result of applying too many processors for the problem size.



(a) (b)

Figure 4.4: Evaluation of speedup of enumeration time for producer-consumer (4.4a) and workload distribution (4.4b) models. The *Fetch* time, or amount of time spent stealing work, is also plotted in Figure 4.4b. Speedup was calculated for the time spent in the *Enumerate* phase of the algorithm, in which the algorithm retrieves the vertices of a given clique, performs the recursive removal procedure, and balances the processor workloads. The speedup for the *Enumerate* phase of the algorithm is plotted against ideal (linear) speedup. In Figure 4.4b, *Fetch* time is plotted on the secondary axis, in seconds.

4.3 Parallel edge addition algorithm

4.3.1 Serial algorithm

We approach the problem of adding edges to a graph as the inverse of the edge removal problem. Specifically, if edges are being added to G to form G_{new}, we consider the perturbation caused by removing those same edges from G_{new} to form G. By the results of Section 4.2.1, the set of maximal cliques being removed from G_{new} by the reverse perturbation (which is exactly $\mathcal{C}_+$, the set of cliques being added to G by the original perturbation) will be the cliques of $\mathcal{C}_{new}$ that contain an edge being "removed from G_{new}." Further, the set of maximal cliques being added to G_{new} by the reverse perturbation (which is exactly $\mathcal{C}_-$) will be the complete subgraphs of the cliques in the previous set that form maximal cliques in G.

Unlike the removal case, though, we have not indexed the cliques of G_{new} by the edges they contain (as we are trying to *calculate* the cliques of G_-). Thus, to calculate the set of cliques in G_{new} that contain one of the added edges, we employ a variation of the Bron-Kerbosch (BK) clique enumeration algorithm. [10]

BK operates by intelligently traversing the subgraphs of a graph so that it can find all of the maximal cliques of the graph while minimizing the number of subgraphs considered. To do so, it maintains three sets in an array called `compsub`: the current clique; the *candidate set*, which are the vertices adjacent to all vertices of the current clique; and the *not* set, which lists vertices that have previously been added to the clique and should not be further explored. BK continues to add vertices to the clique set one at a time, backtracking when the *candidate* and *not* sets are empty (in which case the current clique is maximal) or when a vertex of *not* is adjacent to all of the vertices in the clique and candidate set (in which case all extensions of this clique have already been enumerated). When backtracking, the algorithm moves the added vertices from the current clique to the *not* set, so that cliques containing these vertices will not be explored in other branches of the search tree.

The efficiency of the BK search, though, is due to how the vertex to be added is chosen. At each step in the computation, BK finds the *candidate* or *not* vertex that is nonadjacent to the fewest number of candidate vertices. BK adds, in turn, each one of the *candidate* vertices nonadjacent to this vertex, as well as the vertex itself if it is in the *candidate* set. By adding vertices to the clique set in this order, BK finds large cliques that overlap others first, eliminating a large portion of the search space.

In order to adapt this technique to calculate the cliques containing an arbitrary edge (u, v), we initialize `compsub` to contain u and v as the clique set, the common neighbors of u and v that succeed u and v lexicographically as the *candidate* set, and the *not* set to contain the common neighbors of u and v that that precede both u and v lexicographically.

Once we have calculated the set $\mathcal{C}_+$, we apply the recursive removal procedure described earlier to the cliques of $\mathcal{C}_+$ in order to calculate $\mathcal{C}_-$. However, unlike the edge removal case, we can check the maximality of the resulting subgraphs by looking up the cliques in an index that maps clique hash values to the IDs of maximal cliques of G that correspond to those hash values. As such, we only need to keep track of *counter* vertices that are adjacent to the clique in G_{new} but not in G, so that we can still eliminate duplicate subgraphs as described in Section 4.2.5.

4.3.2 Parallel algorithm

In order to parallelize the edge addition algorithm, we adapt the parallel BK implementation described in [67] to the problem of finding cliques in G_{new} that contain an edge being added.

The implementation of BK described in [67] is a depth-first traversal in which the major unit of work is the *candidate list*, which corresponds to the `compsub` array in the serial version of BK—that is, a *candidate list* represents the current clique along with its *candidate* and *not* lists. As before, we can modify the initial workload of the BK algorithm in order to enumerate only those maximal cliques containing the added edges. We distribute the set of added edges (and thus the corresponding initial *candidate list* structures) among the processors in a round-robin fashion.

The load balancing strategy in [67], which we also adopt, is performed on two levels—local and remote work sharing. Local work sharing occurs when one thread on a processor runs out of work, at which time it checks the other threads' work stacks (in some randomized order) to see if another thread on the same processor has additional work. Remote work sharing is handled similarly, in that when all threads on a shared-memory processor have exhausted their work stacks, that processor polls the other processors to see if another processor still has work in its work stack. Again, polling is performed in a random order so as to avoid having a single processor inundated with work requests.

Once a processor or thread has located some work that it can steal, the target thread or process transfers a single *candidate list* structure from the bottom of its work stack to each thread requesting work. The rationale behind this load balancing strategy is that a single *candidate list* structure may represent a wildly variable amount of work, and those *candidate list* structures that were generated earliest (and therefore reside on the bottom of the work stack) are the most likely to represent a large amount of work.

As with the edge removal case, we treat the recursive removal operation on the resulting cliques of $\mathcal{C}_+$ as an indivisible unit of work, as the perturbation being applied to the graph—and hence the typical individual subgraph division operation—is assumed to be small. Duplicate subgraph pruning is performed as described in Section 4.3.1, and the hash value index is read into memory. For all of the graphs that we tested, this index was able to be read into memory in its entirety; however, for larger graphs, it may be necessary to split the index and read in only a section of the index at a time into memory. In this event, it may be more effective to distribute the index among the processors and pass the potential cliques of $\mathcal{C}_-$ to the processor that possesses the appropriate section of the hash value index.

4.3.3 Scalability results

Using the index access and load balancing strategies outlined in Sections 4.2.4 and 4.3.2, we performed several experiments on large-scale data to test the scalability of the edge addition algorithm. For this experiment, we chose to use a dataset derived from the Medline database [2] described in [8]. We chose to use this dataset because of its large size (2.6 million vertices), sparsity (1.9 million total edges), and weighted nature. In particular, we do not report results for randomly generated edge addition perturbations on the previously-described Gavin *et al.* data, as these random perturbations of similar magnitude to the previous edge removal perturbations affected just hundreds of maximal cliques and were calculated in milliseconds. By applying edge weight thresholds of 0.85 and 0.80 to the Medline data, though, we are able to generate graphs of 713,000 and 987,000 edges, respectively, resulting in an edge perturbation of about 38.5% on the smaller graph. This perturbation adds 73,623 maximal cliques to the 70,926 cliques of

0.85-weight graph and removes 34,745, for a total of 109,804 maximal cliques in the 0.80-weight graph. Initial test results appear in Table 4.2.

Table 4.2: Scalability results for edge weight–induced perturbation on the Medline graph. All times are given in seconds and represent the longest duration that a single processor spent on the given task. The timing results were broken down as follows. *Init* time represents the amount of time taken by the algorithm in allocating memory for the algorithm data structures and reading the graph and indices into memory. *Root time* represents the amount of time taken by the algorithm in generating the initial set of *candidate list* structures for the modified BK algorithm. *Enumerate time* represents the combined amount of time required to run the BK algorithm to detect the set of cliques in $\mathcal{C}_+$ as well as performing the recursive removal procedure, index lookups required to calculate $\mathcal{C}_-$, and load balancing. *Idle time* represents the time spent by a processor that has finished its workload and has no other work to steal.

Procs	Init	Root	Enumerate	Idle
1	0.876	0.000	1.459	0.000
2	0.951	0.000	0.773	0.005
4	1.197	0.000	0.489	0.002
8	1.381	0.000	0.249	0.007

As can be seen in Table 4.2, the *Init* phase of the algorithm plays a significant role in the overall runtime of the algorithm. Also, it is clear that the time spent in the *Enumerate* phase of the algorithm is too short to permit a full scalability study. As such, we evaluate the scalability of the edge addition algorithm to larger processor counts by using the Medline data to generate larger workloads. In order to increase the problem size evenly, we formed successively larger graphs made up of independent components identical to original graph, linearly increasing the number of vertices, edges, perturbation size, maximal cliques, and resultant index data. We term these independent components "copies," and we increased the number of copies in our graph from 1 to 6 as we increased the number of processors from 1 to 64. We only present results as to the scalability of the *Enumerate* time—the *Root* and *Idle* phases last less than 0.07 seconds in all cases, and the *Init* time, which consists of reading in the graph and index data from the disk, does not scale and eventually dominates the algorithm runtime. The results for our experiments appear in Figure 4.5.

Interestingly, despite the relatively large perturbation size, the edge addition algorithm is far faster than enumerating the maximal cliques of the perturbed graph using BK, particularly as the size of the graph (number of copies) is increased. We do not present full results here, but as a point of reference, enumerating the maximal cliques of the four-copy Medline graph took over 20 minutes using 128 processors on Jaguar, with more than 99% of that time being spent

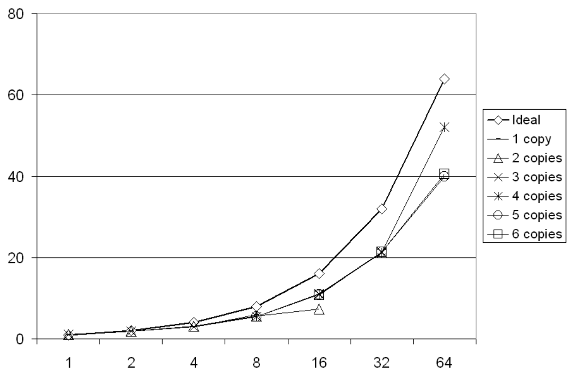


Figure 4.5: Normalized speedup versus number of processors. Speedups were calculated as $(t_1 * n_c)/t_{c,p}$, where t_1 is the *Enumerate* time for calculating the perturbation on one copy of the graph using a single processor, n_c is the number of copies of the graph, and $t_{c,p}$ is the *Enumerate* time for calculating the perturbation on c copies using p processors. *Enumerate time* represents the combined amount of time required to run the BK algorithm to detect the set of cliques in $\mathcal{C}_+$ as well as performing the recursive removal procedure, index lookups required to calculate $\mathcal{C}_-$, and load balancing.

in the initial workload generation (*Root*) phase (compared to around 8 seconds on 4 processors for the edge addition algorithm).

From Figure 4.5, we see that the *Enumerate* time for the perturbed edge algorithm scaled almost perfectly as the size of the graph (number of copies) increased. The scalability of the *Enumerate* time is within two-thirds of ideal, though this may be due to the short times involved—the *Enumerate* time for 6 copies of the graph was only 0.216 seconds. As the memory requirements for the algorithm do not scale linearly, we were unable to test our algorithm for graphs larger than 6 copies.

4.4 Conclusion

In summary, we have adapted the problem of recalculating the maximal cliques of a perturbed graph to be calculated in parallel. We have analyzed several aspects of the parallelization, including how we choose our main work unit for parallelization, how we balance the workload among the processors, and how we achieve scalable performance when accessing our various indices. Moreover, we have presented empirical results that show reasonable scaling for our approach up to 64 processors, as well as how several of our implementation decisions have improved algorithm performance and scalability. Many of the techniques we have developed in this chapter could also be applied to other Graph Perturbation Theory problems, particularly, our technique for scalable index accesses.

Chapter 5

Conclusion and Future Work

This thesis has proposed two novel approaches to the problem of uncovering correlations within noisy and uncertain data. The first of these approaches pioneered the concept of Graph Perturbation Theory, in which graph data is indexed and utilized to update graph properties in response to changes due to the uncertain nature of the underlying data. The second approach incorporates the idea of "knowledge priors" in order to identify relevant correlations more efficiently. We developed novel theoretical results characterizing these problems, and we have applied our theoretical work to produce practical algorithm implementations. Moreover, we have presented empirical results demonstrating the efficacy and applicability of our implementations. Additionally, we have proposed and implemented a parallelization of our Perturbed Graph Clique Enumeration algorithm, and we have demonstrated its scalability empirically.

The thesis, however, does not purport to represent the fully comprehensive closure of all valid or effective approaches to the problem of mining noisy and uncertain graph data. One major avenue of inquiry is in the improvement or adaptation of the algorithms described. For instance, a great deal of the work in Chapter 3 was devoted to reducing the search space of the algorithm. Applying this general concept to the algorithm from Chapter 2 (and its parallelization), we might be able to reduce the amount of work required for the algorithm. In particular, it is possible to reduce the size of the *counter* set as the algorithm progresses because we can "deduce" that some *counter* vertices will not be able to contradict the maximality of a subgraph (or perhaps that another *counter* vertex would succeed in contradicting maximality first). Reducing the size of the *counter* set would reduce the amount of work required to maintain the associated *counter_onceadj* and *counter_nonadj* values as vertices are removed from the subgraph. There would be a computational cost for being able to remove *counter* vertices, but there is potential for large gains, as the *counter* set (initially the set of all vertices that are adjacent to *any* vertex of the original clique) can be an order of magnitude larger than the subgraph itself.

Additionally, the performance of the algorithms presented in this thesis is influenced by the order in which the search space is explored—that is, how to choose the vertices to remove from the subgraph for the algorithm from Chapter 2 and how to choose the vertex to add to the subgraph for the algorithm from Chapter 3. Informal testing suggests that the performance of the Chapter 3 algorithm can be changed by several orders of magnitude based on how the subgraph is expanded. The questions of the optimal strategy for each of the two algorithms and under what conditions these strategies are the most effective remain to be explored in depth.

An unexplored possibility for extending the work on parallelizing Graph Perturbation algorithm is to examine the potential of *index splitting*. *Index splitting* refers to the technique of dividing the information in an index (and the data it represents) into chunks and distributing these chunks among the processors. This technique would allow the algorithm to scale to larger and more dense graphs, as the index would not need to fit into memory in its entirety, but care would need to be taken within the algorithm to avoid index "thrashing." Another possible consideration would be applying data compression techniques to the indices or graph data to reduce the I/O time necessary for reading the indices or index chunks from the disk.

Another possible avenue for further exploration is in applying the techniques described in this thesis to other domains. In this thesis, the empirical results being presented were primarily to networks from the biological domain (from [26], [28], [2], etc.), though we did not attempt to use these results to draw meaningful biological conclusions. In the paper [34], the algorithm described in Chapter 3 was applied to biological network data to discover genes associated with biological hydrogen production, but, otherwise, the techniques described in this thesis have not been otherwise practically applied. As the need to uncover meaningful correlations in uncertain relational data seems a sufficiently generic problem, there are many heretofore undiscovered relationships waiting to be discovered, buried within the mountains of data currently being generated.

Further, the concept of Graph Perturbation Theory is not limited to maintaining a set of maximal cliques for a graph. Other graph properties, such as the set of all shortest paths of the graph, have both practical applications and sufficient computational complexity to benefit significantly from a perturbed approach.

As a final thought, our choice of formal definition for the terms *dense* and *enriched* had a profound impact on the development of the enumerative algorithm described in Chapter 3. Other definitions would almost certainly require different theoretical considerations and different data structures to be populated and manipulated, and optimizing an algorithm to identify such structures is an interesting problem unto itself. On a more speculative note, though, there may be deep relationships between the different possible definitions of such an "interesting" subgraph and the data structures and calculations required for an algorithm used to identify these subgraphs. Discovering such relationships could lead not only to advancements in algorithms for graph data, but also to an advancement in potentially any algorithm that performs a search.

REFERENCES

[1] Purification enrichment scores. `http://interactome-cmp.ucsf.edu/`.

[2] Medline fact sheet. `http://www.nlm.nih.gov/pubs/factsheets/medline.html`, June 2010.

[3] James Abello, Mauricio G. C. Resende, Sandra Sudarsky, and Ra Sudarsky. Massive quasi-clique detection. In *Latin American Theoretical Informatics (LATIN)*, pages 598–612, 2002.

[4] M. Kalim Akhtar and Patrik R. Jones. Engineering of a synthetic hydf-hyde-hydg-hyda operon for biohydrogen production. *Analytical Biochemistry*, 373(1):170–172, 2008.

[5] E. A. Akkoyunlu. The enumeration of maximal cliques of large graphs. *SIAM Journal on Computing*, 2(1):1–6, 1973.

[6] R. Albert and A. L. Barabasi. Statistical mechanics of complex networks. *Reviews of Modern Physics*, 74(1):47–97, 2002.

[7] R. Albert, H. Jeong, and A. L. Barabasi. Internet - diameter of the world-wide web. *Nature*, 401(6749):130–131, 1999.

[8] Amit Awekar, Nagiza F. Samatova, and Paul Breimyer. Incremental all pairs similarity search for varying similarity thresholds. In *SNA-KDD '09: Proceedings of the 3rd Workshop on Social Network Mining and Analysis*, pages 1–10, New York, NY, USA, 2009. ACM.

[9] A. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, and J. Wiener. Graph structure in the web. *Computer Networks-the International Journal of Computer and Telecommunications Networking*, 33(1–6):309–320, 2000.

[10] Coen Bron and Joep Kerbosch. Algorithm 457: Finding all cliques of an undirected graph. *Communications of the ACM*, 16(9):575–577, 1973.

[11] Dongbo Bu, Yi Zhao, Lun Cai, Hong Xue, Xiaopeng Zhu, Hongchao Lu, Jingfen Zhang, Shiwei Sun, Lunjiang Ling, Nan Zhang, Guojie Li, and Runsheng Chen. Topological structure analysis of the protein-protein interaction network in budding yeast. *Nucl. Acids Res.*, 31(9):2443–2450, 2003.

[12] Gareth Butland, Jie Wei Zhang, Wenhong Yang, Anthony Sheung, Peter Wong, Jack F. Greenblatt, Andrew Emili, and Deborah B. Zamble. Interactions of the *Escherichia coli* hydrogenase biosynthetic proteins: Hybg complex formation. *FEBS Letters*, 580(2):677–681, 2006.

[13] J. Camacho, R. Guimera, and L. A. N. Amaral. Robust patterns in food web structure. *Physical Review Letters*, 88(22):–, 2002.

[14] M.W. Carter and D.G. Johnson. Extended clique initialisation in examination timetabling. *Journal of the Operational Research Society*, 52:558–544, 2001.

[15] C. Castellano, F. Cecconi, V. Loreto, D. Parisi, and F. Radicchi. Self-contained algorithms to detect communities in networks. *European Physical Journal B*, 38(2):311–319, 2004.

[16] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. R-mat: A recursive model for graph mining. In *SIAM International Conference on Data Mining*, 2004.

[17] Yu Chen and Gordon M. Crippen. A novel approach to structural alignment using realistic structural and environmental information. *Protein Science*, 14(12):2935–46, 2005.

[18] Norishige Chiba and Takao Nishizeki. Arboricity and subgraph listing algorithms. *SIAM Journal on Computing*, 14(1):210–223, 1985.

[19] Ilka Czech, Alexey Silakov, Wolfgang Lubitz, and Thomas Happe. The [fefe]-hydrogenase maturase hydf from clostridium acetobutylicum contains a co and cn- ligated iron cofactor. *FEBS Letters*, 584(3):638–642, 2010.

[20] G. B. Davis and K. M. Carley. Clearing the fog: Fuzzy, overlapping groups for social networks. *Social Networks*, 30(3):201–212, 2008.

[21] Meghana Deodhar and Joydeep Ghosh. Consensus clustering for detection of overlapping clusters in microarray data. In *ICDMW '06: Proceedings of the Sixth IEEE International Conference on Data Mining - Workshops*, pages 104–108, Washington, DC, USA, 2006. IEEE Computer Society.

[22] J. F. Donges, Y. Zou, N. Marwan, and J. Kurths. The backbone of the climate network. *EPL (Europhysics Letters)*, 87(4):48007 (6pp), 2009.

[23] J. F. Donges, Y. Zou, N. Marwan, and J. Kurths. Complex networks in climate dynamics. *European Physical Journal-Special Topics*, 174:157–179, 2009.

[24] Lisheng Du, Fran Stejskal, and Karl H. Tibelius. Characterization of two genes (hupd and hupe) required for hydrogenase activity in azotobacter chroococcum. *FEMS Microbiology Letters*, 96(1):93–102, 1992.

[25] Logan Everett, Li-San Wang, and Sridhar Hannenhalli. Dense subgraph computation via stochastic search: application to detect transcriptional modules. *Bioinformatics*, 22(14):e117–123, 2006.

[26] Jason Flannick, Antal Novak, Balaji S. Srinivasan, Harley H. McAdams, and Serafim Batzoglou. Graemlin: General and robust alignment of multiple large interaction networks. *Genome Research*, 16(9):1169–1181, September 2006.

[27] Wei Gao and Kam-Fai Wong. Natural document clustering by clique percolation in random graphs. In *Information Retrieval Technology*, pages 119–131. Springer, 2006.

[28] Anne-Claude Gavin, Patrick Aloy, Paola Grandi, Roland Krause, Markus Boesche, Martina Marzioch, Christina Rau, Lars Juhl Jensen, Sonja Bastuck, Birgit Dumpelfeld, Angela Edelmann, Marie-Anne Heurtier, Verena Hoffman, Christian Hoefert, Karin Klein, Manuela Hudak, Anne-Marie Michon, Malgorzata Schelder, Markus Schirle, Marita Remor, Tatjana Rudi, Sean Hooper, Andreas Bauer, Tewis Bouwmeester, Georg Casari,

Gerard Drewes, Gitte Neubauer, Jens M. Rick, Bernhard Kuster, Peer Bork, Robert B. Russell, and Giulio Superti-Furga. Proteome survey reveals modularity of the yeast cell machinery. *Nature*, 440(7084):631–636, 2006.

[29] A. Gozolchiani, K. Yamasaki, O. Gazit, and S. Havlin. Pattern of climate network blinking links follows el niño events. *EPL (Europhysics Letters)*, 83(2):28005 (5pp), 2008.

[30] Helen M. Grindley, Peter J. Artymiuk, David W. Rice, and Peter Willett. Identification of tertiary structure resemblance in proteins using a maximal common subgraph isomorphism algorithm. *Journal of Molecular Biology*, 229(3):707–721, 1993.

[31] Olivier Guerrini, Benedicte Burlat, Christophe Leger, Bruno Guigliarelli, Philippe Soucaille, and Laurence Girbal. Chracterization of two 2[4fe4s] ferredoxins from clostridium acetobutylicum. *Curr Microbiol*, 56:261–267, 2008.

[32] Eric Harley, Anthony Bonner, and Nathan Goodman. Uniform integration of genome mapping data using intersection graphs. *Bioinformatics*, 17(6):487–494, 2001.

[33] K. A. Heller and Z. Ghahramani. A nonparametric bayesian approach to modeling overlapping clusters. In *Eleventh International Conference on Artificial Intelligence and Statistics (AISTATS-2007)*, 2007.

[34] William Hendrix, Andrea M. Rocha, Mark Elmore, Joseph Trien, and Nagiza F. Samatova. Discovery of enriched biological motifs using knowledge priors with application to biohydrogen production. In *BIOCOMP 2010*, volume 1, pages 17–23, 2010.

[35] William Hendrix, Matthew C. Schmidt, Paul Breimyer, and Nagiza F. Samatova. On perturbation theory and an algorithm for maximal clique enumeration in uncertain and noisy graphs. In *U '09: Proceedings of the 1st ACM SIGKDD Workshop on Knowledge Discovery from Uncertain Data*, pages 48–56, 2010.

[36] William Hendrix, Matthew C. Schmidt, Paul Breimyer, and Nagiza F. Samatova. Theoretical underpinnings for maximal clique enumeration on perturbed graphs. *Theoretical Computer Science*, 411(28-28):2520–2536, 2010.

[37] Monika Rauch Henzinger and Valerie King. Randomized dynamic graph algorithms with polylogarithmic time per operation. In *Proceedings of the 25th annual ACM symposium on the Theory of Computing*, pages 519–527, 1995.

[38] F. Hoppner, R. Kruse, F. Klawonn, and T. Runkler. *Fuzzy Cluster Analysis*. John Wiley and Son Ltd., Chichester, 1999.

[39] A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: a review. *ACM Comput. Surv.*, 31(3):264–323, September 1999.

[40] Lars J. Jensen, Michael Kuhn, Manuel Stark, Samuel Chaffron, Chris Creevey, Jean Muller, Tobias Doerks, Philippe Julien, Alexander Roth, Milan Simonovic, Peer Bork, and Christian von Mering. String 8–a global view on proteins and their functional interactions in 630 organisms. *Nucleic Acids Research*, 37(suppl_1 (Database Issue)):D412–416, 2009.

[41] David S. Johnson, Mihalis Yannakakis, and Christos H. Papadimitriou. On generating all maximal independent sets. *Information Processing Letters*, 27(3):119–123, 1988.

[42] H. C. Johnston. Cliques of a graph: Variations on the bron-kerbosch algorithm. *International Journal of Parallel Programming*, 5(3):209–238, 1976.

[43] Paul W. King, Matthew C. Posewitz, Maria L. Ghirardi, and Michael Seibert. Functional studies of [fefe] hydrogenase maturation in an escherichia coli biosynthetic system. *J. Bacteriol.*, 188(6):2163–2172, 2006.

[44] Paul D. W. Kirk and Michael P. H. Stumpf. Gaussian process regression bootstrapping: exploring the effects of uncertainty in time course data. *Bioinformatics*, 25(10):1300–1306, 2009.

[45] R. Krishnapuram, A. Joshi, O. Nasraoui, and L. Yi. Low-complexity fuzzy relational clustering algorithms for web mining. *Fuzzy Systems, IEEE Transactions on*, 9(4):595–607, Aug 2001.

[46] Vipin Kumar, Ananth Y. Grama, and Nageshwara Rao Vempaty. Scalable load balancing techniques for parallel computers. *Journal of Parallel and Distributed Computing*, 22(1):60–79, 1994.

[47] Zvi Lotker, Boaz Patt-Shamir, and Adi Rosén. Distributed approximate matching. In *Proceedings of the 26th annual ACM Symposium on Principles of Distributed Computing*, pages 167–174, 2007.

[48] E. Loukakis and C. Tsouros. A depth first search algorithm to generate the family of maximal independent sets of a graph lexicographically. *Computing*, 27(4):349–366, 1981.

[49] Kazuhisa Makino and Takeaki Uno. New algorithms for enumerating all maximal cliques. In *Algorithm Theory – SWAT 2004*, pages 260–272. Springer, 2004.

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

[51] Kaushik Nath and Debabrata Das. Improvement of fermentative hydrogen production: various approaches. *Applied Microbiology and Biotechnology*, 65(5):520–529, 2004.

[52] M. E. J. Newman. The structure of scientific collaboration networks. *Proceedings of the National Academy of Sciences of the United States of America*, 98(2):404–409, 2001.

[53] M. E. J. Newman. The structure and function of complex networks. *Siam Review*, 45(2):167–256, 2003.

[54] M. E. J. Newman. Modularity and community structure in networks. *Proceedings of the National Academy of Sciences of the United States of America*, 103(23):8577–8582, 2006.

[55] V. Olman, F. L. Mao, H. W. Wu, and Y. Xu. Parallel clustering algorithm for large data sets with applications in bioinformatics. *IEEE-ACM Transactions on Computational Biology and Bioinformatics*, 6(2):344–352, 2009.

[56] Byung-Hoon Park, Nagiza F Samatova, Tatiana Karpinets, Andrew Jallouk, Scott Molony, Scott Horton, and Steven Arcangeli. Data-driven, data-intensive computing for modelling and analysis of biological networks: application to bioethanol production. In *SciDAC 2007*, volume 78, Boston, Massachusetts, June 2007.

[57] J. Pei, D. Jiang, and A. Zhang. Mining cross-graph quasi-cliques in gene expression and protein interaction data. In *Proceedings, 21st International Conference on Data Engineering (ICDE 2005)*, pages 353–356, April 2005.

[58] Matthew C. Posewitz, Paul W. King, Sharon L. Smolinski, Liping Zhang, Michael Seibert, and Maria L. Ghirardi. Discovery of Two Novel Radical S-Adenosylmethionine Proteins Required for the Assembly of an Active [Fe] Hydrogenase. *Journal of Biological Chemistry*, 279(24):25711–25720, 2004.

[59] Monika Rauch. Fully dynamic biconnectivity in graphs. In *Proceedings of the 33rd Symposium on Foundations of Computer Science*, pages 50–59, 1992.

[60] S. Redner. How popular is your paper? an empirical study of the citation distribution. *European Physical Journal B*, 4(2):131–134, 1998.

[61] Oleg Rokhlenko, Ydo Wexler, and Zohar Yakhini. Similarities and differences of gene expression in yeast stress conditions. *Bioinformatics*, 23(2):e184–e190, 2007.

[62] R. Rowe, G. Creamer, S. Hershkop, and S. J. Stolfo. Automated social hierarchy detection through email network analysis. In *9th WebKDD and 1st SNA-KDD 2007 workshop on Web mining and social network analysis*, pages 109–117, 2007.

[63] N. F. Samatova, M. Branstetter, A. R. Ganguly, R. Hettich, S. Khan, G. Kora, J. T. Li, X. S. Ma, C. Pan, A. Shoshani, and S. Yoginath. High performance statistical computing with parallel r: applications to biology and climate modelling. *SciDAC 2006: Scientific Discovery Through Advanced Computing*, 46:505–509, 2006.

[64] N. F. Samatova, P. Breimyer, W. Hendrix, M. C. Schmidt, and T. M. Rhyne. An outlook into ultra-scale visualization of large-scale biological data. *Ultra Vis: 2008 Workshop on Ultrascale Visualization*, pages 29–39, 2008.

[65] N. F. Samatova, M. C. Schmidt, W. Hendrix, P. Breimyer, K. Thomas, and B. H. Park. Coupling graph perturbation theory with scalable parallel algorithms for large-scale enumeration of maximal cliques in biological graphs. *Scidac 2008: Scientific Discovery through Advanced Computing*, 125:12053, 2008.

[66] G. Scardoni, M. Petterlini, and C. Laudanna. Analyzing biological network parameters with centiscape. *Bioinformatics*, 25(21):2857–2859, 2009.

[67] Matthew C. Schmidt, Nagiza F. Samatova, Kevin Thomas, and Byung-Hoon Park. A scalable, parallel algorithm for maximal clique enumeration. *Journal of Parallel and Distributed Computing*, 69(4):417–428, April 2009.

[68] E. Segal, A. Battle, and D. Koller. Decomposing gene expression into cellular processes. *Pac Symp Biocomput*, pages 89–100, 2003.

[69] Volker Stix. Finding all maximal cliques in dynamic graphs. *Computational Optimization and Applications*, 27(2):173–186, 2004.

[70] S. H. Strogatz. Exploring complex networks. *Nature*, 410(6825):268–276, 2001.

[71] Helen E. Stutz and Sharon J. Reid. Gltx from clostridium saccharobutylicum ncp262: glutamate synthase or oxidoreductase? *Biochimica et Biophysica Acta (BBA) - Gene Structure and Expression*, 1676(1):71–82, 2004.

[72] David L Tabb, Melissa R Thompson, Gurusahai Khalsa-Moyers, Nathan C VerBerkmoes, and W Hayes McDonald. Ms2grouper: group assessment and synthetic replacement of duplicate proteomic tandem mass spectra. *Journal of the American Society for Mass Spectrometry*, 16(8):1250–61, August 2005.

[73] Kevin Thomas, Nagiza F. Samatova, Matthew C. Schmidt, and Byung-Hoon Park. Efficient scaling up of parallel graph algorithms for Genome-Scale biological problems on cray XT. In *CUG 2008 Proceedings*, 2008.

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

[75] A. A. Tsonis. Introducing networks in climate studies. *Nonlinear Dynamics in Geosciences*, pages 1–15, 2007.

[76] A. A. Tsonis. Dynamical changes in the enso system in the last 11,000 years. *Climate Dynamics*, 33(7–8):1069–1074, 2009.

[77] A. A. Tsonis and P. J. Roebber. The architecture of the climate network. *Physica A-Statistical Mechanics and Its Applications*, 333:497–504, 2004.

[78] A. A. Tsonis and K. L. Swanson. Topology and predictability of el nino and la nina networks. *Physical Review Letters*, 100(22):–, 2008.

[79] A. A. Tsonis, K. L. Swanson, and P. J. Roebber. What do networks have to do with climate? *Bulletin of the American Meteorological Society*, 87(5):585–+, 2006.

[80] A. A. Tsonis, K. L. Swanson, and G. L. Wang. Estimating the clustering coefficient in scale-free networks on lattices with local spatial correlation structure. *Physica A-Statistical Mechanics and Its Applications*, 387(21):5287–5294, 2008.

[81] Shuji Tsukiyama, Mikio Ide, Hiromu Ariyoshi, and Isao Shirakawa. A new algorithm for generating all the maximal independent sets. *SIAM Journal on Computing*, 6(3):505–517, 1977.

[82] Paulette M. Vignais, Bernard Billoud, and Jacques Meyer. Classification and phylogeny of hydrogenases. *FEMS Microbiology Reviews*, 25(4):455–501, 2001.

[83] D. J. Watts and S. H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393(6684):440–442, 1998.

[84] C. Weinstein, W. Campbell, B. Delaney, and G. O'Leary. Modeling and detection techniques for counter-terror social network analysis and intent recognition. *2009 IEEE Aerospace Conference, Vols 1–7*, pages 3361–3375, 2009.

[85] R. Wetzker, C. Zimmermann, and C. Bauckhage. Detecting trends in social bookmarking systems: A del.icio.us endeavor. *International Journal of Data Warehousing and Mining*, 6(1):38–57, 2010.

[86] Y. Xia, H. Y. Yu, R. Jansen, M. Seringhaus, S. Baxter, D. Greenbaum, H. Y. Zhao, and M. Gerstein. Analyzing cellular biochemistry in terms of molecular networks. *Annual Review of Biochemistry*, 73:1051–1087, 2004.

[87] K. Yamasaki, A. Gozolchiani, and S. Havlin. Climate networks around the globe are significantly affected by el nino. *Physical Review Letters*, 100(22):–, 2008.

[88] K. Yamasaki, A. Gozolchiani, and S. Havlin. Climate networks based on phase synchronization analysis track el-nino. *Progress of Theoretical Physics Supplement*, (179):178–188, 2009.

[89] Y. Yan, C. Wang, A. Y. Zhou, W. N. Qian, L. Ma, and Y. Pan. Efficient indices using graph partitioning in rdf triple stores. *ICDE: 2009 IEEE 25th International Conference on Data Engineering, Vols 1–3*, pages 1263–1266, 2009.

[90] Zhiping Zeng, Jianyong Wang, Lizhu Zhou, and George Karypis. Out-of-core coherent closed quasi-clique mining from large dense graph databases. *ACM Trans. Database Syst.*, 32(2):13, 2007.

[91] Bing Zhang, Byung-Hoon Park, Tatiana Karpinets, and Nagiza F Samatova. From pull-down data to protein interaction networks and complexes with biological relevance. *Bioinformatics (Oxford, England)*, 24(7):979–86, 2008.

[92] S. H. Zhang, G. X. Jin, X. S. Zhang, and L. N. Chen. Discovering functions and revealing mechanisms at molecular level from biological networks. *Proteomics*, 7(16):2856–2869, 2007.

[93] Yuzhou Zhang, Jianyong Wang, Zhiping Zeng, and Lizhu Zhou. Parallel mining of closed quasi-cliques. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–10, April 2008.