

Graph Linear Complexity

Jason Winerip

Nicholas Pippenger, Advisor

Michael Orrison, Reader

May, 2008

HARVEY MUDD
COLLEGE

Department of Mathematics

Copyright © 2008 Jason Winerip.

The author grants Harvey Mudd College the nonexclusive right to make this work available for noncommercial, educational purposes, provided that this copyright statement appears on the reproduced materials and notice is given that the copying is by permission of the author. To disseminate otherwise or to republish requires written permission from the author.

Abstract

This thesis expands on the notion of linear complexity for a graph as defined by Michael Orrison and David Neel in their paper "The Linear Complexity of a Graph". It considers additional classes of graphs and provides upper bounds for additional types of graphs and graph operations.

Contents

Abstract	iii
1 Introduction	1
1.1 Overview	1
1.2 Background	2
2 Original Results	5
2.1 Graph Operations	5
2.2 Proof of Necessity of Multiplication	8
2.3 Generalization of Johnson Graphs	12
3 Conclusion	15
3.1 Summary of Results	15
3.2 Future Work	15
Bibliography	17

List of Figures

2.1	Example of a join and the associated edge partitioning sub-graphs	7
2.2	The graph G , depicted as 3 disjoint copies of K_4 joined with an isolated vertex, and as 3 copies of K_5 identified at a vertex	9

Chapter 1

Introduction

1.1 Overview

This thesis draws on the definition of linear complexity for graphs given by Michael Orrison and David Neel in their paper “Linear Complexity of a Graph” (2006). We begin with some basic definitions from graph theory.

A *graph* $G = \{V, E\}$ is composed of vertices and edges. We label the vertices v_1, \dots, v_n for a graph with n vertices and two vertices v_i and v_j are adjacent (denoted $v_i \sim v_j$) if the graph contains the edge $\{v_i, v_j\}$. We consider only undirected graphs, so $v_i \sim v_j \Leftrightarrow v_j \sim v_i$.

Every graph has an associated *adjacency matrix* which is an $n \times n$ matrix whose entry in row i column j is equal to the number of edges from v_i to v_j . For simple graphs, which is all this thesis will consider, there are never multiple edges between vertices, and a vertex cannot have any edges to itself. Moreover, since we are dealing with undirected graphs, this implies the adjacency matrix is a symmetric 0 – 1 matrix with 0’s along the diagonal.

This is enough background to now rigorously define the linear complexity of a graph as used in this paper. Let

$$(f_{-n+1}, \dots, f_0, f_1, \dots, f_m)$$

be a sequence of linear forms in the indeterminates x_1, \dots, x_n i.e., linear combinations of the x_i ’s with coefficients from \mathbb{R} . Such a sequence is a *linear computation sequence* of length m if, for $i \leq 0$, $f_i = x_{i+n}$ and, for $1 \leq i \leq m$, either

$$f_i = cf_j \text{ or } f_i = \pm f_j \pm f_k$$

for some $c \in \mathbb{R}$ and some $j, k, < i$. This sequence is said to *compute* a set of linear forms F if $F \subseteq \{0, \pm f_i \mid -n < i \leq m\}$. The *linear complexity* of a set of forms is the minimum value m such that there is a sequence of length m that computes that set of forms. In the case of an $n \times n$ matrix A , if $X = [x_1, \dots, x_n]^T$ then we note that AX is a vector consisting of n linear combinations of the indeterminates x_1, \dots, x_n . Considering each element of this vector as an element in a set of forms, the *linear complexity of a matrix* is simply the linear complexity of the set of elements in the vector AX . The *linear complexity of a graph* is then defined to be the linear complexity of its adjacency matrix. Since adjacency matrices are unique up to permuting rows and columns, and since this is equivalent to reindexing the indeterminates or output forms, the complexity is independent of our choice of adjacency matrix.

In Orrison and Neel's paper, many basic bounds on linear complexity are laid out. In particular, they exactly determine the linear complexity of some common classes of graphs, namely trees, cycles, complete graphs, and k -partite cliques. However, other than these relatively simple classes of graphs, exact values for linear complexity are not found, and only very general upper bounds are given in many cases.

1.2 Background

Prior work by Orrison and Neel has established various useful theorems about linear complexity of certain types of graphs. For example, Orrison and Neel define a graph to be *reducible* if it has a vertex of degree 1 (it is only adjacent to 1 vertex), called a leaf, or a redundant vertex v_i . A vertex v_i is a *redundant vertex* if it has the same neighborhood, or set of adjacent vertices, denoted $N(v_i)$, that is the same as the neighborhood of some other vertex, v_j . The notation $I(G)$ denotes the irreducible subgraph of G , which is simply the subgraph that remains after removing all redundant vertices and all leaves. This is useful because $L(G) = L(I(G)) + |V(G)| - |V(I(G))|$, which actually provides equality instead of a bound.

Another bound given by Orrison and Neel relates to upper bounds on complexity based on partitioning edge sets of a graph. They prove that, if G is a graph on n vertices such that $E(G)$ is the union of k disjoint subsets of edges such that the j^{th} subset induces the subgraph G_j of G , and if the i^{th} vertex is in b_i of the induced subgraphs, then $L(G) \leq \sum_{j=1}^k L(G_j) + \sum_{i=1}^n (b_i - 1)$

Orrison and Neel also exactly determined the linear complexity of a k -

partite clique on n vertices. A k -partite clique is a graph where the vertex set can be partitioned into k subsets, and two vertices are adjacent if and only if they are in different subsets. If $k = 2$, $L(G) = n - 2$. If $k = 3$, $L(G) = n$ and otherwise $L(G) = n + k - 2$. The proof of this involves reduction to a complete graph, but these results also obviously imply the values for complete graphs by letting $n = k$. This particular class of graphs is useful because, for one thing, graphs can be partitioned into bipartite cliques, giving a general upper bound based on the complexity of the bipartite cliques. Because cliques have particularly low complexity relative to their number of edges, these partitions give reasonable upper bounds for complexity.

Other useful results from Orrison and Neel's paper include a general upper bound on complexity $\propto O(m \log(n^2/m) / \log n)$ for graphs with m edges and n vertices. This bound comes from a bipartite clique partition constructed by repeatedly finding sufficiently large bipartite cliques in a graph. Orrison and Neel simply show that any graph's complexity is bounded by twice the order of any clique partition, and the order of the bound comes from an algorithm to find a clique partition. Although finding the minimum clique partition is NP-complete, there is a fast algorithm for constructing a partition whose size is $O(m \log(n^2/m) / \log n)$ (Feder and Motwani, 1991).

Nicholas Pippenger's paper on monomials provides an asymptotically similar bound in a much more general case (Pippenger, 1980). The bound proved by Pippenger deals with computations using additions only and considering the maximum number of computations over all matrices of a given size with given entries. Considered for the case of $n \times n$ matrices with 0,1 entries, the upper bound of complexity for a graph of n vertices is $n^2 / \log n^2 + o(n^2 / \log n^2)$. However, since this does not consider anything about the structure, and in particular considers matrices that may not be symmetric, the lower bound established does not hold directly.

Chapter 2

Original Results

2.1 Graph Operations

Before considering the complexity of specific classes of graphs, we look at the effect of various graph operations on complexity. We begin with products of graphs, and in addition to the notion of a direct product of graphs, for which Orrison and Neel found an upper bound, another notion of products of graphs, called the *strong product*, is sometimes used. I will denote the strong product of G and H as $G \cdot H$, and we define $G \cdot H$ to have the vertex set $V(G) \times V(H)$ with an edge from (v_0, w_0) to (v_1, w_1) if and only if $v_0 \sim v_1$ in G and $w_0 \sim w_1$ in H . We note that this operation is associative and commutative, so the product of more than two graphs is well-defined.

We note that this is not the only notion of product defined for graphs, and another product, called the direct product was previously analyzed by Orrison and Neel. The direct product of two graphs $G \times H$ again has vertex set $V(G) \times V(H)$ but has an edge from (v_0, w_0) to (v_1, w_1) if $v_0 \sim v_1$ and $w_0 = w_1$ or $v_0 = v_1$ and $w_0 \sim w_1$. Orrison and Neel showed that, if $G = G_1 \times \cdots \times G_d$ then

$$L(G) \leq |V(G)| \left(\sum_{j=1}^d \frac{L(G_j)}{|V(G_j)|} + (d-1) \right)$$

Theorem 2.1 *If G is the strong product of the graphs G_1, \dots, G_d then*

$$L(G) \leq |V(G)| \sum_{j=1}^d \frac{L(G_j)}{|V(G_j)|}$$

Proof. Although this result is strikingly similar to the upper bound for the direct product, we prove it in a very different way. It will also be illustrated by considering an example, namely $K_3 \cdot K_4$. We start by considering the product of 2 graphs, $G = G_1 \cdot G_2$. To compute the linear form for a vertex $g = (v_0, w_0) \in V(G)$, we note that the neighbors of g are vertices of the form (v_i, w_j) where $v_i \sim v_0$ and $w_j \sim w_0$. Let $f_{v_0}(w_j)$ be the linear form for v_0 considered as a vertex of G_1 , but with each indeterminate x_i replaced with the indeterminate $x_{(i,j)}$, the indeterminate associated with the vertex (v_i, w_j) in G . For the example, this would give $f_{v_0}(w_0) = x_{(1,0)} + x_{(2,0)}$ since the linear form for v_0 in K_3 , which we denote f_{v_0} , is $x_1 + x_2$. We also note that the linear form associated with w_0 in K_4 , denoted f_{w_0} , is $x_1 + x_2 + x_3$. We now want to construct the linear form for the vertex (v_i, w_j) which we denote $f_{(i,j)}$. In the case of the example, we see that

$$\begin{aligned} f_{(0,0)} &= x_{(1,1)} + x_{(2,1)} + x_{(1,2)} + x_{(2,2)} + x_{(1,3)} + x_{(2,3)} \\ f_{(0,0)} &= \sum_{\{j|x_j \in N(w_0)\}} f_{v_0}(w_j). \end{aligned}$$

Note that, if $x_j \in N(w_0)$ then x_j has a coefficient of 1 in the linear form f_{w_0} . We see that this sum holds in general, since the linear form for (v_i, w_j) consists of the sum over any inderteminate of the form $x_{(a,b)}$ where x_a appears in f_{v_0} and x_b appears in f_{w_0} , which is exactly what this sum generates. Thus, to compute all the linear forms, we can simply compute all the forms associated with G_1 once for each vertex in G_2 and then, for each vertex in G_1 , we replace an indeterminate with the linear form for that vertex and then perform the computation sequence associated with G_2 on these linear forms. The total operations required for this is

$$|V(G_1)|L(G_2) + |V(G_2)|L(G_1) = |V(G)| \left(\frac{L(G_1)}{|V(G_1)|} + \frac{L(G_2)}{|V(G_2)|} \right).$$

Iterating over multiple multiplications gives the stated result, since allowing G_1 to be a product itself, you get cancellation of $|V(G_1)|$ and are left with a sum over any graphs in the product divided by the size of their vertex sets. ■

Another common graph operation is the *join* of two graphs, defined for disjoint graphs G and H as the graph $G + H$ with vertex set $V(G + H) = V(G) \cup V(H)$ with adjacency relation $v_1 \sim v_2$ if either $v_1 \sim v_2$ as vertices in either G or H or $v_1 \in G$ and $v_2 \in H$. Any graph formed this way will have

an edge partition consisting of G , H , and $K_{|V(G)|,|V(H)|}$, and every vertex will appear in exactly two of these graphs. This trivially gives an upper bound of $L(G) + L(H) + 2|V(G)| + 2|V(H)| - 2$, however we can do a bit better by reusing computations.

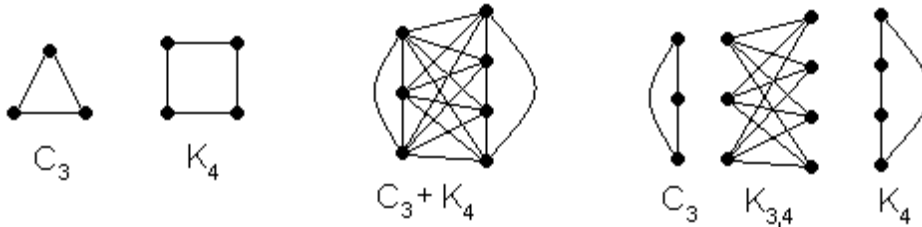


Figure 2.1: Example of a join and the associated edge partitioning sub-graphs

Theorem 2.2 Let $\Delta(G)$ denote the maximum degree of a vertex in graph G . $L(G + H) \leq L(G) + L(H) + 2|V(G)| + 2|V(H)| - \Delta(G) - \Delta(H)$.

Proof. Note that every vertex is adjacent to either all vertices in G or all vertices in H , in addition to whatever vertices it is adjacent to prior to forming the join. Thus, without loss of generality, if we consider a vertex originally in G , the linear form of that vertex as a vertex in $G + H$ is the sum of the linear form for that vertex as a vertex in G plus the sum over all indeterminates associated with vertices in H . This suggests a computation sequence which starts with the computation sequences for G and H , then computes from the longest sum, which must at least consist of $\Delta(G)$ indeterminates, the sum over all indeterminates associated with G and therefore requires at most $|V(G)| - \Delta(G)$ operations. Similar reasoning generates a sum over all indeterminates associated with H . Finally, for every vertex, add the linear form associated with it as a vertex in G or H plus the sum over all vertices from the other component graph, which takes $|V(G + H)| = |V(G)| + |V(H)|$ operations. This generates a computation sequence of length $L(G) + L(H) + 2|V(G)| + 2|V(H)| - \Delta(G) - \Delta(H)$, proving the desired upper bound. ■

Another operation, considered mainly for its interesting results on complexity, is an operation which combines two vertex disjoint graphs G and H by identifying a vertex $v_g \in G$ with the vertex $v_h \in H$ and preserving all

adjacency relationships. In the case of combining two graphs in this way, there is a trivial bound of $L(G) + L(H) + 1$ which comes from an edge partition. That is, you simply perform all computations in the sequences for G and H independently, then add f_{v_g} to f_{v_h} . Naturally, combining n graphs, G_1, \dots, G_n in this way gives an upper bound of $\sum_{i=1}^n L(G_i) + n - 1$. However, in the case of complete graphs, we can prove a better bound, just as with direct products of complete graphs. Noting that we calculate a sum over all indeterminates for a single complete graph, we can improve on the upper bound by taking advantage of these sums. Instead of performing the final subtraction which computes the linear form associated with the vertices being identified, instead add the sums together. Given n complete graphs, this replaces n subtractions with $n - 1$ additions. Finally, multiply the indeterminate associated with the vertex about which the graphs have been combined by n and subtract this from the total sum. Given graphs K_{k_1}, \dots, K_{k_n} , this results in a graph with linear complexity of

$$\sum_{i=1}^n (2k_i - 2) + 1.$$

2.2 Proof of Necessity of Multiplication

Given that the linear forms for any adjacency matrix only ever have coefficients of 1, the existence of graphs such that multiplications are used in an optimal computation sequence is of interest. Although Orrison and Neel used a computation sequence with multiplications to prove an upper bound for products of complete graphs, it is still possible multiplication is not used in the optimal case. However, given one admittedly major assumption, there is a specific graph where a provably optimal sequence consists of a multiplication.

The entire point of this proof is to establish that the graph G on 13 vertices constructed by taking three disjoint copies of K_4 and adding single vertex adjacent to all other vertices has an optimal computation sequence that consists of a multiplication. Because this graph is the only one I am interested in, I am willing to make a major assumption about this graph that most likely does not hold in the general case. Specifically, I am willing to assume that the graph consisting of three disjoint copies of K_4 has linear complexity exactly equal to $3L(K_4)$ or 18. Moreover, I assume that no optimal computation sequence for this specific graph has any sums of indeterminates associated with disconnected vertices, and I call any such sums

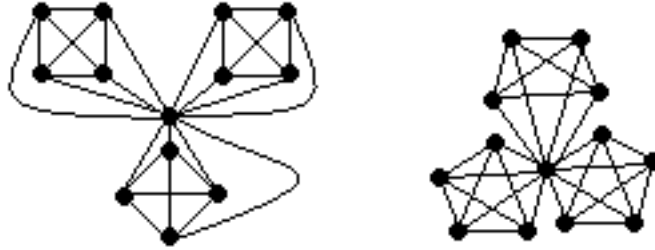


Figure 2.2: The graph G , depicted as 3 disjoint copies of K_4 joined with an isolated vertex, and as 3 copies of K_5 identified at a vertex

cross-component sums. Furthermore, I assume that in the optimal computation sequence for G , it is the case that, given two linear forms that share only one indeterminate, the computation of these two forms is optimally done independently. That is, they never share an intermediate form that is not a single indeterminate. Note that, even if this assumption breaks down, we have an even more bizarre phenomenon, namely a graph on only 13 vertices with linear complexity (as we will see) of 25 that is optimally computed by performing what seem like disjoint computations simultaneously.

We note that disjoint computations are not necessarily optimally performed disjointly, as shown by the existence of fast matrix multiplication algorithms. The simple bound of $4.7n^{\log_2 7}$ for matrix multiplication proved by Strassen shows that, for a large enough matrix, the computation of the product of two $n \times n$ matrices takes $4.7n^{\log_2 7}$ operations but computing it as n disjoint products of a matrix times a vector would take n^3 operations in the general case (Strassen, 1969). We note that this can directly apply to graphs in the case of a n disjoint copies of the same n vertex graph H . Performed disjointly, this takes $nL(H)$ operations, which depending on the number of edges, can exceed $n^{2.81}$. However, this gives the same result as multiplying the adjacency matrix of H with an $n \times n$ matrix consisting of the indeterminates x_1 through x_{n^2} , which using fast matrix multiplication, can be done in $4.7n^{2.81}$ operations. Thus, this assumption certainly does not hold in general, but the linear complexity of 25 we will find for the graph G we are actually interested in is well below $4.7n^{2.81}$, and I do not believe my assumptions break down for this small a case.

We also note that, in the case of K_4 , it is necessary to compute final

forms in the process of computing the sum over all indeterminates, which will be a necessary fact for the following proof. This follows from the fact that the complexity of K_n is $2n - 2$, there are n final forms, and it requires $n - 1$ computations to compute the sum over all indeterminate. Thus, if no final forms are computed in the process of computing the sum over all indeterminates, it would take at least $2n - 1$ operations. Therefore, at least one final form must be computed in the process.

For the proof of a lower bound, we first explicitly define redundant forms. A *redundant form* is, quite simply, a term which appears more than once in a linear computation sequence. Obviously, minimal computation sequences cannot contain redundant forms, since removing the second appearance of a redundant form still computes exactly the same set of linear forms but consists of one fewer form, and is therefore shorter.

Theorem 2.3 *Let G be the graph on 13 vertices constructed by taking 3 disjoint copies of K_4 and adding single vertex adjacent to all other vertices depicted in 2.2. $L(G) = 25$.*

Proof. We start, naturally, with a proof of the upper bound, simply by providing the computation sequence. Let v_1 through v_4 be the vertices of the first K_4 , v_5 through v_8 the second, v_9 through v_{12} the third, and let v_{13} be the central vertex, adjacent to all other vertices, and let f_i be the form associated with v_i . The computations sequence proceeds as follows:

For $j = 0, 1$, and 2 , compute $S_j = x_{13} + \sum_{i=1+4j}^{4+4j} x_i$. This takes 12 total operations and computes in the process f_4, f_8 , and f_{12} .

For the 9 remaining vertices of the complete graphs, we have $f_i = S_{\lfloor \frac{i}{4} \rfloor} - x_i$. This takes 9 total operations.

Compute $\sum_{j=0}^2 S_j$ which gives $3x_{13} + f_{13}$ in 2 operations.

Compute $3x_{13}$, and from this, we get f_{13} . This is 2 more operations.

This computation sequence computes all 13 linear forms and takes a total of 25 operations, giving $L(G) \leq 25$.

To prove this is a lower bound, we assume we have a minimal computation sequence for G , and show that removing v_{13} generates at least 7 unnecessary operations for the computation of the forms for 3 disjoint copies of K_4 . Since we have assumed this graph has linear complexity 18, this will prove the theorem. We let $H = G - v_{13}$, which is graph consisting of 3 disjoint copies of K_4 . Since $d(v_{13}) = 12$, every final form for G other than f_{13} must contain x_{13} . Since v_{13} does not exist in H , the indeterminate x_{13} does not exist in any computation sequence for H . Therefore, since the linear computation sequence for G contains some form f and the

form $f + x_{13}$, the removal of x_{13} generates a redundant form. Moreover, by our assumption, we note that two vertices in different components of H have at most one indeterminate in common as vertices of G , so the linear computation sequence generates these forms using independent computation sequences. Specifically, x_{13} must at some point be introduced into the computation sequence for each component. Since H has three components, this means the operation of adding x_{13} to some other linear form must be performed three times, generating three forms which are redundant in the computation sequence for H .

Again noting $d(v) = 12$, we know that one of the forms for G is $f_{13} = \sum_{i=1}^3 2x_i$. We note that this is clearly a cross-component sum, and therefore cannot exist in the optimal computation sequence for H by our assumption. In fact, it is a sum across 3 components, so there must also be at least 1 other cross-component sum which is no longer necessary in H , giving us at least 2 forms which can be removed. We now consider 2 cases, either there are exactly 2 cross-component sums to remove or there are exactly 3 redundant forms. (3 cross-component sums and 4 redundant forms would be enough unnecessary forms to prove the theorem, so with these as minimums, we must prove that in either case, there are enough other unnecessary forms.)

Case 1, we assume that there are exactly 2 cross-component sums, and therefore f_{13} is computed as the sum of the 3 total sums over each component in 2 operations. In that case, we have as intermediate forms total sums over each component. As noted earlier for the case of H , either some final form f_i has been computed in the process of computing these total sums or this sequence is not an optimal computation sequence for H , in which case there is at least one extra operation per component, which would immediately give at least 8 operations which can be removed, so we assume final forms are completed before the total sum. However, every final form in G other than f_{13} must have x_{13} in the sum. Since we assumed the sum over each component without x_{13} was computed, the indeterminate x_{13} must be added to f_i to make it a final form in the computation sequence for G , but the sum over all indeterminates other than x_{13} cannot build off $f_i + x_{13}$ without spending an operation removing x_{13} , which would generate redundant forms anyway. Thus, removing x_{13} must generate at least 2 redundant forms in each component, giving a total of 6 redundant forms and 2 cross-component sums, or 8 forms which are unnecessary.

Case 2, we can assume there are exactly 3 redundant forms generated in calculating all forms except for f_{13} . We must find a sequence that computes f_{13} in as few operations as possible assuming that, if there are total sums over components, they appear including x_{13} . By adding together these 3

sums, we generate an intermediate cross-component sum, and the form $3x_{13} + \sum_{\gamma \in H} x_\gamma$. This can be transformed into f_{13} through 2 operations, namely generating $3x_{13}$ and subtracting it from this form. Since neither of these forms could possibly be present in the optimal computation sequence for H , we have found a total of 7 forms which are unnecessary in H . Since this has covered all possibilities which could have generated fewer than 7 unnecessary forms, we see that we can take the computation sequence for G and turn it into a sequence for H with at least 7 unnecessary operations being performed. Thus, $L(H) \leq L(G) - 7$. Since $L(H) = 18$, this gives $L(G) \geq 25$ as desired. ■

Although I suspect a parallel construction using copies of K_3 would yield a similar result, K_3 has complexity 3, and as such my lower bound would not hold. However, this graph stemmed from the notion of taking 2 graphs G and H , identifying a vertex $v_g \in G$ with a vertex $v_h \in H$, leaving all other vertices alone, and looking at the complexity. Denoting the resulting graph $GH(v_g, v_h)$, it is trivial to see from bounds using edge partitions that, in the general case, $L(GH(v_g, v_h)) \leq L(G) + L(H) + 1$. However, using complete graphs (and omitting the vertex specification, since all vertices in complete graphs are interchangeable), and extending the notion to multiple graphs by identifying one vertex from each graph, in the case of $K_4K_4K_4$ this bound gives an upper bound of 20. But using a similar computation sequence to the one demonstrated for the K_5 case above, we get a computation sequence of length 19.

2.3 Generalization of Johnson Graphs

The *Johnson graph* $J(n, k)$ is the graph whose vertices are the k -element subsets of $\{1, \dots, n\}$ with the adjacency relation $v \sim w$ if and only if $|v \cap w| = k - 1$. Orrison and Neel showed that $L(J(n, k)) < \binom{n}{k}(2k + 1)$. Johnson graphs can easily be generalized such that two vertices are adjacent if their intersection is of some other fixed size, and thus we define $J(n, k, j)$ to be the graph with the same vertex set as $J(n, k)$ and with $v \sim w$ if and only if $|v \cap w| = k - j$. It is worth noting that two vertices are adjacent in $J(n, k, j)$ if the minimum path between them in $J(n, k)$ is of length j .

Theorem 2.4 $L(J(n, k, j)) \leq j \cdot L(J(n, k)) + (j - 1)5\binom{n}{k}$.

Proof. Since 2 vertices in $J(n, k, j)$ are adjacent if they are a distance j apart in $J(n, k)$, we note that replacing x_j with f_j replaces every instance of x_j

with the indeterminates a distance 1 from v_j , since f_j is the sum over the indeterminates associated with the neighbors of v_j . Replacing x_j with f_j in the computation sequence for $J(n, k, 1)$ gives a sum over twice the indeterminates of all vertices at distance 2, $(n - k) \cdot k$ times the vertex itself (since each vertex has degree $k(n - k)$ and can follow a path of length two to any neighbor then back to itself), and $n - 2$ times each of the indeterminates associated with vertices at distance 1. The $n - 2$ copies of indeterminates at distance 1 come from the fact that, given an initial vertex v with element i and neighbor v' with element i' , we can generate a new neighbor of both vertices by choosing any element other than i and i' . If it appears in the subsets associated with both vertices, replace it with i . If not, replace i' with it. This gives $n - 2$ vertices adjacent to both v and v' . Since the linear form for each vertex in the case where $j = 1$ has already been computed (it was what we began the iteration having calculated) it takes 5 operations per vertex to multiply by the appropriate scalars and then subtract the unnecessary terms. Thus, in $2L(J(n, k)) + 5\binom{n}{k}$ operations we can calculate the linear forms for $J(n, k, 2)$. Replacing indeterminates with these new forms and iterating again generates indeterminates for vertices of original distance 1, 2, or 3, all of which have been previously computed, so each additional iteration ends up requiring $L(J(n, k)) + 5\binom{n}{k}$ operations, which gives the upper bound indicated. ■

Chapter 3

Conclusion

3.1 Summary of Results

This thesis was able to establish some decent upper bounds for multiple graph operations and one class of graphs, as was my initial intention. Although I was generally pleased with my ability to generate new upper bounds for these various operations and graphs, it was my success in proving that linear complexity allowing multiplications gives different results than allowing only additions and subtractions that was the major success of this thesis. Although the underlying assumption for this proof still needs some justification, I feel confident that, in the case considered, these assumptions are valid, since they are made for a single graph with only a few vertices and already low complexity. There just doesn't seem to be space in a computation sequence of length 18 for a 12 vertex graph to do any particularly indirect shortcuts, since there are only 6 free operations in addition to final forms.

3.2 Future Work

There is still much in this area that remains unaddressed. Justifying the assumptions for the proof of multiplication would be one place to start, and addressing these assumptions in general would be nice. Although the question of the behaviour of complexity when considering multiple disjoint copies of the same graph was briefly addressed, I was only able to establish that, with enough edges, the existence upper bound is large enough that fast matrix multiplication would give a better upper bound. Whether it's actually better is another question entirely, and one that I did not have the

16 Conclusion

opportunity to address. In addition, very few specific classes of graphs have been considered here, and most of the graphs previously by Orrison and Neel were even more basic graphs. Moreover, with the exception of k -partite cliques, trees, and cycles characterized by Orrison, all we have is upper bounds for a very small set of graphs, and a very rough upper bound in the general case, with a very far off lower bound. There is therefore certainly room for improvement on many bounds.

Bibliography

- Feder, Tomás, and Rajeev Motwani. 1991. Clique Partitions, Graph Compression and Speeding-up Algorithms. In *STOC '91: Proceedings of the Twenty-third Annual ACM Symposium on Theory of Computing*, 123–133. New York, NY, USA: ACM. doi:<http://doi.acm.org/10.1145/103418.103424>.
- Neel, David, and Michael Orrison. 2006. The linear complexity of a graph. *Electronic Journal of Combinatorics* 13(1):9–19. URL http://www.combinatorics.org/Volume_13/PDF/v13i1r9.pdf.
- Pippenger, Nicholas. 1980. On the Evaluation of Powers and Monomials. *SIAM Journal on Computing* 9:230–250.
- Strassen, V. 1969. Gaussian Elimination is not Optimal. *Numerische Mathematik* 13:354–356.