8-2010

The Maximum Clique Problem: Algorithms, Applications, and Implementations

John David Eblen
jeblen@utk.edu

To the Graduate Council:

I am submitting herewith a dissertation written by John David Eblen entitled "The Maximum Clique Problem: Algorithms, Applications, and Implementations." I have examined the final electronic copy of this dissertation for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Doctor of Philosophy, with a major in Computer Science.

Michael A. Langston, Major Professor

We have read this dissertation
and recommend its acceptance:

Michael W. Berry

Lynne E. Parker

Arnold M. Saxton

Accepted for the Council:

Carolyn R. Hodges

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

The Maximum Clique Problem: Algorithms, Applications, and Implementations

A Dissertation

Presented for the

Doctor of Philosophy

Degree

The University of Tennessee, Knoxville

John David Eblen

August 2010

Acknowledgements

When I began the Ph.D. program seven years ago, I had no idea how long the road would be, nor that there would be so many valuable lessons along the way. My initial aim was to learn more about computer science, a field that continues to fascinate me just as much as it did seven years ago. I ended up learning about computer science and so much more. I wish to thank my advisor, Dr. Michael A. Langston, for being the patient author of many of these lessons, lessons about being a good scientist, about working hard, and about life itself. I wish to thank the many capable fellow graduate students with which I've had the privilege to work. The ones who remain are at this moment working on many exciting projects, some of which are extending the work presented here. I look forward to seeing the results in the months and years to come. Dr. Langston has no shortage of vibrant and intelligent students. I thank all those who served on my committee, Dr. Michael W. Berry, Dr. Arnold M. Saxton, Dr. Lynne E. Parker, and Dr. Bradley T. Vander Zanden, for their support and insights. I want to thank Dr. Parker for agreeing to serve on short notice when I had scheduling conflicts while setting the time for my defense. I also thank Dr. Ivan C. Gerling and Dr. Wendy J. Myrvold for some interesting and fruitful collaborations. I thank my family and friends for their love and encouragement through the years. Most importantly, I thank God for blessing me with the opportunity to pursue my interests and for walking with me every step of the way, even when I didn't realize it.

Abstract

Computationally hard problems are routinely encountered during the course of solving practical problems. This is commonly dealt with by settling for less than optimal solutions, through the use of heuristics or approximation algorithms. This dissertation examines the alternate possibility of solving such problems exactly, through a detailed study of one particular problem, the maximum clique problem. It discusses algorithms, implementations, and the application of maximum clique results to real-world problems. First, the theoretical roots of the algorithmic method employed are discussed. Then a practical approach is described, which separates out important algorithmic decisions so that the algorithm can be easily tuned for different types of input data. This general and modifiable approach is also meant as a tool for research so that different strategies can easily be tried for different situations. Next, a specific implementation is described. The program is tuned, by use of experiments, to work best for two different graph types, real-world biological data and a suite of synthetic graphs. A parallel implementation is then briefly discussed and tested. After considering implementation, an example of applying these clique-finding tools to a specific case of real-world biological data is presented. Results are analyzed using both statistical and biological metrics. Then the development of practical algorithms based on clique-finding tools is explored in greater detail. New algorithms are introduced and preliminary experiments are performed. Next, some relaxations of clique are discussed along with the possibility of developing new practical algorithms from these variations. Finally, conclusions and future research directions are given.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In the past decade, the field of fixed-parameter tractability (FPT) has invigorated theoretical computer science, offering new insights and approaches to solving difficult problems. These new techniques are not apparent from the perspective of classical complexity theory. New theoretical results for various problems appear in publications on a regular basis and are usually constructive. In this dissertation, we consider the problem of moving such results from theory to practical implementation. We seek algorithmic solutions that work well for real software on real computers. It is not enough to implement a single approach that works well for a limited set of problems, though. Therefore, we isolate various algorithmic decisions so that they can be adjusted as needed. This also facilitates research, allowing experimentation with different techniques without reimplementing the core algorithms. From this foundation, we then examine algorithms that use these results for computing solutions to problems involving real data, such as clustering genetic data. The key insight is that exact solutions to these problems, which once were considered intractable and thus impractical to attempt, are quite useful once that barrier is eliminated. Dr. Michael Langston of the University of Tennessee and his students have been involved heavily in developing such algorithms. A former student, Dr. Faisal Abu Khzam of

American Lebanese University, did much work in regard to the vertex cover problem [1], a central problem in the field of FPT. His work inspired this research.

The core problem in this dissertation is clique, a parametric dual of vertex cover. Although the clique problem is not technically fixed-parameter tractable, it is solvable in practice by the same algorithmic techniques as the FPT vertex cover problem. More specifically, this research focuses on the problem of maximum clique. That is, for a given graph we wish to find a largest clique. We emphasize the clique problem because algorithms that locate cliques are especially well-suited for finding groups of related objects, a problem that occurs frequently in practice. Thus, in summary, we translate the vertex cover algorithms into algorithms for finding cliques, creating a base that we can then use to solve some very practical problems.

1.1 Notation and Definitions

Graph theory terms are defined with set theory terminology. The term *set* follows the normal mathematical definition, so that each entry of a set is unique and no ordering is imposed on the members of a set. The size of a set S is denoted as $|S|$. If S' is a subset of a set $S(S' \subseteq S)$, the complement of S' is denoted as $\bar{S}'$. That is, $\bar{S}' = S - S'$.

1.1.1 Basic graph theory notation and definitions

Unless stated otherwise, all graphs are finite, simple, unweighted, and undirected. Specifically, a graph $G = \{V, E\}$ is defined as a set V of *vertices* and a set E of *edges*. Each edge is a set of two vertices from V. If $\{u, v\} \in E$, then u and v are said to be *adjacent*. Otherwise, u and v are *nonadjacent*. The set of vertices adjacent to a vertex, v, is called the *neighborhood* of v and is denoted $N(v)$. The set of vertices nonadjacent to v is the *non-neighborhood* of v, which is denoted $\bar{N}(v)$. A single vertex in one of these two sets is termed a *neighbor* of v or a *non-neighbor* of v, respectively. The *degree* of a vertex is the size of its neighborhood, denoted $|N(v)|$.

A *subgraph* of G is a graph $G' = \{V', E'\}$ where $V' \subseteq V$ and $E' \subseteq E$. For $V' \subseteq V$, let subgraph $G' = \{V', E'\}$, where $E' \subseteq E$ includes exactly those edges $\{u, v\} \in E$ in which $u, v \in V'$. G' is said to be the *subgraph induced* by V'. In this dissertation, all subgraphs are induced unless otherwise noted. The terms *vertex* and *node* frequently are used interchangeably in graph theory. For clarity, this work only uses the term *vertex* for graph elements. The term *node* refers to elements in an algorithmic search tree. A *clique* of a graph G is a set of vertices C in which $\{u, v\} \in C \Rightarrow \{u, v\} \in E$. A *maximum clique* of a graph G is a clique whose size is as large as that of any other clique in G. A *maximal clique* C of a graph G is a clique for which it is not possible to add an additional vertex to C and C remain a clique. An *independent set (IS)* of a graph G is a set of vertices I in which $\{u, v\} \in I \Rightarrow \{u, v\} \notin E$. A *maximum IS* of a graph G is an IS whose size is as large as that of any other IS in G. A *maximal IS* I of a graph G is an IS for which it is not possible to add an additional vertex to I and I remain an IS. A *vertex cover (VC)* of a graph G is a set of vertices C in which $\{u, v\} \in E \Rightarrow u \in C$ or $v \in C$. Intuitively, a VC is a set of vertices that "cover" all edges. A *minimum VC* of a graph G is a VC whose size is as small as that of any other VC in G. The notion of a *minimal VC* is not used. A *matching* for a graph $G = \{V, E\}$ is a set $M \subseteq E$ such that no two edges in M have a common vertex. A *maximal matching* M is a matching for which it is not possible to add an additional edge to M and M still remain a matching. A *coloring* of G is a function $f : V \mapsto \mathbb{N}$ such that $\{u, v\} \in E \Rightarrow f(u) \neq f(v)$. For $v \in V$, $f(v)$ is v's *color*. The *chromatic number* of a graph G is the minimum number of distinct colors over all colorings of G.

1.1.2 Additional notation and definitions

This section introduces additional notation and definitions relevant to this dissertation that may not be found in the common literature on graph theory. For a graph $G = \{V, E\}$, the *common neighborhood* of a set $V' \subseteq V$ is $\bigcap N(v)$ for all vertices

$v \in V'$. It is denoted as $N_\cap(V')$. For a vertex v, define the *neighborhood subgraph* of v to be the subgraph induced by $v \bigcup N(v)$. Neighborhood subgraph is also defined for a set of vertices. For $V' \subseteq V$, the neighborhood subgraph of V' is the subgraph induced by $V' \bigcup N_\cap(V')$.

1.2 Applying Clique to Biological Data

This section briefly outlines how graph-theoretic algorithms, particularly clique, can be applied to the problem of finding relevant networks in biological data. We proceed backwards from an unweighted graph to its biological origins. Generally, graph vertices represent individual genes or another type of molecule, such as proteins. Edges between vertices indicate molecules that are highly correlated. That is, they tend to co-occur in the cell under the same conditions. Thus, a clique should indicate groups of molecules that co-occur, and hence indicates a potential network for further investigation. The strictness of clique (every edge must be present) is advantageous for avoiding false positives in noisy data, at the expense of increased false negatives. In practice, there are techniques, some of which we discuss later, to relax the results and extend the clique to include "nearby" vertices.

The unweighted graph is created by applying a cutoff to a weighted graph, whose edge weights correspond to actual correlation values. Pearson's correlation is a common method for computing these values. The input to such a method is a matrix, gene by condition, giving a signal value for each gene under each condition. Prior to correlation, however, the raw data from experiments must be subject to statistical normalization and verification techniques [54]. This step is crucial. Otherwise, results from a graph-theoretic analysis or any other analysis are worthless.

1.3 Overview

Chapter 2 discusses the theoretical foundation of this work, the field of fixed-parameter tractability (FPT). (The acronym *FPT* stands for both fixed-parameter tractable and fixed-parameter tractability.) In Chapter 3 the maximum clique problem is discussed specifically. The relationship of the clique problem to the FPT vertex cover problem is discussed. Then the overall approach used for solving the maximum clique problem in this work is presented. While Chapter 3 discusses the algorithms independently of any particular implementation, Chapter 4 covers the details of a specific C++ implementation. Then, through experimentation, this implementation is tuned to work well on real-data graphs. Next it is tuned to work well on synthetic graphs. These experiments illustrate how the presented approach can be used to experiment with and develop different algorithms for different classes of graphs. The chapter ends with a description of and experiments with a parallel implementation. Chapter 5 discusses a specific case of using maximum cliques to analyze biological data. Chapter 6 expands on this idea by describing and experimenting with possible approaches for building practical algorithms from clique-finding engines. Chapter 7 concludes and suggests directions for future research.

1.4 Contributions

The specific contributions of this dissertation include the development of a general, configurable algorithm for computing maximum cliques. Key algorithmic decisions are isolated so that they can be easily altered. This includes a general preprocessing algorithm that can be configured to run multiple methods. A configurable, modular, and efficient implementation of these algorithms is created and described in detail. This description includes the high-level design for easy configuration, the low-level design for efficient data structures and for efficient computation of common graph operations, and an effective parallel processing approach. Experiments are performed to show how to configure the software for graphs of real data and for graphs of synthetic data. A new preprocessing approach based on coloring is presented and shown empirically to work well on graphs built from biological data. Analysis of practical algorithms based on clique are done. New practical algorithms are developed and implemented and preliminary testing performed. Finally, several directions for future research are proposed.

Chapter 2

FPT Overview

This brief chapter serves as an introduction to FPT for those unfamiliar with the field. More extensive introductions can be found in the references. Chapter 1 of [14] provides a more colorful and lengthier discussion of the intuition behind FPT, relating it to a "deal with the devil." It then covers the theory behind FPT in excruciating detail, serving as an essential reference for those heavily involved in FPT theory. Conversely, [34] focuses on specific algorithmic techniques and contains several examples, thus serving as a good first book on FPT and as a reference book for FPT practitioners. We first address the main ideas behind FPT and then summarize its theoretical underpinnings. We also discuss new algorithmic techniques that stem from FPT.

2.1 Intuition

Classical complexity theory identifies a set of particularly difficult problems known as $\mathcal{NP}$-hard. Assuming certain time-tested theoretical assumptions and practical observations [21], any solution (algorithm) for such problems must have exponential running time with respect to the size of the input. Intuitively, this means that certain inputs force the algorithm to check all or nearly all possible combinations of items (no clever shortcuts exist). Unfortunately, many NP-hard problems occur in practice, so we cannot simply ignore them. The field of FPT probes more deeply by asking what, exactly, causes such a "combinatorial explosion." The term *fixed-parameter tractable* is fitting. If the source of the explosion, one or more *parameters*, can be contained, or *fixed*, then the problem becomes *tractable*. For example, vertex cover is FPT because algorithms exist where the exponential is isolated to the size of the cover. Thus, if we are only interested in small covers, efficient algorithms exist. The field of FPT, though, is about more than just rearranging parameters by trying different algorithms. It also provides a wide range of techniques to reduce problem instances and to narrow problem search spaces by employing one or more isolated parameters. Problems can often be parameterized in multiple ways. So a generally useful strategy is to craft an algorithm that exploits whatever constraints can be placed on the problem domain. (Of course, this is a useful strategy in general, not just in the context of FPT.)

2.2 Theoretical Background

Classical complexity theory has been developed for decision problems. That is, given an alphabet Σ and the set of all its finite strings, Σ^*, along with a language $L \subseteq \Sigma^*$, we consider algorithms that input some $x \in \Sigma^*$ and decide if $x \in L$. For FPT, we consider algorithms that input x and some parameter k. They must still decide if $x \in L$ but do not need to consider solutions where the parameter is larger than k. If this conditional membership can be correctly decided by some algorithm in time $f(k)n^{\alpha}$, where n is the size of x, that language (problem) is said to be FPT. Thus, k is free to affect the complexity of the problem in any way, as long as the size of the problem contributes in a limited way and is not affected by k. Note that it is trivial to extend this definition to multiple parameters. For this definition to be practically useful we rely on the hope, supported through experience, that neither $f(k)$ nor α become prohibitively large for problems encountered in practice. This is analogous to the same hope in classical complexity that practical problems solvable in polynomial time do not contain polynomial time complexities with prohibitively large exponents.

The above definition of FPT is sufficient for most cases, including those in this work, but see [14] for a more general definition.

2.3 Algorithmic Approaches

Of most importance to this work, attempts to make use of the parameter k have led to a plethora of new methods for tackling difficult problems, which can often be divided into two categories. The first category is *kernelization*, or a method that reduces the problem to some size that is a function of k. The second category is simply an algorithm with search space restricted by k. Theoretically, these are equivalent since a *kernel* (the output of a kernelization algorithm) can be searched by brute force in time bounded by k. Usually, these two techniques are combined and even interleaved, something that will be discussed in much more detail later. The algorithm normally takes the form of a *branching* search tree. Thus, we simply refer to kernelization and branching. In the next chapter, we discuss specific algorithms for the vertex cover problem and the clique problem. Finally, note that we often use the term *preprocessing* rather than kernelization for the clique problem, since it is unlikely to be FPT and thus cannot technically be kernelized. (It is W[1]-hard in FPT terminology.)

2.4 Example - Vertex Cover

This section presents a kernelization algorithm and a branching algorithm for the vertex cover problem, which exploit cases where we are only interested in a small cover of size k or less. The goal is two-fold. Not only does this serve as a clarifying example, but these algorithms will be modified later to solve for the maximum clique problem. To be concise, from this point forward we omit the word "problem" when referring to specific problems. Thus, *clique, independent set, vertex cover,* etc. refer to their respective decision problems, while non-decision variants are still qualified with *minimum, maximal, maximum,* etc.

2.4.1 Kernelization

We can kernelize vertex cover by applying a high degree rule that is based on a simple observation. If a vertex v has degree $k+1$ or more, then it must reside in any vertex cover of size k or less. To see why, note that all edges containing v must be covered. This can be done by either including v in the cover or by including all of v's neighbors in the cover. The latter, though, would imply a cover that is too large. To this we can add two other kernelization rules. First, remove isolated vertices. Second, remove any vertex v of degree one (not in the cover), while placing its single neighbor w into the cover. The latter is possible because we must cover the edge $\{v, w\}$ and, assuming we are looking for only a single vertex cover, there is no reason to select v instead of w. These rules are known as the *0-degree rule* and the *1-degree rule*, respectively. Combining these three rules reduces the graph to at most $k^2 + k$ vertices and at most k^2 edges [34].

2.4.2 Branching

We can build a binary search tree of bounded depth k by another simple observation. Given any vertex v, either v is in the cover or all of v's neighbors are in the cover. Each node of the search tree selects a vertex and branches on these two possibilities. Assuming that the tree also removes isolated vertices as they occur, each child node inherits a new vertex cover problem with parameter k reduced by at least one. Hence, the depth of the search tree is bounded by k and the total search space is bounded by 2^k.

2.4.3 Optimization

We have so far outlined only the "bare bones" of the algorithms for vertex cover. A number of optimizations are usually done in practice. The most important one, perhaps, is interleaving kernelization with the branching search tree. Each search tree node by itself is a new vertex cover instance subject to being reduced. In general, computationally expensive kernelization should be avoided, but some interleaving is essential, such as the removal of isolated and single degree vertices. Graph instances become more and more sparse as searching proceeds and low degree vertices are quite common. Thus, such removal serves as a "garbage collector." Additionally, the order in which vertices are considered greatly impacts performance. Such optimizations will be discussed at length.

Chapter 3

Maximum Clique Algorithms

Although not likely to be FPT, clique is subject to the algorithmic approaches discussed in Chapter 2 due to its close relationship with vertex cover. In this chapter, we build a practical, adjustable algorithm for maximum clique. We start by discussing the theoretical relationship between clique and vertex cover. Then we translate the kernelization and branching algorithms for vertex cover into preprocessing and branching algorithms for clique. Additional useful preprocessing rules are also discussed. We introduce a general algorithm for preprocessing that supports multiple specific algorithms, and we make other modifications to the base algorithms to suit our needs. Throughout, we are primarily concerned with enumerating various algorithmic details that can be adjusted, which is crucial for our goal of applying algorithms in practice. (We need to be able to alter the specific strategy easily for different types of input.) We finish by tying preprocessing and branching together to create a complete algorithm.

3.1 Relationship of Clique to Vertex Cover

Translation of an instance of clique to an instance of vertex cover can perhaps best be understood by first converting the clique instance to an independent set instance, which is then converted to a vertex cover instance. We discuss the latter conversion first. Given a graph G and an independent set I, $\bar{I}$ is a vertex cover of G. This can be seen by supposing it were not a vertex cover. Then there is some edge $\{u, v\} \in G$ such that neither u nor v are in $\bar{I}$. Thus, both are in I, and I could not be an independent set. To convert an instance of clique to an instance of independent set, we merely complement G. To summarize then, we can convert a clique instance to a vertex cover instance as follows. Given an instance of clique to solve, $\{G, k\}$, and an algorithm for vertex cover, A, inputting $\{\bar{G}, |G| - k\}$ to A should return a vertex cover set C or return that no solution exists. In the latter case, no solution for clique exists either. In the former case, $\bar{C}$ is a solution to clique. It can be proven that this relationship extends to the optimization versions of the respective problems. That is, given an algorithm for finding a minimum vertex cover, one can apply it to find a maximum clique. Thus, one way to solve maximum clique is to implement minimum vertex cover and convert maximum clique instances. Since we frequently want to compute maximum cliques in practice, though, we often instead convert the vertex cover FPT algorithms to solve clique directly.

3.2 Preprocessing

For clique, the 0-degree rule and the 1-degree rule discussed in Chapter 2 are analogous to an *$(n-1)$-degree rule* and an *$(n-2)$-degree rule*, respectively. The $(n-1)$-degree rule automatically places any vertex adjacent to all other vertices into the clique. The $(n-2)$-degree rule places any vertex v adjacent to all but one other vertex u into the clique, while excluding u. To see this, note that any clique not containing u can be extended to contain v. Any clique containing u, meanwhile, can have u replaced with v. Since v is adjacent to all other vertices, there is no advantage to selecting u. Finally, the high degree rule from vertex cover is analogous to a low degree rule for clique. That is, any vertex of degree less than $k-1$ cannot be contained in any clique of size k.

In this section, we describe two more preprocessing techniques for clique. The first takes advantage of vertex degrees beyond individual degree counts. The second applies coloring to subgraphs and exploits the fact that cliques have strict coloring requirements (in any proper coloring, every vertex must be a different color). Finally, we discuss algorithmic strategies for preprocessing that are important for practical implementation. We postpone discussing interleaved preprocessing until the section on branching.

3.2.1 Common-neighbor preprocessing

Common-neighbor preprocessing (CNP) is a more thorough version of the low degree rule. It examines the "degree" (common neighborhood size) of pairs of vertices. If the common neighborhood of a pair of adjacent vertices $\{u, v\}$ in graph G has size less than $k - 2$, it is not possible for both vertices to be in a clique of size k. Hence, the edge between them can be deleted. A combination of CNP and the low degree rule serves as a more effective preprocessor than the low degree rule alone, although linear time is replaced with quadratic time. Computing of common neighbors is often useful for kernelizing graph problems, such as cluster editing [12].

3.2.2 Color preprocessing

Color preprocessing is similar to the low degree rule but uses the notion of the *color degree* of a vertex as opposed to its degree. The *color degree* of a vertex v in a graph G is defined as the chromatic number of v's neighborhood subgraph. Observe that a graph containing a clique of size k must have a chromatic number k or higher, since the clique itself requires k colors. Also observe that if v is in a clique of size k, the clique itself must be contained in v's neighborhood subgraph. Thus, if the color degree of v's neighborhood subgraph is less than k, v is not in a clique of size k.

Color preprocessing computes an upper bound u on the color degree of each vertex in G, removing those vertices in which $u < k$. We compute upper bounds rather than exact chromatic numbers in practice due to the computational complexity of coloring. Graph coloring is used in [47] to solve maximum clique but is employed for branching rather than for preprocessing. We later discuss and experiment with this branching strategy.

3.2.3 Preprocessing strategies

In this section, we outline some algorithmic decisions related to preprocessing. First, it can sometimes be beneficial to repeat preprocessing, which requires the algorithm to decide when to halt. An obvious criterion would be to halt once no additional vertices or edges are deleted, but this usually leads to several time consuming iterations with little gain. Instead, we define a stopping criterion consisting of two parameters. The first parameter is either *vertices* or *edges*, specifying that vertices or edges, respectively, should be used in computing the criterion. The second parameter is the fraction of vertices or edges that must be removed in order to repeat preprocessing. There are, of course, other ways of defining a stopping criterion, and a combination of vertices removed and edges removed is possible, but we ignore these in this work.

Another important algorithmic decision is the order in which vertices are investigated. For example, for the low degree rule, we often find that iterating through the vertices in lowest to highest degree order reduces the graph more per repetition than an arbitrary ordering. This is because low degree vertices are eliminated early, reducing the degree of later vertices so that they are more likely to be eliminated in the same pass. Sorting vertices is also an important consideration when branching, and so we discuss it in more detail when we discuss branching.

3.3 General Preprocessing

In this section, we introduce a template for clique preprocessing algorithms, termed *general preprocessing (GP)*, in which the low degree rule, CNP, and color preprocessing are specific examples. GP facilitates both research and experimentation into new preprocessing strategies, as it is both a conceptual tool and a practical algorithm. When using GP, we configure a specific algorithm with two parameters: an integer and a function. We delay describing GP in its full generality and instead develop it slowly through examples and basic concepts.

3.3.1 Generalizing specific preprocessing algorithms

Conceptually, GP builds neighborhood subgraphs for all vertex sets of size *depth* (integer parameter), passing each of these neighborhood subgraphs to a *test function* (function parameter). Note that depth is just the set size. The reason for the use of the term *depth* should become apparent later. (It relates to the depth of a search tree.) The low degree rule and CNP share the same simple test function, graph size (number of vertices). However, the low degree rule runs at depth 1 (only single vertices are considered) while CNP runs at depth 2 (pairs of vertices are considered). In GP terminology, the low degree rule is *size preprocessing depth 1* while CNP is *size preprocessing depth 2*. Color preprocessing is *color preprocessing depth 1*. That is, each iteration builds the neighborhood subgraph for a single vertex and runs a test function that uses graph coloring. We could also do this for all vertex pairs, *color preprocessing depth 2*, which, like CNP, would seek to remove edges. In practice, the neighborhood subgraph may not need to be built (the low degree rule can be done by simply computing vertex degrees) and other shortcuts can be taken, such as skipping nonadjacent vertex pairs at depth 2.

3.3.2 Basic concepts

All test functions attempt to prove, with as little effort as possible, that a certain subgraph cannot contain a clique of size k. If successful, then the vertex or edge can be deleted. Note that the test function attempts to solve a clique instance, a smaller piece of the larger clique problem. This recursive property leads to an alternate and more elegant recursive form of the general algorithm that we introduce later. Note that the $(n-1)$-rule and the $(n-2)$-rule do not fall under the umbrella of GP, as they are not readily described as test functions on neighborhood subgraphs.

Before discussing GP more formally, it is helpful to examine the relationship between depths 1 and 2. In practice depth 2 preprocessing also deletes vertices. This can be done in multiple ways, such as running depth 1 before depth 2 or integrating single vertex tests into depth 2 preprocessing. In general, then, given the same test function we expect depth 2 to be more effective than depth 1. This should match intuition, since depth 2 is more thorough ($|G|^2$ iterations versus $|G|$ iterations).

3.3.3 Further examples

An implementation of GP provides a tool for experimentation. Different test functions can be coded and tried without reimplementing the preprocessing search tree. (The final recursive version involves a tree and is more intricate than simply iterating through vertex sets.) We present two more test functions as examples. The first is *match preprocessing*, which finds a maximal matching for the complement graph $\bar{G}$. It can be proven that at most half of the vertices in this matching can occur in a clique. Thus, the size of the matching plus the number of vertices remaining serves as an upper bound on the maximum clique size. The second test function is a variation of color preprocessing. As we will see, color preprocessing is highly effective on real data graphs. Unfortunately, properly coloring a graph, even with a greedy algorithm, is an expensive operation. Luckily, though, a proper coloring of G is also a proper coloring of any subgraph of G. Thus, a reasonable strategy might be to color G once and have a test function that simply counts the number of unique colors for each given subgraph. We term this approach *single-color preprocessing* or *s-color preprocessing* for brevity.

```
bool preproc(graph, minimum_clique_size,
                            depth, test_function) {
  if (test_function(graph, minimum_clique_size)) return 1
  if (depth==0) return 0

  For each vertex v in graph {
    Copy graph to nbrhood_subgraph
    Remove non-nbrs. of v from nbrhood_subgraph excluding v
    if (preproc(nbrhood_subgraph, minimum_clique_size,
                depth-1, test_function))   Remove v from graph
    else (Remove edges from graph between v and vertices
          no longer in nbrhood_subgraph)
  }

  return 0
}
```

Figure 3.1: GP Recursive Form in Pidgin C++

3.3.4 A recursive formulation of GP

Figure 3.1 shows the recursive form of GP in pidgin C++. Statements that are mostly English rather than C++ begin with a capital letter. The depth value equates to the depth of the recursion. Thus, depth values can be any nonnegative integer not greater than k. At depth 0, the test is run once on the whole graph. Note that each run of *preproc*, in isolation, is capable of deleting both vertices and edges for its particular graph instance. The *preproc* function returns 1 if the graph cannot contain a clique of the given size. Otherwise, it returns a preprocessed graph instance by modifying the passed graph (first parameter). Note the way that these return values are used after the recursive call to *preproc*. If a 1 is returned, vertex v can be removed. If not, it is still possible to remove some edges if some vertices were deleted from *nbrhood_subgraph*. At global scope, vertex and edge deletions occurring lower in the recursion at depths greater than 2 can "bubble up" and reduce the graph more than might occur for depth 2 only.

3.3.5 Notes on recursive GP implementation

Since one of the main focuses of this work is the execution of algorithms in practice, we address implementation issues where appropriate. The pseudocode for recursive GP is streamlined for human understanding but not quite suitable for software. Copying the entire graph at each iteration is usually too time consuming and may require too much memory for even modest depths. The implementation in this work has a single global copy of the entire graph, and only a set of vertices is passed to *preproc*. Edges are deleted from the global copy but are restored before *preproc* returns. Various optimizations are also possible, such as sorting vertices before iteration.

3.4 Branching

As with the preprocessing algorithms, we can construct a branching algorithm for clique from the vertex cover branching algorithm. Each node of the analogous clique binary search tree selects a vertex v and branches on two possibilities. First, assume v is not in the clique and remove it from the graph. Second, place v in the clique and remove v and all non-neighbors of v from the graph. When finding a clique of size k in a graph of size n, the number of nodes in this search tree is bounded above by 2^{n-k}. To see this, we show that the quantity $n - k$ decreases by at least one at each branch. (Note the parallels between this section and the discussion of vertex cover branching in Section 2.4.2.) First assume that we filter vertices connected to all other vertices, similar to the filtering of isolated vertices in vertex cover branching. The branch that supposes v is not in the clique decrements n and thus $n - k$. The branch that supposes v is in the clique decrements k by one but decrements n by at least two due to the removal of v and v's non-neighbors.

3.4.1 Alternate branching for maximum clique

In this work, we search exhaustively for a maximum clique, which differs from a decision search tree that halts once a clique of size k is found. One way to solve optimization problems is to perform a binary search by invoking a decision algorithm multiple times with different k values. However, for maximum clique a simple alteration to branching avoids the need to restart for different k values. That is, once a clique C of size k or greater is found, store this clique, set $k = |C| + 1$, and continue searching with the current search tree. Upon completion, the stored clique will be a maximum clique. We must be careful, however. Preprocessing, whether prior to branching or interleaved, may assume that we only wish to find a single clique and may remove vertices in other cliques. (This is why the problem of clique enumeration is not a trivial extension of these algorithms.) Fortunately, of the rules we have examined, only the $(n-1)$-degree rule and $(n-2)$-degree rule make such assumptions. Both of these rules are independent of the parameter k and select vertices that maximize the found clique, so they will leave at least one maximum clique intact.

3.4.2 Vertex sorting

Sorting of vertices is a key algorithmic decision for both preprocessing and branching. It concerns the order in which vertices are iterated through in the main "for each" loop of recursive GP in Figure 3.1 and also in each node of the branching search tree as implemented in Section 3.4.1. Sorting actually consists of two parts, when to sort and how to sort. For the former, we consider two possibilities termed *lazy* and *active*. Lazy sorting sorts vertices only once, before the loop starts. Active sorting sorts vertices at the beginning of every iteration through the loop. Concerning how to sort vertices, two possibilities that have already been mentioned are sorting by vertex degrees or by vertex colors after applying a proper coloring.

3.4.3 Interleaved preprocessing

As with vertex cover, preprocessing for clique can be interleaved with branching to reduce the subgraph at each node. For example, applying the $(n-1)$-degree rule and the $(n-2)$-degree rule is essential, just as the analogous 0-degree rule and 1-degree rule "garbage collector" is essential for vertex cover. (Subgraphs tend to become more dense, rather than more sparse as in vertex cover, as we descend the clique search tree.) Any preprocessing can be applied, but it is usually less compute-intensive than preprocessing done prior to branching. Similar to the procedure for sorting vertices, we can alter not only the type of preprocessing but when it is applied. Thus, we have *lazily-applied interleaved preprocessing* and *actively-applied interleaved preprocessing*, where the former is applied before the loop starts and the latter is applied at the start of every iteration. In general, multiple preprocessing algorithms can be interleaved and can even be applied at different frequencies. We can also specify an upper bound on the search tree depth at which a preprocessing algorithm is applied. Limiting the depth of application may help because branching alone is often quicker than preprocessing and branching on the smaller graphs that occur in the lower nodes of the search tree.

3.4.4 ***In-clique*** versus ***not-in-clique*** branching

During the main loop of each node of the branching search tree described in Section 3.4.1, after selecting a vertex v, we can either assume v is in the clique (*in-clique* branching) or assume v is not in the clique (*not-in-clique* branching). As we will see during experimentation, altering this decision often dramatically affects performance, but the best choice is not obvious.

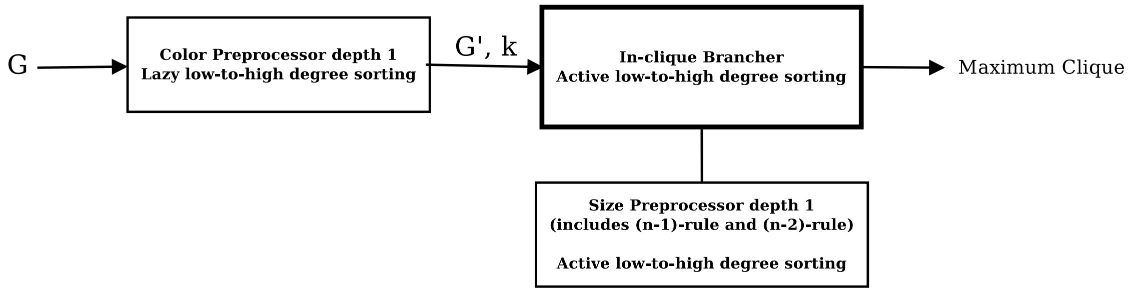

G
Color Preprocessor depth 1
Lazy low-to-high degree sorting
G', k
In-clique Brancher
Active low-to-high degree sorting
Maximum Clique
Size Preprocessor depth 1
(includes (n-1)-rule and (n-2)-rule)
Active low-to-high degree sorting

Figure 3.2: Sample Algorithmic Decision Chart

3.5 The Maximum Clique Algorithm

We now build a complete algorithm for finding a maximum clique in a graph using the tools we have discussed. The algorithm inputs a graph G. The only mandatory component is a branching algorithm. Thus, we can think of it as the main component, which we augment with various preprocessing algorithms. Before invoking branching, we can interject a preprocessing algorithm that produces a revised graph G' and a k value that serves as a lower bound on the clique size. We can also attach various interleaved preprocessing schemes to the branching algorithm. To summarize a specific algorithm, we introduce the notion of an algorithmic decision chart. Figure 3.2 is one such chart illustrating one particular strategy. Each box represents either a branching algorithm or a preprocessing algorithm (either preceding branching or interleaved with branching). The text in each box outlines its particular strategy. Later, we produce empirical evidence that the strategy shown works well for graphs built from real-world data (specifically graphs produced by correlating genetic data). For brevity the chart omits stopping criteria and the frequency of applying a particular interleaved preprocessing algorithm. Unless mentioned otherwise, the following apply. The stopping criterion for preprocessing before branching is to halt when less than 10% of vertices are removed. For interleaved preprocessing, only a single iteration is done upon each invocation. Lastly, interleaved preprocessing algorithms are applied actively at all depths.

Chapter 4

Software Implementations

The practicality of any algorithm can only be verified through actual implementation and experimentation. Therefore, in this chapter we first describe a C++ implementation of the maximum clique algorithms discussed earlier, called the *Maximum Clique Finder (MCF)*. We then tune MCF to improve its efficiency on graphs built from real data. MCF was designed to support such tuning, especially along the lines discussed previously, so that we can easily experiment with different approaches. We also tune MCF for synthetic data to illustrate this flexibility.

4.1 Software Design

The MCF software architecture, as diagrammed in Figure 4.1, mirrors the algorithms outlined previously. The box labeled "Maximum Clique Finder" represents the highest-level functions, which handle program entry and exit and also configure and execute a specific maximum clique algorithm. The other boxes represent C++ classes, except for "Bit Counters," which is a collection of routines for quick counting of one bits. The classes above the dashed line, unlike most C++ classes, primarily encapsulate functionality rather than data. Thus, they can be thought of as highly-configurable functions. The three boxes below the dashed line form the low-level representation of graphs and handle routine graph operations. The "Bit Library"

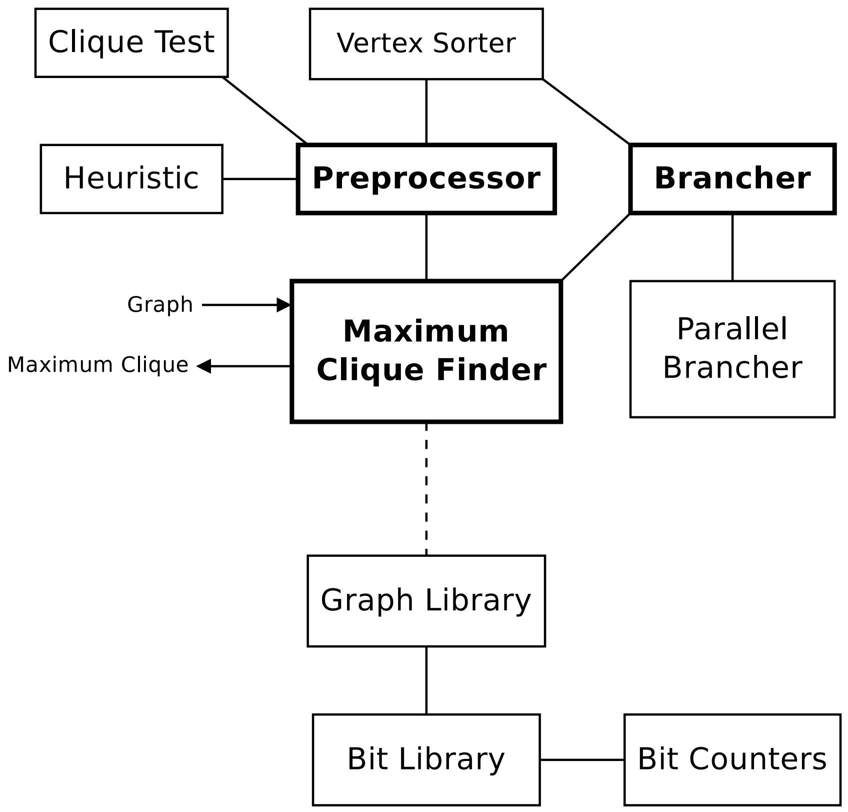


Figure 4.1: MCF Software Architecture

stores arrays of bits in a way that conserves memory and that allows for fast bit manipulation. These arrays are used by the graph representation to store graph edges. The "Preprocessor" and "Brancher" boxes encapsulate the high-level algorithms discussed in Section 3.5. The preprocessor class implements the recursive form of GP from Figure 3.1. The brancher class implements branching as described in Section 3.4.1. The heuristic class encapsulates algorithms that quickly find large cliques (necessary to find an initial k value for preprocessing) and can thus be considered part of preprocessing (conceptually if not technically). The clique-test class encapsulates test functions that can be input to GP. (Recall that a test function is one of the two parameters to GP.) The vertex-sorting class encapsulates strategies for vertex sorting as discussed in Section 3.4.2 and is used in both preprocessing and branching. Interleaved preprocessing, described in Section 3.4.3, is handled by functionality in the brancher class that allows the user to insert into the branching tree a specific preprocessing algorithm. Finally, “Parallel Brancher” is a subclass of “Brancher” that adds parallel capabilities.

4.2 Code Details

In this section, we examine in detail how MCF is implemented to be both highly-configurable and efficient. For the former, we need a flexible design that allows us to change not only parameters but specific sub-algorithms, which can be developed as separate modules that are independent of the main system. For efficiency, the underlying graph library must store the graph compactly and execute common graph operations quickly, taking advantage of low-level bit arithmetic operations known to be fast on most processors.

4.2.1 MCF main operation

Figure 4.2 is pseudocode for the main function of MCF (the “Maximum Clique Finder” box in Figure 4.1), which illustrates how the different classes from Figure 4.1 interact. MCF relies heavily on the strategy design pattern, as discussed in [20] and discussed particularly for C++ in item 35 of [33]. This design pattern, as applied to C++ classes, grants the user the ability to specify and/or alter a class’s internal algorithms. It is well-suited for cases like MCF in which we need the ability to change various supporting algorithms, such as clique testing and vertex selection, without disrupting the core algorithms, such as recursive GP and the branching search tree. In this paradigm, it is common to create functions (actually classes that behave like functions) and pass them as arguments. Note that the created functions in Figure 4.2 correspond directly to the classes in Figure 4.1. The “create_” functions are generic names for specific functions that create the desired item. The pseudocode outlines only the most minimal operation of MCF. Preprocessors and branchers also have many configuration options corresponding to the algorithmic decisions discussed earlier.

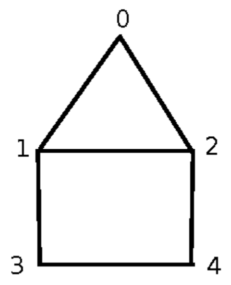
```
Maximum_Clique MCF(graph G) {

  // Heuristic
  Function h = create_heuristic()
  Clique C = h(G)

  // Preprocessing
  Function ct = create_clique_test()
  Function vs = create_vertex_sorter()
  Function p = create_preprocessor(ct, vs, depth (hardcoded))
  Graph G = p(G, |C|)

  // Branching
  Function vs2 = create_vertex_sorter()
  Function b = create_brancher(vs2)
  Clique C = b(G, C)

  return C
}
```

Figure 4.2: Maximum Clique Finder Basic Operation in Pidgin C++

For example, we can alter a preprocessor's stopping criterion. We can implement interleaved preprocessing by creating a preprocessor and inserting it into a brancher, specifying when it is invoked (see Section 3.4.3). Additionally, the $(n-1)$-rule and $(n-2)$-rule can be interleaved automatically and set to be lazily-applied or actively-applied. (Recall from Section 3.3.2 that these rules are outside the GP framework and thus not implementable through the preprocessor class.)

4.2.2 MCF graph library

To be feasible for practical applications, software must use efficient underlying data structures in addition to appropriate algorithms. For MCF the key data structure is the graph library, which provides an interface for common graph operations such as reading graphs from a file, inserting and deleting vertices, computing vertex degrees, and specifying subgraphs. MCF employs a bitwise adjacency matrix, whose entries



$$\begin{matrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 \end{matrix}$$

Figure 4.3: Bitwise adjacency matrix for the "house" graph

indicate whether or not two vertices are adjacent, a "0" meaning "no" and a "1" meaning "yes". Such a setup allows the library to take advantage of low-level bitwise operands, which run in parallel on most hardware platforms. Figure 4.3 shows an example adjacency matrix for a small graph of five vertices. Note that while the matrix is symmetric, and thus half of it is redundant, storing the full matrix allows for faster operations. For example, to compute the degree of a vertex it is only necessary to scan a single row. If only the upper diagonal portion of the matrix is stored, several rows may have to be visited, which are non-contiguous in memory since C++ uses row-major ordering. Furthermore, parallel bitwise operations are not possible with non-contiguous bits.

Now we look at each of the three components below the dashed line in Figure 4.1. The "Bit Counters" component contains a function designed to be fast for the fundamental operation of counting one bits. Given a single array, or row of bits, it employs a lookup table of size $2^{16} = 65536$ that contains the number of ones in any pair of two bytes (16 bits). Dealing with bits and bytes directly can be quite involved. The "Bit Library" encapsulates these details, providing its user with the ability to create an array of bits of any size and use simple function calls to perform common operations. These include counting the number of ones or performing various boolean operations, such as "AND", "OR", and "XOR", on a pair of such arrays. The library handles the details of storing the bits as multiple bytes in an array and quickly performing the operations. As one example, memoization is used when a count of ones is requested. The result is stored so that recounting can be avoided if the array remains unchanged. Finally, the "Graph Library" creates the adjacency matrix using multiple arrays of bits and relying on the bit library as needed. Many common graph operations require only one or two boolean operations on these arrays. For example, computing the common neighbors between two vertices can be done with a single "AND". The graph library allows the user to store induced subgraphs by storing a set of vertices, again represented as an array of bits. Since these do not need to store edge information, they take very little memory and are used to store subgraphs at each node of the branching search tree. One common operation is to remove the non-neighbors of a vertex selected to be in the clique, which again can be done with a single "AND" between the current subgraph and the bit array for the selected vertex.

4.3 Tuning MCF for Real Data Graphs

In this section we configure MCF to process graphs built from biological data by timing various strategies on three representative graphs. These are correlation graphs constructed in the manner discussed in Section 1.2 and represent genes from mouse cerebellum[*], mouse spleen [15], and yeast [22] and are named *CER*, *NOD*, and *YST*, respectively. Table 4.1 gives the dimensions of these three graphs. To reduce the number of different algorithmic combinations, we tune preprocessing first and then tune branching. We will verify that the strategy of Figure 3.2 is better than the others tested. All non-parallel run times in this chapter are the average wall clock time of three runs on an Intel Xeon (3.20 GHz) processor. Runs were restricted to 24 hours. Dedicated compute nodes were used to minimize outside interference.

4.3.1 Preprocessing

We begin by trying different preprocessor types. For these types of graphs vertex degrees vary widely and sorting by degree is well-known as advantageous. The more

[*]Goldowitz Lab at the Centre for Molecular Medicine and Therapeutics, University of British Columbia, Canada

Table 4.1: Graphs Built from Real Data

Graph	Vertices	Edges	Density
CER	21348	972960	0.4%
NOD	22690	7534598	2.9%
YST	6144	6150429	32.6%

interesting question is what type of preprocessing to use, so we will focus solely on this question and leave the sorting method as indicated in Figure 3.2. (For synthetic graphs, we will reconsider the sorting method.) Figure 4.4 summarizes the experimental results. It shows only preprocessing and branching times and does not include additional tasks, such as file input and heuristic searching, that are not relevant to measuring the effectiveness of preprocessing. Branching is relevant, since the aim of preprocessing is to reduce the effort needed by branching and, more broadly, to reduce total processing time of both. (A preprocessor that greatly reduces branching time but increases total time is worthless.) The blue portion of each bar is preprocessing time, while the white portion is branching time. The four test functions mentioned earlier (size, color, s-color, and match) were used and ran at depths 1 and 2. Only matching at depth 2 on the YST graph exceeded 24 hours. Bars that are all blue and extend to the top of the chart indicate methods that took much longer than the other methods. Match 2 preprocessing on the NOD graph took over 13000 seconds, while color 2 on the YST graph took over 38000 seconds. One conclusion that can be drawn from these results is that computationally-expensive algorithms are not necessary for this type of graph. Except for size preprocessing, depth 2 results are worse with preprocessing times often nearly as long or longer than the total running time at depth 1. Color preprocessing at depth 1 clearly outshines the other methods, being both fast relative to branching and at the same time greatly reducing branching time. We see that s-color preprocessing, designed to reduce the time of color preprocessing while hopefully doing just as well, comes up short. At

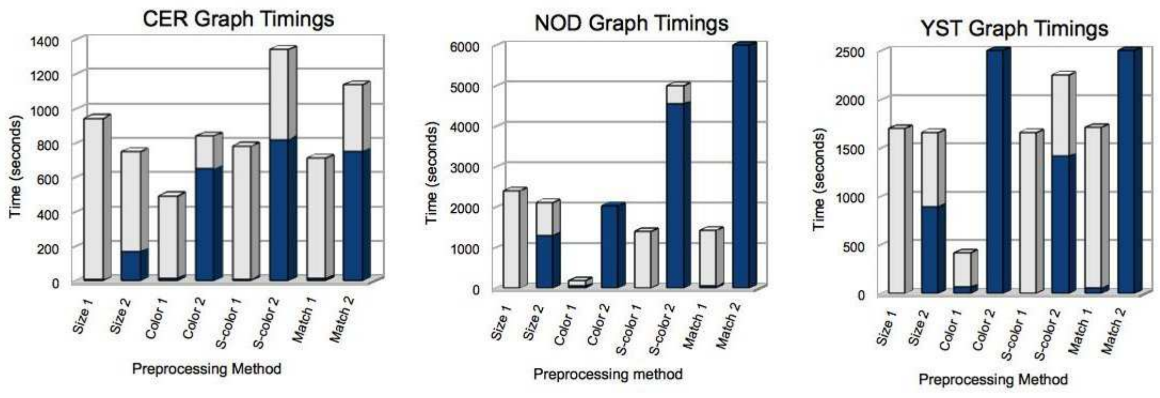


Figure 4.4: Preprocessing Timings

depth 2, in fact, we have a mysterious result concerning this method. The s-color preprocessing algorithm actually takes longer in two of the three cases, even though it is designed to be a faster alternative to color preprocessing. Further analysis shows that s-color preprocessing has slightly more repetitions (reiterating through vertices) when processing in those cases (3 versus 2 repetitions and 4 versus 3 repetitions, respectively). One possible explanation is that s-color preprocessing, because it is less effective in eliminating vertices, reduces the graph more gradually leading to more iterations. Additionally each iteration must process more vertices and so this lazier strategy backfires. This underscores the importance of experimentation, since it is difficult to predict all possible factors that affect run times.

4.3.2 Branching

Now we will try different branching algorithms while leaving preprocessing at the best result from the last section (color preprocessing depth 1). Again we only focus on methods that exploit vertex degrees. Since the vertex sorter computes vertex degrees as a matter of course, we go ahead and include the basic interleaved preprocessing in Figure 3.2 (simple high and low degree rules), as it can be done with little overhead. To do this, we break the abstraction a little by building a vertex sorter that also does rudimentary interleaved preprocessing. (During its routine operation, it checks if a vertex has a degree too high or too low and then applies the necessary reduction.) From this vantage point, eight possible strategies become apparent, as there are three binary decisions to be made. First, we can use either in-clique or not-in-clique branching. Second, we can sort vertices from lowest to highest degree or vice versa. Third, we can choose either a lazy or an active strategy. Figure 4.5 summarizes the results. All strategies that sorted from high to low degree failed to complete within 24 hours. So these strategies are not shown. Additionally, the in-clique, lazy algorithm, when sorting low to high, did complete for all but the YST graph but took an excessively longer amount of time than the other three remaining algorithms. So this strategy is also not shown. The final results are somewhat counterintuitive. The most successful strategy (in-clique and active sorting from lowest to highest degree) begins by assuming something improbable, that low degree vertices are in a maximum clique. To understand why this strategy works, consider that once the search tree finishes exploring the possibility that a particular vertex v is in a maximum clique, v can be removed from the graph. So this strategy quickly eliminates smaller degree vertices, reducing the search space and streamlining later processing. The fact that methods sorting highest to lowest degree fail so spectacularly also supports this conclusion. The two most successful strategies are active strategies, suggesting that persistent attempts to reduce the graph, an aggressive "garbage collection" strategy, seems to be key. The not-in-clique methods that are shown succeed because they still filter low-degree vertices by assuming, initially, that they are not in the clique. These lower-degree vertices, however, have to be revisited later.

4.4 Tuning MCF for Synthetic Graphs

In the previous section we completed experiments justifying the maximum clique algorithm of Figure 3.2 for graphs built from real data. When we apply this algorithm to synthetic graphs, however, the results are quite disappointing, and a different approach is required. By "synthetic" we mean graphs created from mathematical

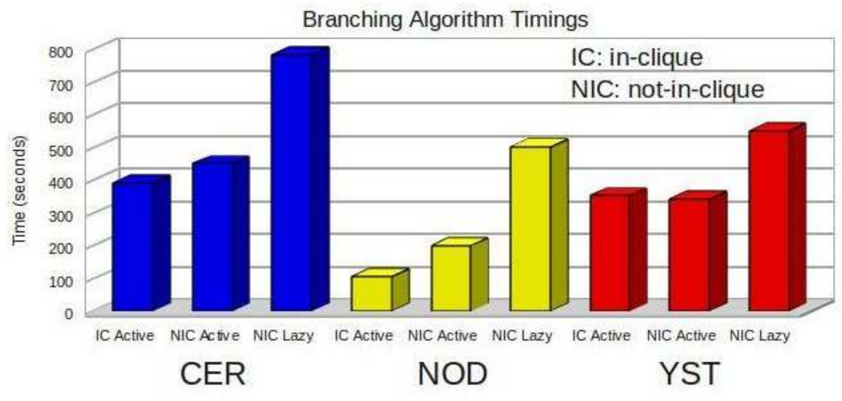


Figure 4.5: Branching Timings

formulas or structures or generated by simple algorithms, such as random graphs. These graphs tend to have a very uniform structure, which makes it harder to locate dense regions. Specifically, vertex degrees may differ only slightly, if at all, and so rules that exploit vertex degrees are less helpful. In this section we consider a different tuning of MCF for such graphs using a graph coloring algorithm presented in [47]. This algorithm has a vertex sorter that colors its subgraph, using the result to select vertices. The cited paper discusses two algorithms, *MCQ* and *MCR*, where MCR is a refinement of MCQ, which was introduced in an earlier paper. Because MCR is an incremental improvement to MCQ with additional optimizations, we focus solely on MCQ in order to highlight the main algorithmic changes without delving into minor details. We modify MCF easily to follow the basic MCQ strategy and show that with this simple adjustment we are able to process synthetic graphs. We also implement MCQ directly and show that our tuned MCF does not outperform the direct implementation, although the results are comparable. Thus while a highly-configurable program such as MCF helps to test and implement various algorithms, it can still be worthwhile to consider crafting specialized codes when one needs to focus on a specific graph class.

4.4.1 The MCQ algorithm

The MCQ algorithm discussed in [47] uses the same standard branching approach as MCF. Its vertex sorter, however, works by initially coloring the vertices, using a simple greedy coloring algorithm, and then iterating through the vertices from highest to lowest color. Curiously, the algorithm selects the highest color vertices first, ones that are likely to be in the clique, and uses an in-clique strategy. Given the observations made for real-data graphs, that the best strategy is to place unlikely candidates in the clique first, this seems suboptimal. MCQ prunes the search tree by a different technique, however. Whereas the best strategy for real-data graphs quickly prunes lower degree vertices, the MCQ strategy prunes by not having to branch on low-colored vertices. Observe that, upon returning from branching, the graph is altered only by removing the vertex that was assumed to be in the clique. Thus the coloring for the remaining subgraph is still valid. Once the color value for the next vertex selected by the vertex sorter drops below the current best clique size, no more searching is necessary.

4.4.2 Tuning MCF

Tuning MCF to implement MCQ is fairly straightforward. MCQ does not preprocess before branching. Thus this is removed except to apply the greedy algorithm to find an initial clique (not done in MCQ where the initial clique is empty). Then the vertex sorter is set to a lazy strategy that sorts the vertices once (after first coloring the graph). Like MCQ, it returns no more vertices available once the color value drops too low. Before branching, vertices are sorted by degree as is done in MCQ. Note that this tuning does not faithfully reproduce every aspect of MCQ, just the key idea of sorting vertices by color. We call this algorithm MCF-MCQ. Figure 4.6 is an algorithmic decision chart for MCF-MCQ. (Note the sparse preprocessing before branching and lack of interleaved preprocessing.) To compare this algorithm's performance to the

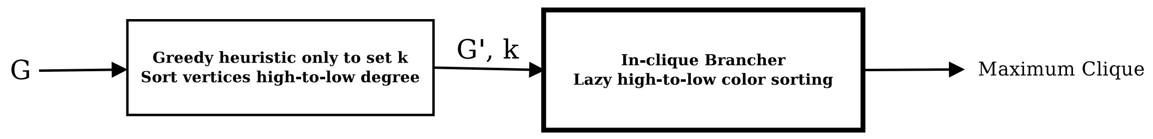

G
Greedy heuristic only to set k
Sort vertices high-to-low degree
G', k
In-clique Brancher
Lazy high-to-low color sorting
Maximum Clique

Figure 4.6: MCF-MCQ Algorithmic Decision Chart

original MCQ, MCQ was also implemented directly using the pseudocode in [47], bypassing MCF. We refer to this algorithm simply as MCQ.

4.4.3 DIMACS test graphs

For the set of synthetic graphs, we use the DIMACS benchmark graphs employed for the second DIMACS implementation challenge [26]. The graphs are available for download at [13]. Out of the 67 available challenge graphs, 26 representative graphs were chosen. Based on preliminary testing, the “c-fat” graphs and Keller graphs were eliminated, the former because they were all easily solved in a matter of seconds, and the latter because none of the tested algorithms could solve them within the 24 hour limit. The remaining 26 graphs represent the remaining graph types, eliminating both the easiest and hardest instances while still keeping some graphs that are quickly solvable, some that are unsolvable, and most that lie between these two extremes.

4.4.4 Results

Running times for MCF-RD, MCF-MCQ, and MCQ on each graph are listed in Table A.1. The most striking result is the difference in the performance of MCF-MCQ versus the earlier algorithm for real-data graphs in Figure 3.2, which we refer to as MCF-RD. For the set of 26 DIMACS synthetic graphs, MCF-MCQ is able to solve 22 while MCF-RD is only able to complete 7. For the real-data graphs NOD and YST, though, MCF-MCQ does not finish. So MCF-RD still performs far better on real-data graphs. Curiously, however, the CER graph is solved by MCF-MCQ in only 155 seconds, faster than the 405 seconds for MCF-RD. Note that the CER graph

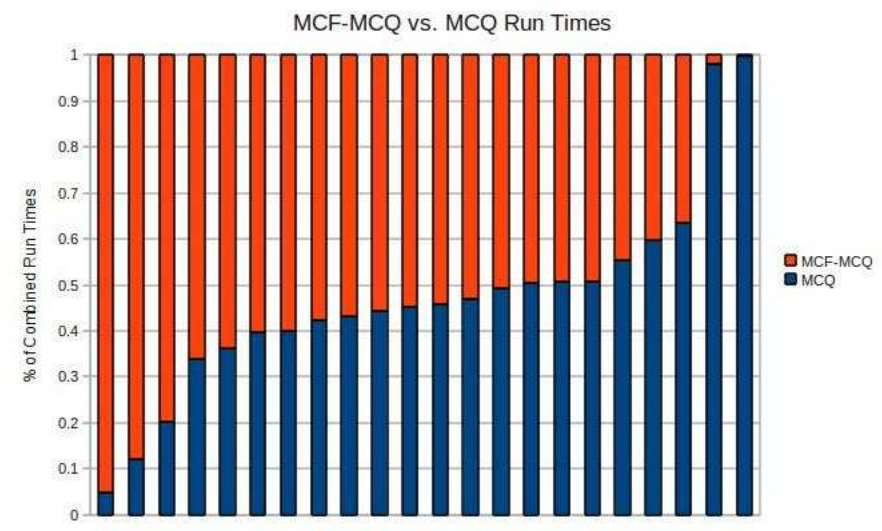


Figure 4.7: MCF-MCQ versus MCQ. Each bar compares the run time of MCF-MCQ and the run time of MCQ on one DIMACS graph. (Raw run times are not shown.) MCF-MCQ run time exceeds that of MCQ in most cases, suggesting that a straight implementation of a maximum clique algorithm may be worthwhile. MCF-MCQ is competitive, however, with only a few cases where it is significantly slower.

is more sparse than either NOD or YST and thus less challenging, which is probably why it is less sensitive to algorithm selection.

Less striking but still revealing is the comparison between MCF-MCQ and MCQ, which is summarized in Figure 4.7. This figure shows the percentage of the combined run time for both algorithms on all 22 DIMACS test graphs that were solved in 24 hours. Overall MCQ does better. Analysis reveals no clear reason. In some cases, the difference can be traced to the initial coloring of vertices. In other cases, though, it seems to be simply the overhead of using the MCF framework.

4.5 Parallel Implementation

This section describes the MCF approach for taking advantage of multiple processors. We first discuss the high-level design and then its implementation.

4.5.1 Parallel high-level design

MCF only parallelizes the branching search tree, which normally consumes most of the processing time (both in practice and in theory). In rare cases, preprocessing time may be a large percentage of total run time, but it can be delayed until branching by applying interleaved preprocessing only at the upper levels of the search tree. Assuming we are able to evenly distribute the workload, the preprocessing work should also be evenly distributed. MCF uses a master-servant approach. Initially, only the master processor runs, as if it were the only processor. At various points in the search tree, the master delegates the remaining subtree to another processor, waiting if no other processor is currently available.

The challenge in creating a good parallel strategy is distributing the workload evenly, especially for non-uniform graphs built from real data [1]. One approach is to traverse the search tree, breadth-first, until enough nodes are found to partition at least one to each processor, but the assigned nodes usually require widely different computational times. Thus, some form of dynamic (continuous) load balancing is needed, which quickly becomes complex. When processors become idle, new jobs must be readily available. This requires working processors to store and communicate subtrees, which again may have a wide range of run times that are difficult to predict. Communication overhead is a serious concern because constant monitoring and refactoring is needed to keep processors busy [7]. Given this complexity, MCF opts for a simple dynamic load balancing approach that tends to work well in practice up to a point, which we will show by experimentation. The premise is to avoid distributing jobs that require a large portion of the total computational time. The advantage of this approach is that no single processor ends up doing the bulk of the work while others sit idle. The other advantage is that refactoring of the workload is unnecessary. Processors need not be interrupted while working in order to rebalance the load. The disadvantage, however, is that dividing the workload into small pieces increases communication overhead. Since we use a master-servant architecture, where a single master distributes jobs, the master eventually becomes a bottleneck.

There are at least two ways of ensuring that jobs remain "small" (do not require excessive amounts of time). First, we can split the workload more finely by having the master process delay distributing nodes until it reaches a certain level of the search tree. MCF normally works best in practice if jobs are distributed at the first level of the search tree. MCF does allow the user to specify a different level for distributing jobs, though, if the workload is not being effectively balanced. (Note that for a binary search tree, finding an appropriate level for job distribution is more of a challenge, since the number of nodes at each level increases more gradually.) Secondly, we can have a branching strategy that avoids creating large jobs. Fortunately the sequential strategy of selecting vertices in order of ascending degree and assuming they are in the clique, which was shown experimentally as the best sequential strategy for real-data graphs earlier, meets this goal. Early jobs are small because low-degree vertices have few neighbors. Later jobs are small because the earlier vertices have been removed. As an example of a poor branching strategy for parallelization, consider the intuitive strategy of first assuming low-degree vertices not in the clique. This approach does not balance well because the first job, perhaps a vertex of degree one, encompasses nearly the entire workload.

4.5.2 Software implementation and results

The "PBrancher" class in Figure 4.1 is an extension (subclass in OOP language) of the Brancher class that adds capabilities for running in parallel. The Brancher class contains hooks in the normal branching code that allow an extension, such as PBrancher, to insert extra functionality, such as having a master process halt normal execution and send the current job to a servant process. (Thus, MCF can be easily modified to use more advanced load balancing strategies.) To illustrate the performance of this parallel approach, we test the NOD graph at increasingly

Table 4.2: Run Times in Seconds for Parallel Runs of the NOD Graph

	Number of Processors						
Cutoff	2	4	8	16	32	64	128
0.40	>24 hours	44101	19289	9454	5072	3074	2259
0.39	>24 hours	>24 hours	51978	26232	14629	9287	6961
0.38	>24 hours	>24 hours	>24 hours	44505	23816	14110	9750
0.37	>24 hours	>24 hours	>24 hours	>24 hours	44792	25335	16249
0.36	>24 hours	>24 hours	>24 hours	>24 hours	84589	45543	27153

smaller cutoff values, creating increasingly denser graphs, on various numbers of processors. All runs were done on quad-core Intel Nehalem 2.67 GHz processors with two processors (8 cores) per node.[†] Due to limitations on computing resources, only one run was performed for each case. As was done for the non-parallel runs, wall clock time was recorded, dedicated compute nodes were used, and runs were restricted to 24 hours. Table 4.2 summarizes the run times. Table 4.3 indicates the speedup each time the number of processors is doubled. Note that ideal linear speedup is 2. Also note that one processor serves as a master, which leads to the appearance of superlinear speedup in two cases. From these tables we observe, as forecasted, that significant speedup is being achieved but fades for greater numbers of processors. Running at increasingly smaller cutoffs, though, reveals another interesting trend. As graphs become denser and more difficult, speedup erodes more slowly. Thus, even with this simple approach we see that there is hope to solve very difficult instances given a sufficient number of processors.

[†]This research used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

Table 4.3: Speedup for Parallel Runs of NOD Graph

	Transition (X Processors to Y Processors)					
Cutoff	2-4	4-8	8-16	16-32	32-64	64-128
0.40	-	2.29	2.04	1.86	1.65	1.36
0.39	-	-	1.98	1.79	1.58	1.33
0.38	-	-	-	1.87	1.69	1.45
0.37	-	-	-	-	1.77	1.56
0.36	-	-	-	-	1.86	1.68

Chapter 5

Graph Algorithms for Integrated Biological Analysis

This chapter was published in the book *Clustering Challenges in Biological Networks* by World Scientific:

J. D. Eblen, I. C. Gerling, A. M. Saxton, J. Wu, J. R. Snoddy and M. A. Langston, Graph Algorithms for Integrated Biological Analysis, with Applications to Type 1 Diabetes Data, in Clustering Challenges in Biological Networks (S. Butenko, W. A. Chaovalitwongse and P. Pardalos, editors), World Scientific, 2009, 207-222.

Only minor modifications have been made to the published work. Previous chapters have discussed the technical aspects of solving maximum clique. This work demonstrates its practical application. MCF computes the maximum cliques that are augmented by the paraclique algorithm. My contribution to this paper was implementing paraclique, the running of all paraclique and k-means experiments, most of the initial writing, and participation in several rounds of editing and revision.

5.1 Overview

Many inbred strains of *Mus musculus*, the common house mouse, are employed in biomedical research. The non-obese diabetic (NOD) mouse is particularly useful as a model of type 1 diabetes mellitus (also called juvenile onset, or insulin dependent, diabetes). In both mice and humans, this disease is characterized by persistent hyperglycemia (elevated blood sugar level) that is induced in genetically susceptible individuals and modified by a variety of environmental triggers including food and infections. It is caused by an abnormal and self-destructive immune response (autoimmunity), which allows mononuclear leukocytes to target the insulin producing beta cells in the pancreas [45, 53, 46]. Eventually this process destroys so many of the beta cells that the body is unable to produce sufficient insulin to retain normal blood glucose levels and diabetes is observed. Our studies in the NOD mouse focus on the very early leukocyte abnormalities that may be associated with initiation of the autoimmune process [24]. If we can gain a better understanding of the initiation of autoimmunity, then we may be able to develop rational intervention strategies that can stop the disease process in its preclinical phase effectively and with minimal side effects.

The importance of melding experimental research with continuing advances in computational analysis is well understood [28, 29, 37]. In the work reported here, we begin with high-throughput NOD mouse data and apply novel clique-centric methods to analyze it. Fixed parameter tractability [14, 2] and various realizations of the paraclique algorithm [9] form the basis of techniques we use to extract dense putative networks from the vast sea of correlations that arise in the analysis of comprehensive transcriptomic data [6, 30]. Proteomics data is added to the mix, thereby introducing challenging new problems in inhomogeneous data interpretation [27]. The results we obtain are evaluated in terms of both statistical quality and biological relevance.

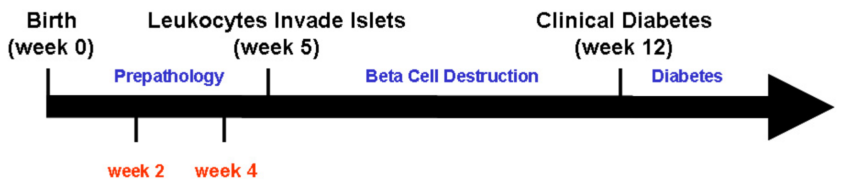


Figure 5.1: At birth, NOD mice have normal blood glucose levels, with no indication of destruction of insulin-producing beta cells (located in the islets of Langerhans). At five weeks of age, the first signs of pathology occur with leukocytes invading the space around the islets. This infiltration progresses to involve additional leukocytes with more invasive and destructive character. By twelve weeks of age or later, so many beta-cells have been destroyed that insulin production capacity has been severely diminished. Because blood glucose can no longer be regulated normally, the mice become diabetic. We sampled leukocytes in the early and late prepathology stage to evaluate defects at the molecular level associated with initiation of the pathology.

5.2 Description of Data

To define abnormalities in the early phases of autoimmunity, we have conducted comprehensive studies of gene expression in young NOD mice and mice from control strains (NON and C57BL/6) that do not develop diabetes or autoimmunity to beta cells [24, 23]. Genes encoded in DNA are transcribed into mRNA, which is then translated into proteins that are the major determinants of a cell's activation and function. To gain a comprehensive picture of how the genetic differences between our strains can affect the development of autoimmunity, we evaluated gene expression at the mRNA level using Affymetrix MOE430A/B arrays, and at the protein level using 2D-gel electrophoresis. We collected mononuclear spleen leukocytes from each of the three strains at both two and four weeks of age. This is a critical window for our analysis, because it represents the prepathology stage before leukocytes begin to infiltrate the islets of Langerhans, which typically occurs in NOD pups when they are about five weeks old. See Figure 5.1. From each of the six strain/age groups, we collected five independent samples for a total of 30 samples in the complete dataset. Experimental details regarding the analysis of mRNA and protein expression levels have been published previously [24].

Because the data is biological, it has a fairly high level of noise. At the time of sample collection, individual mice may or may not have just eaten, been fighting, been scared, been sleeping, etc. These biological parameters can be difficult to control, and can have an influence on expression levels of some genes. In addition to this biologically derived noise, there are also technical sources of noise to be considered. The mRNA gene expression arrays have a very effective normalization and scaling process and very good technical reproducibility on identical samples with percent coefficients of variance usually in the low single digits [43, 50]. In contrast, protein expression data involves technologies that are more complicated and difficult to standardize. Technical reproducibility of protein expression data collected from identical samples often has percent coefficients of variance in the low double digit range [52, 39].

5.3 Correlation Computations

We employ the aforementioned 30 samples to compute a correlation matrix. The matrix entry at location (i, j) denotes the correlation coefficient between the i^{th} and j^{th} items (genes or proteins), normalized to the range [-1.0,1.0]. Because mRNA arrays alone can measure over 45,000 different values, we may be faced with making sense of over a trillion correlate pairs. Close examination of the data reveals a paucity of outliers, so that we are able to use the well-known Pearson's method for the computation of correlation coefficients. Because we are searching for putative pathways and networks, both positive and negative correlations are of equal interest. We therefore take absolute correlation values. Recall that this is biological and hence noisy data. Not every probe set is reliably measured in every sample. Thus we move away from simple correlation and compute a p-value for each pair of correlates, which is the probability that they have a correlation different from zero [54]. See Figure 5.2.

Protein correlations are too weak to find relevant relationships at this level, and so for them we turn to other methods as will be described in Section 6. The correlation

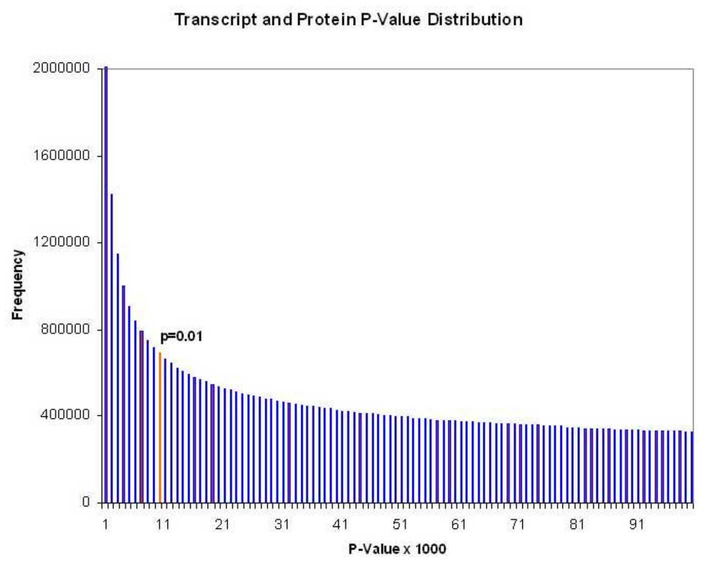


Figure 5.2: The transcriptomic data used in this study provides a broad spectrum of p-values. A threshold p-value of 0.01, for example, creates an unweighted graph with 22750 vertices and roughly 11 million edges.

matrix is transformed into a complete, weighted correlation graph by using a vertex for each transcript and protein, and by weighting the edge between each pair of items with the corresponding correlation matrix entry. From this we can build a simple, unweighted graph as needed with the use of a cut-off value (we favor the use of p=0.01) and a high-pass filter. An edge whose weight is less then the cut-off is discarded. Other edges are retained, but their weights are now ignored.

5.4 Clique and its Variants

We assume the reader is familiar with standard concepts in graph and complexity theory [51, 44]. We begin with the well-known clique problem. A clique is a densest possible subgraph. Each pair of its vertices is connected by an edge. A clique is maximum if it is a largest clique in a graph. A clique is maximal if it is not contained wholly within a larger clique. A clique on five vertices is illustrated in Figure 5.3. Protein correlations are too weak to find relevant relationships at this level, and so

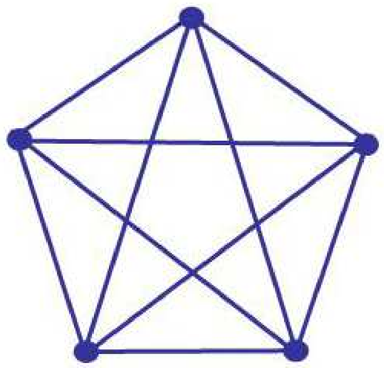


Figure 5.3: A clique of size five.

for them we turn to other methods as will be described in Section 6. The correlation matrix is transformed into a complete, weighted correlation graph by using a vertex for each transcript and protein, and by weighting the edge between each pair of items with the corresponding correlation matrix entry. Clique is widely acknowledged for its many applications in computational molecular biology [41]. In the present setting, its advantages include cluster purity (all edges are present), cluster overlap (genes and gene products are pleiotropic), and resistance to false positives (the bane of many clustering methods). Contrasts with other techniques can be found in [49]. The classic *decision* version of clique is $\mathcal{NP}$-complete. Finding approximate solutions appears no easier, because ensuring solutions within n^{ϵ} in polynomial time implies $\mathcal{P} = \mathcal{NP}$ for any $\epsilon > 0$ [16].

We are of course more interested in *search* and *optimization*. By transforming clique to vertex cover, we can apply notions from fixed-parameter tractability [14, 2] and many years of basic research [18, 19] to solve the *maximum* clique problem effectively in practice. With novel implementations and high performance platforms, we are currently able to find maximum cliques with hundreds of vertices in graphs with tens of thousands of nodes. We must often also solve the *maximal* clique problem [8]. Even when the maximum clique size is modest, we frequently find that the number of maximal cliques is staggering. Thus it is that space as well as time is a critical resource for solving maximal clique, even when supercomputing technologies are used.

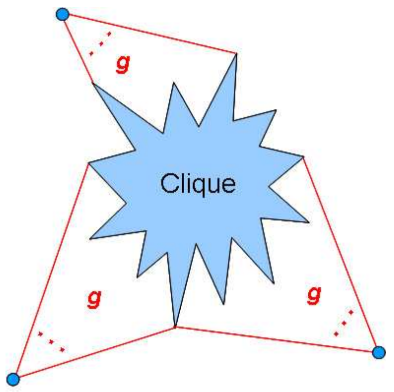


Figure 5.4: Paraclique augments a clique with non-clique vertices in a controlled manner to increase size, decrease overlap and maintain density.

Our work on this general subject, as well as its application to transcriptomic data analysis, is chronicled in [2, 9, 6, 30, 49, 10, 31, 55].

The *paraclique* algorithm was recently introduced in [9], where it was shown to have advantages in the amelioration of noise inherent in high throughput biological data. Clique by itself is highly resistant to false positives. Under certain experimental conditions, however, it can be subject to false negatives. This is because, if even a single edge is missing, the clique is lost. Moreover, we frequently encounter enormous numbers of overlapping cliques [30]. To coalesce these into fewer but larger clusters, and to reduce the significance of noise, paraclique solves something similar to the dense-k-subgraph problem [17], which is NP-complete even on graphs of maximum degree three. Roughly speaking, a paraclique is a clique augmented with non-clique vertices in a highly controlled manner. A user-defined *glom factor*, g, is provided to increase cluster size while limiting the number of missing edges permitted. We glom onto a non-clique vertex only if it is adjacent to at least g clique/paraclique members. This notion is depicted in Figure 5.4. Correlations between non-adjacent vertices may be taken into consideration as well. We refer the reader to [9] for details. Thus, when the application permits, we employ the paraclique algorithm and sacrifice overlap in order to build robust clusters.

Paraclique is also useful from a computational standpoint because it can, depending on the application, obviate the need for maximal clique enumeration. To illustrate, the processing of an NOD file whose maximum clique size was only 20 produced a list containing over four million maximal cliques and requiring over two gigabytes of memory before the enumeration was terminated by the operating system. In contrast, only 25 paracliques were generated. We therefore identify a maximum clique, use paraclique to decompose the graph, and then iterate the process on the remaining subgraph. We halt the process when maximum clique size falls below some reasonable cutoff value (we set this value at 50). In this way, paraclique eliminates the need to compute and store enormous lists of maximal cliques.

5.5 Statistical Evaluation and Biological Relevance

Edge density is arguably the most telling statistical clustering metric. Clique, of course, maximizes density at 100% by definition. With the paraclique algorithm, density will tend to decrease as new nodes are glommed onto a starting clique. How precipitously density falls depends heavily on g. Table 5.1 summarizes the results for paraclique when it was run over the NOD data of this study. Clique size, paraclique size, and edge density are averaged over the paracliques generated. Note the manner in which paraclique increases cluster size with only a gradual reduction in density. (In contrast, we find that enlarging cliques using simple 1- and 2-neighborhoods quickly drops density into the single digits.) As a practical matter, we must balance the desire to handle noise and expand paracliques with the real need to maintain suitably high edge densities. As a rule of thumb, therefore, we seek to maintain a minimum density of at least 90% and henceforth set g at $|C| - 5$. We emphasize that this choice is highly data-dependent, and tunable to each application by design.

Density alone, however, tells only part of the story. To test for biological relevance, we used the Ingenuity Pathways Analysis (IPA) package from Ingenuity® Systems, `www.ingenuity.com`. IPA allows subscribers to upload and test lists of genes (in our

Table 5.1: Paraclique Parameter Variation

Glom Factor	Number of Paracliques	Clique Size	Paraclique Size	Edge Density	Lowest Edge Density
$\|C\| - 1$	32	99.4	104.8	99.8%	99.5%
$\|C\| - 2$	30	99.9	118.8	99.0%	97.9%
$\|C\| - 3$	28	101.6	137.4	97.8%	96.0%
$\|C\| - 4$	27	101.4	151.4	96.4%	92.3%
$\|C\| - 5$	24	106.1	173.8	94.9%	90.3%
$\|C\| - 6$	24	104.7	186.8	92.9%	86.7%
$\|C\| - 7$	22	108.5	205.7	91.4%	83.1%
$\|C\| - 8$	21	110.2	221.1	90.0%	80.0%
$\|C\| - 9$	21	109.3	231.1	88.6%	77.9%
$\|C\| - 10$	19	114.7	250.5	87.7%	76.6%

case Affymetrix probesets) against a manually curated biological interaction database. Probe sets known by the database are mapped to genes, which are then termed *focus genes*. Other probe sets are ignored. Focus genes are analyzed to determine how they are connected to one another based on evidence from the biomedical literature. Based on this analysis, one or more molecular networks are produced. Each typically consists of a mixture of focus genes, sprinkled with additional database genes and gene products that are needed to connect the focus genes and complete the network. We term a focus gene that is placed in such a network a *focus gene utilized*. In general, one cannot expect that all focus genes will become members of a network. The database may have very little information about a focus gene's connectivity. Alternately, a focus gene may be only distantly related to other focus genes. Due to technical constraints, IPA imposes a limit on network size, which is currently set to 35 nodes. As a result, lists with large numbers of focus genes often create multiple networks. Fortunately, these can often be fused together into a single common network using commands that are available on the Ingenuity website and that are designed for this purpose. The more closely connected a group of focus genes are biologically, the more likely it is that the database can connect them all into a network. Thus, an important metric is the *percent focus genes utilized*. This number alone can be misleading, however, because we must bear in mind that IPA may spread the genes across more than one network. A group of 40 focus genes, for example, would be considered more closely related if they could be connected in two networks than if four networks are needed to connect them all. We will therefore also calculate and examine *focus genes utilized per network*, a metric that normalizes for this effect.

As a control, we also tested K-means clustering, a traditional and highly popular algorithm. We invoked it via the R programming language, with the “kmeans” function from the “amap” package [25]. Input values were log transformed. Pearson correlations were employed. We sought to generate 500 clusters, because that should yield clusters of roughly the same size as those produced by the paraclique algorithm. Iteration was performed until convergence. IPA requires that each network be analyzed separately (no batch mode is available), a process that can be quite time consuming. Thus, only a small number, say ten, of clusters could be selected for further analysis. For paraclique, we simply selected the first ten outputs. Deciding on a representative set of K-means outputs was not as straightforward. We therefore chose to select K-means clusters under three different criteria. One criterion was to choose the ten largest clusters. Another was to favor those ten with the highest edge density in the p=0.01 graph. In case this produced unfairly small genesets, we also required that for a cluster to be selected it had to have size at least 50, the same lower bound we use for paraclique. The third criterion was based on paraclique overlap. For this we chose the ten K-means clusters with the highest percentage overlap with some paraclique, again insisting that a cluster had to have at least 50 vertices. Overlap ranged from roughly 45% to 64%, with an average of about 55%. Table 5.2 summarizes these results. All values are averaged over the relevant ten clusters.

By inspection, paraclique is superior to K-means clustering in terms of density. The case for superior biological relevance is perhaps less obvious. We therefore performed ANOVA tests for statistical significance. The number of focus genes per

Table 5.2: Paraclique versus K-Means

Method	Probe Sets	Edge Density	Focus Genes	Genes Utilized	Percent Utilized	Focus Genes per Network
Paraclique	254.3	97.1%	146.9	140.7	95.5%	14.4
Large K-means	244.0	31.5%	143.1	133.5	93.0%	12.8
Dense K-means	80.9	84.6%	52.3	46.7	89.4%	12.8
Overlap K-means	89.0	79.6%	55.7	49.9	89.8%	12.3

network was higher ($p < .001$) for paraclique than for any of the K-means methods. And while paraclique did not differ markedly from Large K-means in terms of cluster size, it was more successful than other K-means methods in both size and percent focus genes utilized ($p < .05$).

5.6 Proteomic Data Integration

We now consider the problem of combining quantitative transcript and protein data for analysis. Only a few studies have been reported (see, for example, [3]). The related problem of combining gene expression with measures of function was recently considered in [4]. There gene ontology, phenotypes and protein-protein interaction were used to devise distance measures and permutation tests for strength of commonality in graphs from these different data sources. Although no quantitative protein values were employed, data derived from *Saccharomyces cerevisiae*, commonly known as baker's or budding yeast, suggested that similarity in expression is related to similarity in function.

Our main goal is to identify biological pathways, each of which is anchored by a protein of interest. We are fortunate that both gene expression array data and protein gel data were collected from the exact same samples. If it were not for the expense involved, we would wonder why this is not done more often. Nevertheless, data integration remains a formidable task. The biggest difficulty we must overcome is probably that transcriptomic and proteomic data are generated by two completely different and unrelated processes. Thus we will not be able to use parametric statistical procedures, including the highly favored Pearson's correlation technique. Another problem is that current technologies for protein sensing are generally inferior to those for transcript detection. Modern expression array platforms can often detect transcripts for more than 50% of the known genes in the relevant organism, and generate highly reproducible quantitative measurements. In contrast, protein identification platforms can seldom cover more than 10% of an organism's estimated number of proteins, and with only moderate quantization and reproducibility. Of course function is a direct consequence of proteins, not mRNA, and so the importance of protein expression cannot be underestimated. Finally, it is well known that gene expression at the mRNA level will not always correlate well with gene expression at the protein level. After all, gene products are subject to post-transcriptional and post-translational modifications, degradation and other factors. Put together, these difficulties make any serious attempt at transcript-protein co-expression analysis a huge challenge. In the sequel, we shall address this challenge with non-parametric methods, graph algorithms and a clique-centric combinatorial approach.

We begin with the establishment of two correlation structures. For transcript-transcript relationships, we retain the Pearson's coefficients already computed. Transcript-protein relationships are typically much weaker and, for reasons already stated, require a non-parametric approach. For these we employ the rank metric provided by Spearman's correlation technique. This naturally leads to the loss of some information; a simple ranked list "flattens" raw data values. Our aim is now two-fold. We still wish to find dense, well-connected subgraphs. Yet these subgraphs must also be anchored as much as possible about some given protein, p, under scrutiny. Of course we could simply choose a putative pathway to be p and those transcripts ranked most highly with it. As we shall show, however, we can do better with the use of graph structure. To accomplish this, we take the transcript graph and add to it a new vertex for protein p. We then use the rank order provided by the Spearman's coefficient list to add edges connecting p with transcript vertices. We add these edges until the subgraph induced by p and its neighbors contains at least 100 maximal cliques each of size at least 40. We then output p along with the 60 or fewer vertices that most highly populate the resultant set of cliques. The values 40, 60 and 100 were chosen based on trial and error combined with our previous experience working with the idiosyncrasies of IPA. Other values may be superior in other applications.

To test this approach, we chose six proteins on which IPA contained information, which were well-expressed in the experimental samples, and which appear to be orthogonal to each other in terms of their biological function. Two of the six, HNRPK and EIF4A1, are of special interest because they are generally known to have increased expression in NOD mice relative to the NON and C57BL/6 strains [24]. The other four are ACTB, GDI2, GNB2L1 and ZBTB1. We also chose three different transcript graphs constructed from respective Pearson correlation thresholds 0.60, 0.70, and 0.80. For each of these 18 tests, maximal cliques were highly overlapping, as expected. As a measure of a cluster's biological relevance, we examine a metric we call *protein links*. Protein links is a count of the number of connections between an anchored protein and the network created by IPA. For each protein, we chose the threshold setting that maximizes protein links, with ties broken in favor of the higher threshold. The lowest threshold, 0.60, had none of the best results. It is probably the case that, in a graph this dense, the transcript-transcript relationships drown out protein-transcript correlations.

As a control, we compared the quality of the transcript sets we produced against the 60 transcripts that simply correlate most highly with the protein. GDI2 and ZBTB1 had fewer than three protein links for all four results (the three threshold values plus the straight correlation list), and so were dropped from further analysis. Results for each of the four remaining proteins are shown in Table 5.3.

From this table, we see that our clique-centric approach builds subgraphs that are no worse and in fact generally better than those simply defined by ranking and selecting correlates. Although protein links are our primary focus, other metrics are

Table 5.3: Clique vs Correlates

Protein	Algorithm	Probe Sets	Focus Genes	Protein Links
ACTB	Clique at 0.70	60	42	6
	Correlates List	60	27	6
EIF4A1	Clique at 0.70	59	50	7
	Correlates List	60	41	2
GNB2L1	Clique at 0.70	60	39	6
	Correlates List	60	37	3
HNRPK	Clique at 0.80	55	38	5
	Correlates List	60	42	3

equally revealing. In the case of ACTB, for example, we find that both methods produce six protein links, but the algorithm based on clique is superior in terms of percent focus genes utilized (100% versus 85.2%) and focus genes per network (14 versus 11.5).

It may also be instructive to compare IPA's outputs visually. Figures 5.5 and 5.6 contain screenshots of merged network diagrams created by IPA for HNRPK. Figure 5.5 was generated from the list of transcripts produced by our clique-centric method; Figure 5.6 was generated from the list produced by mere correlate ranking. Focus genes are depicted in grey. Connections to the anchor protein are rendered in blue. Glyph shapes vary depending on IPA classifiers.

The IPA screenshots shown in Figures 5.5 and 5.6 demonstrate how the two methods we consider create quite different networks, and how the protein is connected to more genes in the network created by the clique-centric algorithm.

5.7 Remarks

We have studied clique-centric algorithms in the context of effective biological data clustering. Statistical quality based primarily on edge density and biological significance based on curated pathway matching have demonstrated the utility

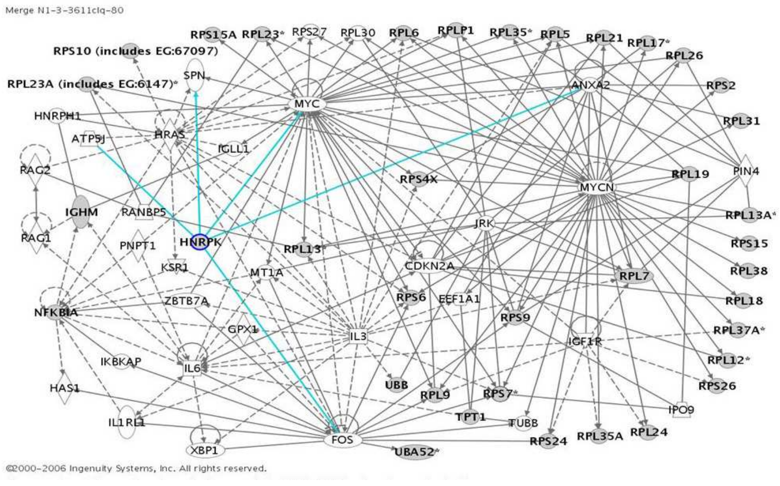


Figure 5.5: The IPA merged network for HNRPK using a clique-centric algorithm.

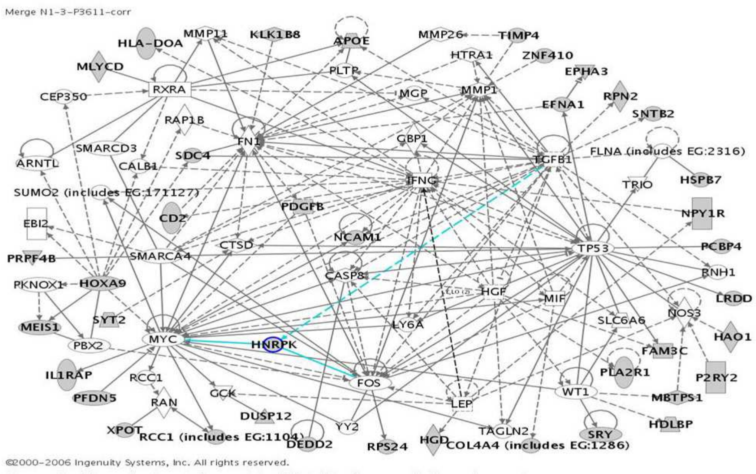


Figure 5.6: The IPA merged network for HNRPK using simple correlation.

of paraclique and related methods. We have also considered the problem of inhomogeneous data integration. Transcriptomic data from gene expression arrays and proteomics data from 2d gels have been reconciled to identify biological networks for further scrutiny.

We emphasize that this work has been limited in scope to the analysis of inhomogeneous data of relevance to type 1 diabetes. It is not meant to provide a comprehensive guide to the literature. Nor is it intended to serve as an exhaustive comparison of clustering methods. Such a task would be an enormous challenge, requiring the implementation of a huge number of algorithms, and necessitating tests across a great many diverse datasets.

There are a variety of ways to modify and enhance paraclique and the other algorithms we describe. In [9], for example, an optional user-defined *threshold parameter* is provided to help guide the search for edges affected by noise. For simplicity, we have ignored this parameter here and considered only the effect of the glom factor. Another enhancement is to glom vertices in stages, invoking paraclique iteratively until a certain threshold is reached. Initial results suggest that this procedure can further increase paraclique size while maintaining both edge density and biological fitness as measured by IPA.

Finally, we observe that pleiotropism is common in gene and gene products. It is thus a major reason for the popularity of soft clustering methods such as clique: a vertex can lie in more than one clique, just as an oligonucleotide or a protein can lie in more than one pathway. Noise and the need for simpler structures motivate the paraclique algorithm. The clusters produced are robust with respect to a few missing edges. Unfortunately, they no longer overlap with the basic paraclique method. It is possible to modify the algorithm so that overlap is permitted. This is a topic of current research within our group. Optimal ways to accomplish this, however, probably depend on the application. The same may be said for the highly challenging task of inhomogeneous data integration. We are currently working on techniques to integrate multiple proteins in a single step, rather than handling them one at a time. This is not as easy as it might sound, and may require the use of three rather than just two forms of correlate pairs.

Acknowledgments

We wish to thank an anonymous referee, whose careful review of our original submission helped us to improve the final presentation. This research has been supported in part by the National Science Foundation under grant CCR-0311500, by the National Institutes of Health under grants 1-P01-DA-015027-01, 5-U01-AA-013512, 1-R01-DK-062103-01 and 1-R01-MH-074460-01, and by the UT-ORNL Science Alliance. A preliminary version of a portion of this paper was presented at the DIMACS Workshop on Clustering Problems in Biological Networks, held at the DIMACS Center at Rutgers University in May, 2006.

Chapter 6

Practical Algorithms

In previous chapters, we have described methods for computing maximum cliques, both high-level algorithms and software implementations. In Chapter 5 we saw the application of paraclique, a practical algorithm that employs the MCF clique-finding engine to analyze data. In this chapter, we explore more fully this idea of building practical data-analysis algorithms on top of clique-finding software. Refer to Section 1.2 for more details about how a graph can be used to represent biological data. More generally, a graph can be used when we have a set of items (vertices) and a relationship between those vertices, which can be used to define edge weights. A clique is a set of items that are all pairwise related. Thus, machinery for finding cliques becomes machinery for finding highly dense cores. This chapter describes algorithms that build upon these cores and are applicable to data mining tasks that seek highly cohesive groups inside a given data set. We start by describing paraclique and how it is used in practice, including how it was used in Chapter 5. We then present a new variation of paraclique that attempts to be more robust. We run some preliminary experiments to compare the two algorithms. Next, we propose a new algorithm based on maximal clique, which is an attempt to improve upon a known algorithm. We then again show some preliminary results. Lastly, we consider relaxations of clique and a more general clique problem. We see that these problems address some of the weaknesses of clique and thus are targets for future research in developing practical algorithms based on computationally hard problems.

6.1 Paraclique and Phased Paraclique

The term *paraclique* was first defined in [9]. A paraclique is a clique plus additional vertices that are "close" to membership in that clique. Paraclique relies on a maximum clique algorithm to find highly dense cores and then extends or "fleshes out" these cores to include vertices omitted due to noise. We first define paraclique and discuss how paracliques can be computed in practice. Next, we introduce a variation of paraclique designed to enhance paraclique construction.

6.1.1 Paraclique

The version of paraclique used in Chapter 5 and in this chapter is given as pseudocode in Figure 6.1. Note that it differs from that given in [9], mainly due to practical experience. It is simpler because there is no outer loop. It was discovered that repeated addition of vertices to P can lead to unrestrained growth for real-data graphs. This is because the glom factor g is unchanged but P grows larger with each iteration, making it easier to add new vertices each time. Another difference is the lack of threshold value t, which is effectively set to zero. This threshold value gives another means to control paraclique growth, but in practice is difficult to apply due to the need to store the weights of all graph edges. For example, a graph of 20,000 vertices, a conservative number for the types of real-data graphs frequently encountered with biological data, would require storing 200 million edge weights or about 800 megabytes of data if each edge is four bytes. Also, note that the number of edges grows quadratically with respect to number of vertices. Finally, note that this definition is more general in that the set C does not have to be a clique, much less a maximum clique. This flexibility is useful later when phased paraclique is introduced.

```
compute_paraclique(graph G = (V,E), glom factor g,
                                      vertex set C)
P = C
for each v in V-P {
  if v is adjacent to at least g members of C { P = P U {v} }
}
return P
```

Figure 6.1: Paraclique Algorithm

For now, though, we can consider C to be a maximum clique returned by MCF or some other program for computing maximum cliques.

6.1.2 Paraclique computation

The *compute_paraclique* function alone is inadequate as an algorithm for finding multiple cohesive groups. Some type of iterative process is needed. The approach we use is to compute a maximum clique C in G, extend it to P with *compute_paraclique* in Figure 6.1, remove P from G, and repeat until the graph is exhausted (no maximum cliques greater than size 3 exist) or the cliques become "small" (an adjustable parameter). The glom factor specifies an absolute number of vertices in C to which a candidate vertex must be adjacent, and hence is a poor parameter by itself. (It neglects the size of the set C.) Thus we set the glom factor $g = |C| - a$ for some nonnegative integer a. The glom factor varies with respect to $|C|$, and the relevant, unchanging parameter is a, the number of vertices in C allowed to be nonadjacent to the candidate vertex.

6.1.3 Phased paraclique

Paraclique requires the user to choose a glom factor, which poses a couple of problems. First, setting the glom factor to $|C| - a$ does not adjust for size differences in cliques since it is simply a raw number of missing edges allowed. Second, paraclique does not take into account the relative "distances" of each vertex to the set C. Intuitively,

```
compute_phased_paraclique(graph G = (V,E), ratio R,
                                       vertex set C)
P = C
a = 1
while (a <= R*|P|) {
  P = compute_paraclique(G, |P|-a, P)
  a = a + 1
}

return P
```

Figure 6.2: Phased Paraclique Algorithm

vertices lacking fewer edges from being in a clique should be favored and a more iterative approach seems more appropriate. An iterative approach would allow the user to set some reasonable stopping criteria for paraclique growth and thus have better control over augmentation. These stopping criteria also can take clique size into account. *Phased paraclique* is an algorithm designed along these lines and is given as pseudocode in Figure 6.2. Phased paraclique relies on the same *compute_paraclique* function in Figure 6.1 but may call it multiple times to augment a single clique. Note that for *each* maximum clique C, phased paraclique does the following. It first calls *compute_paraclique* with graph G, glom factor $g = |C| - 1$, and clique $P = C$ so that all nodes missing only a single edge are added. In the next iteration, *compute_paraclique* is called with graph G, glom factor $|P| - 2$, and P (note that now we are invoking *compute_paraclique* for a non-clique set). This continues, with the glom factor decreasing by one each time, until the stopping criteria for expansion are met. Added vertices must be adjacent not only to adequate numbers of members of the original maximum clique but to vertices added to the paraclique at prior iterations. Thus, the algorithm gives priority to "closer" vertices, and these newly added vertices help to filter vertices in later iterations. Phased paraclique halts expansion based on the current size of the paraclique. Many variations to this approach are possible, of course, such as paraclique densities and application specific measurements. Phased paraclique halts when proceeding would add vertices with too many missing edges, specified by a ratio R. Specifically, phased paraclique halts when $R < a/|P|$, where glom $g = |P| - a$. Note that R is an upper bound on the percentage of vertices in $|P|$ that are allowed to be nonadjacent to a new vertex. For example, if $R = 1/4$, then phased paraclique halts once the glom factor would allow vertices to be added that are nonadjacent to more than a quarter of vertices in $|P|$ on the next iteration.

6.1.4 Comparison of paraclique and phased paraclique

To compare the two algorithms, we use a sparser version of the YST graph from Chapter 4 at correlation value 0.80, which we refer to as YST80. This graph contains 2608 vertices and 26061 edges. Table 6.1 summarizes the results of ten runs of paraclique on the YST80 graph. Paraclique was run until the graph was exhausted, but paracliques of size less than 10 are excluded. Note that as the parameter increases, paraclique maintains density while creating ever larger paracliques. The number of paracliques, however, steadily drops. Table 6.2 summarizes the results for phased paraclique on the same graph. As the ratio increases, more paracliques are found while the size of the paracliques remains steady. There is a steady decrease in density, but that is to be expected as the criteria for paraclique growth is relaxed. Thus paraclique works well for small glom factors, but as the glom increases, paracliques begin to merge and distinct regions are lost. For phased paraclique, it seems that old regions remain and grow larger, and at the same time new regions develop.

6.2 Clique Difference

In this section we introduce a technique based on maximal clique and termed *clique difference*. While computing all maximal cliques is not feasible for large, dense graphs, it is often tractable for graphs built from biological data sets of only a few thousand genes and built with a high threshold. In such cases, having all maximal cliques provides a significant advantage over computing a single maximum clique because

Table 6.1: Results of 10 Runs of Paraclique on YST80

Glom	Number of Paracliques	Avg. Size	Avg. Density
$\lvert G\rvert - 1$	14	21.07	0.99
$\lvert G\rvert - 2$	41	15.51	0.57
$\lvert G\rvert - 3$	42	19.9	0.55
$\lvert G\rvert - 4$	29	26.24	0.52
$\lvert G\rvert - 5$	17	32.35	0.6
$\lvert G\rvert - 6$	14	39	0.58
$\lvert G\rvert - 7$	10	46.1	0.59
$\lvert G\rvert - 8$	9	55.67	0.52
$\lvert G\rvert - 9$	8	61.88	0.5
$\lvert G\rvert - 10$	5	79.6	0.56

vastly more information is available from which to work. Observations of these maximal cliques on real-data graphs reveal that many are nearly identical, differing in only a few vertices. So a logical question to ask is if we could somehow combine them into a smaller set of dense cores. One such approach is that of k-clique community [36], which is implemented by the CFinder software [11]. Clique difference is similar but fuses sets together in stages, allowing for a gradual reduction of the number of sets, a process that can be halted when desired to balance number of sets versus quality. By fusing the most similar sets together first and continuing this process in recursive fashion, set quality could possibly be improved over that of a single step method such as k-clique community.

6.2.1 The k-clique community algorithm

Both k-clique community and clique difference employ an auxiliary graph, which we term a *cluster graph*. A cluster graph is defined for a graph G and a set of clusters of G's vertices and is used to decide which clusters to merge. The clusters define the vertices of the cluster graph. (Adjacency of vertices is defined later, as it varies between the two methods.) Connected components are computed, and clusters all in the same component are merged. The k-clique community algorithm inputs a graph

Table 6.2: Results of 17 Runs of Phased Paraclique on YST80

Ratio	Number of Paracliques	Avg. Size	Avg. Density
1/10	12	23.75	0.98
1/9	11	25.64	0.98
1/8	11	25.82	0.98
1/7	12	24.67	0.93
1/6	11	26.82	0.96
1/5	10	29.5	0.94
1/4	12	27.58	0.92
1/3	14	27.29	0.84
1/2	17	28.06	0.7
2/3	25	25.92	0.57
3/4	32	24.44	0.45
4/5	36	25.78	0.36
5/6	38	26.45	0.32
6/7	36	27.83	0.32
7/8	42	26.64	0.27
8/9	50	25.12	0.26
9/10	47	27.06	0.25

G and a parameter k. It then builds a cluster graph in which the clusters are maximal cliques of size k or greater. Edges are placed between maximal cliques having at least $k-1$ vertices in common. Output is the merged clusters. In practice multiple k values are computed simultaneously, so the user can decide which k best represents the data.

6.2.2 Clique difference algorithm

Clique difference defines edges in terms of the *difference* between clusters. Clique difference does not have a size cutoff for clusters, so the metric must be defined to handle properly clusters whose size greatly differs. A simple approach with a constant overlap value will not work. An approach based on percent overlap is also problematic. Smaller maximal cliques can chain together larger maximal cliques that should not be joined. Instead, clique difference uses the non-overlap between clusters. The non-overlap consists of those vertices not in the intersection. Clusters are merged only if the non-overlap does not exceed a certain value. Note that this avoids merging small and large maximal cliques since in order to merge them the non-overlap value must be greater than their size difference. This is a conservative approach, since it can be argued, for example, that a cluster and a much smaller counterpart that share all but one vertex should be joined. (A conservative approach, though, is in the spirit of applying clique-based methods to noisy data.)

Figure 6.3 displays the clique difference algorithm in pseudocode. The outer loop increments a difference value from one to a user-provided maximum. The inner loop builds a cluster graph of the current set of clusters, defines edges using the current difference value, and merges clusters. The loop halts once no more merging occurs. (The cluster graph is edgeless.) Thus, the idea is to merge clusters in multiple stages as incrementally as possible. Results are reported after each iteration of the outer loop, so that there is one set of results for each difference value, each set at least as condensed as the previous. Figure 6.4 are the results of running clique difference on the YST80 graph. Again, sets less than size 10 are excluded. Clique difference can be halted at any time. In this case, we halt just before the number of sets drops below 100 (101). At that point (right bar), the average cluster density is 0.45 and the average size is 61, which compares quite favorably to the results from paraclique.

6.3 Clique Relaxations

The strict nature of clique is its greatest asset. It provides highly-filtered cores upon which we have built useful and practical algorithms. Ironically, though, this is also a weakness in at least two ways. First, noise may prevent clique from finding a valid core. If a portion of this core is found, algorithms like paraclique can compensate by adding the additional needed nodes. However, if noise causes only a few edge deletions that happen to be widespread in a particular core, even an algorithm like

```
clique_difference(graph, max_difference) {
  clusters = all_maximal_cliques(graph);

  for (diff = 1; diff <= max_difference; diff++) {
    do {
      cgraph = build_cluster_graph(clusters, diff);
      components = connected_components(cgraph);
      clusters = merge_clusters(clusters, components);
    } while (number of clusters was reduced);

    print_current_clusters();
  }
}
```

Figure 6.3: Clique Difference Algorithm in Pidgin C++

paraclique will not locate it. The second weakness of clique is the need of such heuristic algorithms, like paraclique and clique difference, to perform additional processing on the results. Ideally we would have problems similar to clique with efficient exact algorithms that are themselves sufficient. In this section we abstract both vertex cover and clique by introducing two problems that are more general and could possibly address these weaknesses. Viewing clique from this vantage point opens up new possibilities for research.

6.3.1 Generalizations of vertex cover and clique

We begin by viewing vertex cover in a more abstract fashion. A vertex cover should more appropriately be called an “edge cover” as the key property of such a set is that it contains at least one vertex from every edge. (Edge cover is actually a different, unrelated problem.) In general, then, we could talk about covering any particular set of subgraphs of a given graph, not just edges. For example, we could attempt to cover all triangles of the graph by finding a set of vertices such that at least one vertex from every triangle is in the set. More generally, we can specify a set S of subgraph structures that must be covered. To prevent ambiguity, we mandate that only induced

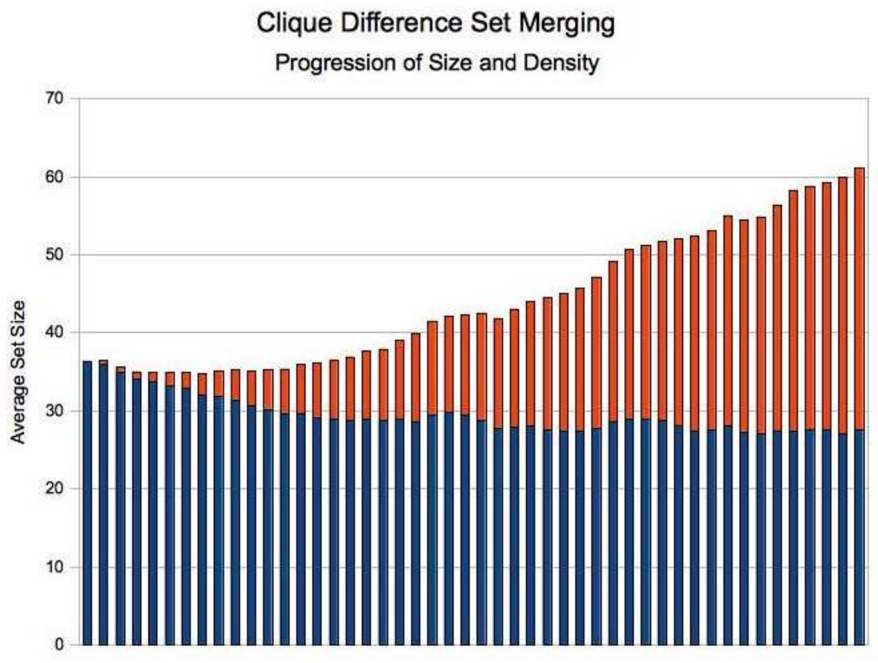


Figure 6.4: Clique difference begins with all maximal cliques (left bar) and iteratively merges them. In this case, there are initially 60882 maximal cliques, which are merged eventually to 101 sets (right bar). The average density in each case is the percentage of the bar that is blue. Even on the rightmost bar, the density is almost 50%.

subgraphs that exactly match those in S must be covered. For example, if we wish to cover all triangles and all paths of length 2 (three vertices), we must include both in S. This rule is critical in order to distinguish different, subtle variations. We define *General Subgraph Cover (GSC)* formally as follows:

General Subgraph Cover Problem

Input: A graph $G = (V, E)$, a positive integer $k \leq |V|$, and a set S of graphs to be covered.

Question: Does there exist a set of vertices $V' \subseteq$V such that $|V'| \leq k$ and V' contains at least one vertex from every induced subgraph of G isomorphic to some graph in S?

Note that vertex cover is equivalent to GSC if S contains only one subgraph, a clique of size 2. We can also define Minimum Genral Subgraph Cover (MGSC), the corresponding optimization problem. Clique can be viewed as a special case of an "avoidance" problem. That is, we are attempting to find a graph structure that avoids a certain induced subgraph, the independent set of size 2. Thus, we can generalize clique by specifying a set of forbidden subgraph structures. (We mandate again that we only forbid induced subgraphs that exactly match.) Then our goal is to find a graph structure that avoids any of the forbidden sets. Thus, we formally define *General Subgraph Avoidance (GSA)* as follows:

General Subgraph Avoidance Problem
Input: A graph $G = (V, E)$, a positive integer $k \leq |V|$, and a set S of graphs to be avoided.
Question: Does there exist a set of vertices $V' \subseteq$V such that $|V'| \geq k$ and V' contains no induced subgraphs isomorphic to a graph in S?

Note that clique is equivalent to GSA if S contains only one subgraph, the independent set of size 2. We can also define Maximum General Subgraph Avoidance (MGSA), the corresponding optimization problem. MGSC and MGSA are related in the same way that vertex cover and independent set are related. Converting an instance only requires complementing the input graph, not the set of graphs to be covered or avoided.

6.3.2 Examples of clique-like subgraph avoidance instances

By varying the set S that is input to GSA, specific clique-like problems can be created. Forbidding graphs of size one or two is either trivial or leads to clique and independent set, respectively. Here we consider forbidding graphs of size three, which leads to a couple of interesting problems. (Many more interesting cases may

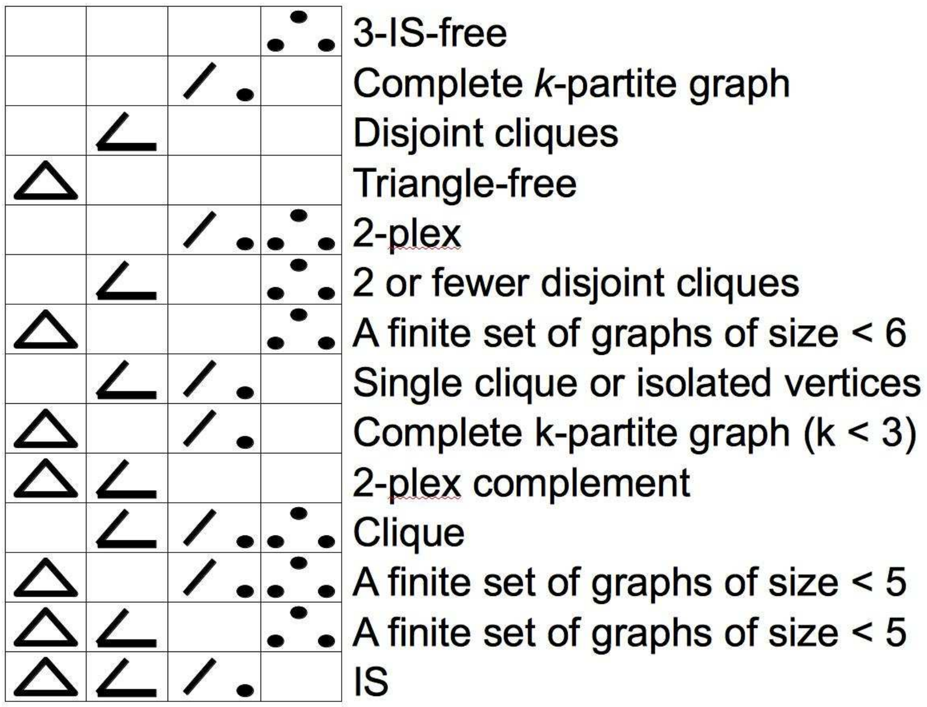

3-IS-free
Complete *k*-partite graph
Disjoint cliques
Triangle-free
2-plex
2 or fewer disjoint cliques
A finite set of graphs of size < 6
Single clique or isolated vertices
Complete k-partite graph (k < 3)
2-plex complement
Clique
A finite set of graphs of size < 5
A finite set of graphs of size < 5
IS

Figure 6.5: Graph classes defined by forbidding induced subgraphs of size 3. Each row is a pictorial list of forbidden subgraphs and the resulting graph class (the set of graphs not containing any of the forbidden subgraphs).

be found by forbidding larger subgraph sizes.) There are four graphs of size three within isomorphism, and hence, there are 14 possible sets of such graphs (ignoring the empty set and complete set). Figure 6.5 lists all 14 variants along with the resulting problem instance (class of graphs without the forbidden subgraphs). Many of the cases are not interesting, producing very restricted classes of graphs, some with only a finite number of members. Of those remaining, some are not helpful for finding dense graph regions, such as triangle-free. Two cases, though, are worthy of further discussion. The 2-plex problem is a special case of k-plex [5, 32], which relaxes clique by allowing missing edges but placing a limit on the number of missing edges per vertex. Research on the k-plex problem, originally introduced in the late 70s [40], has been scarce until recently. It is a seemingly more challenging problem than clique that could be applied to real data.

The 3-IS problem is another interesting variant. Note that it is the complement of a triangle-free graph. While triangle-free graphs are well-studied in the literature, algorithms for efficiently finding maximum subgraphs without a 3-IS, or more generally a k-IS, appear not to be. How would such an algorithm fair in uncovering dense subgraphs? While k-IS relaxes the clique problem even more than k-plex, a little analysis shows that it still maintains a high density, at least for small values of k, by the following result:

Claim: For graph $G = (V, E)$, $M_G = \frac{|V|*(|V|+1)}{2}$ is the maximum possible number of edges $|E|$ (vertex pairs) in graph G. If G is r-IS-free, $|E| \geq \frac{M_G}{r-1}$.

Proof: The proof is by contradiction. Turán's Theorem [48] states that a K_{r+1}-free graph (where K_i indicates a clique of size i) contains at most $(1 - \frac{1}{r}) * \frac{n^2}{2}$ edges, where n is the number of vertices. Suppose there exists a graph $G = (V, E)$ that is r-IS-free where $|E| < \frac{M_G}{r-1}$. Consider graph $\bar{G} = (\bar{V}, \bar{E})$, which is K_r-free. However, $|\bar{E}| = M_G - |E| > M_G - \frac{M_G}{r-1} = M_G * (1 - \frac{1}{r-1}) > (1 - \frac{1}{r-1}) * \frac{|V|^2}{2}$, contradicting Turán's Theorem.

Thus, a 3-IS-free subgraph must have at least 50% of edges adjacent, a 4-IS-free subgraph 33%, and so forth, giving a reasonable expectation that locating such subgraphs would provide highly dense cores.

6.3.3 Relation of subgraph cover to hitting set

The 3-hitting set problem is defined below, as copied from [35]:

3-Hitting Set Problem

Input: A collection C of subsets of size three of a finite set S and a positive integer k.

Question: Is there a subset $S' \subseteq S$ with $|S'| \leq k$ that allows S' to contain at least one element from each subset in C?

Any GSC instance where the set of graphs to be covered are of size 3 can be converted easily to a hitting set problem. This can be done by creating a subset for each triple of vertices to be covered. (Specifically, these subsets become parameter C, the graph vertices become S, and k remains the same.) Note that any GSA instance can also be converted to hitting set by first converting to GSC. (This is done by complementing the input graph, as described earlier.) More generally, any GSC or GSA instance can be converted to a general hitting set problem. Thus, the development of efficient, practical algorithms for hitting set would lead to a general approach for solving multiple GSC or GSA instances. This is one possible direction for future research.

Chapter 7

Conclusions

In this dissertation, we explored the idea of computing exact solutions to NP-complete problems in practice. We did this by focusing on one particular case, maximum clique, and attempting to engineer both a practical algorithm and a practical software implementation. Any such solution should be highly configurable, since the nature of NP-complete problems seems to be that different types of inputs require different strategies. We addressed this need by designing a system around basic algorithms (preprocessing and branching) and abstracting out various sub-algorithms. To implement this in software, we employed the strategy design pattern, which allows us to alter internal algorithms easily. This is done by passing the various functionality to the basic algorithms. The way we do this is somewhat counterintuitive, since the functions are C++ classes, which are normally thought of as "nouns" by OOP (object-oriented programming) programmers. A functional programming solution might be more natural. It is an interesting question whether or not current functional languages could be efficient enough, though, since practical solutions must also employ efficient underlying data structures. We were able to exploit the bitwise operations that C++ allows. A good implementation must also follow good software engineering practices. We separated the various algorithms into separate modules and provided layers of abstraction. Programmers of high-level MCF algorithms need not worry about the bitwise operations that make graph operations fast, since the graph library hides such details. Programmers of new preprocessing or sorting strategies need not bother with the details of how the basic preprocessing or branching code functions.

We also explored how to use the output of maximum clique and maximal clique to create practical algorithms for data mining and finding dense regions of graphs. Paraclique provides a mechanism to extend a maximum clique into a feasible dense region for further study. We introduced two new methods for taking advantage of cliques. Phased paraclique attempts to grow paracliques more gradually than the original algorithm. Clique difference compresses the multitude of maximal cliques into something more manageable. Algorithms based on clique, with its stingy requirement that all edges must be present, should have very few false positives, a plus when dealing with noisy data. Finally, we examined in detail a possible avenue for future research. Relaxations of clique may find dense regions of graphs that clique algorithms miss. At the same time, they can be shown to maintain high density. They could potentially do better than clique, possibly without the need for post-processing.

7.1 Future Research Directions

In this section, we outline some directions for future research. We look at possible directions for advancement from three perspectives: algorithms, applications, and implementations.

7.1.1 Algorithm advancement

MCF is a tool for research into different preprocessing and branching schemes. One interesting line of research then would be experimenting with new or known "clique tests," "vertex sorters," parameter settings, etc. for classes of graphs not discussed in this dissertation. The general preprocessing algorithm discussed in Section 3.3, once implemented, allows for quick testing of various preprocessing approaches with the amount of computation (depth) adjustable for each approach. This preprocessing can be done before branching or interleaved. Dense graphs are one example of a graph class that could be investigated. Possible approaches are discussed in [42]. It would also be interesting to develop algorithms and software along the same lines as MCF for other problems. The dominating set problem is one possibility. The practical benefits of having an efficient solver for dominating set are not clear, but there is reason for optimism. The vertices of a dominating set represent a minimum set of "hubs" that connect to all vertices in the graph. This has potential benefits for biological data, as well as other data that represents networks, since the hubs play a vital role. Of course, this research touches on all three areas (algorithms, applications, and implementations). Another research direction has already been mentioned, the development of algorithms for clique variants like k-plex and k-IS-free graphs, and also development of algorithms for hitting set, which could solve several clique-like problems at once.

7.1.2 Application advancement

We have seen various algorithms that employ a clique-finding engine for examining real-world data. There is much more research to be done, however. New variations on these algorithms are one possible research direction. Another is a more thorough analysis of their results, using additional data sets and biological metrics. We have primarily considered only transcriptomic data. Other types of biological data, such as protein-protein interaction (PPI) data, should also be tested. For graphs of PPI data, is the presented configuration of MCF for real data still the best? Does the behavior of the algorithms in Chapter 6 change? The application of clique variants is another interesting line of research, but good solvers must be available first. Finally, is it possible to prove some guarantees on the output of algorithms? For example, does phased paraclique or clique difference guarantee some lower bound on the density of the sets generated? Is it possible to specify more formally the informal graph types considered (real-data graphs and synthetic graphs) and define certain properties? If so, more guarantees on the output are possible, as well as a formal investigation of why certain algorithms perform better on certain graphs.

7.1.3 Implementation advancement

There are several avenues for advancing implementations of MCF and MCF-like tools. One avenue is to improve upon the parallel approach employed by MCF. We saw in Chapter 4 that MCF scales well on large jobs for over a hundred processors, at which point the gains from adding more processors begin to dwindle. Scaling MCF to thousands of processors would be an interesting challenge. Parallelizing the branching search tree is only one possibility. Would it be possible to parallelize the underlying graph operations? Perhaps vector processing or GPUs could be employed. Would it be worthwhile to consider parallelizing preprocessing? In Section 4.5.1 we mention delaying preprocessing until parallel branching as one way of coping with computationally-expensive preprocessors. Experimentation, though, would be required to see which approach is more effective. Further, if parallelizing preprocessing proves to be effective, would it be worth reconsidering more computationally-expensive preprocessors, such as color preprocessing at depth 2? Other approaches to solving maximum clique may be worth investigating. MCF uses an adjacency matrix, which is efficient but has some drawbacks. Since the space needed is quadratic with respect to vertices, very large graphs, say with millions of vertices, are not solvable with MCF. Adjacency matrices scale better for sparse graphs but can be slower. A combination of these two approaches may be worth exploring. One approach for solving large graphs has been explored in [38]. In this paper, MCF (called *Maximum Clique Solver (MCS)*) is used as a sub-program to solve smaller instances created by the main program. Another line of research is in implementing the practical algorithms on top of MCF and other clique-finding engines. This, however, is not that pressing in terms of implementation, because the clique-finding engine dominates the run time. So clever programming is not as critical.

7.2 Contributions

The specific contributions of this dissertation include the development of a general, configurable algorithm for computing maximum cliques. Key algorithmic decisions are isolated so that they can be easily altered. This includes a general preprocessing algorithm that can be configured to run multiple methods. A configurable, modular, and efficient implementation of these algorithms is created and described in detail. This description includes the high-level design for easy configuration, the low-level design for efficient data structures and for efficient computation of common graph operations, and an effective parallel processing approach. Experiments are performed to show how to configure the software for graphs of real data and for graphs of synthetic data. A new preprocessing approach based on coloring is presented and shown empirically to work well on graphs built from biological data. Analysis of practical algorithms based on clique are done. New practical algorithms are developed and implemented and preliminary testing performed. Finally, several directions for future research are proposed.

7.3 Parting Reflections

In this dissertation, we focused on solving a provably hard problem, clique, efficiently in practice. We then considered practical algorithms for analyzing real-world data, algorithms that are based on an efficient clique solver. On the surface, though, this approach to solving real-world problems seems anything but practical. Why would we attempt to build practical applications around a core that is NP-complete? It would seem more logical to apply heuristics or to work from a problem that is known to have a low time complexity. Clique, though, offers a guarantee that problems of seemingly lesser complexity do not. That is, 100% density is ensured. NP-complete problems in general tend to offer such rigorous guarantees. Thus, algorithms based on exact results from NP-complete problems may offer many benefits. This dissertation demonstrates that with good design and testing on cases of interest, such as real data, such problems can be solved efficiently for practical applications. Therefore, when NP-complete problems arise in practice, we have another possibility besides heuristics and approximation algorithms. Computing exact results and working around the complexity issues is a longer, riskier road, of course, but one that could hold enormous benefits.

Bibliography

Bibliography

[1] F. N. Abu-Khzam. *Topics in Graph Algorithms: Structural Results and Algorithmic Techniques, with Applications*. PhD thesis, University of Tennessee, 2003. 2, 36

[2] F. N. Abu-Khzam, M. A. Langston, P. Shanbhag, and C. T. Symons. Scalable parallel algorithms for FPT problems. *Algorithmica*, 45:269–284, 2006. 41, 45, 46

[3] J. S. Bader, A. Chaudhuri, J. M. Rothberg, and J. Chant. Gaining confidence in high-throughput protein interaction networks. *Nature Biotechnology*, 22:78–85, 2004. 50

[4] R. Balasubramanian, T. LaFramboise, D. Scholtens, and R. Gentleman. A graph-theoretic approach to testing associations between disparate sources of functional genomics data. *Bioinformatics*, 20(18):3353–3362, 2004. 50

[5] B. Balasundaram, S. Butenko, and I. V. Hicks. Clique relaxations in social network analysis: The maximum k-plex problem. *Operations Research*, 2010. To Appear. 68

[6] N. E. Baldwin, E. J. Chesler, S. Kirov, M. A. Langston, J. R. Snoddy, R. W. Williams, and B. Zhang. Computational, integrative, and comparative methods for the elucidation of genetic coexpression networks. *J Biomed Biotechnol*, 2(2):172–180, 2005. 41, 46

[7] N. E. Baldwin, R. L. Collins, M. A. Langston, M. R. Leuze, C. T. Symons, and B. H. Voy. High performance computational tools for motif discovery. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, 2004. 36

[8] C. Bron and J. Kerbosch. Algorithm 457: Finding all cliques of an undirected graph. *Commun. ACM*, 16(9):575–577, 1973. 45

[9] E. J. Chesler and M. A. Langston. Combinatorial genetic regulatory network analysis tools for high throughput transcriptomic data. In *RECOMB Satellite Workshop on Systems Biology and Regulatory Genomics*, 2005. 41, 46, 55, 58

[10] E. J. Chesler, L. Lu, S. Shou, Y. Qu, J. Gu, J. Wang, H. C. Hsu, J. D. Mountz, N. E. Baldwin, M. A. Langston, J. B. Hogenesch, D. W. Threadgill, K. F. Manly, and R. W. Williams. Complex trait analysis of gene expression uncovers polygenic and pleiotropic networks that modulate nervous system function. *Nature Genetics*, 37:233–242, 2005. 46

[11] Clusters & communities: Overlapping dense groups in networks. `http://www.cfinder.org`. 62

[12] F. Dehne, M. A. Langston, X. Luo, S. Pitre, P. Shaw, and Y. Zhang. The cluster editing problem: Implementations and experiments. In *Parameterized and Exact Computation*. Springer Berlin, 2006. 14

[13] Dimacs implementation challenges. `http://dimacs.rutgers.edu/Challenges/index.html`. 34

[14] R. G. Downey and M. R. Fellows. *Parameterized Complexity*. Springer, 1999. 7, 9, 41, 45

[15] J. D. Eblen, I. C. Gerling, A. M. Saxton, J. Wu, J. R. Snoddy, and M. A. Langston. Graph algorithms for integrated biological analysis, with applications

to type 1 diabetes data. In *Clustering Challenges in Biological Networks*. World Scientific, 2008. 28

[16] U. Feige, S. Goldwasser, L. Lovasz, S. Safra, and M. Szegedy. Approximating the maximum clique is almost np-complete. In *IEEE Symposium on the Foundations of Computer Science*, pages 2–12, 1991. 45

[17] U. Feige, D. Peleg, and G. Kortsarz. The dense k-subgraph problem. *Algorithmica*, 29:410–421, 2001. 46

[18] M. R. Fellows and M. A. Langston. Nonconstructive tools for proving polynomial-time decidability. *Journal of the ACM*, 35:727–739, 1988. 45

[19] M. R. Fellows and M. A. Langston. On search, decision, and the efficiency of polynomial-time algorithms. *Journal of Computer and Systems Sciences*, 49:769–779, 1994. 45

[20] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995. 25

[21] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. WH Freeman & Co., 1979. 7

[22] A. P. Gasch, M. Huang, S. Metzner, D. Botstein, S. J. Elledge, and P. O. Brown. Genomic expression responses to dna-damaging agents and the regulatory role of the yeast atr homolog mec1p. *Molecular Biology of the Cell*, 12:2987–3003, October 2001. 28

[23] I. C. Gerling, C. Ali, and N. Lenchik. Characterization of early developments in the splenic leukocyte transcriptome of NOD mice. *Ann N Y Acad Sci*, 1005:157–160, 2003. 42

[24] I. C. Gerling, S. Singh, N. I. Lenchik, D. R. Marshall, and J. Wu. New data analysis and mining approaches identify unique proteome and transcriptome

markers of susceptibility to autoimmune diabetes. *Mol Cell Proteomics*, 5(2):293–305, 2006. 41, 42, 52

[25] R. Ihaka and R. Gentleman. R: A language for data analysis and graphics. *Journal of Computational and Graphical Statistics*, 5:299–314, 1996. 49

[26] D. S. Johnson and M. A. Trick. Cliques, coloring, and satisfiability. second dimacs implementation challenge, volume 26 of dimacs series in discrete mathematics and theoretical computer science. *American Mathematical Society*, 1996. 34

[27] R. Kirova, M. A. Langston, X. Peng, A. D. Perkins, and E. J. Chesler. A systems genetic analysis of chronic fatigue syndrome: Combinatorial data integration from snps to differential diagnosis of disease. In *International Conference for the Critical Assessment of Microarray Data Analysis (CAMDA)*, 2006. 41

[28] H. Kitano. Computational systems biology. *Nature*, 420(6912):206–210, 2002. 41

[29] H. Kitano. Systems biology: A brief overview. *Science*, 295(5560):1662–1664, 2002. 41

[30] M. A. Langston, L. Lan, X. Peng, N. E. Baldwin, C. T. Symons, B. Zhang, and J. R. Snoddy. A combinatorial approach to the analysis of differential gene expression data: The use of graph algorithms for disease prediction and screening. In K. F. Johnson and S. M. Lin, editors, *Methods of Microarray Data Analysis IV, Papers from CAMDA '03*, pages 223–238. Kluwer Academic Publishers, 2005. 41, 46

[31] M. A. Langston, A. D. Perkins, A. M. Saxton, J. A. Scharff, and B. H. Voy. Innovative computational methods for transcriptomic data analysis. In *ACM Symposium on Applied Computing*, 2006. 46

[32] B. McClosky and I. V. Hicks. Combinatorial algorithms for max k-plex. *Journal of Combinatorial Optimization*, 2010. To Appear. 68

[33] S. Meyers. *Effective C++: 55 Specific Ways to Improve Your Programs and Designs*. Addison-Wesley, third edition, 2005. 25

[34] R. Niedermeier. *Invitation to Fixed-Parameter Algorithms*. Oxford, 2006. 7, 10

[35] R. Niedermeier and P. Rossmanith. An efficient fixed-parameter algorithm for 3-hitting set. *Journal of Discrete Algorithms*, 1(1):89–102, 2003. Combinatorial Algorithms. 70

[36] G. Palla, I. Derényi, I. Farkas, and T. Vicsek. Uncovering the overlapping community structure of complex networks in nature and society. *Nature*, 435:814–818, June 2005. 62

[37] E. Pennisi. Systems biology. tracing life's circuitry. *Science*, 302(5651):1646–1649, 2003. 41

[38] G. L. Rogers, A. D. Perkins, C. A. Phillips, J. D. Eblen, F. N. Abu-Khzam, and M. A. Langston. Using out-of-core techniques to produce exact solutions to the maximum clique problem on extremely large graphs. In *ACS/IEEE International Conference on Computer Systems and Applications (AICCSA'09)*. IEEE Computer Society, 2009. 74

[39] I. Seefeldt, G. Nebrich, I. Römer, L. Mao, and J. Klose. Evaluation of 2-de protein patterns from pre- and postnatal stages of the mouse brain. *Proteomics*, 6(18):4932–4939, 2006. 43

[40] S. B. Seidman and B. L. Foster. A graph theoretic generalization of the clique concept. *Journal of Mathematical Sociology*, 6:139–154, 1978. 68

[41] J. C. Setubal and J. Meidanis. *Introduction to Computational Molecular Biology*. PWS Publishing Company, 1997. 45

[42] E. C. Sewell. A branch and bound algorithm for the stability number of a sparse graph. *INFORMS Journal on Computing*, 10:438–447, 1998. 73

[43] L. Shi, L. H. Reid, W. D. Jones, R. Shippy, J. A. Warrington, S. C. Baker, P. J. Collins, F. de Longueville, E. S. Kawasaki, K. Y. Lee, Y. Luo, Y. A. Sun, J. M. Willey, R. A. Setterquist, G. M. Fischer, W. Tong, Y. P. Dragan, D. J. Dix, F. W. Frueh, F. M Goodsaid, D. Herman, R. V. Jensen, C. D. Johnson, E. K. Lobenhofer, R. K. Puri, U. Schrf, J. Thierry-Mieg, C. Wang, M. Wilson, P. K. Wolber, L. Zhang, W. Slikker, L. Shi, L. H. Reid, and M. A. Q. C. Consortium. The microarray quality control (maqc) project shows inter- and intraplatform reproducibility of gene expression measurements. *Nat Biotechnol*, 24(9):1151–1161, 2006. 43

[44] M. Sipser. *Introduction to the Theory of Computation*. Course Technology, 1996. 44

[45] H. E. Thomas and T. W. Kay. Beta cell destruction in the development of autoimmune diabetes in the non-obese diabetic (NOD) mouse. *Diabetes Metab Res Rev*, 16(4):251–261, 2000. 41

[46] J. Tian, A. P. Olcott, L. R. Hanssen, D. Zekzer, B. Middleton, and D. L. Kaufman. Infectious th1 and th2 autoimmunity in diabetes-prone mice. *Immunol Rev*, 164:119–127, 1998. 41

[47] E. Tomita and T. Kameda. An efficient branch-and-bound algorithm for finding a maximum clique with computational experiments. *Journal of Global Optimization*, 37:95–111, January 2007. 15, 32, 33, 34

[48] P. Turán. On an extremal problem in graph theory. *Matematikai és Fizikai Lapok*, 48:436–452, 1941. 69

[49] B. H. Voy, J. A. Scharff, A. D. Perkins, A. M. Saxton, B. Borate, E. J. Chesler, L. K. Branstetter, and M. A. Langston. Extracting gene networks for low-dose radiation using graph theoretical algorithms. *PLoS Comput Biol*, 2(7):e89, 2006. 45, 46

[50] M. T. Wayland and S. Bahn. *Chapter 5, Reproducibility of Microarray Studies: Concordance of Current Analysis Methods.*, volume 158, pages 109–125. ScienceDirect, 2006. 43

[51] D. B. West. *Introduction to Graph Theory*. Prentice Hall, 1996. 44

[52] A. M. Wheelock and A. R. Buckpitt. Software-induced variance in two-dimensional gel electrophoresis image analysis. *Electrophoresis*, 26(23):4508–4520, 2005. 43

[53] J. W. Yoon, H. S. Jun, and P. Santamaria. Cellular and molecular mechanisms for the initiation and progression of beta cell destruction resulting from the collaboration between macrophages and t cells. *Autoimmunity*, 27(2):109–122, 1998. 41

[54] J. H. Zar. *Biostatistical Analysis*. Prentice Hall, fourth edition, 1999. 4, 43

[55] Y. Zhang, F. N. Abu-Khzam, N. E. Baldwin, E. J. Chesler, M. A. Langston, and N. F. Samatova. Genome-scale computational approaches to memory-intensive applications in systems biology. In *Supercomputing*, 2005. 46

Appendix

Table A.1: Run Times in Seconds on DIMACS Graphs

Graph	MCF-RD	MCF-MCQ	MCQ
MANN_a27	>24 hours	9.89	8.13
MANN_a45	>24 hours	7579.66	7806.54
MANN_a81	>24 hours	>24 hours	>24 hours
brock400_1	>24 hours	3778.39	2514.06
brock400_2	>24 hours	4271.19	1087.12
brock400_3	>24 hours	2734.67	2181.75
brock400_4	>24 hours	556.65	965.56
hamming10-2	>24 hours	0.00	2.07
hamming10-4	>24 hours	>24 hours	>24 hours
hamming8-2	>24 hours	0.00	0.02
johnson16-2-4	17.74	1.34	0.69
johnson32-2-4	>24 hours	>24 hours	>24 hours
p_hat1000-1	56.47	3.23	2.38
p_hat1000-2	>24 hours	1646.52	1462.11
p_hat1000-3	>24 hours	>24 hours	>24 hours
p_hat700-1	7.71	0.43	0.32
p_hat700-2	10215.78	23.06	28.54
p_hat700-3	>24 hours	18229.50	18487.50
san1000	>24 hours	7.10	7.33
san400_0.5_1	34022.23	0.04	0.05
san400_0.7_1	>24 hours	3.63	3.07
san400_0.7_2	>24 hours	39.57	2.02
san400_0.7_3	>24 hours	9.54	9.27
san400_0.9_1	>24 hours	476.81	65.46
sanr400_0.5	47.25	3.44	1.95
sanr400_0.7	30773.06	953.24	628.98

Vita

John D. Eblen was born in Knoxville, Tennessee, USA. In 1999 he graduated from the University of Tennessee (UT) with a double major in computer science and mathematics. In 2001 he received an MS in computer science at UT under the direction of Dr. Michael W. Berry. He then worked in the bioinformatics industry for awhile developing algorithms and software for gene finding under the guidance of Dr. Ed C. Uberbacher and Dr. George J. Maalouf. In 2010 he received his Ph.D. in computer science at UT under the direction of Dr. Michael A. Langston.