

Algorithms for Bounding Folkman Numbers

by

Jonathan Coles

Master's Thesis submitted to the Faculty of the Computer Science
Department of the Rochester Institute of Technology in partial fulfillment
of the requirements for the degree of

Master of Science
in
Computer Science

March 11, 2005

Stanisław Radziszowski, Ph.D., Advisor

Christopher Homan, Ph.D., Reader

Rhys Price-Jones, Ph.D., Observer

Hans-Peter Bischof, Ph.D., Graduate Coordinator

Copyright ©2005 Jonathan Coles. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with Invariant Sections being “Acknowledgements,” “GNU Free Documentation License,” Front-Cover Texts being “Algorithms for Bounding Folkman Numbers,” “by Jonathan Coles,” and no Back-Cover Texts. A copy of the license may be found on <http://www.fsf.org>.

Acknowledgements

So many people have influenced who I am today. So many people have guided me to this point. The following is only a small subset of those who deserve my heartfelt thanks: Staszek, for his unwavering guidance during the last year. Hans-Peter, for all the extraordinary, life-changing opportunities he has given me. Rhys, for encouraging new ways of thinking and indirectly pushing me to explore my artistic side. Chris, for reviewing this work. Jen for graciously editing several drafts and pushing me towards the finish line. Sarah for being a friend for so many years. My parents and grandparents, for their constant encouragement and support throughout my life, but especially over the last six years. Thank you all so much.

Contents

1	Introduction	1
1.1	The Mathematician's Party	1
1.2	History	3
1.3	Definitions	4
1.4	Vertex Folkman Numbers	7
1.5	Edge Folkman Numbers	11
2	$F_v(2, 2, 3; 4)$	14
2.1	Overview	14
2.2	Testing for the Folkman Property	15
2.3	Computing the Solution	17
2.4	The Solution	19
2.5	Results	22
3	$F_e(3, 3; 4)$	24
3.1	Background	24
3.2	Methods	24
3.3	Experiments	26
3.4	Future Work	29
4	Final Thoughts	30
4.1	Technical Improvements	30
A	Tools	32
A.1	flib	32
A.2	nauty	34
A.3	Condor	34
A.4	SAT Solvers	35

List of Figures

1.1	A graph and its complement	4
1.2	Examples of K_3 , K_4 , and K_5 graphs	5
1.3	Mycielski graph	8
2.1	The Nenov graph Γ_3	15
2.2	Illustration of the extension process.	20
2.3	The Folkman graph F_{16} with $ \text{Aut}(G) = 16$	23
3.1	Sample DIMACS file	27
3.2	Phase transition of 3-SAT [32]	28
A.1	All graphs of order 3 output from running <i>geng 3</i>	34
A.2	Sample <i>Condor</i> job submission file	35

List of Tables

1.1	Exact known vertex Folkman numbers for two colors	9
1.2	Exact known vertex Folkman numbers for at least three colors	11
1.3	Edge triangle Folkman numbers	12
1.4	Known two-color edge Folkman numbers	13
1.5	Open edge Folkman number problems	13
2.1	Estimated processing times for algorithm INH2234	17
2.2	Properties of graphs in $H_v(2, 2, 3; 4; 14)$	22
2.3	Properties of Γ_3 and F_{16}	23

Abstract

For an undirected, simple graph G , we write $G \rightarrow (a_1, \dots, a_k)^v$ ($G \rightarrow (a_1, \dots, a_k)^e$) if for every vertex (edge) k -coloring of G , a monochromatic K_{a_i} is forced in some color $i \in \{1, \dots, k\}$. The vertex (edge) Folkman number is defined as

$$F_v(a_1, \dots, a_k; p) = \min\{|V(G)| : G \rightarrow (a_1, \dots, a_k; p)^v, K_p \not\subseteq G\}$$

$$F_e(a_1, \dots, a_k; p) = \min\{|V(G)| : G \rightarrow (a_1, \dots, a_k; p)^e, K_p \not\subseteq G\}$$

for $p > \max\{a_1, \dots, a_k\}$. Folkman showed in 1970 that these numbers always exist for valid values of p .

This thesis concerns the computation of a new result in Folkman number theory, namely that $F_v(2, 2, 3; 4) = 14$. Previously, the bounds stood at $10 \leq F_v(2, 2, 3; 4) \leq 14$, proven by Nenov in 2000. To achieve this new result, specialized algorithms were executed on the computers of the Computer Science network in a distributed processing effort. We discuss the mathematics and algorithms used in the computation. We also discuss ongoing research into the computation of the value of $F_e(3, 3; 4)$; the current bounds stand at $16 \leq F_e(3, 3; 4) \leq 3 \times 10^9$. The upper bound on this number was once the subject of an Erdős prize—claimed by Spencer in 1988.

Chapter 1

Introduction

This thesis has four main goals: First, to understand Folkman number theory. Second, to exploit the theory to solve particular problem instances using computers. Third, to develop efficient algorithms to perform the computation; this requires understanding the nature of the problems and how to structure programs so that the complexity of the algorithms is limited. And finally, to calculate a new Folkman number.

In order to understand Folkman number theory, it helps to have a mental picture of the problem. Keeping the following story in mind will help put some of the mathematics in perspective.

1.1 The Mathematician's Party

Fourteen people are invited to a party where some of these people know each other and some do not. It just so happens that no four people are mutual acquaintances. As the guests mingle, they wander between the living room and the kitchen. The host notices that regardless of who is in either room, there is always a room in which three people all know each other. Being a mathematician, the host finds this rather interesting, but doesn't give it too much thought until he is invited to a party a few weeks later.

At this party there are, again, fourteen guests where no four people are mutual acquaintances. He notices that there are three rooms, and that as the guests move around, at least one of the following is always true: There are at least two people in the kitchen who know each other; at least two people in the living room who know each other; or at least three people in

the library who know each other.

Now his interest is piqued further. There must be something to this other than sheer coincidence. Later that night he discovers that there is a whole field of mathematics called Folkman number theory which describes the situations he has observed. It explains why there had to be fourteen people at both parties. Had there been only thirteen, he never would have noticed the interesting patterns. The two parties represent perfect examples of what are called Folkman graphs. To understand what a Folkman graph is, it is first necessary to discuss the concept of a graph.

In the sense used here, a *graph* is a mathematical object consisting of a set of lines, called *edges*, and a set of points, called *vertices*. One can represent many things as graphs. For instance, one may use a graph to represent a map, where each city is a vertex and the roads connecting different cities are edges.

To translate the parties into *Folkman* graphs, represent each person by a vertex and connect two vertices with an edge if the two people know each other. Represent which room they are in by coloring each point. For the first party, there will be at most two colors, while for the second party there will be at most three colors. All the colors do not have to be used; it is acceptable for rooms to be empty.

Because the vertices are the objects that are colored, as opposed to the edges, this particular area of Folkman theory is called *vertex* Folkman theory. For the purposes of this story, *vertex* Folkman graphs will be referred to simply as Folkman graphs.

Returning to the party, it doesn't matter who gets which color: for the first party, no matter how the points are colored, there will always be, somewhere, three points that form a triangle and have the same color. For the second party, there will either be two connected points with the first color, two connected points with the second color, or three mutually connected points in the third color. This will be true regardless of how the points are colored. The conditions are forced because of the properties of the specific relationships of knowing and not knowing. Graphs where these kinds of conditions are forced, regardless of the coloring, are called Folkman graphs.

Thinking more abstractly, one can create any graph on any number of points and connect them in any way. However, it won't always be the case that the graph produced is a Folkman graph.

Folkman number theory asks the question: Given the number of rooms, the number of mutual acquaintances for each room, and the maximum num-

ber of mutual acquaintances allowed at the party, what is the fewest number of people so that no matter how the people move among the room, the acquaintance requirement is upheld? Also of interest is the question: Given the same information as the previous question plus a given number of people, what specific set of relationships (i.e., whether two people know or do not know each other) upholds the acquaintance requirements. In other words, what does the Folkman graph look like?

To describe the parties more concisely, all the information about what is being asked for in the two party examples can be described with the notation $F_v(3, 3; 4)$ and $F_v(2, 2, 3; 4)$, for the first and second parties, respectively. The party questions are then to what are $F_v(3, 3; 4)$ and $F_v(2, 2, 3; 4)$ equal? Answering that $F_v(2, 2, 3; 4) = 14$ —that fourteen people are required—is the main result of this thesis.

1.2 History

Folkman number theory has its origins in the study of Ramsey theory. In the classical form, Ramsey theory states that there exists a number $R(r, l)$ such that if the edges of a complete graph of order $\geq R(r, l)$ are two-colored, a monochromatic K_r will be forced in the first color, or a monochromatic K_l will be forced in the second color. Ideas from Ramsey theory have led to new insights into the structure of large mathematical objects, often in areas only indirectly related to graph theory.

Frank Plumpton Ramsey [15] first developed Ramsey theory in the late 1920s, although he based some of the work on theorems developed earlier by Schur [55] in 1916 and van der Waerden [61] in 1927. At only 26 years old, Ramsey died in 1930, and it wasn't until 1935 that Ramsey theory was rediscovered by Paul Erdős and George Szekeres. Erdős would go on to develop Ramsey theory into a completely new branch of mathematics. A thorough listing of known Ramsey numbers and bounds is given in Radziszowski's [50] survey on small Ramsey numbers. For more complete history on Ramsey theory and a survey of its applications, refer to the paper "Ramsey Theory Applications" by Vera Rosta [52].

Folkman number theory extends Ramsey theory by imposing restrictions on the maximum clique that a graph is allowed to contain. Jon Folkman [11] first showed in 1970 that all Folkman graph sets, under certain conditions, are non-empty. This momentous proof spurred an investigation into the so-

called Folkman *critical* graphs—the least ordered graphs in each set. These graphs are interesting because they have the property that dropping any vertex forces the graphs to lose the Folkman property.

The study of Folkman numbers concerns either *vertex* or *edge* Folkman graphs. Vertex Folkman graphs are colored by the vertices and edge Folkman graphs are colored by the edges. Most of the research in the area has focused on vertex Folkman graphs. A historical account of the more interesting points of both vertex and edge Folkman numbers follows, but in order to understand the theorems some definitions are required.

1.3 Definitions

Throughout this thesis, a graph G is a simple undirected graph with vertex set $V(G)$ and edge set $E(G)$. The complement of G is denoted by \overline{G} . The complement of a graph has all the vertices of the original, while the edges in the complement exist only if two points are *not* connected in the original. More precisely:

$$E(\overline{G}) = \{\{u, v\} : u, v \in V(G) \wedge \{u, v\} \notin E(G)\}$$

Figure 1.1 shows an example of a graph and its complement.

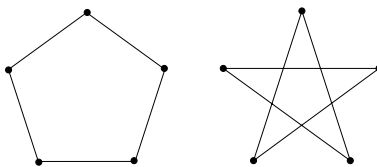


Figure 1.1: A graph and its complement

Two vertices u, v are adjacent in G if $\{u, v\} \in E(G)$. A complete graph K_n has n mutually adjacent vertices. Examples of K_3 , K_4 , and K_5 graphs are shown in Figure 1.2.

A graph with $n > 0$ vertices such that the edges form a cycle is denoted C_n . The pentagon in Figure 1.1 and the triangle in Figure 1.2 are examples for C_5 and C_3 graphs, respectively.

Given a vertex v in G , the neighborhood of v , $N_G(v)$, is the set of vertices adjacent to v . The minimum number of colors necessary to color the vertices

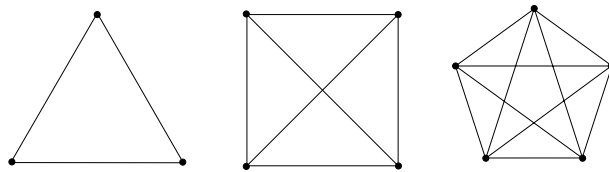


Figure 1.2: Examples of K_3 , K_4 , and K_5 graphs

of G such that no two vertices connected by an edge are colored the same is called the chromatic number of G and is denoted by $\chi(G)$. The maximal clique in G , $\text{cl}(G)$, is equal to the largest n such that K_n is a subgraph of G . Notice that it is always true that $\chi(G) \geq \text{cl}(G)$. The maximal independent set in G , $\alpha(G)$, is equal to the largest n such that K_n is a subgraph of \overline{G} . The set $\text{tri}(G)$ contains all K_3 subsets of G . In other words, all the triangles in G . The join $G = G_1 + G_2$ of graphs G_1 and G_2 is the union of G_1 and G_2 such that $V(G) = V(G_1) \cup V(G_2)$ and $E(G) = E(G_1) \cup E(G_2) \cup \{\{u, v\} : u \in V(G_1) \text{ and } v \in V(G_2)\}$.

The Ramsey graphs $\mathcal{R}(r, l; n)$ are graphs on n vertices with no clique of order r and no independent set of order l . The Ramsey number $R(r, l)$ is the smallest number n such that no graph of order $\geq n$ is a Ramsey graph. Another way to think about Ramsey numbers is that the Ramsey number is the least n such that for all edge 2-colorings of a complete graph on n vertices a monochromatic K_r is forced in the first color, or a monochromatic K_l is forced in the second color. Both descriptions are equivalent.

The notation $G \rightarrow (r, l)^v$ ($G \rightarrow (r, l)^e$), read G “arrows” (r, l) , means that for every vertex (edge) 2-coloring of G there is a K_r in the first color or a K_l in the second color. For positive integers a_1, \dots, a_k , $G \rightarrow (a_1, \dots, a_k)^v$ ($G \rightarrow (a_1, \dots, a_k)^e$) if and only if for every vertex (edge) k -coloring of G there exists a monochromatic K_{a_i} , for some color $i \in \{1, \dots, k\}$. In general, arrowing is Π_2^P -complete [53]. To see that arrowing is at least **NP**-complete, consider that $G \rightarrow (2, n)$ decides if G has a clique on n vertices.

It may be assumed that $a_i \geq 2$ because if $a_i = 1$ for some $i \in \{1, \dots, k\}$, then

$$G \rightarrow (a_1, \dots, a_k)^{v(e)} \iff G \rightarrow (a_1, \dots, a_{i-1}, a_{i+1}, \dots, a_k)^{v(e)} .$$

For $p > \max\{a_1, \dots, a_k\}$, the vertex (edge) Folkman graph families are

defined by

$$H_v(a_1, \dots, a_k; p) = \{G \rightarrow (a_1, \dots, a_k)^v, K_p \not\subseteq G\} .$$

$$H_e(a_1, \dots, a_k; p) = \{G \rightarrow (a_1, \dots, a_k)^e, K_p \not\subseteq G\} .$$

In 1970, Folkman [11] showed that these sets are non-empty. Also, let $H_{v(e)}(a_1, \dots, a_k; p; n)$ denote the set of Folkman graphs on n vertices.

A vertex (edge) Folkman number is defined by

$$F_v(a_1, \dots, a_k; p) = \min\{|V(G)| : G \in H_v(a_1, \dots, a_k; p)\}$$

$$F_e(a_1, \dots, a_k; p) = \min\{|V(G)| : G \in H_e(a_1, \dots, a_k; p)\}$$

Some basic properties cited in [42] are stated below.

Definition 1.3.1 Given integers a_1, \dots, a_k , $a_i \geq 2$, let

$$m = \sum_1^k (a_i - 1) + 1 \text{ and } a = \max\{a_1, \dots, a_k\} .$$

Proposition 1.3.1 If m satisfies Definition 1.3.1, then $K_m \rightarrow (a_1, \dots, a_k)^v$

This is a simple consequence of the pigeon-hole principle: Take K_m and group the vertices into k parts; one part must contain a K_{a_i} in part i for some $i \in \{1, \dots, k\}$.

Proposition 1.3.2 For any $r \geq 2$, $G \rightarrow \underbrace{(2, \dots, 2)}_r^v \iff \chi(G) \geq r + 1$.

If $G \rightarrow \underbrace{(2, \dots, 2)}_r^v$ then for every r -coloring there are always two neighboring vertices with the same color. This obviously implies that the chromatic number for that graph is $\geq r + 1$. More generally, there is Proposition 1.3.3:

Proposition 1.3.3 If $G \rightarrow (a_1, \dots, a_k)^v$, then $\chi(G) \geq m$.

Proposition 1.3.4 Let $G \rightarrow (a_1, \dots, a_k)^v$ and $\{b_1, \dots, b_t\} \subseteq \{a_1, \dots, a_k\}$. Then $G \rightarrow (b_1, \dots, b_t)^v$.

Proposition 1.3.5 Let $A \subseteq V(G)$ be an independent set of G and $G_1 = G - A$. Let $G \rightarrow (a_1, \dots, a_k)^v$. Then $G_1 \rightarrow (a_1, \dots, a_i - 1, \dots, a_k)^v$.

Proposition 1.3.6 For any permutation φ of the symmetric group S_r ,

$$G \rightarrow (a_1, \dots, a_k)^v \iff G \rightarrow (a_{\varphi(1)}, \dots, a_{\varphi(k)})^v .$$

The order may not matter, but it is common to write a_1, \dots, a_k in increasing order: it is often assumed that $\max\{a_1, \dots, a_k\} = a_k$.

1.4 Vertex Folkman Numbers

For this section, many theorems will make reference to m , which is assumed to satisfy Definition 1.3.1.

The study of vertex Folkman number theory has progressed as have many mathematical studies: beginning with simple cases and slowly imposing restrictions. Let K_q , $q \geq 1$, be the forbidden graph. The simplest case in vertex Folkman numbers is where there is no clique restriction; in other words, $q > m$. From Proposition 1.3.1, K_m is the smallest graph such that $K_m \rightarrow (a_1, \dots, a_k)^v$.

A natural question to ask is what happens when $q = m$. In 1996 Łuczak et al. [24] showed that for $q = m$, $F_v(a_1, \dots, a_k; q) = a_k + q$. They also showed that the critical graph is $K_{a_k+m} - C_{2a_k+1}$.

Most recent work has since focused on the cases where $q = m - 1$. There is much less work that considers the cases for $q < m - 1$. In 1995, with the help of a computer search, Jensen and Royle [20] showed that $F_v(2, 2, 2, 2; 3) = 22$. This is the only known vertex Folkman number where $q = m - 2$.

Proving the exact value of a given Folkman number is very challenging. More often than not, proofs only give general bounds, but leave the exact value unknown. To prove an upper bound on a Folkman number, it suffices to find any single Folkman graph. To prove the lower bound, however, can be much more difficult, as it requires a proof that shows that there does not exist any Folkman graph with fewer vertices than the lower bound. Functions that give exact Folkman numbers are rare and difficult to come by. Completing all the cases for Theorem 1.4.1 spanned more than two decades.

Theorem 1.4.1 *For any $r \geq 3$,*

$$F_v(\underbrace{2, \dots, 2}_r; r) = \begin{cases} 11 & r = 3 \text{ or } r = 4 \\ r + 5 & r \geq 5 \end{cases}$$

Mycielski [33] first gave an 11-vertex graph $G \in H_v(2, 2, 2; 3)$, Figure 1.3, in 1955 which Chvátal [6] later used in 1974 to show that $F_v(2, 2, 2; 3) = 11$. Nenov [40, 39, 36] showed in the early 1980s that $F_v(2, 2, 2, 2; 4) = 11$. Not until 1997 did Łuczak et al. [23] show that $F_v(\underbrace{2, \dots, 2}_r; r) = r + 5, r \geq 5$. This

last result the authors claim to be a “folklore result: people either know it, or can prove it overnight.” The proof is reproduced here to give an impression of how manual proofs of Folkman numbers may be presented:

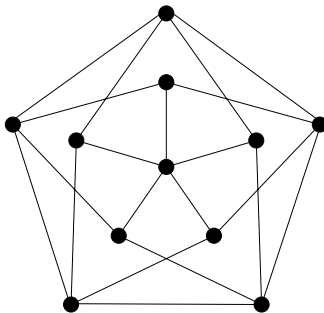


Figure 1.3: Mycielski graph—the smallest 4-chromatic triangle-free graph

Theorem 1.4.2 [23] *For every integer, $r \geq 5$,*

$$F_v(\underbrace{2, \dots, 2}_r; r) = r + 5 .$$

Proof. From [46], $G = K_{r-5} + C_5 + C_5$ has no K_r and $\chi(G) \geq r + 1$, hence $F_v(2, \dots, 2; r) \leq r + 5$. For the lower bound, consider the following. Let G be a K_r -free graph on $r + 4$ vertices. We shall show that $\chi(G) \leq r$. Let M be a maximal matching in \overline{G} . Since $G \not\supset K_r$, the matching M consists of at least three edges. If $|M| \geq 4$ then it is easy to properly color G with at most r colors: just assign the same color to the endpoints of each edge in M .

This takes $|M|$ colors, and the remaining $r + 4 - 2|M|$ vertices are colored each with a different color. Assume thus, that M consists of exactly three edges $\{u_i, v_i\}$, $i = 1, 2, 3$. The remaining $r - 2$ vertices form an independent set I in \overline{G} . If for some i , both u_i and v_i send an edge into I , then either there is another matching in \overline{G} of size 4, or there is a triangle consisting of u_i, v_i and a vertex in I , and again $\chi(G) \leq r$. Let S be the subset of $V(M)$ consisting of the vertices which do not send an edge into I . Since $G \not\supset K_r$, the induced subgraph $\overline{G}[S]$ must be a clique. Hence, for $|S| \geq 4$ we have $\chi(G) \leq (6 - |S|) + (r - 2) \leq r$. In view of the above remark, the only case left is when $S = \{v_1, v_2, v_3\}$. Then all u_1, u_2 , and u_3 have the same unique neighbor w in I . (Otherwise, again, there would be a matching of size 4). This, in turn, implies, that there is at least one edge in the induced subgraph $\overline{G}[u_1, u_2, u_3]$. Otherwise, the set $I \setminus \{w\} \cup \{u_1, u_2, u_3\}$ would form a clique K_r in G . Assume that $\{u_1, u_2\}$ is an edge of \overline{G} . Then a proper r -coloring of

G can be obtained by coloring v_1, v_2, v_3 by one color, u_1, u_2, w by a second color, and the remaining vertices by $r - 2$ different colors. \square

Nenov, in a paper from 2000 [42], laid out a general lower bound shown as Theorem 1.4.3.

Theorem 1.4.3 *Let a_1, \dots, a_k be positive integers. Let a and m satisfy (1.3.1) and $m \geq a + 2$. Then $F_v(a_1, \dots, a_k; m - 1) \geq m + a + 2$.*

In the same paper, Nenov provided a number of general bounds, a few of the more interesting ones are presented below. There are very few known values.

Theorem 1.4.4 *If $p \geq 3$, then $2p + 4 \leq F_v(3, p; p + 1) \leq 4p + 2$.*

$F_v(3, p; p + 1)$ falls into the category of two-colored graphs. Two colored graphs are especially interesting from a computer perspective. Each vertex can have only one of two colors, so the colors can be treated as binary numbers, combinations of which are very easy to represent in a computer¹. Some exact two-colored results are shown in Table 1.1.

Folkman number	References
$F_v(2, p; p + 1) = 2p + 1 \quad p \geq 2$	[24]
$F_v(3, 3; 4) = 14$	[38, 49]
$F_v(3, 4; 5) = 13$	[44]
$F_v(4, 4; 6) = 14$	[34]

Table 1.1: Exact known vertex Folkman numbers for two colors

When $p > 4$ the lower bound from Theorem 1.4.4 pushes the number of graphs a computer has to analyze beyond current capabilities. Attacking these problems will require clever hand proofs or much more powerful computer power combined with efficient algorithms.

Theorem 1.4.5 *For any $p \geq 3$ and $r \geq 2$,*

$$2p + r + 2 \leq F_v(\underbrace{2, \dots, 2}_r, p; p + r - 1) \leq 4p + r$$

¹Appendix A discusses details of how graphs can be represented in a computer.

Let $p = 3$ and $r = 2$, the simplest case. Then $10 \leq F_v(2, 2, 3; 4) \leq 14$. In his paper, where this theorem is stated, Nenov writes, “The exact value of $F_v(2, 2, 3; 4)$ is unknown.” The main result from the current work, discussed in Chapter 2, is to prove that $F_v(2, 2, 3; 4) = 14$.

When $p = 4$, the upper bound can be reduced:

Theorem 1.4.6 [42] *For any $r \geq 2$,*

$$r + 10 \leq F_v(\underbrace{2, \dots, 2}_r, 4; r + 3) \leq r + 11$$

Theorem 1.4.7 [42] *Let $p \geq 3$ and $r \geq 1$. Then*

$$2p + 2r + 2 \leq F_v(\underbrace{3, \dots, 3}_r, p; 2r + p - 1) \leq 4p + 2r$$

As with Theorem 1.4.5, when $p = 4$, the upper bound can be reduced:

Theorem 1.4.8 [42] *For $r \geq 1$*

$$2r + 10 \leq F_v(\underbrace{3, \dots, 3}_r, 4; 2r + 3) \leq 2r + 11$$

Theorem 1.4.9 [42] *Let $r \geq 2$. Then*

$$F_v(\underbrace{3, \dots, 3}_{r+1}; 2r + 2) \leq 2r + 10$$

In a later paper in 2003 [46], Nenov improved upon this and found an exact value for a general case:

Theorem 1.4.10 *Let $r \geq 3$. Then*

$$F_v(\underbrace{3, \dots, 3}_r; 2r) = 2r + 7$$

The ability to use computers to solve any of the above problems for specific values is significantly hampered by the number of graphs that exist with ≥ 13 vertices. The time required to analyze all the graphs is beyond the means of present day computers. Also, with three or more colors, the

complexity of the algorithm increases, which increases the time to analyze a *single* graph, let alone all the graphs with a given number of vertices.

There are, however, a few values that do lie within the capabilities of computer searches or even human trial and error. The main result of this thesis is the computation and value of $F_v(2, 2, 3; 4)$. Very few vertex Folkman numbers are known for three colors, and this result adds one more to the list. Some known values for Folkman numbers with at least three colors are shown in Table 1.2.

Folkman number	References
$F_v(2, 2, 4; 5) = 13$	[43]
$F_v(2, 2, 2, 4; 6) = 14$	[35]
$F_v(2, 3, 4; 6) = 14$	[35]
$F_v(2, 2, 3; 4) = 14$	[this work]

Table 1.2: Exact known vertex Folkman numbers for at least three colors

1.5 Edge Folkman Numbers

Most of the literature has focused on vertex Folkman numbers. The abundance of generalized bounds shown in the previous section are not common for edge Folkman numbers. There are, however, some results and open problems that should be discussed.

Edge Folkman numbers are closely related to Ramsey numbers.

Lemma 1.5.1 *If the forbidden clique has order $q > R(r, l)$ then $F_e(r, l; q) = R(r, l)$.*

Proof. Recall the definition of a Ramsey number: The least n such that for all edge 2-colorings of a K_n a monochromatic K_r is forced in the first color, or a monochromatic K_l is forced in the second color. The edge Folkman number is the least n such that there exists a graph not containing a K_q subgraph and for all edge 2-colorings a monochromatic K_r is forced in the first color or a monochromatic K_l is forced in the second color. Ramsey theory says this graph is $K_{R(r,l)}$. Hence, $F_e(r, l; q) = R(r, l)$ and $K_{R(r,l)}$ is the critical graph. \square

When $q \leq R(r, l)$ the situation becomes more difficult. Lin [22] showed in 1972 that certain generalizations could be made:

Theorem 1.5.1 $F_e(a_1, \dots, a_k; R(a_1, \dots, a_k)) \geq R(a_1, \dots, a_k) + 2$ with equality holding if and only if $G^* \rightarrow (a_1, \dots, a_k)^e$, where $G^* = C_5 + \overline{K}_{R(a_1, \dots, a_k) - 3}$.

Theorem 1.5.2 $F_e(a_1, \dots, a_k; R(a_1, \dots, a_k) - 1) \geq R(a_1, \dots, a_k) + 4$

There is also a relationship between edge and vertex Folkman numbers. In a two-color scenario []:

$$F_e(r, l; n) \leq F_v(R(r - 1, l), R(r, l - 1); n - 1) + 1 .$$

As with vertex Folkman numbers, the difficulty arises when cliques of a given order are forbidden. Table 1.3 shows a history of triangle edge Folkman numbers, so called because the corresponding Folkman graphs cannot be split into two or more triangle-free parts.

Folkman number	Critical graph(s)	Reference
$F_e(3, 3; \geq 7) = 6$	K_6	Lemma 1.5.1
$F_e(3, 3; 6) = 8$	$C_5 + K_3$	[14]
$F_e(3, 3; 5) = 15$	659 graphs	[49]
$F_e(3, 3; 4) \leq 3 \times 10^9$	probabilistic	[49]
$F_e(3, 3, 3; 17) = 19$	$C_5 + \overline{K}_{14}$	[22]
$F_e(3, 3, 3; 16) \geq 21$		[22]

Table 1.3: Edge triangle Folkman numbers

The history of $F_e(3, 3; 4)$ is particularly interesting. The problem has existed since the late 1960s. A more detailed history is discussed in chapter 3. Other two-colored Folkman numbers are shown in Table 1.4.

There are still many open problems. Some particular ones are listed in Table 1.5.

Folkman number	Critical graph	Reference
$F_e(3, 4; \geq 10) = 9$	K_9	Lemma 1.5.1
$F_e(3, 4; 9) = 14$	$K_4 + C_5 + C_5$	[41]
$F_e(3, 5; 14) = 16$	$C_5 + \overline{K}_{11}$	[22]
$F_e(4, 4; 18) = 20$	$C_5 + \overline{K}_{15}$	[22]

Table 1.4: Known two-color edge Folkman numbers

Folkman number	Reference
$27 \leq F_e(3, 7; 22) \leq \text{Unknown}$	[22]
$21 \leq F_e(3, 3, 3; 16) \leq \text{Unknown}$	[22]
$14 \leq F_e(3, 4; 8) \leq 314$	[37, 23]
$16 \leq F_e(3, 3; 4) \leq 3 \times 10^9$	[49]

Table 1.5: Open edge Folkman number problems

Chapter 2

$F_v(2, 2, 3; 4)$

2.1 Overview

In 2000, Nenov [42] showed that $10 \leq F_v(2, 2, 3; 4) \leq 14$; he proved the upper bound using the 14-node graph Γ_3 , depicted in Figure 2.1. No graph with fewer than 14 vertices is known to exist in $H_v(2, 2, 3; 4)$. It is shown here that there are *no* such graphs and thus, $F_v(2, 2, 3; 4) = 14$.

Proving exact values of Folkman numbers by hand is often very difficult since deriving the lower bound requires a non-existence proof. Computers can be of great help, but because showing non-existence often entails large searches, the algorithms must be carefully designed to work as efficiently as possible.

When using computers to prove theorems, it is important that one has high confidence in the results from the algorithms. There are at least two ways to do this. First, different algorithms can be implemented that achieve the same result. If the results agree, then it is highly likely that the results are correct. The second way to achieve confidence in the results, is to have two different people implement the algorithms. Both methods were used for this proof.

To find the exact value of $F_v(2, 2, 3; 4)$ with the aid of computers, there are two main approaches. One approach is to check all graphs of order < 14 for inclusion in $H_v(2, 2, 3; 4)$, starting with graphs of order 10. If a graph is found to be Folkman, then the current order is the Folkman number. If not, the next higher order must be checked. A second approach is to find all Folkman graphs on 14 vertices and drop a vertex from each one in all

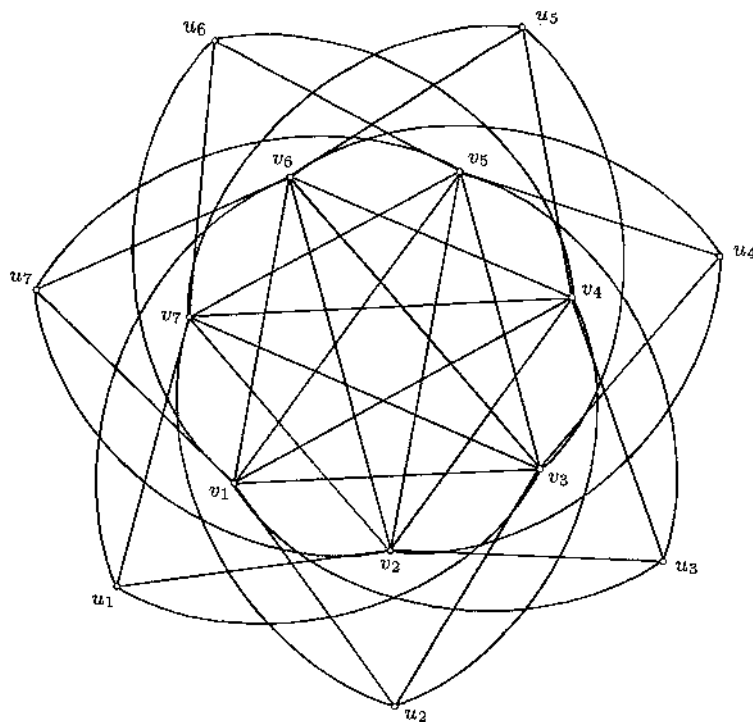


Figure 2.1: The Nenov graph Γ_3 taken from [42].

possible ways. If one of the resulting graphs is Folkman, then 13 is the new upper bound and the process can be repeated on the Folkman graphs with 13 vertices. Once no smaller Folkman graphs are obtained, the Folkman number has been found.

2.2 Testing for the Folkman Property

Regardless of the approach, we must have an algorithm for testing whether or not a graph is in $H_v(2, 2, 3; 4)$. From the set parameters, any graph that contains K_4 can be discarded. For the remaining graphs, any colorings of a graph that don't have a K_2 in the first or second colors must force a K_3 in the third, otherwise the graph is not Folkman. Such colorings occur when two independent sets are colored with the first and second colors, forcing any remaining vertices not in the union of the two independent sets to contain a

Algorithm 2.2.1: INH2234(G)

```

if  $K_4 \subseteq G$ 
  then return ( false )
 $M = \text{ALLMAXCLIQUES}(\overline{G})$ 
 $T = \{\{a, b, c\} : a, b, c \in V(G) \text{ and } abc \text{ is a triangle in } G\}$ 
for each  $A, B \in M$ 
  do  $\left\{ \begin{array}{l} C \leftarrow V(G) \setminus (A \cup B) \\ \text{if } C \text{ doesn't contain a triangle in } T \\ \text{then return ( false )} \end{array} \right.$ 
return ( true )

```

K_3 in the third color.

The algorithm for determining if $G \in H_v(2, 2, 3; 4)$ is fairly straightforward: Discard G if it contains a K_4 . Otherwise, check that for each pair of independent sets $A, B \in G$, the induced subgraph on $V(G) \setminus (A \cup B)$ contains a K_3 . A pseudo-code outline of the algorithm is presented as Algorithm 2.2.1, INH2234. For this to be efficient, we need Lemma 2.2.1.

Lemma 2.2.1 *It is sufficient to consider only maximal independent sets when determining whether $G \in H_v(2, 2, 3; 4)$.*

Proof. Let A, B be maximal independent sets in G and let $V(G) \setminus (A \cup B)$ induce a subgraph \mathcal{S} in G . In order for $G \in H_v(2, 2, 3; 4)$, \mathcal{S} must contain a K_3 . Now let $A' \subset A$ and $B' \subset B$. Since K_3 is a subgraph of \mathcal{S} , it follows that the induced subgraph on $V(G) \setminus (A' \cup B')$ also contains a K_3 . Thus, in Algorithm 2.2.1 it is sufficient to consider only the maximal independent sets A and B . \square

For graphs of order 14, the number of maximal independent sets is usually very small (around 40), so the complexity of the algorithm is not a great obstacle. The algorithm ALLMAXCLIQUES [21] is used to find these sets. Although ALLMAXCLIQUES returns maximal cliques, if the input is \overline{G} , the result will be maximal independent sets in G .

For efficiency, it is also necessary to have a precomputed table of triangles in G . This table is used to check if an induced subgraph contains a triangle.

The table is not very large, with an upper bound of 364 elements for K_{14} . The algorithm to build the table is a simple triple-nested brute force search over all vertices. With at most 14 vertices, the $O(n^3)$ complexity is insignificant.

2.3 Computing the Solution

To examine a single graph, running INH2234 is virtually instantaneous. However, the number of graphs that need to be examined explodes as the order increases. Table 2.1 shows how the number of graphs increases and predicted processing time using a 1 GHz Pentium III CPU.

$ V(G) $	# of Graphs	Total Time
8	12346	0.97 seconds
9	274668	17.51 seconds
10	12005168	~9 minutes
11	1018997864	~14 hours
12	165091172592	~96 days
13	50502031367952	~80 years
14	29054155657235488	~46,000 years

Table 2.1: Estimated processing times for algorithm INH2234

Because of the time requirements it is impractical to check all the graphs on 12 or more vertices. Although parallelizing the search using many computers is possible, it would not be a practical solution for graphs on even 13 vertices.

There is an alternative, however, that avoids searching all graphs on less than 14 vertices. If it were possible to generate all the Folkman graphs on 14 vertices, the existence of Folkman graphs on 13 vertices could be decided by dropping a vertex in all possible ways from each graph and testing for the Folkman property.

We write $G - v$ to denote a graph with deleted vertex v and edges incident to v . Let

$$S = H_v(2, 2, 3; 4; 14)$$

and

$$D = \{G - v : G \in S\} ,$$

then there is a Folkman graph on 13 vertices if and only if

$$D \cap H_v(2, 2, 3; 4; 13) \neq \emptyset .$$

The difficulty lies in generating S . It is easy to test if a graph is in S , but finding all of them could require testing all graphs on 14 vertices, which has already been shown to be impractical.

Some of the maximal Folkman graphs can be retrieved from the set of Ramsey graphs $\mathcal{R}(4, 4; 14)$. This set contains graphs on 14 vertices which have neither K_4 nor $\overline{K_4}$. The complete set can be found on Brendan McKay's website [27]. Of the the 130,816 Ramsey graphs, 1,507 are Folkman graphs.

For each Folkman graph, edges can be added in all possible ways, making sure to exclude K_4 . The result is a set of maximal Folkman graphs. More precisely, let

$$R = \mathcal{R}(4, 4; 14) \cap S$$

$$T = \{G' : \exists G \in R, E(G) \subseteq E(G'), V(G) = V(G') \wedge K_4 \not\subseteq G'\}$$

$$T_{max} = \{G \in T : \text{for each } e \in E(\overline{G}), K_4 \in G + e\}$$

Algorithm EXTENDTOMAXIMALS shows the pseudo code for this operation.

The minimal Folkman graphs, using algorithm REDUCETOMINIMALS ¹, are found in a similar way:

$$R = \mathcal{R}(4, 4; 14) \cap S$$

$$U = \{G - e : G \in R \wedge (G - e) \in S\}$$

$$T_{min} = \{G \in U : \text{for each } e \in E(G), G - e \notin U\}$$

The process can be performed repeatedly to discover new Folkman graphs; taking the minimal graphs, extending them to maximal graphs and then reducing the maximal graphs back to minimal graphs. Eventually the process will stop when either all Folkman graphs are found, or when an extension does not yield new maximal graphs.

¹An algorithm for efficiently discovering new maximal independent sets when a vertex is dropped from a graph was needed for the implementation of REDUCETOMINIMALS. The algorithm can be seen in Appendix A.

Algorithm 2.3.1: EXTENDTOMAXIMALS(G)

```

 $max \leftarrow \mathbf{true}$ 
for all  $u, v \in V(G), u \neq v$ 
  do  $\left\{ \begin{array}{l} \mathbf{if} \{u, v\} \notin E(G) \\ \mathbf{then} \left\{ \begin{array}{l} \mathbf{let} W \text{ be a graph} \\ V(W) \leftarrow N_G(u) \cup N_G(v) \cup \{u\} \cup \{v\} \\ E(W) \leftarrow \{\{x, y\} : x, y \in V(W) \wedge \{x, y\} \in E(G)\} \\ \mathbf{if} K_4 \not\subseteq W \\ \mathbf{then} \left\{ \begin{array}{l} max \leftarrow \mathbf{false} \\ \text{EXTENDTOMAXIMALS}(G + \{u, v\}) \end{array} \right. \end{array} \right. \end{array} \right.$ 
if  $max = \mathbf{true}$  and  $K_4 \not\subseteq G$ 
  then output ( $G$ )

```

There is no guarantee that this process will yield all the Folkman graphs on 14 vertices, but it was thought that it would give insight into the distribution of the graphs. Of the graphs generated by the end of the process, Γ_3 was not amongst the results, ruling out the possibility that the process had yielded all the Folkman graphs.

2.4 The Solution

There is, however, a better plan of attack to generate all Folkman graphs than experimenting. The Ramsey number $R(3, 4) = R(4, 3) = 9$ guarantees that all graphs of order ≥ 9 have either a K_4 or a $\overline{K_3}$. The Folkman graphs in $H_v(2, 2, 3; 4; 14)$ clearly do not have a K_4 , so they must have a $\overline{K_3}$. Using this observation, all graphs without K_4 on 11 vertices can be extended to graphs on 14 vertices by connecting three independent vertices to the triangle-free subsets of each graph in all possible ways. It is necessary to avoid subsets with triangles so that a K_4 is not formed. Figure 2.2 gives an illustration. The algorithm is called EXTEND:

1. For each K_4 -free graph on 11 vertices, do steps 2, 3, 4 below. The graphs are generated using *geng* from the *nauty* software package [27] and then filtered for those without K_4 . [easy]

Algorithm 2.3.2: REDUCETOMINIMALS(G)

```
 $min \leftarrow \mathbf{true}$   
for all  $u, v \in V(G), u \neq v$   
  do  $\left\{ \begin{array}{l} \mathbf{if} \{u, v\} \in E(G) \\ \quad \mathbf{then} \left\{ \begin{array}{l} \mathbf{if} \text{INH2234}(G - \{u, v\}) \\ \quad \mathbf{then} \left\{ \begin{array}{l} min \leftarrow \mathbf{false} \\ \text{REDUCETOMINIMALS}(G - \{u, v\}) \end{array} \right. \end{array} \right. \end{array} \right.$   
if  $min = \mathbf{true}$  and  $\text{INH2234}(G)$   
  then output ( $G$ )
```

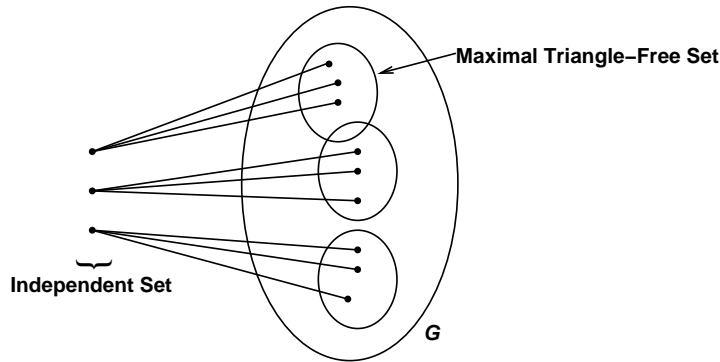


Figure 2.2: Illustration of the extension process.

2. Extend graph to 14 vertices by efficiently adding $\overline{K_3}$ with incident edges. Each new vertex is incident with a maximal triangle-free subset to avoid creating a K_4 . This is done in all possible ways with obvious isomorphs skipped. The output will contain all maximal Folkman graphs in addition to other Folkman and non-Folkman graphs. [hard]
3. Eliminate isomorphs using *nauty* software tools. [easy]
4. Filter for Folkman property using INH2234. [easy]

In addition to the custom built algorithms INH2234 and EXTEND, the software packages *nauty* [27] and *Condor* [7] were crucial. *nauty* includes

highly optimized and efficient tools for handling graphs. Developed by Brendan McKay from the Australian National University, *nauty* contains programs to quickly generate all non-isomorphic graphs of a given order as well as identify and eliminate isomorphic graphs. *nauty* has been used in numerous research projects for many years. *Condor* is a distributed processing package created at the Computer Science Department at the University of Wisconsin.

The advantage of the procedure EXTEND is that finding all graphs on 11 vertices without K_4 is feasible: There are 138,892,304 such graphs. The extension process is computationally challenging, yet easily parallelizable; the work was divided over 153 machines in the RIT Computer Science Department labs using *Condor*. Various types of machines were used: Sun Blade 150, Sun Blade 1500, and Sun Fire 880. There were 917 different processes running for a combined processing time of 85 days, 4 hours, and 43 minutes. Since these processes were running in parallel, the complete extension process was completed in just under 3 days. The extension process generated, in particular, all maximal Folkman graphs.

To find all Folkman graphs on 14 vertices, the maximal graphs were reduced using the algorithm REDUCESIZE. This algorithm inputs a graph and removes edges in all possible ways, outputting only those that are Folkman. REDUCESIZE was applied to all of the maximal Folkman graphs. The resulting set of non-isomorphic graphs combined with the maximal graphs was the complete set of Folkman graphs on 14 vertices. Isomorphs were eliminated using *nauty*. The set included the Nenov graph Γ_3 . The processing time for this stage was small.

To achieve confidence in the results, the approach just described was slightly modified to provide an alternate path to the same output: All the Folkman graphs on 14 vertices were generated by extending graphs on 10 vertices instead of 11. The process is as follows: Extend graphs on 10 vertices by adding an independent set of 4 vertices and connecting them to maximal triangle-free subsets of the graph in all possible ways. However, by using an independent set of order 4, the extensions avoid graphs which have no independent set of order 4. This is not a problem, as the missing graphs are those Folkman graphs which are in the Ramsey graph set $\mathcal{R}(4, 4; 14)$, computed in [31]. This alternate technique is sufficiently different from the first approach, allowing for a very high confidence in the correctness of the results.

Since the extension from 10 to 14 vertices is guaranteed to generate all maximal Folkman graphs with $\alpha(G) \geq 4$, the other maximal Folkman graphs were extracted from $\mathcal{R}(4, 4; 14)$. This set was then reduced, using REDUCESIZE as before. The final set of non-isomorphic graphs was exactly the same as previously found after extending from 11 vertices and reducing.

To ensure that no graph on 13 vertices is in $H_v(2, 2, 3; 4)$, the last algorithm REDUCEORDER drops a vertex in all possible ways from each of the Folkman graphs on 14 vertices and the algorithm INH2234 then checks for the Folkman property. No Folkman graphs were found on 13 vertices, thus proving that $F_v(2, 2, 3; 4) = 14$.

2.5 Results

Table 2.2 illustrates various properties of all Folkman graphs in $H_v(2, 2, 3; 4; 14)$. There are 12,227 such graphs in total; interestingly, all of them have chromatic number equal to 5.

$ E(G) $		maxdeg(G)	mindeg(G)	$\alpha(G)$		$ \text{Aut}(G) $			
	#		#		#		#		
42	1	7	527	4	451	3	1507	1	11367
43	6	8	11080	5	5759	4	10557	2	802
44	51	9	393	6	5996	5	160	4	44
45	453	10	227	7	21	6	2	7	1
46	2279					7	1	8	10
47	4555							14	2
48	3628							16	1
49	1138								
50	114								
51	2								

Table 2.2: Properties of graphs in $H_v(2, 2, 3; 4; 14)$.

The order of the automorphism group of the Nenov graph Γ_3 , pictured in Figure 2.1, is equal to 14. The graph F_{16} , presented in Figure 2.3, has the largest automorphism group among all 12,227 graphs in $H_v(2, 2, 3; 4)$, with $|\text{Aut}(F_{16})| = 16$. Let $g_1 = (0\ 1)$, $g_2 = (3\ 4)(6\ 7)(8\ 9)(10\ 11)(12\ 13)$, and $g_3 =$

$(2\ 3)(4\ 5)(7\ 8)(11\ 12)$ be the permutations of the set $\{0, \dots, 13\}$. Then the full automorphism group of F_{16} is generated by g_1, g_2 , and g_3 .

Table 2.3 lists the specific properties of these two graphs in relation to the properties shown in Table 2.2.

G	$\chi(G)$	$ E(G) $	$\max\deg(G)$	$\min\deg(G)$	$\alpha(G)$	$ \text{Aut}(G) $
Γ_3	5	42	8	4	7	14
F_{16}	5	45	7	5	4	16

Table 2.3: Properties of Γ_3 and F_{16} .

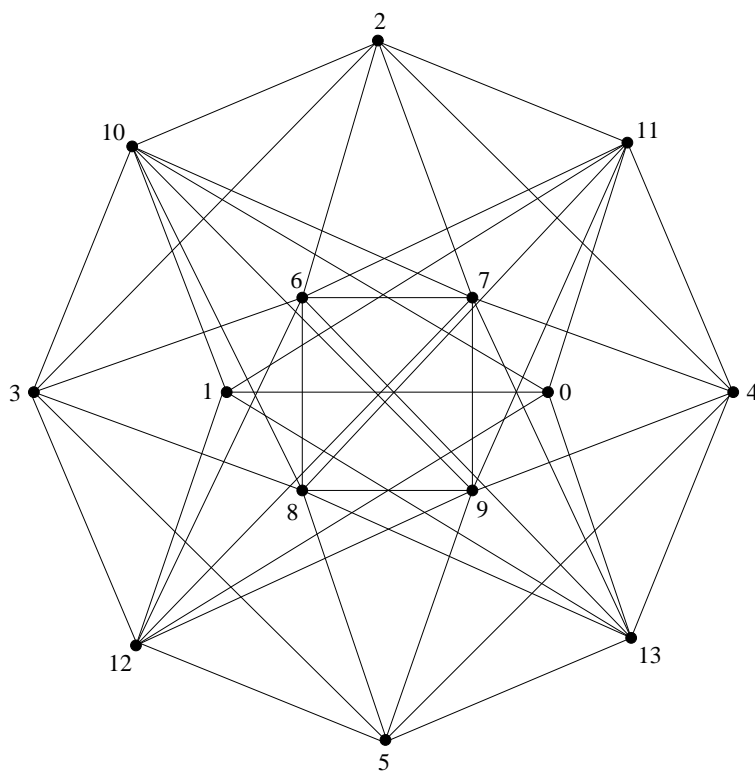


Figure 2.3: The Folkman graph F_{16} with $|\text{Aut}(G)| = 16$.

Chapter 3

$F_e(3, 3; 4)$

3.1 Background

Finding values of $F_e(3, 3; k)$ is one of the oldest problems in Folkman number theory. For $k \geq 7$ the value is trivially 6, with K_6 as the smallest satisfying graph. For $k = 6$ Graham [14] showed in 1968 that $F_e(3, 3, 6) = 8$ using $C_5 + K_3$. Schäuble [54] first showed in 1969 that $F_e(3, 3; 5) \leq 42$, but it wasn't until 1998 that Piwakowski, Radziszowski, and Urbański [49] proved, using a computer-based proof, that $F_e(3, 3; 5) = 15$. The smallest parameters of an edge Folkman number whose value is still unknown is $F_e(3, 3; 4)$.

In the late 1960s, Erdős proposed finding a solution for $F_e(3, 3; 4)$; later, in 1970, Folkman showed the existence of such a graph, but the order was extremely large. Frankl and Rödl [12] proved in 1986 that $F_e(3, 3; 4) \leq 7 \times 10^{11}$, and in 1988, Spencer [57] brought the upper bound down to 3×10^8 . An error in Spencer's proof discovered by Hovey in 1989, unfortunately raised the bound to 3×10^9 [58]. All attempts to reduce this bound to a more reasonable value have failed.

3.2 Methods

To reduce the upper bound it suffices to show that a smaller graph is in $H_e(3, 3; 4)$. The problem, of course, is finding such a graph; the graph will have the property that it will be a K_4 -free graph which is not the union of two triangle-free graphs.

Exoo suggested using a $(4, 4, 4)$ - and $(4, 12)$ -Ramsey graph defined in 1968

by Hill and Irving [18], G_{127} , which has vertices

$$\{a_i : 0 \leq i < 127\}$$

and edges

$$\{(a_i, a_j) : j - i \equiv \alpha^3 \pmod{127}, 0 \leq i < j < 127, \alpha \in \mathbb{N}\} .$$

With this construction, G_{127} has 127 vertices, 2667 edges, 9779 triangles, and no K_4 's.

In dealing with a graph on 127 vertices the test for the Folkman property is more difficult than with $F_v(2, 2, 3; 4)$. A brute force attempt would mean examining every two coloring of the edges of G_{127} , of which there are 2^{2667} , and then for each of those determining if the coloring has a triangle in the first color or a triangle in the second color. Such a task is well beyond the computing power of current workstations. A better technique is to redefine the problem as a satisfiability (SAT) problem.

SAT problems are expressed in conjunctive normal form (CNF): The conjunction of clauses where each clause is the disjunction of distinct boolean literals. A literal is a variable or negated variable. A k -CNF expression has k literals per clause. An example of a 3-CNF expression using the literals x, y, z is given below:

$$(x \vee y \vee z) \wedge (\bar{x} \vee \bar{y} \vee \bar{z}) .$$

A *satisfying* solution to such a problem—one that reduces the expression to true—might be to assign true to x and false to y . For this example there are many such solutions. However, when there are more variables and many more clauses, the difficulty in finding a satisfying assignment becomes much greater. In fact, SAT is NP-complete for k -CNF expressions, $k > 2$ [2]. For randomized algorithms, the expected running time is $O(c^n)$ for some constant c and n variables. In the trivial case, $c = 2$, Dantsin et. al [9] found a deterministic algorithm for k -CNF SAT (k -SAT) where $c = 2 - \frac{2}{k+1}$. Most recently, in 2004, Iwama and Tamaki showed that for 3-SAT there is an algorithm with $c = 1.324$.

Some of the current best solvers include *zchaff* [13] developed at Princeton, and *march_eq*, developed at the Delft University of Technology in the Netherlands [17]. More information about these tool can be found in Appendix A.

Since $H_e(3, 3; 4)$ uses only two colors, each edge in G_{127} is one of two colors, or alternatively, true or false. To define the Folkman property of G_{127} in terms of a SAT problem, each edge is assigned a variable and every two clauses represents a triangle in one of the two colors. Let the formula ϕ be

$$\phi = (a_i \vee a_j \vee a_k) \wedge (\bar{a}_i \vee \bar{a}_j \vee \bar{a}_k) \wedge \dots$$

where $\{a_i, a_j, a_k\} \in \text{tri}(G_{127})$.

There are 2667 variables and 19558 clauses in ϕ . The clauses are defined so that there will be no satisfying solution if the graph cannot be split into two triangle-free parts. This is precisely the condition needed for G_{127} to be in $H_e(3, 3; 4)$. Formally,

$$G \not\rightarrow (3, 3)^e \iff \phi \text{ is satisfiable .}$$

Logically, this can be rewritten to say

$$\phi \text{ is not satisfiable} \iff G \rightarrow (3, 3)^e .$$

Thus, showing that $G_{127} \in H_e(3, 3; 4)$ reduces to proving that ϕ is unsatisfiable.

SAT program such as *zchaff* and *march_eq* take their input from files in DIMACS format. The unofficial format for satisfiability problems from the Center for Discrete Mathematics and Theoretical Computer Science (DIMACS) is a text file with the first line describing the number of variables and the number of clauses n in a SAT formula. The remaining n lines describe the clauses. Variables are represented as positive integers and negated variables are represented as negative integers. An example of a DIMACS SAT file is presented in Figure 3.1.

3.3 Experiments

Even with the powerful solvers mentioned earlier, showing ϕ to be unsatisfiable has proven difficult. The SAT problem is so large that current attempts to attack the problem directly have taken so long as to be deemed impractical. However, since the goal is showing unsatisfiability, if any smaller subset of clauses can be shown unsatisfiable, then the whole graph is unsatisfiable.

Because of the regularity of G_{127} , the graph has interesting properties that can be used to extract very regular subgraphs. Each vertex has exactly

```

p cnf 6 6
1 2 3 0      (x1 ∨ x2 ∨ x3) ∧
-1 -2 -3 0   (x̄1 ∨ x̄2 ∨ x̄3) ∧
3 4 5 0      (x3 ∨ x4 ∨ x5) ∧
-3 -4 -5 0   (x̄3 ∨ x̄4 ∨ x̄5) ∧
4 5 6 0      (x4 ∨ x5 ∨ x6) ∧
-4 -5 -6 0   (x̄4 ∨ x̄5 ∨ x̄6)

```

Figure 3.1: Sample DIMACS file (left) where each line corresponds to a clause (right)

42 neighbors. Selecting one vertex and removing it and all its neighbors produces the graph G_{84} with 84 vertices. Choosing an independent set of 10 vertices and removing these creates a graph on 74 vertices. The corresponding formula for this graph is easily shown to be satisfiable. Reducing the number of vertices in the independent set increases the number of vertices in the resulting graph. These graphs are named $G_{74}, G_{75}, \dots, G_{83}$. Several programs were written to generate these graphs and convert them to input files for the SAT solvers. All of the graphs were satisfiable, but the time needed to solve them increased as the number of vertices increased.

Another aspect under consideration is the ratio between the number of clauses and the number of variables. Papers such as [32, 26, 56] discuss empirical evidence that there exists a phase transition when the ratio exceeds some value r , $3.53 \leq r \leq 4.596$. The probability of finding a satisfying solution for 3-CNF SAT problems quickly diminishes as r increases. The exact value of r is unknown, but conjectured to be approximately 4.2. A graph comparing the probability of satisfiability to the r is shown in Figure 3.2.

The ratio for G_{127} well exceeds the proposed transition ratio of 4.2, which strongly suggests that G_{127} is, in fact, not satisfiable. However, strong evidence is not sufficient for a proof, and the time needed to determine satisfiability increases greatly as the number of vertices chosen from G_{127} approaches 86 and beyond. There have been other, similar attempts to find subgraphs that are not satisfiable [51], but they have had difficulty because of the time required to find a solution.

Other experiments involved generating the input files for the SAT solvers in different ways. As with any search, the order of the input can greatly

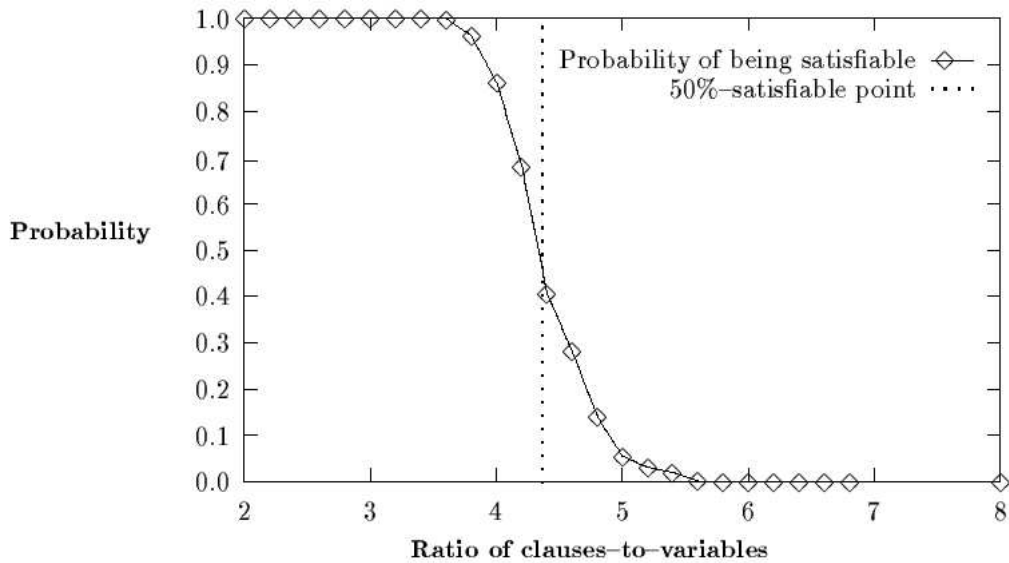


Figure 3.2: Phase transition of 3-SAT [32]

affect how the search is conducted, which affects the running time of the algorithm. Indeed, changing the order of the clauses did yield better search times, but only in some instances.

One experiment involved writing all the negated clauses at the end, while another wrote the negated clauses in reverse order from the corresponding positive clauses. Despite some positive results, in general there was no single technique that consistently proved better for all input graphs.

The most recent experiments have examined symmetry-breaking clauses. These clauses are added to the original formula such that they do not affect the outcome of the solution, but may allow SAT solvers to finish faster.

Symmetry-breaking clauses are derived from special propositional-formula bipartite graphs constructed from the clauses of a formula. Every clause is assigned a vertex and every variable is assigned two vertices, one for each possible state. A clause vertex is connected by an edge to each vertex that corresponds to a variable in the clause. An edge also connects each pair of variable vertices. The resulting graph is then analyzed to find symmetries. Those symmetries are exploited to generate additional clauses that reduce the search space of the original formula. Extensive research into this field has been conducted by Aloul et al. [3, 4, 5], Darga [10], and Crawford et al.

[8] to name a few.

In addition, a new tool *saucy* [10] extends *nauty* and provides optimized graph symmetry detection in cases where a graph has “inherent structure.”. Specifically, *saucy* exploits the sparsity in many propositional-formula graphs. Doing so results in considerable performance and efficiency gains over *nauty*. This program is used in the process of generating symmetry-breaking clauses.

Work on symmetry-breaking clauses and *saucy* by researchers at the University of Michigan has produced a program called *Shatter* [10], which accepts a propositional formula and outputs a new formula in DIMACS format containing the original formula plus symmetry-breaking clauses. Because of the optimization in *saucy*, this process takes only a few moments to complete.

Shatter was used in several attempts to simplify ϕ . Since the output of *Shatter* is another DIMACS file, any SAT solver may be used to process it. Attempts at using *march_eq* and *zchaff* still yielded the same difficulties as before: The search space is still too large and the programs never showed any signs of real progress. Even with simpler graphs on fewer vertices did not yield to symmetry-breaking.

3.4 Future Work

There is a rich area for further research into the phase transition from easily solvable to virtually impossible. Better SAT solvers will undoubtedly be of use because they will allow much larger problems to be solved in less time. The real key to determining the satisfiability of G_{127} will be finding an appropriate subgraph that can be shown unsatisfiable. Such a result would improve the current upper bound 23,622,047-fold.

Chapter 4

Final Thoughts

This thesis has provided a new result in Folkman number theory by showing $F_v(2, 2, 3; 4) = 14$. It has also explored new areas of attack in lowering the upper bound of $F_e(3, 3; 4)$. Folkman number theory and Ramsey theory are rich areas of interest that manage to encompass both the pure mathematical world and the computing world.

Results that are mathematically challenging to obtain are beginning to fall to the ever increasing power of computers. This trend will only continue into the future. There are still many problems left to solve, however. One must be extremely careful with enumeration algorithms similar to those used in this thesis. To guarantee the complete enumeration of a set is often not only difficult but error prone. The future of mathematical computing will depend on solid algorithmic design and the provability of correctness. These problems have existed since the advent of computing, leaving many to harbor doubt over the usefulness of computers in the field of mathematics. To overcome these doubts will be an fascinating challenge to tackle.

4.1 Technical Improvements

Many other theses could be written using similar techniques described here, but there are a number of things that should be done differently in the future.

The *nauty* package contains a wealth of functionality that can be better utilized by integrating the custom software with the *nauty* libraries. Instead of only using the individual programs, many of the graph manipulation functions have already been written. The macros to create graphs, add and

remove vertices, and complement graphs—to name a few—are already available. In hindsight, it was wasteful and error-prone to rewrite these. In addition, perhaps the isomorph testing functions can be used directly from within an algorithm, rather than outputting the graphs from one program and then running *shortg* to eliminate the isomorphs. This may improve the running time of the algorithm.

There are also further refinements to the extension algorithm that would reduce the number of graphs that need extending. By considering for extension only those graphs that have the property that adding an edge creates a K_3 , the extension process would be significantly faster. Another enhancement, from 1.3.3, would be to eliminate graphs with chromatic number less than 4. These additional constraints were not applied in the original work because initially it was debated how much improvement they would yield. A more brute force approach was possible and conducting the extension in parallel using *Condor* was another point of research interest. These constraints would, in fact, have been the source of huge performance gains but the results had already been calculated and the improvements were no longer needed.

Appendix A

Tools

A.1 fib

fib and the associated utilities such as *filter*, *extend*, and *in_h223_4* represent the core software written by the author for the purpose of this thesis. The source code is available online and in Appendix B. The following highlights some of the implementation details, while the main algorithms are discussed in pseudo-code form in previous chapters.

The graphs used for $F_v(2, 2, 3; 4)$ had no more than 14 vertices, which meant that any information about sets of vertices could be stored in a single 32-bit integer. If an element was in the set, then the corresponding bit was turned on. Macros provided easy access to and modification of the bit values.

Graphs are represented as an adjacency matrix with each row of the matrix stored in one 32-bit integer. The matrix is combined with other information about the graph, such as order, number of edges, maximum and minimum degree. The struct also contains three important tables: a table of triangles, a table for caching the presence of a triangle, and a table of maximum independent sets.

The table of triangles is built by three nested loops iterating over all vertex triplets and testing if they form a triangle. It is not always necessary for this table to exist, so it is only built if the entire list of triangles is needed.

The triangle caching table is indexed by a set and contains 1 if the set contains a triangle, and 0 if it does not. When the presence of a triangle is to be tested the cache is consulted first before searching the set for a triangle. If one is found, the cache is updated accordingly.

The table of maximum independent sets is crucial to testing if a graph is in $H_v(2, 2, 3; 4)$ and so its implementation had to be efficient. The algorithm used is derived from ALLMAXCLIQUES by Kreher and Stinson [21].

As noted in Chapter 2, the algorithm for REDUCETOMINIMALS used an optimization that allowed the table of maximum independent sets to grow for a given graph without having to search the graph as edges were dropped. The present maximum independent sets can be extended. The algorithm is presented as Algorithm A.1.1.

Algorithm A.1.1: EXTENDMISTABLE(G, x, y)

comment: $\{x, y\}$ is a dropped edge in G

global M_ℓ ($\ell = 0, \dots, k$)

comment: M_ℓ is a mis in G such that $\{x, y\} \in E(M_\ell)$

let $MIS \leftarrow \{\}$

for $i \leftarrow 1$ **to** k

do	{	if $x \in M_i \wedge y \notin M_i$	then $MIS \leftarrow MIS \cup \{M_i\}$
		if $x \in M_i \wedge y \notin M_i$	then $\left\{ \begin{array}{l} \text{if } N(y) \cap M_i = \{x\} \\ \text{then } MIS \leftarrow MIS \cup \{M_i \cup \{y\}\} \\ \text{else if } N(y) \cap M_i \geq 2 \\ \text{then } MIS \leftarrow MIS \cup \{M_i\} \end{array} \right.$
		if $x \notin M_i \wedge y \in M_i$	then $\left\{ \begin{array}{l} \text{if } N(x) \cap M_i = \{y\} \\ \text{then } MIS \leftarrow MIS \cup \{M_i \cup \{x\}\} \\ \text{else if } N(x) \cap M_i \geq 2 \\ \text{then } MIS \leftarrow MIS \cup \{M_i\} \end{array} \right.$
		if $x \in M_i \wedge y \notin M_i$	then $\left\{ \begin{array}{l} S = M_i \setminus (N(y) \setminus x) \\ \text{if } N(S \cup y) = V(G) \setminus S \\ \text{then } MIS \leftarrow MIS \cup \{S \cup \{y\}\} \end{array} \right.$

UNIQUESORT(MIS)

return (MIS)

A.2 nauty

The *nauty* (*no automorphisms, yes?*) package [27] proved essential to this thesis. Developed by Brendan McKay at the Australian National University, *nauty* includes tools for working with graphs. The program *geng* can efficiently generate all non-isomorphic graphs of a given order. Historically, it was the first program to produce all graphs on 11 vertices. Also critical to this thesis was the *shortg* program, which eliminates isomorphic graphs from the input. Without these two programs this work would have been impossible.

Graphs are represented in an efficient printable-ASCII format. The upper triangle of a graph's adjacency matrix is compacted into a bit array. Only the six bits of each byte are used so that the output can be easily read by a text editor. This also allows for simple visual comparison of two graphs. The graphs are output using a canonical labelling that ensures that isomorphic graphs produce the same output. The specific details of the output can be found on the *nauty* website [27]. A sample of the output of generating all graphs on 3 vertices is shown in Figure A.1.

```
$ geng 3
>A geng -d0D2 n=3 e=0-3
B?
B0
BW
Bw
>Z 4 graphs generated in 0.00 sec
```

Figure A.1: All graphs of order 3 output from running *geng 3*

A.3 Condor

Condor is a distributed processing package created at the Computer Science department at the University of Wisconsin. *Condor* was designed with high-throughput, as opposed to high-performance, computing as its goal. High-throughput refers to systems “that can deliver large amounts of processing capacity over long periods of time [7].” High-performance systems

```

universe      = standard
getenv        = True
executable    = extend
input         = input/in.%(Process)
output        = output/11to14.%(Process).g6
error         = output/11to14.%(Process).report
log           = output/11to14.%(Process).log
arguments     = -e 11 14 -r 60 --noX
Notification  = Error

```

```
queue 917
```

Figure A.2: Sample *Condor* job submission file

are designed to perform many calculations over seconds or minutes instead of months or years. *Condor* is used to distribute the processing of many programs across a network of machines.

The software was installed on the RIT Computer Science lab computers and allowed up to 153 machines to be in use at one time. Each computer ran the EXTEND algorithm with a different subset of graphs. The computers were connected using the Network File System (NFS), which made the collection of the output trivial.

The properties for a *Condor* job are described in a file which is submitted to the *Condor* system. Figure A.2 shows the file used for the extension from 11 vertices to 14 vertices. Each process of the job has a unique number from 0 to n , where n is one less than the number of jobs. That number can be used to select different input, output, and log files.

A.4 SAT Solvers

The two SAT solvers used in the experiments for $F_e(3, 3; 4)$ were *zchaff* [13], developed at Princeton, and *march_eq* [17] developed at the Delft University of Technology. Both programs have been winners of different categories at the international SAT 2004 competition. *zchaff* specializes in general SAT problems, while *march_eq* has added heuristics for problems which have less random structure and are expected to be unsatisfiable. Such a solver was

ideal for tackling G_{127} . Both of these programs accept input in DIMACS format [1], which is a standard format for describing CNF problems. An example can be found in Figure 3.1.

Appendix B

Source Code

The essential source code to the programs developed by the author for this thesis can be downloaded from <http://www.jpcoles.com/uni/rit/thesis>. The source code is free software, licensed under the GNU General Public License, Version 2, a copy of which can be found on <http://www.fsf.org>.

```

/*
 * Folkman Graph Property Library Header
 *
 *
 * Copyright 2003, 2004 Jonathan Coles
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA
 */

#ifndef FLIB_H
#define FLIB_H

#include <stdlib.h>
#include <istream>
#include <assert.h>

using namespace std;

//=====//
//                               Graph struct                               //
//=====//

typedef struct
{
    uint *data;
    uint length;
} bool_array;

//
// struct to hold the order and adjacency matrix of a graph
//
typedef struct
{
    uint order;

    uint edge_count;

    uint max_degree,
        min_degree;

    uint max_mis;
    uint chi;

    uint adj_size;

    uint *N;
    uint *adj;
    uint *deg;

    uint num_triangles;
    uint *triangles;
    char *tri_cache;

    uint mis_table_size;
    uint *mis_table;
} Graph;

typedef struct
{
    uint list_len;
    uint alloc_len;
    int **IS;
    uint **Bucket;
} mis_builder_cache;

typedef bool (*f_hX_4)(Graph *g, bool print);

//=====//
//                               Prototypes                               //
//=====//

void print_bits(uint m, uint n, const char *end="\n");
void print_set(uint m, uint s, const char *end="\n");
void print_set(FILE *fp, uint m, uint s, const char *end="\n");
void print_array(FILE *fp, uint size, uint *a, const char *end="\n");

uint bit_count(uint n);
Graph *make_graph(const uint order, const int adj_size=-1, Graph *g=NULL);
void free_graph(Graph *g);
void free_tri_table(Graph *g);
void free_mis_table(Graph *g);
void print_graph(const Graph *g);
void add_edge(Graph *g, const uint v0, const uint v1);
void add_edges(Graph *g, const uint v0, const uint vs);
void remove_edge(Graph *g, const uint v0, const uint v1);
int add_vertex(Graph *g);
void remove_vertex(Graph *g, const uint v0);
void update_min_degree(Graph *g);
void update_max_degree(Graph *g);
Graph *copy_graph(const Graph *g0, Graph *g=NULL);
void complement(Graph *g);

void init_tri_table(Graph *g);
bool has_k_n_helper(Graph *g,
                    uint n, uint k, uint s,
                    uint *vertices, uint subgraph);

bool has_k_n(Graph *g, uint n, uint subgraph);

```

```

bool has_k4(const Graph *g, const uint subgraph);
bool has_triangle_cached(Graph *g, uint subgraph);
bool has_triangle(const Graph *g, const uint subgraph);
void build_mis_table(Graph *g);
void build_tri_table(Graph *g);
bool h223_4(Graph *g, const bool print=false);
bool h23_4(Graph *g, bool print=false);
bool h22_4(Graph *g, bool print=false);
bool in_hX_4(f_hX_4 f, Graph *g, uint f_repeat, double *t, bool check_k4, bool print=false);
Graph *read_graph(FILE *in, Graph *g=NULL);
Graph *read_graph(char *bytes, Graph *g=NULL);
void write_graph(const Graph *g, FILE *fp);
void write_graph_file(const Graph *g, char *filename, char *mode);
Graph *make_gamma3_graph();
void reduce(Graph *g, FILE *out);
void find_maximals(Graph *g);
bool maximal(Graph *g, uint n);
bool minimal(Graph *g);
void extend_to_maximals(Graph *g, uint n);
uint reduce_to_minimals(Graph *g, bool all=false, FILE *out=stdout);
void build_set_neighbours(Graph *g);
void update_set_neighbours(Graph *g, uint vs, uint *tmp_gN);
uint N(Graph *g, const uint set);
uint chromatic(const Graph *g);
void build_max_tri_table(Graph *g, uint *table, uint *num_entries);
void print_graph_info(FILE *fp, Graph *g, char *show_info, char *eol="\n");
uint chi(Graph *g);

//=====//
//                               //
//                               //
//=====//

#define foreach_pair(x,y,n) \
for (uint x=0; x < n-1; x++) \
for (uint y=x+1; y < n; y++)

#define foreach_vertex(g,i) for (uint i=0; i < g->order; i++)

#define foreach_vertex_pair(g,i,j) foreach_pair(i,j,g->order)

#define foreach_subset(i,s) for (uint i=0; i <= s; i++)

#define foreach_elem(g,u,i) \
if (i) for (uint __i__ = i, u=0; u < g->order; u++) if (elem_of(u, __i__))

#define for_if(c) if (c) for

inline uint set_elem(const uint x)
{ return 1 << x; }

/* Return true if x is an element of set */
inline bool elem_of(const uint x, const uint set)
{ return set & set_elem(x); }

/* Return true if m is a subset of s */

inline bool subset(const uint m, const uint s)
{ return m && ((s & m) == m); }

/* Return true if s0 intersect s1 */
inline bool intersects(const uint s0, const uint s1)
{ return (s0 & s1) != 0; }

/* Add x to the set set */
inline void add_elem(const uint x, uint &set)
{ set |= set_elem(x); }

/* Remove x from the set set */
inline void remove_elem(const uint x, uint &set)
{ set &= ~set_elem(x); }

inline uint V(const Graph *g)
{ return (1 << g->order)-1; }

/* Return the set of all vertices in a graph */
inline uint all_vertices(const Graph *g)
{ return V(g); }

/* Return true if v0 is connected to v1 in graph g */
inline bool connected_to(const Graph *g, const uint v0, const uint v1)
{
    assert(v0 < g->order);
    assert(v1 < g->order);

    return elem_of(v0, g->adj[v1]);
}

inline bool connected(const uint v0, const uint adj)
{ return elem_of(v0, adj); }

/* Return the degree of vertex v in graph g */
inline uint deg(const Graph *g, const uint v)
{ return g->deg[v]; }

inline uint adj(const Graph *g, const uint v)
{ return g->adj[v]; }

/* Return true if subgraph does not contain K4 (non-caching) */
inline bool is_k_n_free(Graph *g, uint n, uint subgraph)
{ return !has_k_n(g, n, subgraph); }

/* * Return true if subgraph does not contain a triangle (non-caching) */
inline bool triangle_free(const Graph *g, const uint subgraph)
{ return !has_triangle(g, subgraph); }

/* Return true if subgraph does not contain K3 (caching) */
inline bool triangle_free_cached(Graph *g, uint subgraph)
{ return !has_triangle_cached(g, subgraph); }

/* Return true if subgraph does not contain a K4 (non-caching) */

```

```

inline bool k_4_free(const Graph *g, const uint subgraph)
{ return !has_k_4(g, subgraph); }

inline bool in_h223_4(Graph *g, uint f_repeat, double *t, bool check_k4, bool print)
{ return in_hX_4(h223_4, g, f_repeat, t, check_k4, print); }

inline bool in_h22_4(Graph *g, uint f_repeat, double *t, bool check_k4, bool print)
{ return in_hX_4(h22_4, g, f_repeat, t, check_k4, print); }

inline bool in_h23_4(Graph *g, uint f_repeat, double *t, bool check_k4, bool print)
{ return in_hX_4(h23_4, g, f_repeat, t, check_k4, print); }

#endif // FLIB_H

/*
 * Folkman Graph Property Library
 *
 * Copyright 2003, 2004 Jonathan Coles
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA
 */

#include <stdio.h>
#include <stdlib.h>
#include <memory.h>
#include <assert.h>

#include "timing.h"
#include "flib.h"

void unique_sort(uint *list, uint *len);
void buildheap(uint *list, uint len);
void heapify(uint *list, uint i, uint j);

using namespace std;

#define TESTS 0

#define _free(ptr) \
{ free(ptr); ptr = NULL; }

//printf(stderr, "free() on line %i\n", __LINE__);

CPUDEFS
//
// Program output verbosity level
// 0 is normal
//
extern int verbosity;

#if 0
bool_array *make_bool_array(uint length)
{
    bool_array *a = (bool_array *)calloc(length, sizeof(bool_array));

    a->data = (uint *)calloc((length >> (sizeof(uint) << 3)) + 1, sizeof(uint));
}

bool bool_array_index(bool_array *a, uint index)
{
    assert(index < a->length);
    return (bool)(a[index >> 3] & (index & ((sizeof(uint) << 3) - 1)));
}

void set_bool_array_index(bool_array *a, uint index)
{
    assert(index < a->length);
    a[index >> (sizeof(uint) << 3)] |= (index & ((sizeof(uint) << 3) - 1));
}
#endif

//=====//
//                               connect                               //
//=====//

/*
 * Although connect is inline it is not made available in the header because
 * it must only be used as a helper and/or optimization. By itself, it does
 * not preserve the consistency of the graph. Be careful of how it is used.
 */
inline void connect(Graph *g, uint v0, uint v1)
{
    assert(v0 < g->order);
    assert(v1 < g->order);

    add_elem(v0, g->adj[v1]);
    add_elem(v1, g->adj[v0]);
    g->deg[v0]++;
    g->deg[v1]++;
}

//=====//
//                               disconnect                               //
//=====//

```

```

/*
 * Although disconnect is inline it is not made available in the header because
 * it must only be used as a helper and/or optimization. By itself, it does
 * not preserve the consistency of the graph. Be careful of how it is used.
 */
inline void disconnect(Graph *g, uint v0, uint v1)
{
    assert(v0 < g->order);
    assert(v1 < g->order);

    remove_elem(v0, g->adj[v1]);
    remove_elem(v1, g->adj[v0]);
    g->deg[v0]--;
    g->deg[v1]--;
}

//=====//
//                                     bit_count                               //
//=====//

#if 1
#if 0
#define bit_count(n) __builtin_popcount(n)
#else
uint bit_count(uint n)
{
    char table[16] = {0, 1, 1, 2, 1, 2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4};
    uint sum = 0;

    for (uint i = 0; i < (sizeof(n) << 3); i += 4)
        sum += table[(n >> i) & 0xf];

    return sum;
}
#endif
#endif
#define m1 ((uint) 0x55555555)
#define m2 ((uint) 0x33333333)

uint bit_count(const uint b) {
    uint n;
    const uint a = b - ((b >> 1) & m1);
    const uint c = (a & m2) + ((a >> 2) & m2);
    n = ((uint) c) + ((uint) (c >> 32));
    n = (n & 0x0F0F0F0F) + ((n >> 4) & 0x0F0F0F0F);
    n = (n & 0xFFFF) + (n >> 16);
    n = (n & 0xFF) + (n >> 8);
    return n;
}
#endif

//=====//
//                                     print_bits                               //
//=====//

void print_bits(FILE *fp, uint m, uint n, const char *end="\n");

/* print at most m bits in n */
void print_bits(uint m, uint n, const char *end)
{
    print_bits(stdout, m, n, end);
}

void print_bits(FILE *fp, uint m, uint n, const char *end)
{
    uint mask = (1 << (m-1));
    while (mask != 0)
    {
        if (n & mask)
            fprintf(fp, "1");
        else
            fprintf(fp, "0");

        mask >>= 1;
    }

    if (end != NULL) fprintf(fp, end);
}

//=====//
//                                     print_set                               //
//=====//

void print_set(uint m, uint s, const char *end)
{
    print_set(stdout, m, s, end);
}

void print_set(FILE *fp, uint m, uint s, const char *end)
{
    fprintf(fp, "{");
    bool comma=false;

    if (m && (s & 1))
        fprintf(fp, "0%s", comma ? ", " : ""), comma=true;

    for (uint mask=1; mask < m; mask++)
    {
        if (s & (1 << mask))
            fprintf(fp, "%s%i", comma ? ", " : "", mask), comma=true;
    }

    fprintf(fp, "}");

    if (end != NULL) fprintf(fp, end);
}

//=====//
//                                     print_array                             //
//=====//

```

```

//=====//
void print_array(FILE *fp, uint size, uint *a, const char *end)
{
    fprintf(fp, "{");

    if (size == 0) return;

    fprintf(fp, "%i", a[0]);

    for (uint i=1; i < size; i++)
        fprintf(fp, ",%i", a[i]);

    fprintf(fp, "}");

    if (end != NULL) fprintf(fp, end);
}

//=====//
//                               make_graph                               //
//=====//

Graph *make_graph(const uint order, const int adj_size, Graph *g)
{
    if (g == NULL)
    {
        g = (Graph *)calloc(1, sizeof(Graph));
        assert(g != NULL);
    }

    #if 0
    g->edge_count = 0;
    g->max_degree =
    g->min_degree = 0;
    g->max_mis = 0;
    g->mis_table_size = 0;

    g->N = NULL;
    g->triangles = NULL;
    g->num_triangles = 0;
    g->tri_cache = NULL;
    g->mis_table = NULL;
    #endif

    g->order = order;
    g->adj_size = adj_size == -1
        ? g->order
        : adj_size;

    g->adj = (uint *)calloc(g->adj_size, sizeof(uint));
    g->deg = (uint *)calloc(g->adj_size, sizeof(uint));

    assert(g->adj != NULL);
    assert(g->deg != NULL);
}
else
{
    assert(g->adj_size >= order);
    g->order = order;
    memset(g->adj, 0, g->adj_size * sizeof(uint));
    memset(g->deg, 0, g->adj_size * sizeof(uint));
    // don't worry about other members, they are realloc'd
}

return g;
}

//=====//
//                               copy_graph                               //
//=====//

Graph *copy_graph(const Graph *g0, Graph *g)
{
    if (g == NULL)
    {
        g = (Graph *)malloc(sizeof(Graph));
        assert(g != NULL);

        g->adj_size = g0->order;
        g->adj = (uint *)malloc(g->adj_size * sizeof(uint));
        g->deg = (uint *)malloc(g->adj_size * sizeof(uint));

        assert(g->adj != NULL);
        assert(g->deg != NULL);
    }

    #define ALLOC(var,num,type) \
    if (g0->var != NULL) \
    { g->var = (type *)malloc(num * sizeof(type)); assert(g->var != NULL); } \
    else \
    { g->var = NULL; }

    ALLOC(N, g->adj_size, uint);
    ALLOC(triangles, (1<g0->order), uint);
    ALLOC(tri_cache, (1<g0->order), char);
    ALLOC(mis_table, g0->mis_table_size, uint);
    #undef SETUP

    }
    else
    {
        assert(g->adj_size >= g0->order);

        g->order = g0->order;
        g->edge_count = g0->edge_count;
        g->max_degree = g0->max_degree;
        g->min_degree = g0->min_degree;
        g->max_mis = g0->max_mis;
        g->mis_table_size = g0->mis_table_size;
        g->num_triangles = g0->num_triangles;
    }
}

```

```

    assert(g->adj != NULL);
    assert(g->deg != NULL);
    assert(g0->adj != NULL);
    assert(g0->deg != NULL);

    memcpy(g->adj, g0->adj, g0->order * sizeof(uint));
    memcpy(g->deg, g0->deg, g0->order * sizeof(uint));

#define COPY(var,num,type) \
if (g->var != NULL && g0->var != NULL) \
{ memcpy(g->var, g0->var, num * sizeof(type)); }

    COPY(N, g0->order, uint);
    COPY(triangles, (1<g0->order), uint);
    COPY(tri_cache, (1<g0->order), char);
    COPY(mis_table, g0->mis_table_size, uint);
#undef COPY

    return g;
}

//=====//
//                               free_graph                               //
//=====//

void free_graph(Graph *g)
{
    if (g != NULL)
    {
        free_tri_table(g);
        free_mis_table(g);

        _free(g->N);
        _free(g->adj);
        _free(g->deg);
        _free(g);
    }
}

//=====//
//                               print_graph                               //
//=====//

void print_graph(const Graph *g)
{
    uint i, j;

    for (i=0; i < g->order; i++)
    {
        printf("%i:\t", i);

        for (j=0; j < g->order; j++)
            printf("%i", elem_of(j, adj(g, i)) ? 1 : 0);

        printf("\n");
    }
}

}

//=====//
//                               add_edge                               //
//=====//

void add_edge(Graph *g, const uint v0, const uint v1)
{
    assert(v0 < g->order);
    assert(v1 < g->order);
    assert(v0 != v1);

    connect(g, v0, v1);

    g->max_degree = max(g->max_degree, max(deg(g, v0), deg(g, v1)));

    update_min_degree(g);

    g->edge_count++;
}

//=====//
//                               add_edges                               //
//=====//

/* Connect v0 to the vertices in vs */
void add_edges(Graph *g, const uint v0, const uint vs)
{
    for (uint u=0; u < g->order; u++)
        if (elem_of(u, vs))
            connect(g, v0, u);

    update_min_degree(g);
    update_max_degree(g);
}

//=====//
//                               remove_edge                               //
//=====//

/* Remove the edge between v0 and v1 */
void remove_edge(Graph *g, const uint v0, const uint v1)
{
    assert(v0 < g->order);
    assert(v1 < g->order);

    disconnect(g, v0, v1);

    g->min_degree = min(g->min_degree, min(deg(g, v0), deg(g, v1)));

    update_max_degree(g);

    g->edge_count--;
}

```

```

//=====//
//                               update_min_degree                               //
//=====//

/* (helper) Recalculate minimum degree in the graph */
void update_min_degree(Graph *g)
{
    g->min_degree = g->order-1;
    register uint *degs = g->deg;
    register uint *end_deg = degs + g->order;
    while (degs != end_deg)
    {
        register uint deg = *degs++;
        if (deg < g->min_degree)
            g->min_degree = deg;
    }
}

//=====//
//                               update_max_degree                               //
//=====//

/* (helper) Recalculate maximum degree in the graph */
void update_max_degree(Graph *g)
{
    g->max_degree = 0;
    register uint *degs = g->deg;
    register uint *end_deg = degs + g->order;
    while (degs != end_deg)
    {
        uint deg = *degs++;
        if (deg > g->max_degree)
            g->max_degree = deg;
    }
}

//=====//
//                               add_vertex                                     //
//=====//

int add_vertex(Graph *g)
{
    assert(g != NULL);

    if (g->order+1 > g->adj_size)
    {
        g->adj = (uint *)realloc(g->adj, (g->order + 1) * sizeof(uint));
        assert(g->adj != NULL);
        g->adj_size = g->order + 1;
    }

    g->adj[g->order] = 0;
    g->deg[g->order] = 0;

    g->order++;
    return g->order-1;
}

//=====//
//                               remove_vertex                                   //
//=====//

void remove_vertex(Graph *g, const uint v0)
{
    assert(v0 < g->order);

    //memmove(g->adj + v0, g->adj + v0 + 1, (g->order - 1 - v0) * sizeof(uint));

#ifdef 1
    for (uint i=v0; i < g->order-1; i++)
        g->adj[i] = g->adj[i+1];
#endif

    g->order--;

    //
    // This is tricky so let's take an example. Let g be a graph of order 10,
    // v0=4, and one row in the adjacency table be 100111010.
    //
    // s = 100111010 ^ (1 << 4) = 100111010 ^ 0000010000 = 1001101010
    // m = (1 << (4+1))-1 = (1 << 5)-1 = 0000011111
    // adj[] = ((s & ~m) >> 1) | (s & m)
    //          = ((1001101010 & 111100000) >> 1) | (1001101010 & 0000011111)
    //          = (1001100000 >> 1) | 0000001010
    //          = 100110000 | 0000001010
    //          = 100111010
    //
    // The result is 9 bits with v0 dropped.
    //
    for (uint i=0; i < g->order; i++)
    {
        uint s = adj(g, i) & ~set_elem(v0);
        uint m = (1 << (v0+1)) - 1;
        g->adj[i] = ((s & ~m) >> 1) | (s & m);
    }
}

//=====//
//                               complement                                       //
//=====//

void complement(Graph *g)
{
    for (uint i=0; i < g->order; i++)
    {
        g->adj[i] = ~(adj(g, i) | set_elem(i));
        g->deg[i] = bit_count(adj(g, i)); //g->order - deg(g, i);
    }
}

```

```

    }

    update_min_degree(g);
    update_max_degree(g);
}

//=====//
//                               //
//                               //
//=====//

/*
 * Initialize the tri_table. To reduce memory thrashing,
 * this function will only allocate new memory if the order
 * of g changes between calls. If g is NULL the memory
 * will be freed.
 */
void init_tri_table(Graph *g)
{
    g->num_triangles = 0;
    g->triangles = (uint *)realloc(g->triangles, (1<<g->order) * sizeof(uint));

    g->tri_cache = (char *)realloc(g->tri_cache, (1<<g->order) * sizeof(char));
    memset(g->tri_cache, 2, (1<<g->order) * sizeof(char));
}

void free_tri_table(Graph *g)
{
    if (g != NULL)
    {
        _free(g->triangles);
        g->num_triangles = 0;

        _free(g->tri_cache);
    }
}

//=====//
//                               //
//                               //
//=====//

void build_tri_table(Graph *g)
{
    register uint j, k, l;

    if (g->order < 3) return;

    for (j=0; j <= g->order - 3; j++)
    {
        register uint rj = adj(g, j);
        for (k=j+1; k <= g->order - 2; k++)
        {
            register uint rk = adj(g, k);
            for (l=k+1; l <= g->order - 1; l++)
            {
                if ( connected(k, rj)

```

```

                && connected(l, rj)
                && connected(l, rk))
            {
                uint tri = set_elem(j) | set_elem(k) | set_elem(l);
                g->triangles[g->num_triangles++] = tri;
                g->tri_cache[tri] = 1;
            }
        }
    }
}

//=====//
//                               //
//                               //
//=====//

uint N(Graph *g, const uint set)
{
    if (g->N[set] == 0xFFFFFFFF)
    {
        uint n = 0;
        foreach_elem(g, v, set) n |= adj(g, v);
        g->N[set] = n & ~set;
    }
    return g->N[set];
}

//=====//
//                               //
//                               //
//=====//

void build_set_neighbours(Graph *g)
{
    g->N = (uint *)realloc(g->N, (1 << g->order) * sizeof(uint));
    memset(g->N, 0xFF, (1 << g->order) * sizeof(uint));
}

//=====//
//                               //
//                               //
//=====//

bool has_triangle(const Graph *g, const uint subgraph)
{
    //
    // Subgraph must have at least 3 vertices
    //
    uint w = subgraph;
    w &= w - 1;
    w &= w - 1;
    if (!w) return false;

    uint list[50];
    uint list_len=0;

    //

```

```

// Rather than check these bits every time through each
// loop in the main algorithm, just extract the ones
// we need now.
//
for (uint i=0; i < g->order; i++)
{
    w = adj(g, i) & subgraph;
    w &= w - 1;
    if (w && elem_of(i, subgraph)) list[list_len++] = i;
}

//
// If we don't have three points, we don't have a triangle
//
if (list_len < 3) return false;

//
// Check triples of points
//
register uint j, k, l;
for (j=0; j <= list_len - 3; j++)
{
    register uint rj = adj(g, list[j]);
    for (k=j+1; k <= list_len - 2; k++)
    {
        register uint lk = list[k];
        register uint rk = adj(g, lk);
        for (l=k+1; l <= list_len - 1; l++)
        {
            if ( connected(lk, rj)
                && connected(list[l], rj)
                && connected(list[l], rk))
                return true;
        }
    }
}

return false;
}

//=====//
//                                     has_k_4                               //
//=====//

bool has_k_4(const Graph *g, const uint subgraph)
{
    //
    // Subgraph must have at least 4 vertices
    //
    uint w = subgraph;
    w &= w - 1;
    w &= w - 1;
    w &= w - 1;
    if (!w) return false;

    uint list[50];
    uint list_len=0;

    for (uint i=0; i < g->order; i++)
    {
        w = adj(g, i) & subgraph;
        w &= w - 1;
        w &= w - 1;
        if (w && elem_of(i, subgraph)) list[list_len++] = i;
    }

    if (list_len < 4) return false;

    for (uint j=0; j <= list_len - 4; j++)
    {
        register uint rj = adj(g, list[j]);
        for (uint k=j+1; k <= list_len - 3; k++)
        {
            uint lk = list[k];
            register uint rk = adj(g, lk);
            for (uint l=k+1; l <= list_len - 2; l++)
            {
                uint ll = list[l];
                register uint rl = adj(g, ll);
                for (uint m=l+1; m <= list_len - 1; m++)
                {
                    if ( connected(lk, rj)
                        && connected(ll, rj)
                        && connected(ll, rk)
                        && connected(list[m], rj)
                        && connected(list[m], rk)
                        && connected(list[m], rl))
                    {
                        return true;
                    }
                }
            }
        }
    }

    return false;
}

//=====//
//                                     has_k_n_helper                               //
//=====//

bool has_k_n_helper(Graph *g,
                    uint n, uint k, uint s,
                    uint *vertices, uint subgraph)
{
    if (k == 0)
    {
        //fprintf(stdout, "HERE\n");
        foreach_pair(i, j, n)

```

```

    {
        // fprintf(stdout, "[%i %i] ", i, j);
        if (!connected_to(g, vertices[i], vertices[j]))
            return false;
    }
    //fprintf(stdout, "\n");
    return true;
}
else
{
    //fprintf(stdout, "HERE 2\n");
    for (uint i=s; i <= g->order - k; i++)
    {
        //
        // only consider vertices in the subgraph
        //
        if (elem_of(i, subgraph))
        {
            //
            // only consider a vertex if it is connected to
            // at least one other vertex in the subgraph
            // and connects to at least n-1 other vertices
            //

            uint w = 1; //g->adj[i] & subgraph;

            //for (uint j=0; j < n-2; j++)
            //w &= w - 1;

            if (w)
            {
                vertices[n-k] = i;
                if (has_k_n_helper(g, n, k-1, i+1, vertices, subgraph))
                    return true;
            }
        }
    }
}

return false;
}

//=====//
//                                     has_k_n                               //
//=====//

/*
 * Return true if subgraph contains a K_n
 */
bool has_k_n(Graph *g, uint n, uint subgraph)
{
    if (n < 1)          return false;
    if (n == 1)        return true; // ???
    if (g->order < n)   return false;
    if (g->order == 0)  return false;

    if (g->max_degree < n-1) return false;
    //fprintf(stdout, "HERE 1\n");

    if (n == 3) return has_triangle(g, subgraph);
    if (n == 4) return has_k_4(g, subgraph);

    //fprintf(stdout, "HERE 1\n");

    uint vertices[n];
    return has_k_n_helper(g, n, n, 0, vertices, subgraph);
}

//=====//
//                                     has_triangle_cached                               //
//=====//

/*
 * Return true if subgraph contains a K_4
 *
 * (This caches the results and assumes that init_tri_table()
 * has been called sometime in the past with the same graph g)
 */
bool has_triangle_cached(Graph *g, uint subgraph)
{
    if (g->tri_cache[subgraph] != 2)
        return (bool)g->tri_cache[subgraph];

    for (uint i=0; i < g->num_triangles; i++)
        if (subset(g->triangles[i], subgraph))
            return (bool)(g->tri_cache[subgraph] = 1);

    return (bool)(g->tri_cache[subgraph] = 0);
}

/*
 * Recursively try to add vertex v and then its successor (v+1)
 * to the set provided it doesn't connect to any vertex already
 * in the set. Once all vertices are exhausted, add the set to
 * the table.
 */
static void build_max_tri_table_helper(Graph *g, uint set, uint v, uint *table, uint *num_entries)
{
    if (v == g->order)
    {
        if (set && triangle_free_cached(g, set))
            table[( *num_entries )++] = set;
    }
    else
    {
        build_max_tri_table_helper(g, set | set_elem(v), v+1, table, num_entries);
        build_max_tri_table_helper(g, set, v+1, table, num_entries);
    }
}

void build_max_tri_table(Graph *g, uint *table, uint *num_entries)

```



```

    }
    return false;
}

static void cache_call(mis_builder_cache *mbc, Graph *g, uint i, int *IS, uint *Bucket)
{
    if (mbc[i].list_len == mbc[i].alloc_len)
    {
        mbc[i].alloc_len += 30;
        uint alen = mbc[i].alloc_len;

        mbc[i].IS = (int **)realloc(mbc[i].IS, alen * sizeof(int *));
        mbc[i].Bucket = (uint **)realloc(mbc[i].Bucket, alen * sizeof(uint *));

        for (uint j=mbc[i].list_len; j < alen; j++)
        {
            mbc[i].IS[j] = (int *)malloc(g->order * sizeof(int));
            mbc[i].Bucket[j] = (uint *)malloc(g->order * sizeof(uint));
        }
    }

    mbc[i].list_len++;
    uint len = mbc[i].list_len;

    //for (uint j=0; j < g->order; j++) mbc[i].IS[len-1][j] = IS[j];
    //for (uint j=0; j < g->order; j++) mbc[i].Bucket[len-1][j] = Bucket[j];

    memcpy(mbc[i].IS[len-1], IS, g->order * sizeof(int));
    memcpy(mbc[i].Bucket[len-1], Bucket, g->order * sizeof(uint));
}

/*
 * Based on code from maxclique.c by Tim Carr, Kevin O'Neill, and Mark Simmons
 * which was based on the algorithm of Tsukiyama, Ide, Ariyoshi, and Shirakawa
 */
static void build_mis_table_helper(Graph *g, uint i, int *IS, uint *Bucket,
    mis_builder_cache *mbc, uint adj_list[][33])
{
    static uint recur_count = 0;

    if (g == NULL)
    {
        recur_count = 0;
        return;
    }

    #if 0
    if (find_cached_call(mbc, g, i, IS, Bucket)) return;
    #endif

    //recur_count++;

    #if 0
    fprintf(stderr, "%i\t", i);
    print_array(stderr, g->order, IS); fprintf(stderr, " ");
    print_array(stderr, g->order, Bucket); fprintf(stderr, "\n");
    #endif

    #if 0
    cache_call(mbc, g, i, IS, Bucket);
    #endif

    if (i >= g->order-1) {
        uint *mis = &g->mis_table[g->mis_table_size++];
        *mis = 0;
        uint count=0;
        for (uint i=0; i < g->order; i++)
        {
            if (IS[i] == 0)
            {
                add_elem(i, *mis);
                count++;
            }
        }

        if (count && count > g->max_mis) g->max_mis = count;

        //print_set(g->order, *mis); printf("\n");
    }
    else
    {
        static double total_time = 0;
        static double last_total_time = 0;
        double start_time = 0;
        double end_time = 0;

        uint *adj_x = &adj_list[i][1];
        uint adj_x_len = adj_list[i][0];

        uint x = i + 1;

        //
        // precalculated in build_mis_table
        //
        //foreach_vertex(g, y)
        //if (y <= i && elem_of(y, adj(g, x))) adj_x[adj_x_len++] = y;

        uint c = 0;

        for (uint y=0; y < adj_x_len; y++)
            if (IS[adj_x[y]] == 0) c++;
    }
}

```

```

if (c == 0)
{
    uint cc[32];
    memcpy(cc, IS, g->order * sizeof(uint));

    for (uint y=0; y < adj_x_len; y++)
        IS[adj_x[y]]++;

    build_mis_table_helper(g, x, IS, Bucket, mbc, adj_list);

    memcpy(IS, cc, g->order * sizeof(uint));
}
else
{
    IS[x] = c;
    build_mis_table_helper(g, x, IS, Bucket, mbc, adj_list);
    IS[x] = 0;

    bool f = true;

    uint cc[32];
    memcpy(cc, IS, g->order * sizeof(uint));

    uint bb = Bucket[x];

start_time = CPUTIME;
    for (uint y=0; y < adj_x_len; y++)
    {
        if (IS[adj_x[y]] == 0)
        {
            add_elem(y, Bucket[x]);

            uint adj_adj_x_y = adj(g, adj_x[y]);
            foreach_vertex(g, z)
            {
                if (z > i) break;

                if (elem_of(z, adj_adj_x_y))
                {
                    IS[z]--;
                    if (IS[z] == 0) f = false;
                }
            }
            IS[adj_x[y]]++;
        }
    }
end_time = CPUTIME;

    if (f) build_mis_table_helper(g, x, IS, Bucket, mbc, adj_list);

    memcpy(IS, cc, g->order * sizeof(uint));
    Bucket[x] = bb;
}
}

#endif
if (end_time && start_time)
{
    total_time += (end_time - start_time);

    if (total_time - last_total_time > 2)
    {
        fprintf(stderr, "%f %f\n", total_time, (end_time - start_time));
        last_total_time = total_time;
    }
}
#endif

}

#endif

/*
 * Build a table of maximum independent sets in graph g.
 * To reduce memory thrashing, new memory is allocated
 * only when the order of g changes between calls.
 */
void build_mis_table(Graph *g)
{
    init_mis_table(g);

    //g->mis_table_size = 0;

    int *IS = (int *)calloc(g->order, sizeof(int));
    uint *Bucket = (uint *)calloc(g->order, sizeof(uint));

    static mis_builder_cache *mbc
        = (mis_builder_cache *)calloc(g->order, sizeof(mis_builder_cache));

    #if 1
    for (uint i=0; i < g->order; i++)
    {
        mbc[i].list_len = 0;
    }
    #if 0
    mbc[i].alloc_len = 0;
    mbc[i].IS = NULL;
    mbc[i].Bucket = NULL;
    #endif
}
#endif

uint adj_list[g->order][33];

for (uint i=0; i < g->order-1; i++)
{

```

```

uint x = i + 1;
adj_list[i][0] = 0;

uint adj_x = adj(g, x);
foreach_vertex(g, y)
{
    if (y > i) break;

    if (elem_of(y, adj_x))
    {
        uint len = ++adj_list[i][0];
        adj_list[i][len] = y;
    }
}

build_mis_table_helper(g, 0, IS, Bucket, mbc, adj_list);
//build_mis_table_helper(NULL, 0, NULL, NULL, NULL);

#if 0
for (uint i=0; i < g->order; i++)
{
    for (uint j=0; j < mbc[i].alloc_len; j++)
    {
        free(mbc[i].IS[j]);
        free(mbc[i].Bucket[j]);
    }

    free(mbc[i].IS);
    free(mbc[i].Bucket);
}
#endif

_free(mbc);

_free(Bucket);
_free(IS);
}
#endif
#endif

void free_mis_table(Graph *g)
{
    if (g != NULL)
    {
        _free(g->mis_table);
        g->mis_table = NULL;
    }
}

//=====//
//                               h223_4                               //
//=====//

/*
* Return true if graph g is a member of H(2,2,3;4).
*
* Only graphs that do not contain K_4 must be considered.
* Call is_k_4_free() first to check.
*
* build_mis_table() and init_tri_table() must be called before
* this function.
*/

#define foreach_mis_pair(x,y) foreach_pair(x,y, g->mis_table_size)

bool h223_4(Graph *g, const bool print)
{
    if (g->num_triangles == 0) return false;

    //
    // loop over pairs of mis's
    //
    foreach_mis_pair(i, j)
    {
        //
        // union the two mis's and check the remaining vertices
        //
        const uint k = V(g) & ~(g->mis_table[i] | g->mis_table[j]);

        if (triangle_free_cached(g, k)) return false;
    }

#if 0
    if (print)
    {
        print_set(g->order, mis_table[i]); printf("\t");
        print_set(g->order, mis_table[j]); printf("\t");
        print_set(g->order, k); printf("\n");
    }
#endif

    return true;
}

//=====//
//                               maximal                               //
//=====//

/*
* Given a graph g which is known not to contain K_n, add an edge
* and return if the result causes g to contain K_n
*/
bool maximal(Graph *g, uint n)
{
    for (uint v0=0; v0 < g->order-1; v0++)
    {
        uint row = adj(g, v0);
        for (uint v1=v0+1; v1 < g->order; v1++)

```

```

    {
        if (!connected(v1, row))
        {
            connect(g, v0, v1);

            bool free = is_k_n_free(g,
                n,
                set_elem(v0)
                | set_elem(v1)
                | row
                | adj(g, v1));

            disconnect(g, v0, v1);

            if (free) return false;
        }
    }
}

return is_k_n_free(g, n, all_vertices(g));
}

//=====//
//                               //
//                               //
//=====//

void extend_to_maximals(Graph *g, uint n)
{
    bool max = true;

    for (uint v0=0; v0 < g->order-1; v0++)
    {
        uint row = adj(g, v0);
        for (uint v1=v0+1; v1 < g->order; v1++)
        {
            if (!connected(v1, row))
            {
                connect(g, v0, v1);

                bool free = is_k_n_free(g,
                    n,
                    set_elem(v0)
                    | set_elem(v1)
                    | adj(g,v0)
                    | adj(g,v1));

                if (free)
                {
                    max = false;
                    extend_to_maximals(g, n);

                    if (verbosity >= 3)
                    {
                        write_graph(g, stdout);
                        max = true;
                    }
                }
            }
        }
    }
}

}

disconnect(g, v0, v1);
}
}

if (max && is_k_n_free(g, n, all_vertices(g))) write_graph(g, stdout);
}

//=====//
//                               //
//                               //
//=====//

/*
 * must call init_tri_table(), build_tri_table(), and build_mis_table() first
 */
bool minimal(Graph *g)
{
    char *tmp_tri_table = NULL;
    uint *tmp_triangles = NULL;
    uint tmp_num_triangles = 0;
    uint *tmp_mis_table = NULL;
    uint tmp_mis_table_size = 0;

    swap(g->tri_cache, tmp_tri_table);

    swap(g->triangles, tmp_triangles);
    swap(g->num_triangles, tmp_num_triangles);

    swap(g->mis_table, tmp_mis_table);
    swap(g->mis_table_size, tmp_mis_table_size);

    for (uint v0=0; v0 < g->order-1; v0++)
    {
        uint row = adj(g, v0);
        for (uint v1=v0+1; v1 < g->order; v1++)
        {
            if (connected(v1, row))
            {
                disconnect(g, v0, v1);

                init_tri_table(g);
                build_tri_table(g);
                build_mis_table(g);

                bool folkman = h223_4(g, false);

                connect(g, v0, v1);

                if (folkman)
                {
                    free_tri_table(g);
                }
            }
        }
    }
}

```

```

        free_mis_table(g);

        swap(g->tri_cache, tmp_tri_table);
        swap(g->triangles, tmp_triangles);
        swap(g->num_triangles, tmp_num_triangles);
        swap(g->mis_table, tmp_mis_table);
        swap(g->mis_table_size, tmp_mis_table_size);

        return false;
    }
}

free_tri_table(g);
free_mis_table(g);

swap(g->tri_cache, tmp_tri_table);
swap(g->triangles, tmp_triangles);
swap(g->num_triangles, tmp_num_triangles);
swap(g->mis_table, tmp_mis_table);
swap(g->mis_table_size, tmp_mis_table_size);

return h223_4(g, false);
}

//=====//
//                               extend_mis_table                               //
//=====//

static void extend_mis_table(Graph *g, uint x, uint y,
                             uint *tmp_mis_table, uint tmp_mis_table_size)
{
#define ADD_MIS(x) g->mis_table[g->mis_table_size++] = (x)

    uint ex = set_elem(x),
          ey = set_elem(y);

    for (uint k=0; k < tmp_mis_table_size; k++)
    {
        uint mis = tmp_mis_table[k];

        // 1
        if (!elem_of(x, mis) && !elem_of(y, mis))
            ADD_MIS(mis);

        // 2.1
        if (elem_of(x, mis) && !elem_of(y, mis))
        {
            uint s = N(g, ey) & mis;
            if (s == ex) ADD_MIS(mis | ey);
            else if (bit_count(s) >= 2) ADD_MIS(mis);
        }

        // 2.2
        if (!elem_of(x, mis) && elem_of(y, mis))
        {
            uint s = N(g, ex) & mis;
            if (s == ey) ADD_MIS(mis | ex);
            else if (bit_count(s) >= 2) ADD_MIS(mis);
        }

        // 3.1
        if (elem_of(x, mis) && !elem_of(y, mis))
        {
            uint S = (mis & ~(N(g, ey) & ~ex)) | ey;
            if (N(g, S) == (V(g) & ~S)) ADD_MIS(S | ey);
        }
    }

    unique_sort(g->mis_table, &g->mis_table_size);
}

//=====//
//                               reduce_to_minimals                               //
//=====//

uint reduce_to_minimals(Graph *g, bool all, FILE *out)
{
    bool min = true;
    uint count = 0;

    char *tmp_tri_table = NULL;
    uint *tmp_triangles = NULL;
    uint tmp_num_triangles = 0;
    uint *tmp_mis_table = NULL;
    uint tmp_mis_table_size = 0;

    swap(g->tri_cache, tmp_tri_table);
    swap(g->triangles, tmp_triangles);
    swap(g->num_triangles, tmp_num_triangles);
    swap(g->mis_table, tmp_mis_table);
    swap(g->mis_table_size, tmp_mis_table_size);

    for (uint v0=0; v0 < g->order-1; v0++)
    {
        uint row = adj(g, v0);
        for (uint v1=v0+1; v1 < g->order; v1++)
        {
            if (connected(v1, row))
            {
                uint vs = set_elem(v0) | set_elem(v1);

                init_tri_table(g);

                for (uint i=0; i < tmp_num_triangles; i++)
                {
                    if (!subset(vs, tmp_triangles[i]))
                        g->triangles[g->num_triangles++] = tmp_triangles[i];
                }
            }
        }
    }
}

```

```

init_mis_table(g);
g->mis_table_size = 0;
extend_mis_table(g, v0, v1, tmp_mis_table, tmp_mis_table_size);

// with the caching scheme for N()
// THIS MUST COME SOMETIME AFTER extend_mis_table()
disconnect(g, v0, v1);

if (h223_4(g, false))
{
    min = false;

    uint *tmp_gN = NULL;
    swap(g->N, tmp_gN);
    build_set_neighbours(g);
    count += reduce_to_minimals(g, all);
    _free(g->N);
    swap(g->N, tmp_gN);

    if (all)
    {
        write_graph(g, out);
        min = true;
    }
}

connect(g, v0, v1);
}
}

free_tri_table(g);
free_mis_table(g);

swap(g->tri_cache, tmp_tri_table);
swap(g->triangles, tmp_triangles);
swap(g->num_triangles, tmp_num_triangles);
swap(g->mis_table, tmp_mis_table);
swap(g->mis_table_size, tmp_mis_table_size);

if (min && h223_4(g, false)) write_graph(g, out), count++;

return count;
}

//=====//
//                               read_graph                               //
//=====//

/*
 * Construct a graph by reading from in assuming graph6 format without
 * a header.
 */
Graph *read_graph(FILE *in, Graph *g)
{
    char bytes[518];

    if (feof(in)) return NULL;

    fgets(bytes, 517, in);

    if (feof(in)) return NULL;

    if (bytes == NULL) return NULL;

    //fprintf(stderr, "[%i]", bytes);

    return read_graph(bytes, g);
}

//=====//
//                               read_graph                               //
//=====//

Graph *read_graph(char *bytes, Graph *g)
{
#define GET_BYTE bytes[byte_index++]

    int byte_index=0;

    if (bytes == NULL) return NULL;

    uint order = GET_BYTE;

    if (order == 126)
    {
        order = (GET_BYTE << 12) - 63;
        order |= (GET_BYTE << 6) - 63;
        order |= (GET_BYTE << 0) - 63;
    }
    else
    {
        order -= 63;
    }

    g = make_graph(order, -1, g);

    uint i=0, j=0;

    while (true)
    {
        char c = GET_BYTE;
        if (c == '\n') break;

        c -= 63;

        for (char n=(1 << 5); n != 0; n >>= 1)
        {
            if (i == j)

```

```

        {
            i = 0;
            j++;
            if (j == order) break;
        }

        if (c & n) add_edge(g, i, j);

        i++;
    }
}

return g;
}

//=====//
//                               write_graph                               //
//=====//

/*
 * Write a graph to out using graph6 format without a header
 */
#define PUT_BYTE(x) *bytes_ptr++ = x
void write_graph(const Graph *g, FILE *fp)
{
    char bytes[40];
    char *bytes_ptr = bytes;

    if (g->order < 63)
        PUT_BYTE((char)(g->order + 63));
    else
    {
        PUT_BYTE((char)126);
        PUT_BYTE((char)((g->order >> 12)&0xFF + 63));
        PUT_BYTE((char)((g->order >> 6)&0xFF + 63));
        PUT_BYTE((char)((g->order >> 0)&0xFF + 63));
    }

    uint n=6;
    char c=0;

    for (uint j=1; j < g->order; j++)
    {
        #if 1
            uint row = adj(g, j);
            for (uint i=0; i < j; i++)
            {
                n--;

                //c |= (int)connected_to(g, i, j) << n;
                c |= (int)connected(i, row);

                if (n == 0)
                {
                    PUT_BYTE(c + 63);

                    c = 0;
                    n = 6;
                }
                c <<= 1;
            }
        #endif

        if (n != 6) PUT_BYTE((c << (n-1)) + 63);

        PUT_BYTE('\n');
        //PUT_BYTE('\0');
        //out << bytes;
        //fprintf(fp, bytes);
        fwrite(bytes, 1, bytes_ptr - bytes, fp);
    }

    //=====//
    //                               write_graph_file                               //
    //=====//

    /*
     * Write a graph to file filename using graph6 format without a header
     */
    void write_graph_file(const Graph *g, char *filename, char *mode="w")
    {
        FILE *fp = fopen(filename, mode);
        if (fp != NULL)
        {
            write_graph(g, fp);
            fclose(fp);
        }
    }

    //=====//
    //                               in_hX_4                               //
    //=====//

    bool in_hX_4(f_hX_4 f, Graph *g, uint f_repeat, double *t, bool check_k4, bool print)
    {
        if (g->order < 4)
        {
            fprintf(stderr, "Order of graph < 4.\n");
            return false;
        }

        if (check_k4 && has_k_4(g, all_vertices(g)))
        {
            return false;
        }

        if (g->max_degree == 0) return false;
    }
}

```

CT
CT

```

bool ret=false;
double t0 = CPUTIME;
init_tri_table(g);
build_tri_table(g);
build_mis_table(g);

for (uint i=0; i < f_repeat; i++)
    ret = f(g, print);
double t1 = CPUTIME;

*t = t1-t0;

return ret;
}

//=====//
//                               h23_4                               //
//=====//

bool h23_4(Graph *g, bool print)
{
    //
    // loop over mis's
    //
    for (uint i=0; i < g->mis_table_size-1; i++)
    {
        //
        // remove the current mis from the set of vertices
        // and check the remaining vertices for triangles.
        //
        uint k = all_vertices(g) ^ g->mis_table[i];

        if (triangle_free_cached(g, k)) return false;
    }

    return true;
}

//=====//
//                               h22_4                               //
//=====//

bool h22_4(Graph *g, bool print)
{
    //
    // loop over mis's
    //
    for (uint i=0; i < g->mis_table_size-1; i++)
    {
        uint k = all_vertices(g) ^ g->mis_table[i];

        for (uint j=0; j < g->order; j++)
            if (elem_of(j, k) && (k & adj(g, j)))
                return false;
    }
}

return true;
}

//=====//
//                               make_gamma3_graph                               //
//=====//

Graph *make_gamma3_graph()
{
    Graph *g = make_graph(14);
    for (int i=0; i < 7; i++)
    {
        add_edge(g, i, (i+2)%7);
        add_edge(g, i, (i+3)%7);
        add_edge(g, i, (i+4)%7);
        add_edge(g, i, (i+5)%7);
    }

    for (int i=0; i < 7; i++)
    {
        add_edge(g, i+7, (i+1)%7);
        add_edge(g, i+7, (i+2)%7);
        add_edge(g, i+7, (i+5)%7);
        add_edge(g, i+7, (i+6)%7);
    }

    return g;
}

//=====//
//                               reduce                               //
//=====//

void reduce(Graph *g, FILE *out)
{
    assert(g->order >= 2);

    Graph *g0 = make_graph(g->order);

    foreach_vertex(g, v)
    {
        copy_graph(g, g0);

        remove_vertex(g0, v);
        write_graph(g0, out);
    }

    free_graph(g0);
}

//=====//
//                               extend                               //
//=====//

```

```

void extend(Graph *g, FILE *out)
{
    Graph *g0 = make_graph(g->order);

    foreach_vertex(g, v)
    {
        copy_graph(g, g0);

        add_vertex(g);
        write_graph(g0, out);
    }

//=====//
//                               sort                               //
//=====//

int compar(const void *x, const void *y)
{
    //fprintf(stderr, "%i %i\n", *((uint *)x), *((uint *)y));
    return *((uint *)x) - *((uint *)y);
}

void unique_sort(uint *list, uint *len)
{
    if (*len == 0) return;

    //fprintf(stdout, "%x %i\n", list, *len);
    qsort(list, *len, sizeof(uint), &compar);

    uint cur = list[0];
    uint j=1;
    for (uint i=1; i < *len; i++)
    {
        if (list[i] > cur)
        {
            cur = list[i];
            list[j++] = cur;
        }
    }

    *len = j;
}

#if 0
void sort(uint *list, uint len)
{
    //int j;
    buildheap(list, len);
    for (uint i=len; i >= 1; i--)
    {
        swap(list[1], list[i]);
        if (i > 1) heapify(list, 1, i-1);
    }
}

#endif

#if 0
bcount = 0;
for (i=1; i<=ind;)
{
    j = i;
    bcount++;
    if (bcount>=BSIZE)
    {
        printf("block number %d too small\n",BSIZE);
        exit(1);
    }
    while (j<=ind && (!less(j,i))) j++;
    bsize[bcount] = j-i;
    bstart[bcount] = i;
    i = j;
}
#endif

void buildheap(uint *list, uint len)
{
    for (uint i=len/2; i; i--)
        heapify(list, i, len);
}

void heapify(uint *list, uint i, uint j)
{
    uint p = i << 1;
    uint q = p + 1;

    if (p > j) return;

    uint k = i;

    if (list[p] < list[i]) k = p;
    if (q <= j && list[q] < list[k]) k = q;
    if (k == i) return;

    swap(list[i], list[k]);

    heapify(list, k, j);
}

//=====//
//                               chromatic                               //
//=====//

uint chromatic(const Graph *g)
{
    //uint *color = (uint *)calloc(g->order, sizeof(uint));
    //
    // this is faster
    static uint color[32 * sizeof(uint)];
}

```

```

static uint color_class[32 * sizeof(uint)];

memset(color, 0, 32 * sizeof(uint));
memset(color_class, 0, 32 * sizeof(uint));

uint k=1;
foreach_vertex(g, v)
{
    uint h=1;
    uint adj_v = adj(g, v);
    while (h < k && intersects(adj_v, color_class[h]))
    {
        h++;
    }
    #if 0
        bool done = true;
        foreach_elem(g, a, adj(g,v))
        {
            if (color[a] == h)
            {
                h++;
                done = false;
                break;
            }
        }
        if (done) break;
    #endif
    if (h == k)
    {
        k++;
        color_class[h] = 0;
    }

    add_elem(v, color_class[h]);
    color[v] = h;
}

//free(color);

return k-1;
}

void print_graph_info(FILE *fp, Graph *g, char *show_info, char *eol)
{
    uint i=0;
    bool good = false;
    bool label = false;
    if (strlen(show_info) != 0 && show_info[0] == ':')
    {
        label = true;
        i++;
    }
}

```

```

for (; i < strlen(show_info); i++)
{
    switch (show_info[i])
    {
        case 'o':
            if (label) fprintf(fp, "%c:", show_info[i]);
            good = true;
            fprintf(fp, "%i ", g->order);
            break;
        case 'e':
            if (label) fprintf(fp, "%c:", show_info[i]);
            good = true;
            fprintf(fp, "%i ", g->edge_count);
            break;
        case 't':
            if (label) fprintf(fp, "%c:", show_info[i]);
            good = true;
            if (g->num_triangles == 0)
            {
                init_tri_table(g);
                build_tri_table(g);
            }
            fprintf(fp, "%i ", g->num_triangles);
            break;
        case 'm':
            if (label) fprintf(fp, "%c:", show_info[i]);
            good = true;
            if (g->mis_table_size == 0)
                build_mis_table(g);
            fprintf(fp, "%i ", g->mis_table_size);
            break;
        case 'M':
            if (label) fprintf(fp, "%c:", show_info[i]);
            good = true;
            if (g->mis_table_size == 0)
                build_mis_table(g);
            fprintf(fp, "%i ", g->max_mis);
            break;
        case 'D':
            if (label) fprintf(fp, "%c:", show_info[i]);
            good = true;
            fprintf(fp, "%i ", g->max_degree);
            break;
        case 'd':
            if (label) fprintf(fp, "%c:", show_info[i]);
            good = true;
            fprintf(fp, "%i ", g->min_degree);
            break;
        case 'X':
            if (label) fprintf(fp, "%c:", show_info[i]);
            good = true;
            if (g->chi == 0)
            {
                if (g->mis_table_size == 0)
                    build_mis_table(g);
            }
            break;
    }
}

```

CT
∞

```

        g->chi = chi(g);
    }

    fprintf(fp, "%i ", g->chi);
    break;
}
}
if (good) fprintf(fp, eol);
}

static uint cover(Graph *g, uint pat, uint mis, uint m, uint size)
{
    uint i, j, locpat, mlow;

    size--;
    mlow = m;
    for (i = 0; i < m; i++)
    {
        j = g->mis_table[mis];
        mis = mis + 1;
        mlow--;
        //fprintf(stderr, "$$ %i %i %i\n", g->mis_table_size, mis, mlow);
        locpat = pat & ~j;
        if (!locpat)
            return 1;
        if (!size || !mlow)
            continue;
        if (cover(g, locpat, mis, mlow, size))
            return 1;
    }
    return 0;
}

uint chi(Graph *g)
{
    uint j = 1;

    uint i=0;

    while (!cover(g, V(g), i, g->mis_table_size, j)) j++;

    return j;
}

//=====//
//                               TESTS                               //
//=====//

#if TESTS

static bool independent_set(Graph *g, uint subset)
{

```

```

    register int j, k;

    static char *table = NULL;

    if (table == NULL)
    {
        table = (char *)malloc((1 << g->order) * sizeof(char));
        assert(table != NULL);

        memset(table, 2, (1 << g->order) * sizeof(char));
    }
    else
    {
        if (table[subset] != 2)
            return (bool)table[subset];
    }

    for (j=g->order-1; j >= 1; j--)
    {
        if (elem_of(j, subset))
        {
            for (k=j-1; k >= 0; k--)
            {
                if (elem_of(k, subset) && connected_to(g, j, k))
                {
                    //print_bits(32, subset);
                    //fprintf(stdout, "%i\n", subset);
                    table[subset] = 0;
                    return false;
                }
            }
        }
    }

    table[subset] = 1;
    return true;
}

static bool check_indep_sets(Graph *g)
{
    build_mis_table(g);
    for (uint i=0; i < g->mis_table_size; i++)
        if (!independent_set(g, g->mis_table[i])) return false;

    return true;
}

int verbosity = 0;

int main(int argc, char **argv)
{
    Graph *g0, *g1, *g2;

    #if 0
    g0 = make_gamma3_graph();

```

```

        write_graph(g0, stdout);
        build_mis_table(g0);
        return 0;
    #else
    #if 0
        g0 = make_graph(3);
        connect(g0, 0, 1);
        connect(g0, 0, 2);
        connect(g0, 1, 2);
        build_mis_table(g0);
        return 0;
    #endif
    #endif

    #if 1
        g0 = make_graph(2);
        print_graph(g0);
        remove_vertex(g0, 0);
        print_graph(g0);

        g0 = make_graph(2);
        connect(g0, 0, 1);
        print_graph(g0);
        remove_vertex(g0, 0);
        print_graph(g0);

        g0 = make_graph(3);
        connect(g0, 0, 1);
        connect(g0, 0, 2);
        connect(g0, 1, 2);
        print_graph(g0);
        remove_vertex(g0, 0);
        print_graph(g0);
    #endif

    #if 0
        g0 = make_graph(10);

        //
        // init_k3_table()
        //
        assert(g0->tri_cache == NULL);
        init_tri_table(g0);
        assert(g0->tri_cache != NULL);

        g1 = make_graph(g0->order);

        char *ptr = tri_table;
        init_tri_table(g1);
        assert(g->tri_cache != NULL);
        assert(ptr == g->tri_cache);

        g2 = make_graph(5);

        init_tri_table(g2);
        assert(g->tri_cache != NULL);

        free_graph(g0);
        free_graph(g1);
        free_graph(g2);

        //
        // is_k_n_free()
        // has_k_n()
        //
        g0 = make_graph(1);
        assert(is_k_n_free(g0, 2, all_vertices(g0)));
        free_graph(g0);

        g0 = make_graph(2);
        assert(is_k_n_free(g0, 2, all_vertices(g0)));
        add_edge(g0, 0, 1);
        assert(has_k_n(g0, 2, all_vertices(g0)));
        free_graph(g0);

        g0 = make_graph(4);
        assert(is_k_n_free(g0, 2, all_vertices(g0)));
        assert(is_k_n_free(g0, 3, all_vertices(g0)));
        assert(is_k_n_free(g0, 4, all_vertices(g0)));
        add_edge(g0, 0, 1);
        add_edge(g0, 1, 2);
        add_edge(g0, 2, 3);
        add_edge(g0, 3, 0);
        assert(has_k_n(g0, 2, all_vertices(g0)));
        assert(is_k_n_free(g0, 3, all_vertices(g0)));
        assert(is_k_n_free(g0, 4, all_vertices(g0)));
        assert(check_indep_sets(g0));

        add_edge(g0, 0, 2);
        assert(has_k_n(g0, 2, all_vertices(g0)));
        assert(has_k_n(g0, 3, all_vertices(g0)));
        assert(is_k_n_free(g0, 4, all_vertices(g0)));
        assert(check_indep_sets(g0));

        add_edge(g0, 3, 1);
        assert(has_k_n(g0, 2, all_vertices(g0)));
        assert(has_k_n(g0, 3, all_vertices(g0)));
        assert(has_k_n(g0, 4, all_vertices(g0)));
        assert(check_indep_sets(g0));
    }

    uint list[10] = {0,1,2,3,4,5,6,7,8,9};
    uint len = 10;

    unique_sort(list, &len);
    for (uint i=0; i < len; i++) printf("%i ", list[i]);
    printf("\n");
}

```

```

uint list[10];
uint len = 0;

unique_sort(list, &len);
for (uint i=0; i < len; i++) printf("%i ", list[i]);
printf("\n");
}
{
uint list[10] = {0,0,0,1,1,1,1,2,2,2};
uint len = 10;

unique_sort(list, &len);
for (uint i=0; i < len; i++) printf("%i ", list[i]);
printf("\n");
}
{
uint list[10] = {0,9,0,1,2,1,1,4,2,4};
uint len = 10;

unique_sort(list, &len);
for (uint i=0; i < len; i++) printf("%i ", list[i]);
printf("\n");
}
#endif
}
#endif

/*
 * Argument Parsing Library Header
 *
 * Copyright 2003, 2004 Jonathan Coles
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA
 */

#ifndef ARGV_H
#define ARGV_H

void begin_args(int *argc, char ***argv, int *opt_argc, char ***opt_argv);
void end_args(char ***optv);
bool next_arg(int *argc, char ***argv, int *optc, char **optv);

#endif

/*
 * Argument Parsing Library
 *
 * Copyright 2003, 2004 Jonathan Coles
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "args.h"

void begin_args(int *argc, char ***argv, int *opt_argc, char ***opt_argv)
{
    *opt_argc = 0;
    *opt_argv = new char*[10];

    (*argc)--;
    (*argv)++;
}

void end_args(char ***optv)
{
    delete *optv;
    *optv = NULL;
}

bool next_arg(int *argc, char ***argv, int *optc, char **optv)
{
    bool need_opt = true;

    *optc = 0;

```

```

if (*argc == 0) return false;

while (*argc > 0)
{
    if (strcmp("--", **argv) && strcmp("-", **argv))
    {
        for (int i=2; i >= 1; i--)
        {
            if (!strcmp("--", **argv, i))
            {
                if (!need_opt) return true;
                need_opt = false;

                *optv = (**argv) + i;

                goto not_arg;
            }
        }

        *optv = **argv;
    }

    not_arg:

    (*optc)++;
    optv++;

    (*argc)--;
    (*argv)++;

    if (need_opt) break;
}

return true;
}

#ifndef __TIMING_H__
#define __TIMING_H__

#include <sys/time.h>
#include <sys/resource.h>

extern int getrusage();
#define CPUDEFS static struct rusage ruse;
#define CPUTIME (getrusage(RUSAGE_SELF,&ruse),\
ruse.ru_utime.tv_sec + ruse.ru_stime.tv_sec + \
1e-6 * (ruse.ru_utime.tv_usec + ruse.ru_stime.tv_usec))

#endif

/*
 * Folkman Graph Property Library Header for Large Graphs

```

```

*
*
* Copyright 2004 Jonathan Coles
*
* This program is free software; you can redistribute it and/or modify
* it under the terms of the GNU General Public License as published by
* the Free Software Foundation; either version 2 of the License, or
* (at your option) any later version.
*
* This program is distributed in the hope that it will be useful,
* but WITHOUT ANY WARRANTY; without even the implied warranty of
* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
* GNU General Public License for more details.
*
* You should have received a copy of the GNU General Public License
* along with this program; if not, write to the Free Software
* Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA
*/

#ifndef FLIBLG_H
#define FLIBLG_H

#include <stdlib.h>
#include <istream>
#include <assert.h>

using namespace std;

//=====//
//                               Graph struct                               //
//=====//

typedef struct
{
    uint *data;
    uint length;
} bool_array;

typedef struct
{
    uint *block;
    uint nblocks;
    uint len;
} flset;

typedef struct
{
    uint order;
    uint adj_size;

    uint max_degree;
    uint min_degree;
    uint edge_count;

    flset **adj;

```

```

uint *deg;

uint **edge_numbering;
uint edge_numbering_count;

flset **mis_table;
uint mis_table_size;
uint max_mis;

flset **triangles;
uint num_triangles;

} Graph;

//=====//
//                                     Prototypes                          //
//=====//

flset *new_flset(uint size);
void copy_flset(flset *A, flset *B);
void invert_set(flset *fls);
void delete_elem_flset(uint v, flset *A);
bool equal_set(flset *A, flset *B);
void free_flset(flset *fls);

void print_graph_info(FILE *fp, Graph *g, char *show_info, char *eol="\n");
void print_bits(uint m, uint n, const char *end="\n");
void print_set(flset *s, const char *end="\n");
void print_set(FILE *fp, flset *s, const char *end="\n");
void print_array(FILE *fp, uint size, uint *a, const char *end="\n");

uint bit_count(uint n);
Graph *make_graph(const uint order, const int adj_size=-1, Graph *g=NULL);
Graph *make_random_graph(const uint order, const double p);
Graph *read_graph(char *bytes, Graph *g);
void write_graph(const Graph *g, FILE *fp);
void write_graph_file(const Graph *g, char *filename, char *mode="w");
void init_tri_table(Graph *g);
void build_tri_table(Graph *g);
void free_graph(Graph *g);
void free_tri_table(Graph *g);
void free_mis_table(Graph *g);
void print_graph(const Graph *g);
void add_edge(Graph *g, const uint v0, const uint v1);
void add_edges(Graph *g, const uint v0, const uint vs);
void remove_edge(Graph *g, const uint v0, const uint v1);
int add_vertex(Graph *g);
void remove_vertex(Graph *g, const uint v0);
void update_min_degree(Graph *g);
void update_max_degree(Graph *g);
Graph *copy_graph(const Graph *g0, Graph *g=NULL);
//void complement(Graph *g);
Graph *make_g127();
Graph *make_g84_2();
Graph *make_g84(uint svertex=0);

Graph *make_g74(int svertex=-1, uint *x=NULL);
Graph *make_gX(flset *x, int svertex=0);

void number_edges(Graph *g);
uint **map_edge_numbers(Graph *g);
bool in_array(uint x, uint *a, uint l);
uint bit_count(uint n);
uint set_count(flset *A);

void init_mis_table(Graph *g, uint size=0);
void build_mis_table(Graph *g);

//=====//
//                                     Macros and inline functions          //
//=====//

#define _free(ptr) \
{ free(ptr); ptr = NULL; }

#define foreach_pair(x,y,n) \
for (uint x=0; x < n-1; x++) \
for (uint y=x+1; y < n; y++)

#define foreach_vertex(g,i) for (uint i=0; i < g->order; i++)
#define foreach_vertex_r(g,i) for (uint __i__=g->order, i=__i__-1; __i__ > 0; __i__--, i--)

#define foreach_vertex_pair(g,i,j) foreach_pair(i,j,g->order)

#define foreach_subset(i,s) for (uint i=0; i <= s; i++)

#define foreach_elem(g,u,i) \
for (uint u=0; u < g->order; u++) if (elem_of(u, i))

#define foreach_flselem(u,i) \
for (uint u=0; u < (i)->len; u++) if (elem_of(u, i))

#define for_if(c) if (c) for

/* Return true if x is an element of set */
inline bool elem_of(const uint x, const flset *set)
{ return set->block[x]>>5 & (1 << (31-(x&31))); }

inline bool empty_set(const flset *fls)
{
for (uint i=0; i < fls->nblocks; i++)
if (fls->block[i] != 0) return false;
return true;
}

/* Return true if m is a subset of s */
inline bool subset(const uint m, const uint s)

```

```

{ return m && ((s & m) == m); }

/* Add x to the set set */
inline void add_elem(const uint x, flset *set)
{ set->block[x>>5] |= (1 << (31-(x&31))); }

/* Remove x from the set set */
inline void remove_elem(const uint x, flset *set)
{ set->block[x>>5] &= ~(1 << (31-(x&31))); }

/* Return true if v0 is connected to v1 in graph g */
inline bool connected_to(const Graph *g, const uint v0, const uint v1)
{
    assert(v0 < g->order);
    assert(v1 < g->order);

    return elem_of(v0, g->adj[v1]);
}

inline bool connected(const uint v0, const flset *adj)
{ return elem_of(v0, adj); }

/* Return the degree of vertex v in graph g */
inline uint deg(const Graph *g, const uint v)
{ return g->deg[v]; }

inline flset *adj(const Graph *g, const uint v)
{ return g->adj[v]; }

inline flset *V(const Graph *g)
{
    flset *fls = new_flset(g->order);
    for (uint i=0; i < g->order; i++)
        add_elem(i, fls);
    #if 0
    for (uint i=0; i < fls->nblocks-1; i++)
        fls->block[i] = 0xFFFFFFFF;

    fls->block[fls->nblocks-1] =
    #endif
    return fls;
}

inline flset *intersection2(flset *A, flset *B)
{
    #if 0
    flset *C = new_flset(A->len);
    for (uint i=0; i < C->nblocks; i++)
        C->block[i] = A->block[i] & B->block[i];
    #else
    flset *C = new_flset(A->len);
    for (uint i=0; i < C->len; i++)
        if (elem_of(i,A) && elem_of(i,B))
            add_elem(i, C);
    #endif

    return C;
}

#include "flibLG.h"
#include <stdio.h>

void print_bits(FILE *fp, uint m, uint n, const char *end)
{
    uint mask = (1 << (m-1));
    while (mask != 0)
    {
        if (n & mask)
            fprintf(fp, "1");
        else
            fprintf(fp, "0");

        mask >>= 1;
    }

    if (end != NULL) fprintf(fp, end);
}

void print_set(flset *s, const char *end)
{
    print_set(stdout, s, end);
}

void print_set(FILE *fp, flset *s, const char *end)
{
    fprintf(fp, "{");
    bool comma=false;

    if (s->len && elem_of(0, s))
        fprintf(fp, "%s", comma ? ", " : ""), comma=true;

    for (uint mask=1; mask < s->len; mask++)

```

```

    {
        if (elem_of(mask, s))
            fprintf(fp, "%s%i", comma ? ", " : "", mask), comma=true;
    }

    fprintf(fp, "}");

    if (end != NULL) fprintf(fp, end);
}

void print_graph_info(FILE *fp, Graph *g, char *show_info, char *eol)
{
    uint i=0;
    bool good = false;
    bool label = false;
    if (strlen(show_info) != 0 && show_info[0] == ':')
    {
        label = true;
        i++;
    }

    for (; i < strlen(show_info); i++)
    {
        switch (show_info[i])
        {
            case 'o':
                if (label) fprintf(fp, "%c:", show_info[i]);
                good = true;
                fprintf(fp, "%i ", g->order);
                break;
            case 'e':
                if (label) fprintf(fp, "%c:", show_info[i]);
                good = true;
                fprintf(fp, "%i ", g->edge_count);
                break;
            case 't':
                if (label) fprintf(fp, "%c:", show_info[i]);
                good = true;
                if (g->num_triangles == 0)
                {
                    init_tri_table(g);
                    build_tri_table(g);
                }
                fprintf(fp, "%i ", g->num_triangles);
                break;
            case 'm':
                if (label) fprintf(fp, "%c:", show_info[i]);
                good = true;
                if (g->mis_table_size == 0)
                    build_mis_table(g);
                fprintf(fp, "%i ", g->mis_table_size);
                break;
            case 'D':
                if (label) fprintf(fp, "%c:", show_info[i]);
                good = true;
                fprintf(fp, "%i ", g->max_degree);
                break;
            case 'd':
                if (label) fprintf(fp, "%c:", show_info[i]);
                good = true;
                fprintf(fp, "%i ", g->min_degree);
                break;
        }
    }

    if (good) fprintf(fp, eol);
}

flset *new_flset(uint size)
{
    assert(size > 0);

    flset *fls = (flset *)malloc(sizeof(flset));
    assert(fl != NULL);
    fls->nblocks = (uint)ceil(size/32.);

    fls->block = (uint *)calloc(fls->nblocks, sizeof(uint));
    assert(fls->block != NULL);
    fls->len = size;

    return fls;
}

void invert_set(flset *fls)
{
    for (uint i=0; i < fls->len; i++)
    {
        if (elem_of(i, fls))
            remove_elem(i, fls);
        else
            add_elem(i, fls);
    }
}

#if 0
//printf("-----\n");
for (uint i=0; i < fls->nblocks-1; i++)
{
    //printf("*****\n");
    //print_bits(stderr, 32, fls->block[i], "\n");
    fls->block[i] = ~fls->block[i];
    //print_bits(stderr, 32, fls->block[i], "\n");
}

uint x = fls->block[fls->nblocks-1];
uint y = 1 << (31-((fls->len-1)&31));
//print_bits(stderr, 32, x, "\n");
//print_bits(stderr, 32, ~x, "\n");
//print_bits(stderr, 32, y, "\n");
//print_bits(stderr, 32, ~(y-1), "\n");

fls->block[fls->nblocks-1] = ~x & ~(y-1);
#endif

```

```

#endif
}

bool equal_set(flset *A, flset *B)
{
    //if (A->nblocks != B->nblocks) return false;
    //if (A->len != B->len) return false;

    if (A->len < B->len)
    {
        uint i;
        for (i=0; i < A->nblocks; i++)
            if (A->block[i] != B->block[i]) return false;

        for (; i < B->nblocks; i++)
            if (B->block[i] != 0) return false;
    }
    else
    {
        uint i;
        for (i=0; i < B->nblocks; i++)
            if (B->block[i] != A->block[i]) return false;

        for (; i < A->nblocks; i++)
            if (A->block[i] != 0) return false;
    }

    return true;
}

void delete_elem_flset(uint v, flset *A)
{
    assert(0 <= v && v <= A->len);

    //
    // This is tricky so let's take an example. Let g be a graph of order 10,
    // v0=4, and one row in the adjacency table be 1001111010.
    //
    // s = 1001111010 ^ (1 << 4) = 1001111010 ^ 0000010000 = 1001101010
    // m = (1 << (4+1))-1 = (1 << 5)-1 = 0000011111
    // adj[] = ((s & ~m) >> 1) | (s & m)
    //         = ((1001101010 & 1111100000) >> 1) | (1001101010 & 0000011111)
    //         = (1001100000 >> 1) | 0000001010
    //         = 100110000 | 0000001010
    //         = 100111010
    //
    // The result is 9 bits with v0 dropped.
    //
    uint blk = v >> 5;
    uint index = (1 << (31-(v&31)));

    uint s = A->block[blk] & ~index;
    uint m = index - 1;
    A->block[blk] = (s & ~m) | ((s & m) << 1);

    for (uint j=blk; j < A->nblocks-1; j++)
    {
        A->block[j] |= (A->block[j+1] & (1 << 31)) >> 31;
        A->block[j+1] <<= 1;
    }

    A->len--;
}

void copy_flset(flset *A, flset *B)
{
    assert(B->nblocks <= A->nblocks);
    for (uint i=0; i < B->nblocks; i++)
        B->block[i] = A->block[i];

    B->len = A->len;
}

void free_flset(flset *fls)
{
    if (fls != NULL)
    {
        if (fls->block != NULL)
            _free(fls->block);

        _free(fls);
    }
}

#if 0
void copy_flset(flset fls0, flset fls1)
{
    fls1 = new_flset(fls1.len, fls1);
    memcpy(fls1.blocks, fls0.blocks, fls0.nblocks * sizeof(uint));
}
#endif

/*
 * Although connect is inline it is not made available in the header because
 * it must only be used as a helper and/or optimization. By itself, it does
 * not preserve the consistency of the graph. Be careful of how it is used.
 */
inline void connect(Graph *g, uint v0, uint v1)
{
    assert(v0 < g->order);
    assert(v1 < g->order);

    add_elem(v0, g->adj[v1]);
    add_elem(v1, g->adj[v0]);
    g->deg[v0]++;
    g->deg[v1]++;
}

//=====
// disconnect //

```

```

//=====//
/*
 * Although disconnect is inline it is not made available in the header because
 * it must only be used as a helper and/or optimization. By itself, it does
 * not preserve the consistency of the graph. Be careful of how it is used.
 */
inline void disconnect(Graph *g, uint v0, uint v1)
{
    assert(v0 < g->order);
    assert(v1 < g->order);

    remove_elem(v0, g->adj[v1]);
    remove_elem(v1, g->adj[v0]);
    g->deg[v0]--;
    g->deg[v1]--;
}

//=====//
//                                     make_graph                               //
//=====//

Graph *make_graph(const uint order, const int adj_size, Graph *g)
{
    assert(order != 0);

    if (g == NULL)
    {
        g = (Graph *)calloc(1, sizeof(Graph));
        assert(g != NULL);

        g->order = order;
        g->adj_size = adj_size == -1
            ? g->order
            : adj_size;

        g->adj = (flset **)malloc(g->adj_size * sizeof(flset **));
        assert(g->adj != NULL);

        for (uint i=0; i < g->adj_size; i++)
        {
            g->adj[i] = new_flset(g->adj_size);
            assert(g->adj[i] != NULL);
        }

        g->deg = (uint *)calloc(g->adj_size, sizeof(uint));
        assert(g->deg != NULL);
    }
    else
    {
        assert(g->adj_size >= order);
        g->order = order;
        for (uint i=0; i < g->adj_size; i++)
            memset(g->adj[i], 0, g->adj[i]->nblocks * sizeof(uint));

        memset(g->deg, 0, g->adj_size * sizeof(uint));
    }

    return g;
}

//=====//
//                                     free_graph                               //
//=====//

void free_graph(Graph *g)
{
    if (g != NULL)
    {
        free_tri_table(g);

        for (uint i=0; i < g->adj_size; i++)
            free_flset(g->adj[i]);

        _free(g->adj);
        _free(g->deg);
        _free(g);
    }
}

//=====//
//                                     make_random_graph                         //
//=====//

Graph *make_random_graph(const uint order, const double p)
{
    Graph *g = make_graph(order);

    foreach_vertex_pair(g,x,y)
    {
        if (((float)(1.0*rand()/(RAND_MAX+1.0)) >= 1.0-p)
            add_edge(g, x,y);
    }

    return g;
}

//=====//
//                                     print_graph                               //
//=====//

void print_graph(const Graph *g)
{
    uint i, j;

    for (i=0; i < g->order; i++)
    {
        printf("%i:\t", i);
    }
}

```

```

        for (j=0; j < g->order; j++)
            printf("%i", elem_of(j, adj(g, i)) ? 1 : 0);

        printf("\n");
    }
}

```

```

void add_edge(Graph *g, const uint v0, const uint v1)
{
    assert(v0 < g->order);
    assert(v1 < g->order);
    assert(v0 != v1);
    assert(!connected_to(g, v0, v1));

    connect(g, v0, v1);

    g->max_degree = max(g->max_degree, max(deg(g, v0), deg(g, v1)));

    update_min_degree(g);

    g->edge_count++;
}

```

```

//=====//
//                               read_graph                               //
//=====//

```

```

Graph *read_graph(char *bytes, Graph *g)
{
#define GET_BYTE bytes[byte_index++]

    int byte_index=0;

    if (bytes == NULL) return NULL;

    uint order = GET_BYTE;

    if (order == 126)
    {
        order = (GET_BYTE << 12) - 63;
        order |= (GET_BYTE << 6) - 63;
        order |= (GET_BYTE << 0) - 63;
    }
    else
    {
        order -= 63;
    }

    g = make_graph(order, -1, g);

    uint i=0, j=0;

    while (true)
    {

```

```

        char c = GET_BYTE;
        if (c == '\n') break;

        c -= 63;

        for (char n=(1 << 5); n != 0; n >>= 1)
        {
            if (i == j)
            {
                i = 0;
                j++;
                if (j == order) break;
            }

            if (c & n) add_edge(g, i, j);

            i++;
        }
    }

    return g;
}

//=====//
//                               write_graph                               //
//=====//

/*
 * Write a graph to out using graph6 format without a header
 */
#define PUT_BYTES(x) *bytes_ptr++ = (char)((x)&0xFF)
#define PUT_BYTE(x) *bytes_ptr++ = (char)((x)&0x3F)+63
void write_graph(const Graph *g, FILE *fp)
{
    //char bytes[g->order*(g->order-1)/2 / 6 + 10];
    char *bytes = (char *)malloc(g->order*(g->order-1)/2 / 6 + 10);

    assert(bytes != NULL);
    char *bytes_ptr = bytes;

    if (g->order < 63)
        PUT_BYTE(g->order);
    else
    {
        PUT_BYTES(126);
        PUT_BYTE(g->order >> 12);
        PUT_BYTE(g->order >> 6);
        PUT_BYTE(g->order >> 0);
    }

    uint n=6;
    char c=0;

    for (uint j=1; j < g->order; j++)
    {

```

```

#if 1
    flset *row = adj(g, j);
    for (uint i=0; i < j; i++)
    {
        n--;

        //c |= (int)connected_to(g, i, j) << n;
        c |= (int)connected(i, row);

        if (n == 0)
        {
            PUT_BYTE(c);
            c = 0;
            n = 6;
        }

        c <<= 1;
    }
#endif

    if (n != 6) PUT_BYTE(c << (n-1));

    PUT_BYTES('\n');
    //PUT_BYTE('\0');
    //out << bytes;
    fwrite(bytes, 1, bytes_ptr - bytes, fp);

    free(bytes);
}

//=====//
//                               //
//                               //
//=====//

/*
 * Write a graph to file filename using graph6 format without a header
 */
void write_graph_file(const Graph *g, char *filename, char *mode)
{
    FILE *fp = fopen(filename, mode);
    if (fp != NULL)
    {
        write_graph(g, fp);
        fclose(fp);
    }
}

//=====//
//                               //
//                               //
//=====//

/* Remove the edge between v0 and v1 */
void remove_edge(Graph *g, const uint v0, const uint v1)
{
    assert(v0 < g->order);
    assert(v1 < g->order);
    assert(connected_to(g, v0, v1));

    //if (!connected_to(g, v0, v1)) return;

    disconnect(g, v0, v1);

    g->min_degree = min(g->min_degree, min(deg(g, v0), deg(g, v1)));

    update_max_degree(g);

    g->edge_count--;
}

//=====//
//                               //
//                               //
//=====//

/* (helper) Recalculate minimum degree in the graph */
void update_min_degree(Graph *g)
{
    g->min_degree = g->order-1;
    register uint *degs = g->deg;
    register uint *end_deg = degs + g->order;
    while (degs != end_deg)
    {
        register uint deg = *degs++;
        if (deg < g->min_degree)
            g->min_degree = deg;
    }
}

//=====//
//                               //
//                               //
//=====//

/* (helper) Recalculate maximum degree in the graph */
void update_max_degree(Graph *g)
{
    g->max_degree = 0;
    register uint *degs = g->deg;
    register uint *end_deg = degs + g->order;
    while (degs != end_deg)
    {
        uint deg = *degs++;
        if (deg > g->max_degree)
            g->max_degree = deg;
    }
}

//=====//
//                               //
//                               //
//=====//

//                               //
//                               //
//=====//

```

```

void remove_vertex(Graph *g, const uint v0)
{
    assert(v0 < g->order);

    //memmove(g->adj + v0, g->adj + v0 + 1, (g->order - 1 - v0) * sizeof(uint));

    g->edge_count -= deg(g, v0);

    free_flset(g->adj[v0]);
    for (uint i=v0; i < g->order-1; i++)
    {
        g->adj[i] = g->adj[i+1];
        g->deg[i] = g->deg[i+1];
    }
    g->deg[g->order-1] = 0;
    g->adj[g->order-1] = new_flset(g->adj_size);

    g->order--;

    g->min_degree = g->order;
    g->max_degree = 0;
    foreach_vertex(g, v)
    {
        delete_elem_flset(v0, g->adj[v]);
        g->deg[v] = set_count(g->adj[v]);
        if (g->deg[v] < g->min_degree) g->min_degree = g->deg[v];
        if (g->deg[v] > g->max_degree) g->max_degree = g->deg[v];
    }

    //update_min_degree(g);
    //update_max_degree(g);
}

void number_edges(Graph *g)
{
    if (g->edge_numbering == NULL)
    {
        g->edge_numbering = (uint **)malloc(g->order * sizeof(uint));

        for (uint i=0; i < g->order; i++)
            g->edge_numbering[i] = (uint *)calloc(g->order, sizeof(uint));
    }

    uint c = 1;
    foreach_vertex_pair(g,x,y)
    {
        if (connected_to(g, x, y))
        {
            g->edge_numbering[x][y] = c;
            g->edge_numbering[y][x] = c;
            c++;
        }
    }

    g->edge_numbering_count = c-1;
}

uint **map_edge_numbers(Graph *g)
{
    uint **map = (uint **)malloc((g->edge_count + 1) * sizeof(uint *));
    for (uint i=0; i < g->edge_count+1; i++)
        map[i] = (uint *)calloc(2, sizeof(uint));

    foreach_vertex_pair(g,x,y)
    {
        if (connected_to(g, x, y))
        {
            map[g->edge_numbering[x][y]][0] = x;
            map[g->edge_numbering[x][y]][1] = y;
        }
    }

    return map;
}

//=====//
//                               remove_vertex                               //
//=====//

#if 0
void remove_vertex(Graph *g, const uint v0)
{
    assert(v0 < g->order);

    //memmove(g->adj + v0, g->adj + v0 + 1, (g->order - 1 - v0) * sizeof(uint));

    #if 1
    for (uint i=v0; i < g->order-1; i++)
        memcpy(g->adj[i], g->adj[i+1], g->adj_size * sizeof(uint));
    //g->adj[i] = g->adj[i+1];
    #endif

    g->order--;

    //
    // This is tricky so let's take an example. Let g be a graph of order 10,
    // v0=4, and one row in the adjacency table be 1001111010.
    //
    // s = 1001111010 ^ (1 << 4) = 1001111010 ^ 0000010000 = 1001101010
    // m = (1 << (4+1))-1 = (1 << 5)-1 = 0000011111
    // adj[] = ((s & ~m) >> 1) | (s & m)
    //         = ((1001101010 & 111100000) >> 1) | (1001101010 & 0000011111)
    //         = (1001100000 >> 1) | 0000001010
    //         = 100110000 | 0000001010
    //         = 100111010
    //
    // The result is 9 bits with v0 dropped.
}

```

```

//
for (uint i=0; i < g->order; i++)
{
    uint s = adj(g, i) & ~set_elem(v0);
    uint m = (1 << (v0+1)) - 1;
    g->adj[i] = ((s & ~m) >> 1) | (s & m);
}
}
#endif

//=====//
//                               build_mis_table                               //
//=====//

void init_mis_table(Graph *g, uint size)
{
    if (size == 0)
        size = 1 << (uint)min((int)g->order, 24);

    //printf("init_mis_table: %i\n", size);
    g->mis_table = (flset **)realloc((void *)g->mis_table, size * sizeof(flset *));
    assert(g->mis_table != NULL);
    g->mis_table_size = 0;
    g->max_mis = 0;
}

void all_cliques(Graph *g, uint l, uint x, flset *X, flset *N_l, flset *C_l, flset **B)
{
    flset *N_l2;

    if (l == 0) N_l2 = V(g);
    else      N_l2 = intersection2(adj(g, x), N_l);

    //print_set(N_l2);
    if (empty_set(N_l2))
    {
        //print_set(X);
        uint i=g->mis_table_size;
        g->mis_table[i] = new_flset(X->len);
        copy_flset(X, g->mis_table[i]);

        g->mis_table_size++;
        //g->mis_table[g->mis_table_size++] = X;
        //uint count = bit_count(X);
        //if (count && count > g->max_mis) g->max_mis = count;
    }

    flset *C_l2;

    if (l == 0) C_l2 = V(g);
    else      C_l2 = intersection3(adj(g, x), B[x], C_l);

    //printf("%i: ", l);
    //print_set(C_l2);

    foreach_elem(g, e, C_l2)
    {
        add_elem(e, X);
        all_cliques(g, l+1, e, X, N_l2, C_l2, B);
        remove_elem(e, X);
    }

    free_flset(N_l2);
    free_flset(C_l2);
}

void build_mis_table(Graph *g)
{
    init_mis_table(g);

    for (uint i=0; i < g->order; i++)
    {
        invert_set(g->adj[i]);
        remove_elem(i, g->adj[i]);
    }

    flset *N_l = new_flset(g->order);
    flset *C_l = new_flset(g->order);
    flset *X   = new_flset(g->order);

    flset **B = (flset **)malloc(g->order * sizeof(flset *));
    for (uint i=0; i < g->order; i++)
    {
        B[i] = new_flset(g->order);
        for (uint j=i+1; j < g->order; j++)
            add_elem(j, B[i]);
    }

    all_cliques(g, 0, 0, X, N_l, C_l, B);

    for (uint i=0; i < g->order; i++)
        free_flset(B[i]);
    free(B);

    free_flset(N_l);
    free_flset(C_l);
    free_flset(X);

    for (uint i=0; i < g->order; i++)
    {
        invert_set(g->adj[i]);
        remove_elem(i, g->adj[i]);
    }
}

//=====//
//                               init_tri_table                               //
//=====//

```

```

/*
 * Initialize the tri_table. To reduce memory thrashing,
 * this function will only allocate new memory if the order
 * of g changes between calls. If g is NULL the memory
 * will be freed.
 */
void init_tri_table(Graph *g)
{
    g->num_triangles = 0;
    //g->triangles = (flset *)realloc(g->triangles, (1<<g->order) * sizeof(uint));

    uint i = min((int)g->order, 24);
    g->triangles = (flset **)realloc((void *)g->triangles, (1<<i) * sizeof(flset *));
    assert(g->triangles != NULL);

    //g->tri_cache = (char *)realloc(g->tri_cache, (1<<g->order) * sizeof(char));
    //memset(g->tri_cache, 2, (1<<g->order) * sizeof(char));
}

void free_tri_table(Graph *g)
{
    if (g != NULL)
    {
        for (uint i=0; i < g->num_triangles; i++)
            free_flset(g->triangles[i]);

        _free(g->triangles);
        g->num_triangles = 0;

        //_free(g->tri_cache);
    }
}

//=====//
//                          build_tri_table                          //
//=====//

void build_tri_table(Graph *g)
{
    register uint j, k, l;

    if (g->order < 3) return;

    for (j=0; j <= g->order - 3; j++)
    {
        register flset *rj = adj(g, j);
        for (k=j+1; k <= g->order - 2; k++)
        {
            register flset *rk = adj(g, k);
            for (l=k+1; l <= g->order - 1; l++)
            {
                if ( connected(k, rj)
                    && connected(l, rj)
                    && connected(l, rk))
                {
                    flset *tri = new_flset(g->adj_size);
                    add_elem(j, tri);
                    add_elem(k, tri);
                    add_elem(l, tri);
                    g->triangles[g->num_triangles++] = tri;
                }
            }
        }
    }

    bool in_array(uint x, uint *a, uint l)
    {
        for (uint k=0; k < l; k++)
            if (x == a[k]) return true;

        return false;
    }

    uint bit_count(uint n)
    {
        char table[16] = {0, 1, 1, 2, 1, 2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4};
        uint sum = 0;

        for (uint i = 0; i < (sizeof(n) << 3); i += 4)
            sum += table[(n >> i) & 0xf];

        return sum;
    }

    uint set_count(flset *A)
    {
        uint count=0;

        for (uint i=0; i < A->nblocks; i++)
            count += bit_count(A->block[i]);

        return count;
    }

    Graph *make_g127()
    {
        Graph *g = make_graph(127);

        uint cubes[g->order];
        uint ncubes = 0;

        for (uint i=0; i < g->order; i++)
        {
            cubes[i] = (i*i*i) % g->order;
            ncubes++;
        }

        foreach_vertex_pair(g, i, j)
        {

```

```

        uint v = (j - i);
        if (in_array(v, cubes, ncubes))
            add_edge(g, i, j);
    }
    return g;
}

int dec_order(const void *A, const void *B)
{
    return *(uint *)B - *(uint *)A;
}

Graph *make_g84(uint svertex)
{
    Graph *g = make_g127();

    uint a[g->order];
    uint count=0;

    a[count++] = svertex;
    foreach_flselem(i, adj(g, svertex))
        a[count++] = i;

    qsort(a, count, sizeof(uint), dec_order);

    for (uint i=0; i < count; i++)
        remove_vertex(g, a[i]);

    return g;
}

Graph *make_g74(int svertex, uint *x)
{
    Graph *g = make_g127();

    uint a[g->order];
    uint count=0;

    a[count++] = svertex;
    foreach_flselem(i, adj(g, svertex))
        a[count++] = i;

    uint mis_table_10[10] = {13,22,34,39,41,42,47,59,68,83};

    for (uint i=0; i < 10; i++)
        a[count++] = mis_table_10[i];

    qsort(a, count, sizeof(uint), dec_order);

    for (uint i=0; i < count; i++)
        remove_vertex(g, a[i]);

    return g;
}

}

Graph *make_gX(flset *x, int svertex)
{
    Graph *g = make_g127();

    uint a[g->order];
    memset(a, 0, g->order * sizeof(uint));

    uint count=0;

    a[svertex] = svertex+1, count++;
    foreach_flselem(i, adj(g, svertex))
        if (a[i] == 0) {a[i] = i+1; count++;}

    foreach_flselem(i, x)
        if (a[i] == 0) {a[i] = i+1; count++;}

    qsort(a, g->order, sizeof(uint), dec_order);

    for (uint i=0; i < count; i++)
        remove_vertex(g, a[i]-1);

    return g;
}

Graph *make_g84_2()
{
    Graph *g = make_graph(84);

    uint cubes[127];
    uint ncubes = 0;

    for (uint i=0; i < 127; i++)
    {
        cubes[i] = (i*i*i) % 127;
        ncubes++;
    }

    uint gi=0;
    for (uint i=0; i < 127-1; i++)
    {
        if (in_array(i, cubes, ncubes)) continue;

        uint gj=gi+1;
        for (uint j=i+1; j < 127; j++)
        {
            if (in_array(j, cubes, ncubes)) continue;

            uint v = (j - i);

            if (in_array(v, cubes, ncubes))
                add_edge(g, gi,gj);
        }
    }
}

```

```

        gj++;
    }

    gi++;
}

#if 0
foreach_vertex_pair(g, i, j)
{
    uint v = (j - i);

    if (in_array(v, cubes, ncubes))
        add_edge(g, i, j);
}
#endif

return g;
}

/*
 * Extend input graphs by one vertex without forming a K_4 subgraph.
 * Input graphs are expected not to have K_4 subgraphs.
 *
 * Copyright 2003, 2004 Jonathan Coles
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA
 */

#include <iostream>
#include <istream>
#include <fstream>
#include <assert.h>
#include <unistd.h>
#include <signal.h>
#include <time.h>
#include "flib.h"
#include "timing.h"
#include "args.h"

#define MAKE_FOR_TESTS 0

```

```

#define USE_ALARM 0

void report(uint state = 0);

uint in_count = 0;
uint out_count = 0;
double start_time = 0;
uint report_time = 0;

typedef struct
{
    uint a[32];
    uint b[32];
    uint len;
} Edgeset;

inline void connect(Graph *g, uint v0, uint v1)
{
    assert(v0 < g->order);
    assert(v1 < g->order);

    add_elem(v0, g->adj[v1]);
    add_elem(v1, g->adj[v0]);
    //g->deg[v0]++;
    //g->deg[v1]++;
}

inline void disconnect(Graph *g, uint v0, uint v1)
{
    assert(v0 < g->order);
    assert(v1 < g->order);

    remove_elem(v0, g->adj[v1]);
    remove_elem(v1, g->adj[v0]);
    //g->deg[v0]--;
    //g->deg[v1]--;
}

CPUDEFS

#define HELP \
"\
Extend input graphs (graph6 format) without forming K_4 subgraphs\n\
\n\
Usage: extend [-e # #] [--help]\n\
\n\
    -e # #      Extend graphs of order # to order #\n\
    --help     This help text\n\
"

FILE *in = stdin;
FILE *out = stdout;
FILE *err = stderr;

```

```

uint smart_output = 0;

int verbosity=0;

void sighandler(int sig)
{
    signal(2, sighandler);
#ifdef USE_ALARM
    signal(SIGALRM, sighandler);
    alarm(report_time);

    if (sig == SIGALRM)
    {
        report();
    }
    else
#endif
    {
        fprintf(err, "Caught signal, quitting.\n");
        exit(0);
    }
}

void report(uint state)
{
    double end_time = CPUTIME;
    time_t now = time((time_t *)NULL);

    char *s = ctime(&now);
    s[strlen(s)-1] = 0;

    switch (state)
    {
        case 0:
            fprintf(err, "extend: [%s] %i %i %.2fs %.2fg/s %.2fg/s\n", s,
                    in_count, out_count, (end_time-start_time),
                    (double)in_count / (end_time-start_time),
                    (double)out_count / (end_time-start_time));
            break;
        case 1:
            fprintf(err, "extend: [Date] in out time in_rate out_rate\n");
            break;
        case 2:
            fprintf(err, "extend: [%s] finished.\n", s);
            break;
    }
}

// =====
// extend(Graph *g)
//
// Create new K4-free graphs by connecting a new vertex to every
// triangle-free subset in g. g is assumed to be K4-free. New graphs are
// written to global 'out'.

// Returns the number of new graphs generated.
//
// =====
uint extend_helper(Graph *g,
                  uint *sets, Edgeset *edges,
                  uint s, uint k, uint extension,
                  uint *triangle_free_subsets, uint num_triangle_free_subsets)
{
    uint count = 0;

    if (k == extension)
    {
        #if 0
        Graph *g0 = make_graph(g->order + extension);
        copy_graph(g, g0);
        g0->order = g->order + extension;

        for (uint i=0; i < extension; i++)
        {
            uint cur_set = triangle_free_subsets[sets[i]];
            foreach_elem(g0, u, cur_set) connect(g0, g->order+i, u);
        }

        write_graph(g0, out);
        count = 1;

        free_graph(g0);
        #else
        uint t = g->order;
        g->order += extension;

        uint old_num_triangles = g->num_triangles;
        #if 1
        init_tri_table(g);
        g->num_triangles = old_num_triangles;

        //
        // extend the graph by 'extension' vertices, but also
        // add new triangles to the graph's list of triangles.
        // this is better than having to call init_tri_table()/build_tri_table()
        // on the entire graph.
        //
        for (uint i=0; i < extension; i++)
        {
            g->adj[t+i] = 0;

            uint cur_set = triangle_free_subsets[sets[i]];

            foreach_elem(g, u, cur_set) connect(g, t+i, u);

            Edgeset *e = &edges[sets[i]];

```

```

        for (uint j=0; j < e->len; j++)
        {
            uint tri
                = set_elem(e->a[j]) | set_elem(e->b[j]) | set_elem(t+i);

            g->triangles[g->num_triangles++] = tri;
            g->tri_cache[tri] = 1;
        }
    }
#else
    for (uint i=0; i < extension; i++)
    {
        g->adj[t+i] = 0;
        uint cur_set = triangle_free_subsets[sets[i]];
        foreach_elem(g, u, cur_set) connect(g, t+i, u);
    }

    //init_tri_table(g);
    //build_tri_table(g);
#endif

    // g -> (2,2,3;4) => X(g) >= 5
    //if (chromatic(g) >= 5)

    build_mis_table(g);

    if (h223_4(g, false))
    {
        write_graph(g, out);
        count = 1;
    }

    for (uint i=0; i < extension; i++)
    {
        uint cur_set = triangle_free_subsets[sets[i]];
        foreach_elem(g, u, cur_set) disconnect(g, t+i, u);
    }

    g->num_triangles = old_num_triangles;

    g->order -= extension;
#endif
}
else
{
    for (uint i=s; i < num_triangle_free_subsets; i++)
    {
        sets[k] = i;

        count +=
            extend_helper(g, sets, edges, i, k+1, extension,
                triangle_free_subsets, num_triangle_free_subsets);
    }
}

```

```

    }
    return count;
}

int main(int argc, char **argv)
{
    int optc;
    char **optv;

    int verbosity = 0;
    uint skip_n_lines=0;
    bool separate = false;
    bool check_K4 = true;
    bool check_chi = true;

    double cur_time;
    double last_time;

    cur_time = last_time = CPUTIME;

    uint m = 0;
    uint n = 0;

    fprintf(stderr, "extend:");
    for (int i=1; i < argc; i++)
        fprintf(stderr, " %s", argv[i]);
    fprintf(stderr, "\n");

    begin_args(&argc, &argv, &optc, &optv);

    while (next_arg(&argc, &argv, &optc, optv))
    {
        if (!strcmp("q", optv[0]))
        {
            verbosity--;
        }
        else if (!strcmp("sep", optv[0]))
        {
            separate = true;
        }
        else if (!strcmp("e", optv[0]))
        {
            if (optc == 3)
            {
                m = atoi(optv[1]);
                n = atoi(optv[2]);
            }
        }
        else if (!strcmp("r", optv[0]))
        {
            if (optc == 2)
            {
                report_time = atoi(optv[1]);
            }
        }
    }
}

```

```

    }
    else if (!strcmp("f", optv[0]))
    {
        if (optc > 1 && strncmp("-",optv[1],1)) in = fopen(optv[1], "r");
        if (optc > 2 && strncmp("-",optv[2],1)) out = fopen(optv[2], "w");
        if (optc > 3 && strncmp("-",optv[3],1)) err = fopen(optv[3], "w");

        if (!in) { fprintf(stderr, "Bad input source.\n"); return 1;}
        if (!out) { fprintf(stderr, "Bad output destination.\n"); return 1;}
        if (!err) { fprintf(stderr, "Bad error destination.\n"); return 1;}
    }
    else if (!strcmp("noK4", optv[0]))
    {
        check_K4 = false;
    }
    else if (!strcmp("noX", optv[0]))
    {
        check_chi = false;
    }
    else //if (!strcmp("help", optv[0]))
    {
        fprintf(err, HELP "\n");
        return 0;
    }
}
end_args(&optv);

if (m==0 || n==0 || m>=n)
{
    fprintf(err, HELP "\n");
    return 0;
}

//
// Skip n input lines
//
while (!feof(in) && skip_n_lines)
{
    skip_n_lines--;

    char s[518];
    fgets(s, 517, in);
}

#define BUFSIZE 20 * 1024 * 1024
//char *buf = (char *)malloc(BUFSIZE);
//setbuffer(out, buf, BUFSIZE);

//signal(2, sighandler);
//signal(SIGUSR1, sighandler);

signal(2, sighandler);
#if USE_ALARM
signal(SIGALRM, sighandler);
alarm(report_time);
#endif

#endif

uint extension = n - m;
uint *triangle_free_subsets = (uint *)malloc((1<<m) * sizeof(uint));

Edgeset *edges = (Edgeset *)malloc((1<<m) * sizeof(Edgeset));

Graph *g = make_graph(m, m + extension);
uint *sets = (uint *)malloc(extension * sizeof(uint));

start_time = CPUTIME;

if (verbosity >= 0)
    report(1);

while (!feof(in))
{
    //
    // Read a graph
    //
    if ((g = read_graph(in, g)) == NULL) break;
    in_count++;

    if (g->order != m)
    {
        fprintf(err, "|V(G)| != m. skipping.\n");
        continue;
    }

    if (check_K4 && has_k_4(g, V(g))) continue;

    //
    // g -> (2,2,3;4) => X(g) >= 5
    #if !MAKE_FOR_TESTS
    if (check_chi && chromatic(g) <= 3) continue;
    #endif

    init_tri_table(g);
    //fprintf(err, "sdn\n");
    build_tri_table(g);

    uint num_triangle_free_subsets = 0;

    foreach_subset(i0, V(g))
    {
        // skip nothing, or single vertices
        #if !MAKE_FOR_TESTS
        //if (!i0 || (((i0 - 1) & i0) == 0)) continue;
        #endif

        //
        // if the set is initially triangle free, the set will be maximal
        // iff the addition of any vertex creates a triangle. conversly,
        // the set is not maximal if adding a vertex does not create
        // a triangle.

```

```

        //
        if (triangle_free_cached(g, i0))
        {
            bool max_set = true;
#if !MAKE_FOR_TESTS
            foreach_vertex(g, v)
            {
                if (!elem_of(v, i0))
                {
                    if (triangle_free_cached(g, i0 | set_elem(v)))
                    {
                        max_set = false;
                        break;
                    }
                }
            }
#endif
            if (max_set)
                triangle_free_subsets[num_triangle_free_subsets++] = i0;
        }
    }

    for (uint i=0; i < num_triangle_free_subsets; i++)
    {
        uint index=0;
        foreach_vertex_pair(g, u,v)
        {
            if (subset(set_elem(u) | set_elem(v), triangle_free_subsets[i])
                && connected_to(g,u,v))
            {
                edges[i].a[index] = u;
                edges[i].b[index] = v;
                index++;
            }
        }
        edges[i].len = index;
    }

    uint count =
        extend_helper(g, sets, edges, 0, 0, extension,
            triangle_free_subsets, num_triangle_free_subsets);

    //
    // After all the new graphs write a seperator for some other
    // tools that need to know which graphs belong to the
    // input graph
    //
    if (seperate) fprintf(out, "--\n");

    out_count += count;

#if !USE_ALARM
    if (report_time)
        {
            cur_time = CPUTIME;
            if (cur_time - last_time >= report_time)
            {
                last_time = cur_time;
                report();
            }
        }
#endif

    fflush(out);
    fflush(err);
}

//
// Clean up
//
free_graph(g);
free(sets);
free(triangle_free_subsets);
free(edges);

if (verbosity >= 0)
{
    report();
    report(2);
}

if (in != stdin) fclose(in);
if (out != stdout) fclose(out);
if (err != stderr) fclose(err);
}

/*
 * Filter graphs with K_n subgraphs or Filter graphs without K_n subgraphs.
 *
 *
 * Copyright 2003, 2004 Jonathan Coles
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

```

```

*/
#include <iostream>
#include <istream>
#include "flib.h"
#include "args.h"

int verbosity=0;

FILE *in = stdin;
FILE *out = stdout;
FILE *err = stderr;

void usage()
{
    fprintf(err,
"\
Only print graphs WITHOUT the given properties. \n\
\n\
[-K # | -Kb #] [--with]\n\
");
    exit(1);
}

int main(int argc, char **argv)
{
    int verbosity = 0;
    bool print_inverse = false;
    bool do_K_bar = false;
    uint K = 0;

    int optc;
    char **optv;
    begin_args(&argc, &argv, &optc, &optv);

    while (next_arg(&argc, &argv, &optc, optv))
    {
        if (!strcmp("q", optv[0]))
        {
            verbosity--;
        }
        else if (!strcmp("with", optv[0]))
        {
            print_inverse = true;
        }
        else if (!strcmp("K", optv[0]))
        {
            if (optc > 1)
                K = atoi(optv[1]);
            else
                usage();
        }
        else if (!strcmp("Kb", optv[0]))
        {
            if (optc > 1)

```

```

{
    K = atoi(optv[1]);
    do_K_bar = true;
}
else
    usage();
}
else if (!strcmp("f", optv[0]))
{
    if (optc > 1 && strcmp("-",optv[1],1)) in = fopen(optv[1], "r");
    if (optc > 2 && strcmp("-",optv[2],1)) out = fopen(optv[2], "w");
    if (optc > 3 && strcmp("-",optv[3],1)) err = fopen(optv[3], "w");

    if (!in) { fprintf(stderr, "Bad input source.\n"); return 1;}
    if (!out) { fprintf(stderr, "Bad output destination.\n"); return 1;}
    if (!err) { fprintf(stderr, "Bad error destination.\n"); return 1;}
}
else
{
    usage();
}
}
end_args(&optv);

if (!K) usage();

uint in_count = 0;
uint out_count = 0;
while (!feof(in))
{
    Graph *g;

    if ((g = read_graph(in)) == NULL) break;

    in_count++;

    //print_graph(g);

    uint result = 0;

    if (do_K_bar)
    {
        complement(g);
        //print_graph(g);
        result = is_k_n_free(g, K, V(g));
        complement(g);
    }
    else
    {
        result = is_k_n_free(g, K, V(g));
    }

    //fprintf(out, "result: %i %i\n", result, print_inverse);

    if ((result && !print_inverse) || (!result && print_inverse))

```

```

    {
        //if (!print_inverse)
        {
            write_graph(g, out);
            out_count++;
        }
        /*
        else
        {
            if (print_inverse)
            {
                write_graph(g, out);
                out_count++;
            }
        }
        */

        free_graph(g);
    }

    if (verbosity >= 0)
    {
        fprintf(err,
            "filter: Read %i and generated %i graphs.\n",
                in_count, out_count);
    }

    if (in != stdin) fclose(in);
    if (out != stdout) fclose(out);
    if (err != stderr) fclose(err);
}

/*
 * Check graphs for inclusion in the Folkman graph set H(2,2,3;4)
 *
 * Copyright 2003 Jonathan Coles
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
    * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA
    */

#include <stdio.h>
#include <stdlib.h>
#include <memory.h>
#include <assert.h>
#include <iostream>
#include <istream>
#include <fstream>
#include "timing.h"
#include "flib.h"
#include "args.h"

using namespace std;

#define TESTS 0

CPUDEFS

//
// Program output verbosity level
// 0 is normal
//
int verbosity = 0;

int main(int argc, char **argv)
{
    Graph *g;
    uint i;
    int f_repeat = 1;
    bool print_on_found = false;
    bool check_k4 = true;
    bool print_extra = false;
    bool print_sets = false;
    bool print_inverse = false;

    FILE *in = stdin;

    char *name = "H(2,2,3;4)";

    bool (*f)(Graph *g, uint f_repeat, double *t, bool check_k4, bool print);

    f = in_h223_4;

    int optc;
    char **optv;
    begin_args(&argc, &argv, &optc, &optv);
    while (next_arg(&argc, &argv, &optc, optv))
    {
        if (!optc) continue;

        if (!strcmp("r", optv[0]) && optc == 2)
        {
            f_repeat = atoi(optv[1]);

```

```

}
else if (!strcmp("f", optv[0]) && optc == 2)
{
    if (strcmp("-", optv[1]) != 0)
        in = fopen(optv[1], "rb");
}
else if (!strcmp("v", optv[0]))
{
    verbosity++;
}
else if (!strcmp("q", optv[0]))
{
    verbosity--;
}
else if (!strcmp("print", optv[0]))
{
    print_on_found = true;
}
else if (!strcmp("nok4check", optv[0]))
{
    check_k4 = false;
}
else if (!strcmp("h223", optv[0]))
{
    name = "H(2,2,3;4)";
    f = in_h223_4;
}
else if (!strcmp("h22", optv[0]))
{
    name = "H(2,2;4)";
    f = in_h22_4;
}
else if (!strcmp("h23", optv[0]))
{
    name = "H(2,3;4)";
    f = in_h23_4;
}
else if (!strcmp("extra", optv[0]))
{
    print_extra = true;
}
else if (!strcmp("show-sets", optv[0]))
{
    print_sets = true;
}
else if (!strcmp("inverse", optv[0]))
{
    print_inverse = true;
}
else if (!strcmp("gamma3", optv[0]))
{
    Graph *g = make_gamma3_graph();
    write_graph(g, stdout);
    free_graph(g);
    exit(0);
}
}
end_args(&optv);

double start_time = CPUTIME;
uint index=0;

uint in_set = 0;

while (!feof(in))
{
    if ((g = read_graph(in)) == NULL) break;

    index++;

    double t;
    bool ret = f(g, f_repeat, &t, check_k4, print_sets);

    if (print_inverse)
    {
        if (!ret) { in_set++; }
    }
    else
    {
        if (ret) { in_set++; }
    }

    if (verbosity >= 0)
    {
        fprintf(stderr, "%i. G(%i) is %sin %s %f second(s).\n",
            index, g->order, (ret ? "" : "not "), name, (t/f_repeat));

        if (print_extra)
        {
            fprintf(stderr, "    mindeg=%i maxdeg=%i |mis|=%i |E|=%i\n",
                g->min_degree, g->max_degree, g->max_mis, g->edge_count);
            fprintf(stderr, "    ");
            write_graph(g, stderr);
        }
    }

    if (ret && print_on_found)
        write_graph(g, stdout);
    else if (!ret && print_on_found && print_inverse)
        write_graph(g, stdout);

    free_graph(g);
}

double end_time = CPUTIME;

if (in != stdin)
{
    //((fstream *)in)->close();
    //delete in;
}

```



```

        && !connected(l, rk))
    {
#if 1
        copy_graph(g, g0);
        remove_vertex(g0, l);
        remove_vertex(g0, k);
        remove_vertex(g0, j);
#endif

        write_graph(g0, out);
        (*out_count)++;

    }
}
}
}

free_graph(g0);
//    reduce(g, out);

free_graph(g);
}
}

void _dropKb4(uint *in_count, uint *out_count)
{
    *in_count = 0;
    *out_count = 0;

    while (!feof(in))
    {
        Graph *g;

        if ((g = read_graph(in)) == NULL) break;

        (*in_count)++;

        Graph *g0 = NULL;

        if (g->order > 4)
        {
            g0 = make_graph(g->order, -1, g0);

            register uint j, k, l, m;
            for (j=0; j <= g->order - 4; j++)
            {
                register uint rj = adj(g, j);
                for (k=j+1; k <= g->order - 3; k++)
                {
                    register uint rk = adj(g, k);
                    for (l=k+1; l <= g->order - 2; l++)

```

```

        {
            register uint rl = adj(g, l);
            for (m=l+1; m <= g->order - 1; m++)
            {
                if ( !connected(m, rl)
                    && !connected(m, rk)
                    && !connected(m, rj)
                    && !connected(l, rj)
                    && !connected(l, rk)
                    && !connected(k, rj))
                {
#if 1
                    copy_graph(g, g0);
                    remove_vertex(g0, m);
                    remove_vertex(g0, l);
                    remove_vertex(g0, k);
                    remove_vertex(g0, j);
#endif

                    write_graph(g0, out);
                    (*out_count)++;

                }
            }
        }
    }
}

free_graph(g0);
//    reduce(g, out);

free_graph(g);
}
}

void _reduce(uint *in_count, uint *out_count)
{
    *in_count = 0;
    *out_count = 0;
    while (!feof(in))
    {
        Graph *g;

        if ((g = read_graph(in)) == NULL) break;

        (*in_count)++;

        reduce(g, out);

        *out_count += g->order;
    }
}

```

```

        free_graph(g);
    }
}

void _maximals(uint *in_count, uint *out_count)
{
    *in_count = 0;
    *out_count = 0;
    while (!feof(in))
    {
        Graph *g;

        if ((g = read_graph(in)) == NULL) break;

        (*in_count)++;

        if (maximal(g, 4))
        {
            write_graph(g, out);
            (*out_count)++;
        }

        free_graph(g);
    }
}

void _minimals(uint *in_count, uint *out_count)
{
    *in_count = 0;
    *out_count = 0;
    while (!feof(in))
    {
        Graph *g;

        if ((g = read_graph(in)) == NULL) break;

        (*in_count)++;

        init_tri_table(g);
        build_tri_table(g);
        build_mis_table(g);

        if (minimal(g))
        {
            write_graph(g, out);
            (*out_count)++;
        }

        free_graph(g);
    }
}

void _bicritical(uint *in_count, uint *out_count)
{
    *in_count = 0;

    *out_count = 0;
    while (!feof(in))
    {
        Graph *g;

        if ((g = read_graph(in)) == NULL) break;

        (*in_count)++;

        init_tri_table(g);
        build_tri_table(g);
        build_mis_table(g);

        if (maximal(g, 4) && minimal(g))
        {
            write_graph(g, out);
            (*out_count)++;
        }

        free_graph(g);
    }
}

void _extend_to_maximals(uint *in_count, uint *out_count)
{
    *in_count = 0;
    *out_count = 0;
    while (!feof(in))
    {
        Graph *g;

        if ((g = read_graph(in)) == NULL) break;

        (*in_count)++;

        extend_to_maximals(g, 4);
        //(*out_count)++;
        //
        if (verbosity >= 1) fprintf(err, ".");

        free_graph(g);
    }

    if (verbosity >= 1 && in_count != 0)
        fprintf(err, "\n");
}

void _reduce_to_minimals(uint *in_count, uint *out_count)
{
    *in_count = 0;
    *out_count = 0;
    while (!feof(in))
    {
        Graph *g;

```

```

        if ((g = read_graph(in)) == NULL) break;

        (*in_count)++;

        init_tri_table(g);
        build_tri_table(g);
        build_mis_table(g);
        build_set_neighbours(g);

        *out_count += reduce_to_minimals(g);

        if (verbosity >= 1) fprintf(err, ".");

        free_graph(g);
    }
}

void _reduce_to_all(uint *in_count, uint *out_count)
{
    *in_count = 0;
    *out_count = 0;
    while (!feof(in))
    {
        Graph *g;

        if ((g = read_graph(in)) == NULL) break;

        (*in_count)++;

        init_tri_table(g);
        build_tri_table(g);
        build_mis_table(g);
        build_set_neighbours(g);

        *out_count += reduce_to_minimals(g, true, out);

        if (verbosity >= 1) fprintf(err, ".");

        free_graph(g);
    }
}

void _gnuplot(uint *in_count, uint *out_count)
{
    *in_count = 0;
    *out_count = 0;
    while (!feof(in))
    {
        Graph *g;

        if ((g = read_graph(in)) == NULL) break;

        fprintf(out, "%i\t%i\t%i\n", g->max_degree, g->min_degree, g->edge_count);
        //fprintf(out, "%i\t%i\n", g->order, g->edge_count);
        //fprintf(out, "%i\t%i\n", *in_count, g->edge_count);

        //fprintf(out, "%i\t%i\n", g->edge_count, *in_count);
        //fprintf(out, "%i\t%i\n", g->order, g->edge_count);

        (*in_count)++;

        free_graph(g);
    }
}

void _chromatic3(uint *in_count, uint *out_count)
{
    *in_count = 0;
    *out_count = 0;
    while (!feof(in))
    {
        Graph *g;

        if ((g = read_graph(in)) == NULL) break;

        init_tri_table(g);
        build_mis_table(g);

        if (chi(g) > 3)
        //if (chromatic(g) > 3)
        {
            write_graph(g, out);
            (*out_count)++;
        }

        free_graph(g);
    }
}

void _info(uint *in_count, uint *out_count)
{
    *in_count = 0;
    *out_count = 0;
    while (!feof(in))
    {
        Graph *g;

        if ((g = read_graph(in)) == NULL) break;

        print_graph_info(out, g, "oetmMdDX");
        free_graph(g);
    }
}

void _new_edge_makes_K3(uint *in_count, uint *out_count)
{
    *in_count = 0;
    *out_count = 0;
    while (!feof(in))

```

```

{
    Graph *g;

    if ((g = read_graph(in)) == NULL) break;

    foreach_vertex_pair(g, i,j)
    {
        if (!connected_to(g, i,j))
        {
            if (!intersects(adj(g,i), adj(g,j)))
            {
                goto next_graph;
            }
        }
    }

    write_graph(g, out);
    (*out_count)++;

next_graph:
    free_graph(g);
}

int main(int argc, char **argv)
{
    int optc;
    char **optv;

    f_f = NULL;

    begin_args(&argc, &argv, &optc, &optv);
    while (next_arg(&argc, &argv, &optc, optv))
    {
        if (!optc) continue;

        if (!strcmp("reduce", optv[0]))
            _f = _reduce;
        else if (!strcmp("maximals", optv[0]))
            _f = _maximals;
        else if (!strcmp("extend-to-maximals", optv[0]))
            _f = _extend_to_maximals;
        else if (!strcmp("minimals", optv[0]))
            _f = _minimals;
        else if (!strcmp("reduce-to-minimals", optv[0]))
            _f = _reduce_to_minimals;
        else if (!strcmp("reduce-to-all", optv[0]))
            _f = _reduce_to_all;
        else if (!strcmp("dropKb3", optv[0]))
            _f = _dropKb3;
        else if (!strcmp("dropKb4", optv[0]))
            _f = _dropKb4;
        else if (!strcmp("gnuplot", optv[0]))
            _f = _gnuplot;
        else if (!strcmp("info", optv[0]))

            _f = _info;
        else if (!strcmp("X3", optv[0]))
            _f = _chromatic3;
        else if (!strcmp("noNewK3", optv[0]))
            _f = _new_edge_makes_K3;
        else if (!strcmp("bicritical", optv[0]))
            _f = _bicritical;
        else if (!strcmp("v", optv[0]))
            verbosity++;
        else if (!strcmp("q", optv[0]))
            verbosity--;
        else if (!strcmp("f", optv[0]))
        {
            if (optc > 1 && strcmp("-",optv[1],1)) in = fopen(optv[1], "r");
            if (optc > 2 && strcmp("-",optv[2],1)) out = fopen(optv[2], "w");
            if (optc > 3 && strcmp("-",optv[3],1)) err = fopen(optv[3], "w");

            if (!in) { fprintf(stderr, "Bad input source.\n"); exit(1);}
            if (!out) { fprintf(stderr, "Bad output destination.\n"); exit(1);}
            if (!err) { fprintf(stderr, "Bad error destination.\n"); exit(1);}
        }
        else
        {
            usage();
        }
    }
    end_args(&optv);

    if (_f == NULL) usage();

    uint in_count=0;
    uint out_count=0;

    double start_time = CPUTIME;
    _f(&in_count, &out_count);
    double end_time = CPUTIME;

    fprintf(err, "Input %i graph(s), ouput %i graphs in %f second(s).\n",
            in_count, out_count, (end_time-start_time));

    if (in != stdin) fclose(in);
    if (out != stdout) fclose(out);
    if (err != stderr) fclose(err);
}

/*
 * Reads graphs in Graph6 format and files them into non-isomorphic groups.
 *
 * Copyright 2003, 2004 Jonathan Coles
 *
 * This program is free software; you can redistribute it and/or modify

```

```

* it under the terms of the GNU General Public License as published by
* the Free Software Foundation; either version 2 of the License, or
* (at your option) any later version.
*
* This program is distributed in the hope that it will be useful,
* but WITHOUT ANY WARRANTY; without even the implied warranty of
* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
* GNU General Public License for more details.
*
* You should have received a copy of the GNU General Public License
* along with this program; if not, write to the Free Software
* Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA
*/

#include <stdio.h>
#include <iostream>
#include <fstream>
#include <args.h>
#include "flib.h"

FILE **file_cache = NULL;
uint cache_size = 0;

int verbosity=0;

void free_cache()
{
    if (file_cache != NULL)
    {
        for (uint i=0; i < cache_size; i++)
            if (file_cache[i] != NULL)
                fclose(file_cache[i]);

        delete file_cache;
        file_cache = NULL;
    }
}

FILE *get_out_stdout(Graph *g)
{
    return stdout;
}

FILE *get_out_degree(Graph *g)
{
    if (file_cache == NULL)
    {
        cache_size = g->order * g->order;
        file_cache = new FILE*[cache_size];
        memset(file_cache, 0, sizeof(FILE *) * cache_size);
    }

    uint n = g->max_degree * g->order + g->min_degree;

    if (file_cache[n] == NULL)

```

```

{
    char s[50];
    sprintf(s, "splitg.%i.%i.%i.g6", g->order,
            g->min_degree,
            g->max_degree);

    file_cache[n] = fopen(s, "w");
    if (file_cache[n] == NULL)
    {
        fprintf(stderr, "Couldn't open %s\n", s);
    }
}

return file_cache[n];
}

FILE *get_out_edge(Graph *g)
{
    if (file_cache == NULL)
    {
        cache_size = (g->order * (g->order-1)) / 2 + 1;
        file_cache = new FILE*[cache_size];
        memset(file_cache, 0, sizeof(FILE *) * cache_size);
    }

    uint n = g->edge_count;

    if (file_cache[n] == NULL)
    {
        char s[50];
        sprintf(s, "splitg.%i.%i.g6", g->order, g->edge_count);
        file_cache[n] = fopen(s, "w");
    }

    return file_cache[n];
}

FILE *get_out_both(Graph *g)
{
    if (file_cache == NULL)
    {
        cache_size = g->order * g->order * ((g->order * (g->order-1)) / 2);
        file_cache = new FILE*[cache_size];
        memset(file_cache, 0, sizeof(FILE *) * cache_size);
    }

    uint n = g->edge_count;

    if (file_cache[n] == NULL)
    {
        char s[50];
        sprintf(s, "splitg.%i.%i.%i.%i.g6", g->order,
            g->min_degree,
            g->max_degree,
            g->edge_count);

        file_cache[n] = fopen(s, "w");
    }
}

```

∞
∞

```
    }
    return file_cache[n];
}

int main(int argc, char **argv)
{
    uint smart_output = 0;

    int optc;
    char **optv;

    Graph *g;

    FILE *in = stdin;

    begin_args(&argc, &argv, &optc, &optv);

    while (next_arg(&argc, &argv, &optc, optv))
    {
        if (!optc) continue;

        if (!strcmp("q", optv[0]))
        {
        }
        else if (!strcmp("so", optv[0]))
        {
            smart_output = 1;

            if (optc == 2)
            {
                if (!strcmp("degree", optv[1]))
                    smart_output = 1;
                else if (!strcmp("edge", optv[1]))
                    smart_output = 2;
                else if (!strcmp("both", optv[1]))
                    smart_output = 3;
                else
                {
                    cerr << "Unknown smart output option. Using 'degree'."
                        << endl;
                }
            }
        }
    }
}
```

```
    }

    FILE *(*out_func)(Graph *g) = NULL;

    switch (smart_output)
    {
        case 1:
            out_func = get_out_degree;
            break;
        case 2:
            out_func = get_out_edge;
            break;
        case 3:
            out_func = get_out_both;
            break;
        default:
            out_func = get_out_stdout;
            break;
    }

    while (!feof(in))
    {
        char bytes[518];

        fgets(bytes, 517, in);
        if (feof(in)) break;

        if ((g = read_graph(bytes)) == NULL) break;

        FILE *out = out_func(g);

        if (out == NULL) break;

        fprintf(out, bytes);
        fflush(out);
        free_graph(g);
    }

    free_cache();
}
```

Bibliography

- [1] Satisfiability suggested format. *DIMACS*, 1993.
- [2] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The design and analysis of computer algorithms*. Addison-Wesley Publishing Co., Reading, Mass.-London-Amsterdam, 1975. Second printing, Addison-Wesley Series in Computer Science and Information Processing.
- [3] F. Aloul, I. Markov, and K. Sakallah. Efficient symmetry-breaking for boolean satisfiability. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 271–282, 2003.
- [4] F. Aloul, I. Markov, and K. Sakallah. Shatter: Efficient symmetry-breaking for boolean satisfiability. In *Design Automation Conference (DAC)*, pages 836–839, 2003.
- [5] F. Aloul, A. Ramani, I. Markov, and K. Sakallah. Solving difficult sat instances in the presence of symmetry. In *IEEE Transactions on Computer Aided Design*, volume 22, pages 1117–1137, 2003.
- [6] V. Chvátal. The minimality of the Mycielski graph. In *Graphs and combinatorics (Proc. Capital Conf., George Washington Univ., Washington, D.C., 1973)*, pages 243–246. Lecture Notes in Math., Vol. 406. Springer, Berlin, 1974.
- [7] Condor Project Version 6.6.6. *Computer Sciences Department, University of Wisconsin-Madison*. <http://www.cs.wisc.edu/condor/>.
- [8] J. Crawford, M. Ginsberg, E. M. Luks, and A. Roy. Symmetry breaking predicates for search problems. In *Proceedings of the Fifth International Conference on Knowledge Representation and Reasoning (KR '96)*, pages 148–159, 1996.

- [9] E. Dantsin, A. Goerdt, E. A. Hirsch, R. Kannan, J. Kleinberg, C. Papadimitriou, P. Raghavan, and U. Schöning. A deterministic $(2 - 2/(k + 1))^n$ algorithm for k -SAT based on local search. *Theoret. Comput. Sci.*, 289(1):69–83, 2002.
- [10] P. T. Darga, M. H. Liffiton, K. A. Sakallah, and I. L. Markov. Exploiting structure in symmetry generation for CNF. In *Proceedings of the 41st Design Automation Conference*, pages 530–534, San Diego, California, June 2004.
- [11] J. Folkman. Graphs with monochromatic complete subgraphs in every edge coloring. *SIAM J. Appl. Math.*, 18:19–24, 1970.
- [12] P. Frankl and V. Rödl. Large triangle-free subgraphs in graphs without K_4 . *Graphs Combin.*, 2(2):135–144, 1986.
- [13] Z. Fu. zchaff SAT solver. *SAT Research Group, Princeton University*. <http://www.princeton.edu/~chaff/zchaff.html>.
- [14] R. L. Graham. On edgewise 2-colored graphs with monochromatic triangles and containing no complete hexagon. *J. Comb. Theory*, 4:300, 1968.
- [15] R. L. Graham, B. L. Rothschild, and J. H. Spencer. *Ramsey theory*. John Wiley & Sons Inc., New York, 1980. Wiley-Interscience Series in Discrete Mathematics, A Wiley-Interscience Publication.
- [16] R. L. Graham and J. H. Spencer. On small graphs with forced monochromatic triangles. In *Recent trends in graph theory (Proc. Conf., New York, 1970)*, pages 137–141. Lecture Notes in Math., Vol. 186. Springer, Berlin, 1971.
- [17] M. Heule and H. van Maaren. march_eq SAT solver. *Delft University of Technology*. http://www.isa.ewi.tudelft.nl/sat/march_eq.htm.
- [18] R. Hill and R. W. Irving. On group partitions associated with lower bounds for symmetric Ramsey numbers. *European J. Combin.*, 3(1):35–50, 1982.

- [19] K. Iwama and S. Tamaki. Improved upper bounds for 3-sat. In *SODA '04: Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 328–328. Society for Industrial and Applied Mathematics, 2004.
- [20] T. Jensen and G. F. Royle. Small graphs with chromatic number 5: a computer search. *J. Graph Theory*, 19(1):107–116, 1995.
- [21] D. L. Kreher and D. R. Stinson. *Combinatorial Algorithms: Generation, Enumeration, and Search*. CRC Press, 1999.
- [22] S. Lin. On Ramsey numbers and K_r -coloring of graphs. *J. Comb. Theory*, 20:82–92, 1972.
- [23] T. Łuczak, A. Ruciński, and S. Urbański. On minimal Folkman graphs. *Discrete Math.*, 236(1-3):245–262, 2001. Graph theory (Kazimierz Dolny, 1997).
- [24] T. Łuczak and S. Urbański. A note on restricted vertex Ramsey numbers. *Period. Math. Hungar.*, 33(2):101–103, 1996.
- [25] E. Luks and A. Roy. Symmetry breaking in constraint satisfaction. In *7th International Conference of Artificial Intelligence and Mathematics*, Jan 2002.
- [26] F. Manyà, R. Béjar, and G. Escalada-Imaz. The satisfiability problem in regular CNF-formulas. *Soft Computing - A Fusion of Foundations, Methodologies and Applications*, 2(3):116–123, 1998.
- [27] B. D. McKay. nauty and Ramsey graphs. *The Australian National University*. <http://cs.anu.edu.au/~bdm/>.
- [28] B. D. McKay. Practical graph isomorphism. In *Proceedings of the Tenth Manitoba Conference on Numerical Mathematics and Computing, Vol. I (Winnipeg, Man., 1980)*, volume 30, pages 45–87, 1981.
- [29] B. D. McKay. Isomorph-free exhaustive generation. *J. Algorithms*, 26(2):306–324, 1998.

- [30] B. D. McKay and S. P. Radziszowski. The first classical Ramsey number for hypergraphs is computed. In *Proceedings of the Second Annual ACM-SIAM Symposium on Discrete Algorithms (San Francisco, CA, 1991)*, pages 304–308, New York, 1991. ACM.
- [31] B. D. McKay and S. P. Radziszowski. A new upper bound for the Ramsey number $R(5, 5)$. *Australas. J. Combin.*, 5:13–20, 1992.
- [32] D. Mitchell, B. Selman, and H. Levesque. Hard and easy distributions of SAT problems. In *Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI-92), San Jose, CA*, pages 459–465.
- [33] J. Mycielski. Sur le coloriage des graphs. *Colloq. Math.*, 3:161–162, 1955.
- [34] E. Nediakov and N. D. Nenov. Computation of the vertex Folkman number $F(4, 4; 6)$. pages 123–128, 2001.
- [35] E. Nediakov and N. D. Nenov. Computation of the vertex Folkman numbers $F(2, 2, 2, 4; 6)$ and $F(2, 3, 4; 6)$. *Electron. J. Combin.*, 9(1):Research Paper 10, 7 pp. (electronic). <http://www.combinatorics.org>, 2002.
- [36] N. D. Nenov. Ramsey graphs and some constants related to them. 1980.
- [37] N. D. Nenov. A constant connected with Ramsey $(3, 4)$ -graphs. *Serdica*, 7(4):366–371 (1982), 1981.
- [38] N. D. Nenov. An example of a 15-vertex $(3, 3)$ -Ramsey graph with clique number 4. *C. R. Acad. Bulgare Sci.*, 34(11):1487–1489, 1981.
- [39] N. D. Nenov. Zykov numbers and some of their applications in Ramsey theory. *Serdica*, 9(2):161–167, 1983.
- [40] N. D. Nenov. The chromatic number of any 10-vertex graph without 4-cliques is at most 4. *C. R. Acad. Bulgare Sci.*, 37(3):301–304, 1984.
- [41] N. D. Nenov. On $(3, 4)$ Ramsey graphs without 9-cliques. *Annuaire Univ. Sofia Fac. Math. Inform.*, 85(1-2):71–81 (1993), 1991.
- [42] N. D. Nenov. On a class of vertex Folkman graphs. *Annuaire Univ. Sofia Fac. Math. Inform.*, 94:15–25 (2001), 2000.

- [43] N. D. Nenov. On the 3-colouring vertex Folkman number $F(2, 2, 4)$. *Serdica Math. J.*, 27(2):131–136, 2001.
- [44] N. D. Nenov. On the vertex Folkman number $F(3, 4)$. *C. R. Acad. Bulgare Sci.*, 54(2):21–24, 2001.
- [45] N. D. Nenov. Lower bound for a number of vertices of some vertex Folkman graphs. *C. R. Acad. Bulgare Sci.*, 55(4):33–36, 2002.
- [46] N. D. Nenov. On the triangle vertex Folkman numbers. *Discrete Math.*, 271(1-3):327–334, 2003.
- [47] J. Nešetřil and V. Rödl. The ramsey property for graphs with forbidden complete subgraphs. *J. Comb. Theory*, 20(3):243–249, 1976.
- [48] T. E. O’Neil and C. Ng. Notes on the performance of the CCG algorithm for 3CNF satisfiability. In *Proceedings of the Southern Symposium on Computing, Hattiesburg, MS (December 1998)*.
- [49] K. Piwakowski, S. P. Radziszowski, and S. Urbański. Computation of the Folkman number $F_e(3, 3; 5)$. *J. Graph Theory*, 32(1):41–49, 1999.
- [50] S. P. Radziszowski. Small Ramsey numbers. *Electron. J. Combin.*, 1:Dynamic Survey 1, 30 pp. (electronic). <http://www.combinatorics.org>, 1994.
- [51] S. P. Radziszowski and G. Exoo. Personal communication.
- [52] V. Rosta. Ramsey theory applications. *Electron. J. Combin.*, page Dynamic Survey 13 (electronic). <http://www.combinatorics.org>, 2004.
- [53] M. Schaefer. Graph Ramsey theory and the polynomial hierarchy. *J. Comput. System Sci.*, 62(2):290–322, 2001. Special issue on the Fourteenth Annual IEEE Conference on Computational Complexity (Atlanta, GA, 1999).
- [54] M. Schäuble. Zu einem Kantenfärbungsproblem. Bemerkungen zu einer Note von R. L. Graham. *Wiss. Z. Techn. Hochsch. Ilmenau*, 15(Heft 2):55–58, 1969.
- [55] I. Schur. Über die kongruenz $x^m + y^m = z^m \pmod{p}$. *Jahresbericht der Deutschen Math.-Verein.*, 25:114–117, 1916.

- [56] B. Selman, D. G. Mitchell, and H. J. Levesque. Generating hard satisfiability problems. *Artif. Intell.*, 81(1-2):17–29, 1996.
- [57] J. Spencer. Three hundred million points suffice. *J. Combin. Theory Ser. A*, 49(2):210–217, 1988.
- [58] J. Spencer. Erratum: “Three hundred million points suffice”. *J. Combin. Theory Ser. A*, 50(2):323, 1989.
- [59] S. Tsukiyama, M. Ide, H. Ariyoshi, and I. Shirakawa. A new algorithm for generating all the maximal independent sets. *SIAM J. Comput.*, 6(3):505–517, 1977.
- [60] S. Urbański. Remarks on 15-vertex $(3, 3)$ -Ramsey graphs not containing K_5 . *Discuss. Math. Graph Theory*, 16(2):173–179, 1996.
- [61] B. L. van der Waerden. Beweis einer baudeischen vermutung. *Nieuw Arch. Wiskunde*, 15:212–216, 1927.
- [62] E. W. Weisstein. Graph automorphism. *MathWorld—A Wolfram Web Resource*.
- [63] X. Xiaodong, X. Zheng, and S. P. Radziszowski. A constructive approach for the lower bounds on the Ramsey numbers $R(s, t)$. *J. Graph Theory*, 47(3):231–239, 2004.