University of Nevada
Reno

Efficient Generation of Minimal Graphs Using Independent Path Analysis

A thesis submitted in partial fulfillment of the
requirements for the degree of Master of Science
with a major in Computer Engineering

by

Linda S. Humphrey

Dr. Frederick C. Harris, Jr., Thesis Advisor

December, 2006

University of Nevada, Reno
Statewide • Worldwide

THE GRADUATE SCHOOL

We recommend that the thesis
prepared under our supervision by

LINDA S. HUMPHREY

Entitled

Efficient Generation Of Minimal Graphs Using Independent Path Analysis

be accepted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

Frederick C Harris

Frederick C. Harris, Jr., Ph.D., Advisor

Dwight Egbert

Dwight Egbert, Ph.D., Committee Member

Indira Chatterjee

Indira Chatterjee, Ph.D., Graduate School Representative

Marsha H Read

Marsha H. Read, Ph. D., Associate Dean, Graduate School

December, 2006

Acknowledgments

What a long, strange trip it's been! I would never have believed, sitting there on the floor in CS106 so many years ago (there weren't enough seats that first day), that I would now be completing my Master's in CPE. So many wonderful teachers and friends I have met along the way, far too many to thank individually, but a few deserve special recognition.

First I would like to thank my advisor, Dr. Frederick C. Harris, Jr., who has been a shining light in my life ever since day one of 106. He was always there, ready to encourage, commissurate, or just chat. He never pressured me to finish, but was totally supportive of my efforts, whether they fizzled out (remember the IPSec phase?) or came to fruition. The CSE department is most fortunate to have such a wonderful educator and genuinely caring person.

I am also grateful to my other two committee members: Dr. Dwight Egbert, with special thanks for tolerating me in just about every class he teaches at UNR, and Dr. Indira Chatterjee, who generously agreed to serve on my committee even though we had never met. Your input is most appreciated.

Very special thanks go to my friend and collaborator, Dr. Judy Fredrickson. Without her help and gentle prodding, there would never have been a "period at the end of the sentence". All the brainstorming over lunches at Record St. Cafe, agonizing over code that almost worked, and rejoicing in ultimate success will remain very fond memories.

Thanks also to my friend and fellow traveler Nancy LaTourrette. Her help and companionship through many grad classes we shared made them so much more fun (or in a few cases, bearable). Nachos Borrachos rule!

Finally, I am eternally indebted to my beloved husband, Mike, and son, Scott. Without their encouragement, tolerance of my absences and irratic schedule, and willingness to eat more carry-out food than any person should have to endure, this work would never have been completed. With heartfelt thanks, I dedicate my work to them.

November 1, 2006

Abstract

Determination of the minimal crossing number of a complete graph is an NP-complete problem that has been attacked in various ways over the years with limited success. Much previous work has focused on conjectured solutions that have been verified only for small values of n, where n is the number of vertices in the graph's vertex set. The advent of more powerful computers and the ability to network multiple computers into Beowolf clusters have led to renewed interest in exact solutions to crossing number problems.

In this thesis, we introduce a new method for finding the minimal crossing number of complete graphs. Our algorithm lays down new edges independent of each other to create minimal K_n graphs from minimal K_{n-1} graphs, a technique we have entitled Star Analysis. We validate our algorithm on small complete K_n graphs, then demonstrate its efficiency in generating larger minimal K_n graphs up to $n = 11$. Finally, we extend our process to the generation of minimal $K_{m,n}$ bipartite graphs to document its usefulness in a variety of graph family problems.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The Minimal Crossing Number problem can be defined as the determination of the minimum possible number of crossings required to connect a set of points in a plane in some defined manner. It is a problem that has intrigued investigators since its original description by Turán in 1944 [9]. Since the points and connections are easily represented in graphic format, the Minimal Crossing Number problem has been most commonly studied as a problem in graph theory.

A *graph* is a finite nonempty set of vertices, customarily depicted as points, along with a set of unordered pairs of vertices called edges. [2] A *complete* graph, denoted K_n, is one in which the edge set is complete, i.e. every vertex is *adjacent* (connected) to every other vertex in the graph. When a graph is drawn in a plane, and the number of vertices, n, exceeds four, then at least two of the edges must cross each other at a point not represented by a vertex (Figure 1.1). The problem is minimizing the number of these crossings as the number of vertices increases. A *minimal* K_n graph is one that exhibits the minimal number of edge crossings possible for a graph of n vertices.

Like many NP-complete problems, the Minimal Crossing Number problem is deceptively simple and very easily stated, whereas its solution is complex. Addition of a new vertex to a minimal K_n graph requires that connections be made between the new vertex and each of the existing n vertices, minimizing the number of existing edges that are crossed. How can one know where in the graph to place the new vertex, and in what sequence and direction should the new edges be added? Very

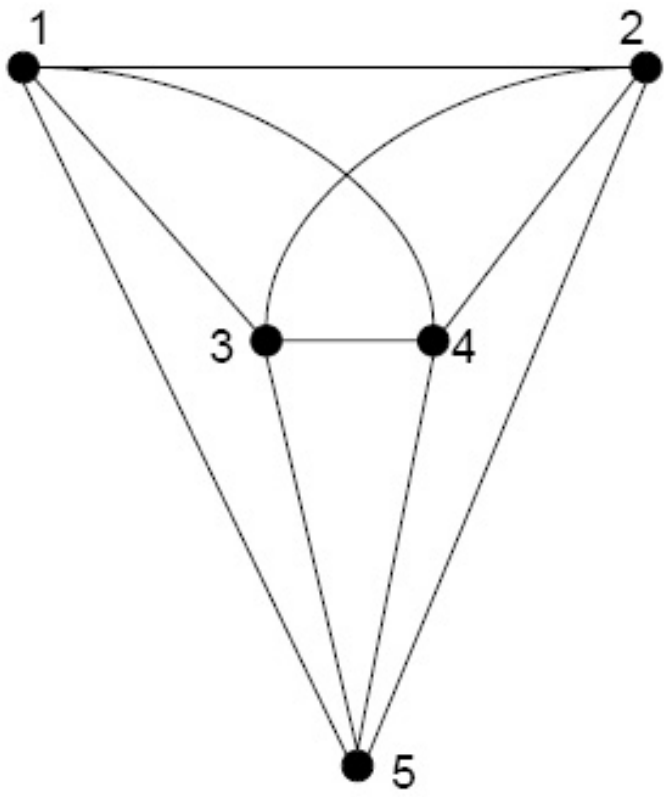


Figure 1.1: Minimal K_5 Graph Demonstrating One Crossing

small complete graphs can be easily created by hand, but the complexity escalates quickly with the number of vertices. Even with all the optimizations to be presented, given a complete graph of size K_7, modest in size relative to the real-world problems in need of solutions, one must draw 76 graphs to exhaustively search all the possible vertex placements and edge configurations that could be used to create a minimal K_8. Within a few more generations of graphs, manual solutions are out of the question. From K_8 to K_9, 4,590 graphs must be drawn and from K_9 to K_{10}, 56,618 graphs must be drawn [5]. Many weeks or longer of processor time are required to perform an exhaustive search for this many graphs. Prior to the work presented here, the situation was orders of magnitude worse.

This thesis presents a novel method of generating minimal K_n graphs. Beginning with previous work involving exhaustive search and proceeding through several levels of optimization, we have arrived at an algorithm that allows efficient creation of all possible minimal K_n graphs from minimal K_{n-1}. The algorithm is not specific to complete graphs, so application to other graph families is possible.

The remainder of this thesis is broken down as follows: Chapter 2 provides a brief set of pertinent definitions from graph theory, following which review of the literature relevant to the Minimal Crossing Number problem is presented. Chapter 3 introduces our algorithm, which we have entitled Star Analysis. Both an overview of the process itself and a description of the underlying data structures and implementation details are presented. Chapter 4 describes the results we have obtained applying the Star Analysis algorithm to several crossing number problems. Finally, a summary and suggestions for future work are found in Chapter 5.

Chapter 2

Literature Review

2.1 Definitions

Several basic definitions from the field of graph theory were given in Chapter 1. We will now provide the remainder of the definitions necessary for the discussions that follow. Unless otherwise noted, all definitions are derived from [2].

Definition 1 *The **order** of a graph is the cardinality of its vertex set, denoted **n***

Definition 2 *The **size** of a graph is the cardinality of its edge set, denoted **m***

Definition 3 *The **degree** of a vertex, **v**, is the number of edges incident with v, denoted **deg v**. In the case of a complete graph, the degree is n-1 for all vertices. The sum of the degrees of the vertices of a graph is **2m**.*

Definition 4 *An **n x n adjacency matrix** is a representation of a graph of order n in which an entry at position x,y is 1 if the vertices x and y are adjacent, otherwise is 0. Zero entries occur along the main diagonal of the matrix since vertices are not considered to be adjacent to themselves.*

Definition 5 *Two graphs are **isomorphic** if there is a one-to-one mapping of their vertex sets that preserves adjacency. Isomorphic graphs therefore have identical adjacency matrices.*

Definition 6 *A graph is **k-partite**, $k \geq 1$, if its vertex set can be partitioned into k subsets called **partite sets**, such that each element of the edge set of the graph*

connects a vertex V_i to a vertex V_j, $i \neq j$. For $k = 2$, such graphs are called ***bipartite graphs*** *(Figure 2.1). A* ***complete k-partite graph****, denoted $K_{n1,n2,n3,...,nk}$, is one in which each vertex V_i is connected to every vertex in each of the other partite sets.*

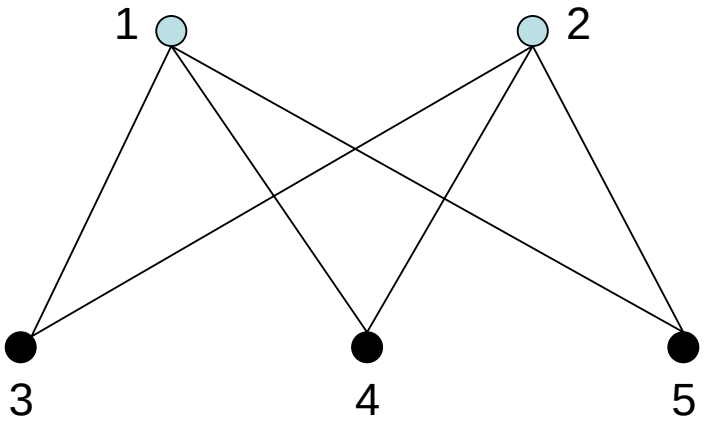


Figure 2.1: Complete Bipartite Graph, $K_{2,3}$

Definition 7 *A graph is* ***planar*** *if it can be embedded in the plane (or on the sphere) with no edge crossings apart from those occurring at vertices. If it is embedded in the plane, it is called a* ***plane graph****. A graph that cannot be embedded in the plane (because of edge crossings) is referred to as* ***drawn*** *in the plane.*

Definition 8 *A* ***region*** *of a plane graph is a maximal portion of the plane for which any two points may be joined by a curve A such that each point of A neither corresponds to a vertex of the graph nor lies on any curve corresponding to an edge of the graph.*

Definition 9 *The* ***boundary*** *of a region R of a plane graph consists of all points x corresponding to the vertices and edges of the graph having the property that x can be joined to a point of R by a curve, all of whose points that differ from x belong to R.*

Definition 10 *A region in an embedding is called a* ***2-cell region*** *if any simple closed curve in that region can be continuously deformed or contracted in that region to a single point.*

Definition 11 *An embedding is a **2-cell embedding** if all the regions in the embedding are 2-cell.*

Theorem 1 *[Euler's Formula] If a graph, G, is a connected (essentially the same as complete) plane graph with n vertices, m edges and r regions than the following relationship holds:*

$$n - m + r = 2$$

Proof of Theorem 1 can be found in [2], page 155. A consequence of Theorem 1 is that every two embeddings of a connected planar graph in the plane have the same number of regions.

Definition 12 *The **crossing number** of a graph, denoted **cr(G)** or ν**(G)**, is the minimum number of edge crossings among its drawings in the plane. Per Definition 7, such a graph is nonplanar unless its crossing number is 0. (A subset of crossing number problems considers the **rectilinear crossing number**, which has the added requirement that each edge is a straight line segment [10]. These graphs will not be considered further here.)*

Definition 13 *A **minimal graph** is one that exhibits the crossing number when drawn in the plane.*

Definition 14 *A **good drawing** of a graph, G, is one in which:*

- *adjacent edges never cross*
- *two nonadjacent edges cross at most once*
- *no edge crosses itself*
- *no more than two edges cross at a point of the plane and*
- *the (open) arc in the plane corresponding to an edge of the graph contains no vertex of the graph*

2.2 History of the Minimal Crossing Number Problem

2.2.1 Initial Description

Credit for the original description of the Crossing Number Problem is generally given to Paul Turán, based on his personal experience in a brick factory near Budapest during World War II [9]. In the factory were kilns, where bricks were made, and storage yards, where they were stacked. The former were connected to the latter by rail tracks, forming what we now call a complete bipartite graph. Turán noticed that problems often arose at the points where rails crossed, with trains jumping the tracks and bricks falling to the ground. He mused that those problems and the extra work they created could have been minimized had someone thought through the track layout in advance to minimize the number of crossings. Turán subsequently delineated his problem in lectures given in 1952 in Warsaw and Wroclaw, and it became known as Turán's Brick Factory Problem.

In the Warsaw audience was a mathematician named Zarankiewicz, who became one of the first to take up the challenge. In 1954, he conjectured that the crossing number of a complete bipartite graph is as follows [23]:

Conjecture 1 *[Zarankiewicz] The crossing number of the complete bipartite graph $K_{m,n}$ satisfies the equality*

$$\nu(K_{m,n}) = \left\lfloor \frac{m}{2} \right\rfloor \left\lfloor \frac{m-1}{2} \right\rfloor \left\lfloor \frac{n}{2} \right\rfloor \left\lfloor \frac{n-1}{2} \right\rfloor$$

He noted that his conjecture was concurrently and independently discovered by Urbanik.

2.2.2 Early Work

Zarankiewicz's conjecture was verified by Kleitman [12] for the case when $\min(m, n) \leq 6$, and by Woodall [20] for the case when $m = 7$, $n \leq 10$. Unfortunately, the conjecture was thrown into question in 1965/66 when both Kainen and Ringel noticed a flaw in

Zarankiewicz's paper, later reiterated by Guy [9]. The flaw was the assumption by Zarankiewicz that among the m graphs $\mathrm{K}_{1,n}$ that compose the graph $\mathrm{K}_{m,n}$, there will always be two that do not contain a crossing. Zarankiewicz's conjecture stands to the present as an upper bound on the crossing number of a complete bipartite graph, having been neither proven nor disproven for equality.

Guy continued to pursue the problem of Minimal Crossing Number of a complete graph, introducing an upper bound in 1960 that persists to the present [8].

Theorem 2 *[Guy] The crossing number of the complete graph K_n satisfies the inequality*

$$\nu(K_n) \leq \tfrac{1}{4} \left\lfloor \tfrac{n}{2} \right\rfloor \left\lfloor \tfrac{n-1}{2} \right\rfloor \left\lfloor \tfrac{n-2}{2} \right\rfloor \left\lfloor \tfrac{n-3}{2} \right\rfloor.$$

He went on to conjecture that equality holds for all n.

Conjecture 2 *[Guy] The crossing number of the complete graph K_n satisfies the equality*

$$\nu(K_n) = \tfrac{1}{4} \left\lfloor \tfrac{n}{2} \right\rfloor \left\lfloor \tfrac{n-1}{2} \right\rfloor \left\lfloor \tfrac{n-2}{2} \right\rfloor \left\lfloor \tfrac{n-3}{2} \right\rfloor.$$

This conjecture has been proven for $n \leq 10$, but remains to be shown for $n > 10$. Guy has speculated privately that he now anticipates divergence from his conjecture with larger n.

Surprisingly little work has been done on the problem of crossing numbers, considering its many applications in circuit design [13, 14], graphics [1], network configuration, etc. The explanation for this undoubtedly lies in the difficulty in making further progress. This difficulty became more understandable in 1983, when Garey and Johnson proved that the Minimal Crossing Number problem is NP-complete [7]. Their proof consists of transforming a known NP-complete problem, the Optimal Linear Arrangement problem, into the Minimal Crossing Number problem. They suggested that the intractable nature of the Minimal Crossing Number problem meant that further research should be directed toward inexact methods that only estimate

crossing numbers. Indeed, subsequent research tended to focus on related problems and subproblems and is difficult to synthesize into a coherent whole, whereas the attempt to search further for exact results for crossing number problems was largely abandoned.

2.3 Harris and Harris Algorithm

2.3.1 Introduction

In 1996, the first algorithm to calculate the exact crossing number of a complete graph was proposed by Harris and Harris [11]. The algorithm is based on a tree structure, using exhaustive depth-first search limited by branch-and-bound methodology. As it is the starting point for all the work to be discussed below, we will examine it now in some detail.

2.3.2 Edmonds' Rotational Embedding Scheme

A key part of the Harris and Harris algorithm is Edmonds' Rotational Embedding Scheme [3]. The scheme is used to determine the planarity of partial graphs prior to mapping to the tree structure. It is formally presented in [2], pages 196-197 as Theorem 7.14, which will be excerpted here.

> Let G be a nontrivial connected graph with $V(G)$ (the vertex set of G) $= \{v_1, v_2, \ldots, v_n\}$. For each 2-cell embedding of G on a surface there exists a unique n-tuple $(\pi_1, \pi_2, \ldots, \pi_n)$, where for $i = 1, 2, \ldots, n$, $\pi_i : V(i) \rightarrow V(i)$ is a cyclic permutation that describes the subscripts of the vertices adjacent to v_i in counterclockwise order about v_i. Conversely, for each such n-tuple $(\pi_1, \pi_2, \ldots, \pi_n)$, there exists a 2-cell embedding of G on some surface such that for $i = 1, 2, \ldots, n$ the subscripts of the vertices adjacent to v_i and in counterclockwise order about v_i are given by π_i.

The scheme is best illustrated with an example, taken from [11]. Figure 2.2 displays a planar embedding of a graph. The counterclockwise permutations associated

with each of the vertices are as follows:

$$\pi_1 = (6, 4, 2) \quad \pi_2 = (1, 4, 3)$$
$$\pi_3 = (2, 4) \quad \pi_4 = (3, 2, 1, 5)$$
$$\pi_5 = (4, 6) \quad \pi_6 = (5, 1)$$

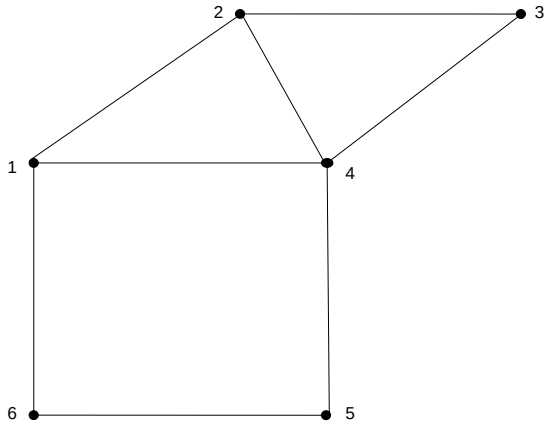


Figure 2.2: A Planar Embedding of a Graph

These permutations allow us to determine the edges of the graph, and thereby the regions of the graph. As an example, we will use the permutations to determine the edges of the region bounded by (1,2), (2,4), and (4,1). The edges are traced as follows:

1) Start with edge (1,2).
2) From the permutation π_2 determine which vertex follows 1; it is 4. Therefore the second edge is (2,4).
3) From the permutation π_4 determine which vertex follows 2; it is 1. Therefore the third edge is (4,1).
4) From the permutation π_1 determine which vertex follows 4; it is 2. This yields edge (1,2), which was the original edge, so we are finished.

The graph in Figure 2.2 has four regions; three are obvious and the fourth is the external region bounded by (6, 5), (5, 4), (4, 3), (3, 2), (2, 1), and (1, 6). Notice that the external region must be traced counterclockwise in order for the permutations to define it, whereas all the other regions are traced clockwise.

Once the regions are identified and counted, Euler's formula can be employed to determine if a planar embedding exists. If Euler's formula is not satisfied, we know that one or more crossings occurred when the graph was drawn.

2.3.3 Description of the Algorithm

The Harris and Harris algorithm commences by taking the unconnected vertex set of the graph and adding edges. After the addition of each edge, the Rotational Embedding Scheme is employed to count the resulting regions, and then Euler's formula is used to test for planarity of the resulting partial graph. Edges continue to be added until the partial graph is no longer planar, at which time the last edge is removed to return to the last planar configuration. This process is repeated for every permutation of the vertices. The result is all the partial graphs possible from that vertex set that have planar embeddings.

At this point the remaining search space is mapped onto a tree. The root of the tree has a branch for each of the planar partial graphs created above. Starting with the first branch of the tree, we begin the depth-first search process, building and searching simultaneously.

Depth-first search is a tree traversal algorithm that searches as deeply as possible into the tree for a solution before moving on to adjacent branches. Only after a terminal leaf is reached, which represents a solution for that branch of the tree, does the algorithm backtrack up to the previous division. Then, it again goes to the deepest position of the adjoining branch before backtracking. In this manner, the entire tree is exhaustively searched.

The added wrinkle of branch and bound provides one method of improving efficiency. The current optimal solution is maintained as the search progresses, so that whenever that optimum is exceeded, the algorithm searches no deeper in the current subtree. Aborting the search prevents fruitless time spent exploring areas of the tree known not to contain a new optimum while still exhaustively searching all viable paths.

Beginning with the first embedding, we again add edges to the partial graph. This time, though, we know that each added edge must result in a crossing. Each level further down into the tree we go without finding our terminal vertex represents yet another crossing. As we progress downward, decisions must be made about which region and edge to cross at each level. Once the requirements of a good drawing are satisfied, a branch is created for every available region and edge combination we could cross on our journey to the desired vertex. For each region and exit edge, we create a new cross vertex at the point of the crossing. This allows the partial graph to remain planar, although with one vertex that is not a part of the graph, and one edge that is actually only an edge segment.

After the cross vertex is created, we check to see if we can connect from it to the terminal vertex directly. If so, we have reached a leaf. We count the number of levels, representing new cross vertices, and this value is the cost of reaching that vertex by that path. If more than one vertex needs to be connected to complete this partial graph, the cost is summed for each path and a starting optimum, or minimal crossing number is established. This becomes the bound as we move on to explore the next partial graph.

Once the tree has been exhaustively searched, we have determined the overall minimal crossing number for the graph. There may be one or more embeddings that result in that crossing number; discovering which of these are isomorphic, and therefore not unique is a problem we will address below.

Again, an example serves to clarify the process. This example, taken from [11], uses a K_5 graph, the first complete graph that has a crossing number greater than zero. This simple example has only one starting partial graph, shown in Figure 2.3. It is the result of connecting edges while confirming after each addition that the graph is still planar with Euler's formula.

When we attempt to connect the final edge, from i to j, Euler's formula tells us that the graph can no longer be embedded in the plane; a crossing has occurred. We backtrack and begin the depth-first search. Note that although other partial graphs

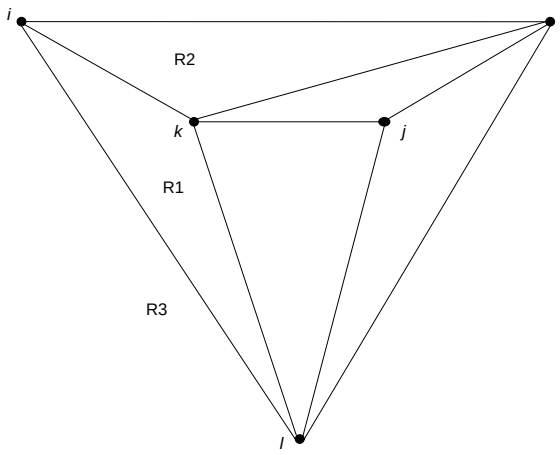


Figure 2.3: Partial Planar K_5 Graph

could be drawn with differently numbered vertices remaining to be connected, they are all isomorphic to this one.

Our next step is to connect all remaining edges, which in this example consists of only one edge, i to j. Looking at Figure 2.3, we see three possible regions to cross to which vertex i is adjacent, namely regions R1, R2, and R3. We begin by selecting R1. R1 has 3 edges but the constraints of drawing a good graph prevent us from crossing two of them, specifically the two incident with i. Thus edge kl is selected, a cross vertex is created on this edge, and an edge segment is placed connecting the cross vertex with i (Figure 2.4a). Next, we check whether the new cross vertex can be connected with j while the graph remains planar. In this instance, it can and so the edge segment is laid down and the edge from i to j is complete with one crossing (Figure 2.4b). As there are no more vertices to connect, the graph is complete with a crossing number of 1. The algorithm then backtracks and tests laying down an edge through each of the other two regions. Although it finds two more ways to draw the edge, it does not improve upon the crossing number of 1.

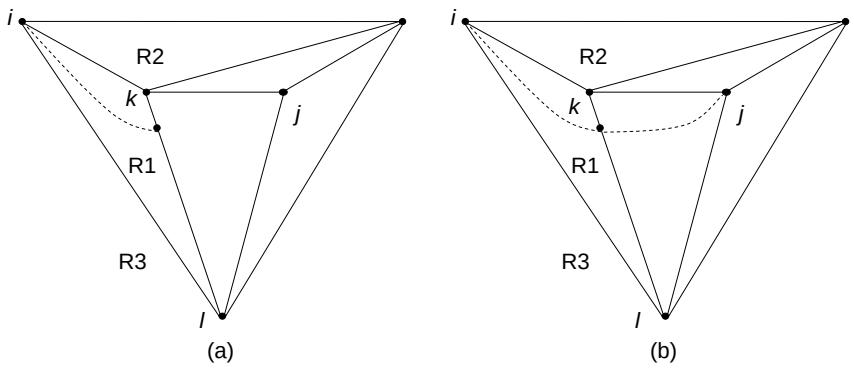


Figure 2.4: Completing Final Edge of K_5

2.4 Optimization of Harris and Harris

2.4.1 Parallel Implementation

Exhaustive search algorithms can become computationally expensive as the search space grows. The Harris and Harris algorithm as presented above has only the branch and bound limitation in place, so the next obvious step in its evolution was to improve performance by further optimization.

The first method chosen was to implement the algorithm in parallel, reported by Tadjiev and Harris [18, 19]. They reconfigured the algorithm to run on a network of Pentium 133 machines running Linux, linked with PVM in a host-node format. For simplicity, they chose to do a static partitioning of the search tree among the processors. A static system has a data set that is completely defined prior to run-time that can be evenly divided among the processors. Although the process ran faster in parallel for all graph families tested, meaningful improvement in performance required at minimum a K_8. As one might anticipate, speedup was improved by increasing the number of processors. Comparable results were seen with a Solaris/MPI system.

2.4.2 Load-Balanced Parallel Processing with a Queuing System

It became apparent during the work described above that the processors were not finishing concurrently, the main drawback of a statically partitioned data set. One can envision an unbalanced tree in which static partitioning does not result in an even work load distribution among processors. Furthermore, a heterogeneous network cluster, which is not uncommon, can result in load imbalance even when the work load is evenly distributed. The next step then was to divide the work efficiently among processors to ensure that no processor is sitting idle while others work. This problem was tackled by Yuan, et al. using a generic work queuing system to provide dynamic load balancing [21, 22].

Work queues can be implemented in various ways, each with its own pros and cons. At one extreme is a central work queue managed by a master processor that distributes jobs to slave processors as they finish working and request them. This approach suffers from the delays inherent in having the entire system managed by a single processor. Bottlenecks can occur if multiple slaves request jobs at once, and the network load is heavy with job and control message passing. At the other extreme is the decentralized system, in which local processors manage their own work queues. The downside of this approach is its similarity to static partitioning.

The system created by Yuan et al. is a balance between these two extremes. A master processor creates the jobs, defined as the smallest possible unit of work, and places them on a central queue. A job is dispatched to each processor, which places the job into its own local queue. In the course of performing its assigned work, the slave creates additional jobs that are added to its local queue. Further message passing occurs when a slave works through its job queue and becomes idle, as it requests more work from the central queue. Alternatively, if the local queue overflows, a user-defined parameter, as jobs are created and added to it, the slave sends work back to the central queue for redistribution. The primary downside to this approach, not surprisingly, is the amount of message passing network traffic it

creates.

Several interesting results were found running the crossing number problem in this parallel generic queuing system. Like a government bureaucracy, the more processors that are used in the network, the more jobs that are created and the higher the volume of network traffic, with the result being little or no improvement in run time over a smaller parallel system [21]. Nonetheless, with an optimal number of processors and optimal queue size, shorter run times were seen than without load balancing.

2.4.3 Graph Isomorphism

As per Definition 5, we describe two graphs as isomorphic if there exists a one-to-one mapping of their vertex sets that preserves adjacency. Numbering of vertices is an arbitrary construct that allows us to define regions, but has no bearing on the underlying graph structure. As one can imagine, larger graphs with many vertices, some native and some crossing, can be created from a myriad of paths, which results in graphs that not only have differently numbered vertices but that may look completely different from each other when drawn. Many of these graphs will in fact be isomorphic. Figure 2.5 demonstrates the difference between isomorphic and nonisomorphic graphs; Figures 2.5a and b are isomorphic to each other, but Figure 2.5c is not isomorphic to either a or b.

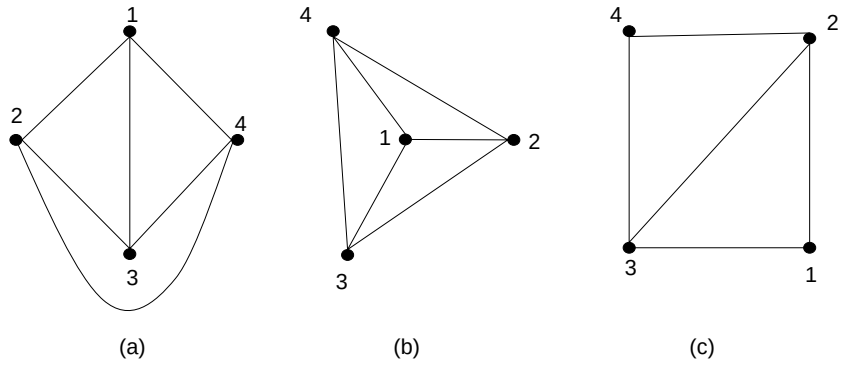


Figure 2.5: Comparison of Isomorphic and Nonisomorphic Graphs

The search space can be significantly reduced by selecting only one member of each isomorphic family to test for the next higher value of n. Table 2.1 displays the relationship between the total number of minimal graphs and the number of isomorphic families found for $5 \leq n \leq 10$.

n	5	6	7	8	9	10
Graphs generated	12	4	76	20	4,590	56,618
Iso families	1	1	5	3	1,453	5,679

Table 2.1: Relationship Between Total Minimal Graphs and Isomorphic Families

Determining graph isomorphism is an intensive process [4]. One of the best current methods uses a canonical labeling for each graph to be tested. Graph G is isomorphic to graph H if and only if the canonical label for G is identical to the canonical label for H. The most widely used algorithm for canonical labeling is called *NAUTY*, standing for "No AUTomorphisms, Yes?" [15] Written by Brendan McKay, the software package that implements the *NAUTY* algorithm has been modestly described by its author as the "world's fastest isomorphism testing program."

2.4.4 Region Restriction

A further optimization was implemented by Fredrickson et al. [5, 6]. They added another constraint to the definition of a good drawing: they specified that no region of the graph should be crossed more than once by a single edge.

As an example, consider the graphs in Figure 2.6 below. Figures 2.6a, b, and c meet the definition of a region restricted good drawing for placement of edge uv and would be generated under the new algorithm. Figure 2.6d does not meet the definition and would no longer be generated. Note that region restriction does not prevent non-minimal graphs from being drawn, but it does eliminate an entire class of graphs that cannot be minimal. Proof that restricting an edge from reentering a previously entered region does not prevent any possible minimal graphs from being generated can be found in [5].

To illustrate further, consider Figure 2.7. A branch of the search tree is created for

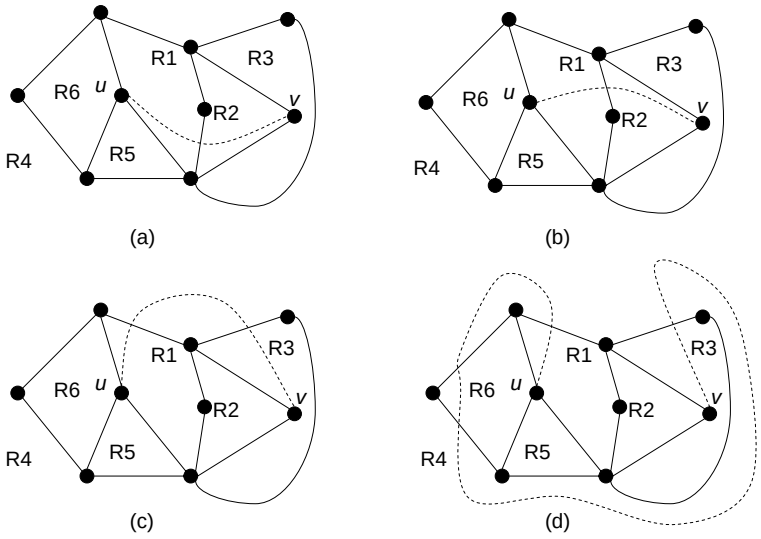


Figure 2.6: Region Restricted Good Drawings of Edge uv (a,b,c) and Not Good Drawing (d)

every possible path between u and v. Thus an edge will be begun or continued through every available region at each level, as seen for the starting regions in Figure 2.7a. Any edge entering a region adjacent to one it has already passed through after having crossed a subsequent region (Figure 2.7b) will have more crossings than an edge that entered that region directly (Figure 2.7c).

Fredrickson later summarized the optimization into a new definition of a good drawing of a graph [5]:

Definition 15 *Let*

$$f : u = u_0, e_1, u_1, e_2, \ldots, u_{k-1}, e_k, u_k = v$$

denote the $u - v$ path laid down in graph G where ei is an edge segment through one region of G. Let $H = G + f$. The graph H is a region restricted good graph if every region of H has at most two vertices of f on its boundary.

She then provided

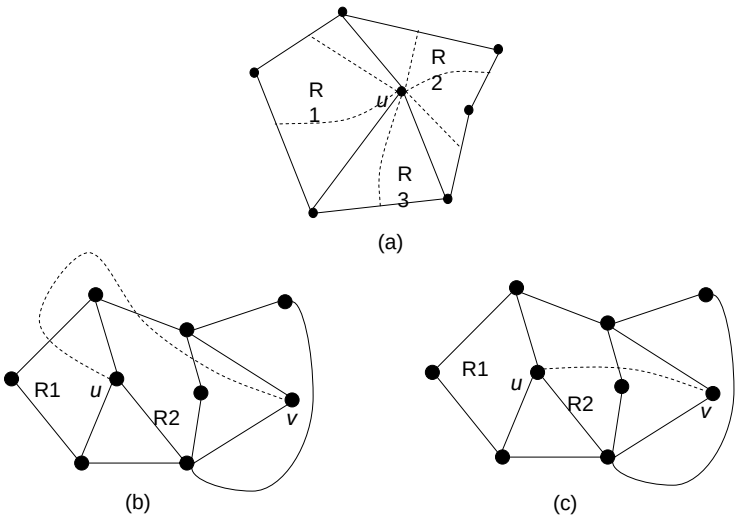


Figure 2.7: Possible Paths for Laying Down Edge uv

Theorem 3 *[Fredrickson] Employing the definition of a region restricted good drawing of a graph while creating minimal K_n from minimal K_{n-1} does not eliminate any possible minimal graphs from being generated.*

The authors of [5, 6, 21] found that the substantial reduction in search space resulting from tightening the definition of a good drawing (see Table 2.2) produced order of magnitude improvements in run time up to K_8. K_9 did not complete running after several weeks, and remained to be solved after further optimization. Additional processors resulted in further run time improvements.

Vertices (n)	$\nu(K_n)$	**Good Graph**	**Restricted Good Graph**
5	1	3	3
6	3	203	71
7	9	1,498,775	19,979
8	18	*	46,697,854

Table 2.2: Number of Jobs Required to Find $\nu(K_n)$ With and Without Region Restriction

2.4.5 Radical Region Restriction

One final optimization was introduced prior to the work to be presented here. A further restriction on placement of edges through regions, known as radical region restriction, was presented in [5]. Radical region restriction is based on preliminary analysis of the shortest path from u to v, in terms of the number of regions crossed. It determines, for each of the n - 1 edges that need to be created, the minimal number of regions that need to be crossed to complete the edge. The algorithm only follows branches of the tree that do not exceed the minimum. Logically, crossing the fewest number of regions results in the fewest number of edge crossings.

This refinement is illustrated in Figure 2.8. Four of the ten possible uv edge placements for the given graph are displayed. Only Figures 2.8a and b have paths

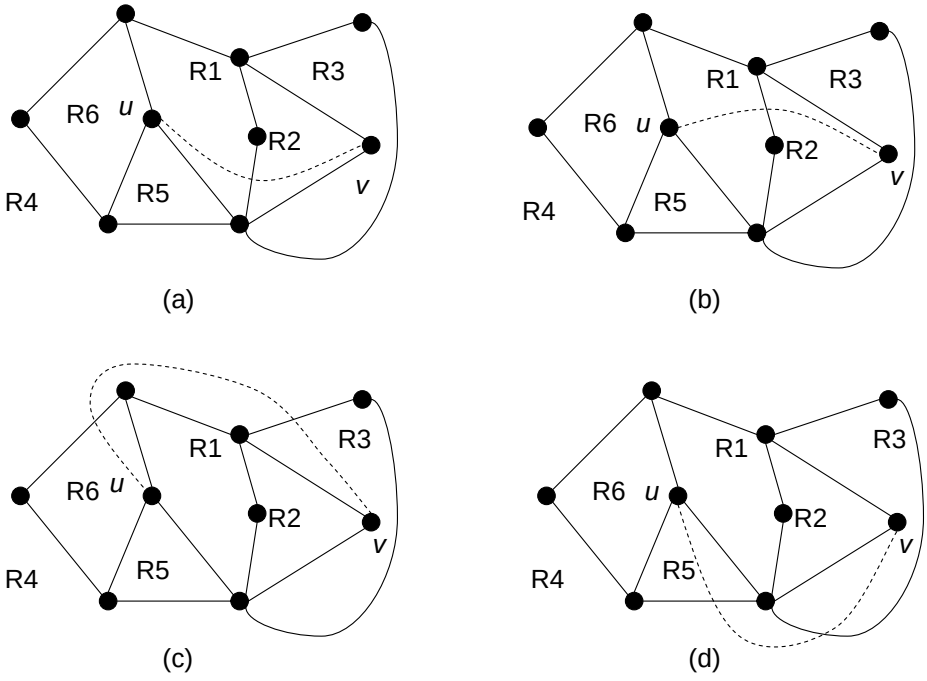


Figure 2.8: Options for Placement of Edge uv

that cross the minimum number of regions (two) and therefore have the fewest edge crossings (one). The other figures display paths that cross more regions (three), result in two edge crossings each, and could not contribute to a minimal graph. Radical

region restriction would prevent the creation of these paths.

When radical region restriction was used to generate minimal K_n graphs up to K_{10}, substantial reduction in running time was seen as compared with basic region restriction (Table 2.3) [5]. Even with radical region restriction, minimal K_{10} required approximately a week to generate owing to the large number of isomorphic K_9's it had to start with (1,453).

Vertices (n)	$\nu(K_n)$	**Region Restriction**	**Radical Region Restriction**
8	18	2 hr 36 min	5 min
9	36	>> 1 week	8 min
10	60	unknown	$\approx$ 1 week

Table 2.3: Runtime for Region Restriction *versus* Radical Region Restriction

2.5 Bipartite Graphs

Before presenting the current work, we will briefly digress and consider bipartite graphs. As mentioned earlier, the original description of the crossing number problem involved a complete bipartite rather than a complete graph. The complete bipartite graph representing Turán's Brick Factory Problem can be visualized as in Figure 2.9, adapted from Richter and Thomassen [16], where vertices 1, 2, and 3 represent the brick ovens, and are the first partite set, and vertices 4, 5, 6, and 7, the second partite set, represent the yards where the bricks were stacked for storage.

This representation can be redrawn as a minimal graph (Figure 2.10) that has a crossing number of 2. It is clear from these two representations of $K_{3,4}$ that the specific drawing in the plane is critical with respect to crossing number, and may not be immediately obvious, even for small m, n.

Bipartite graphs are useful in VLSI routing simulations [17], and might be of value in any situation where interconnections occur between two groups of nodes, but not within the groups, such as electrical wiring for example. A discussion of the relations between the crossing numbers of complete graphs and complete bipartite

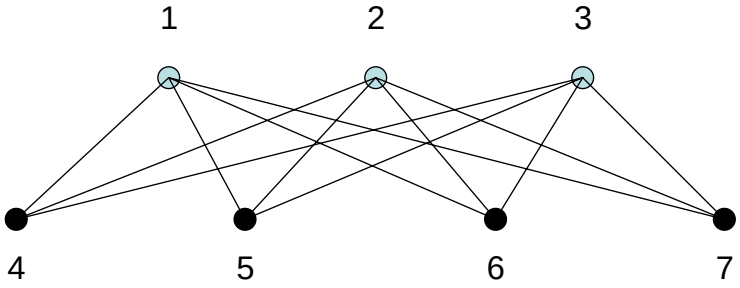


Figure 2.9: Complete Bipartite Graph $K_{3,4}$ Illustrating Turán's Brick Factory Problem

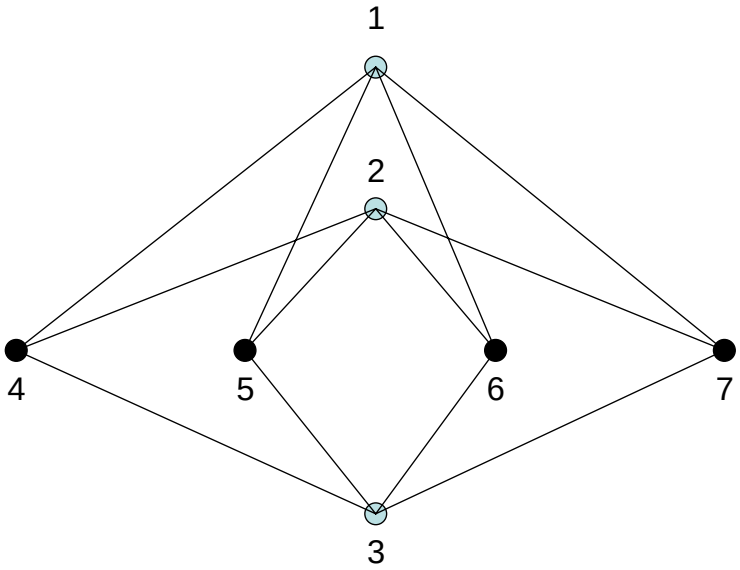


Figure 2.10: Minimal Complete Bipartite Graph $K_{3,4}$

graphs can be found in [16].

There are several interesting properties of bipartite graphs, summarized by Woodall in his work confirming Zarankiewicz's conjecture for m = 7, $n \leq 10$ [20]. These are given in the following theorems:

Theorem 4 *[Woodall] Every $G = K_{m,n}$ with $cr(G) = k$ contains a drawing of $K_{m-1,n}$ with a number of crossings $cr \leq k(m-2)/m$.*

Theorem 5 *[Woodall] If m is even and the Zarankiewicz conjecture holds for $K_{m-1,n}$, then it holds for $K_{m,n}$.*

Theorem 5 allows us to limit our consideration to graphs where both m and n are odd.

Minimizing the crossing number of a complete bipartite graph is a sister problem to that of the crossing number of a complete graph. It is desirable that any solution to the crossing number problem be applicable to the bipartite family of graphs as well. Preferably, a general solution applicable to many graph family problems could be found.

Chapter 3

Star Analysis

3.1 Introduction

All the refinements introduced up to this point resulted in a system that quickly finds minimal graphs up to K_9, and finds K_{10} minimal graphs in a runtime of approximately one week. Above K_{10}, the runtimes are anticipated to increase drastically. Generation of minimal K_{10} requires that the process be run on fourteen hundred and fifty-three different (i.e. nonisomorphic) minimal K_9 graphs, each of which contains sixty-five regions. Generation of minimal K_{11} begins with five thousand six hundred and seventy-nine nonisomorphic minimal K_{10} graphs, each with 97 regions. So the quest for optimization of the process continues.

A brief recap of the current state of the algorithm will facilitate the discussions that follow. Minimal K_n graphs are created from representatives of each isomorphic family of K_{n-1}, beginning with K_5. As minimal K_5 only has one isomorphic family, only a single so-called "feeder" graph is needed to progress to minimal K_6. The algorithm begins by taking the feeder graph and numbering its vertices. Native vertices, those that are in fact in the graph's vertex set, are numbered from one to five. Number six is reserved for the new vertex to be added, and number seven is used to identify the single crossing vertex ($\nu(K_5) = 1$).

The next step is to place vertex six into each of the eight regions of minimal K_5. This creates eight starting partial graphs, one of which is shown in Figure 3.1.

For each, a preprocessing step makes connections between vertex six and the

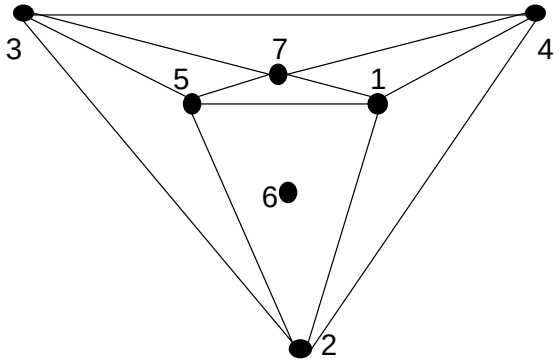


Figure 3.1: K_5 to K_6: K_5 Graph With New Vertex 6 Placed

native vertices bounding the region in which it was placed; in other words, any connections that can be made without new crossings (Figure 3.2). Any connections not

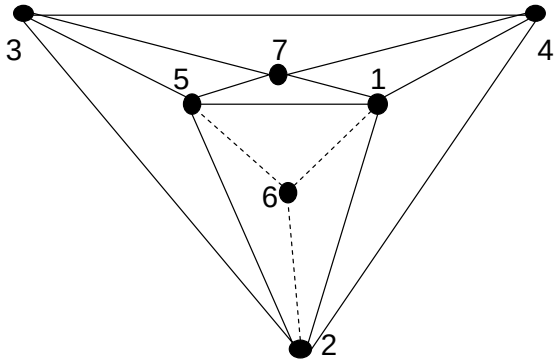


Figure 3.2: K_5 to K_6: Initial Connections

able to be completed directly are placed on an edges-to-add list. The partial graphs are then fed into the parallel processing system.

The parallel system begins by reading in all the partial graphs and edges-to-add lists. The master processor packages each pair of one graph and its associated list into a "job" packet and enqueues it on the master queue. It then distributes jobs to each of the slaves via MPI. Edges are created one segment at a time, where an edge segment is that portion of an edge that crosses a single region. All possible region passages,

within the bounds of the radical region restriction definition of a good drawing, are tested. When an edge segment is laid down, it terminates on one of the edges of the region it has crossed, creating a new crossing vertex. After creating the segment and crossing vertex, the processor looks for the terminal connecting vertex, based on the edges-to-add list, in the new region across the crossing segment. Only the specific vertex being sought by that job will be connected to complete the edge. If that vertex is not found in the boundary of the new region, the processor packages the new partial graph into a job packet and places it on its own work queue. If a region has, say, three legal region-bounding edges to cross, three new jobs will be created and enqueued, each with one new edge segment and one new crossing vertex, as a result of following each path. Thus as multiple segments are placed, the number of job packets in the work queue grows. If the number exceeds a user-defined limit, excess packets are sent back to the master for redistribution. When all job queues are empty and all edges complete, the process terminates. The completed K_6 graph with (a) and without (b) crossing vertices labeled is shown in Figure 3.3.

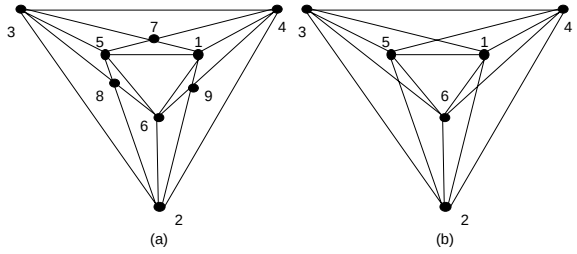


Figure 3.3: K_5 to K_6: K_6 Complete

A key feature of the process is a preliminary step that provides the bound for the search. As described in Section 2.4.5, we want to take only those paths that are less than or equal to the length of the shortest possible path across the regions to each vertex. This number is determined by a scan of the graph before any edge segments are laid down. Nonetheless, all possible graphs are then built up to the point at which the minimum path length is exceeded. Building and then discarding all these nonminimal partial graphs generates a tremendous amount of extra, unnecessary

processing, enqueuing, dequeuing, and message passing. We reasoned that the process could be expedited by not building any graphs at all, but rather by simply identifying the shortest paths to each vertex. By saving all paths, we would have the added advantage of identifying possibly important nonminimal paths for further study. We could then selectively choose which graphs to build based on criteria we could change. Eliminating all the laying down of edge segments also allows us to dispense with the parallel system as it currently exists, along with its massive overhead.

3.2 Overview of Algorithm

The process of Star Analysis begins similarly to its predecessor. Representative graphs from each isomorphic family of K_{n-1} are taken by a preprocessing program and prepared for evaluation by vertex numbering and inserting a new vertex, n, into each region of each representative graph. As opposed to the previous approach, direct connections are not made at this time, rather all edges are placed on the edges-to-add list.

The second program in the Star Analysis module is run once for each feeder graph by a calling script. In each iteration, Star Analysis receives a graph in the form of a Region List (described below), the edges-to-add list, and the region number in which the new vertex should be placed. It examines all possible paths to connect the new vertex to each of the existing ones, placing the paths in a Path List. The completed Path List with all paths is saved to a file. Paths are created independently of each other and in no particular order.

A third program in the module generates the actual graphs. Again, a calling script is employed, which can be varied to select minimal paths only, minimal-plus-one paths, or any desired combination. Graphs resulting from each of the regional vertex placements that meet the chosen criteria are created. The graphs are again represented in the form of region lists, which are saved into files.

The next step is a program that takes the Region List representations of the graphs and converts them into adjacency matrices. Adjacency matrices are required

by the *NAUTY* program, which determines isomorphism among them and sorts them into isomorphic families. The final step is to pluck a representative of each isomorphic family from the sorted graphs and create files for the K_n to K_{n+1} run.

3.3 Data Structures

3.3.1 Region List

The Region List class is adapted from that used previously by [21], modified to function in the current environment. The Region List is constructed as a list of lists, where each sublist represents a single region of a graph. Individual regions are defined by the vertices along their boundaries. The Region List class provides a wealth of functionality for dealing with a region list. It facilitates adding and deleting regions or vertices within regions, examining regions for the presence of specific vertices, adding edges to a graph and updating the adjoining regions, and searching for regions based on a variety of parameters. An example of a starting K_5 graph with its Region List representation is shown in Figure 3.4

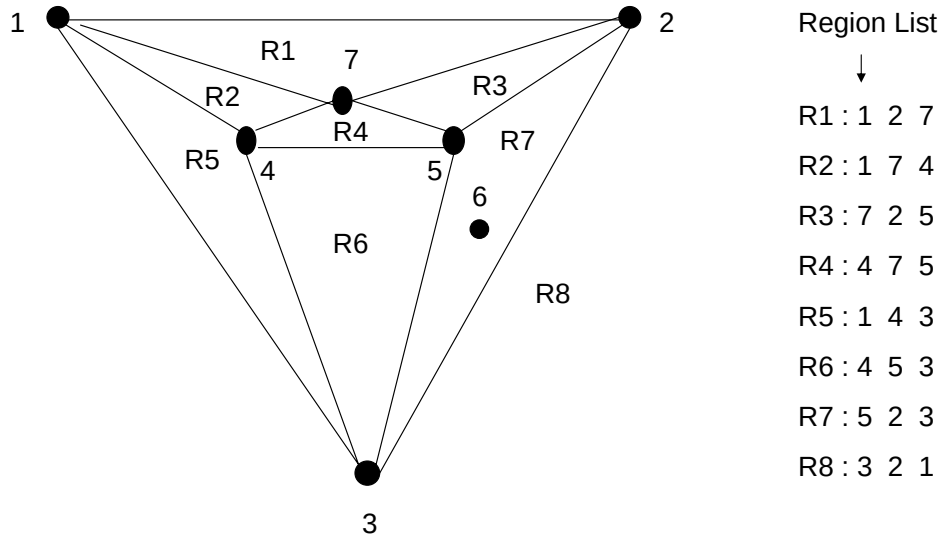


Figure 3.4: A K_5 Graph with New Vertex 6 in Region 7 and its Associated Region List

3.3.2 Distance Tree

The program that analyzes the graphs for shortest paths is based on an object-oriented structure we have entitled a Distance Tree. A Distance Tree class object is a full tree; that is, each level of the tree is complete, and it is built breadth-first.

The tree is constructed of TreeNode class objects. A TreeNode object consists of five data members: a pair of vertices, representing an edge segment of the graph; a region number, representing the region that will be entered upon crossing the aforementioned edge; a vertex number, representing an end vertex if one is found in the region; a pointer that points to the parent node for back tracing when the path is complete; and an array of pointers pointing to nodes at the next level of the tree.

The root of the tree represents the newly placed vertex; its TreeNode, at Level 0, has edge values {0,0}, a region value representing the region it is in, an initial vertex number of 0, a NULL parent pointer and a list-of-pointers pointer for creating the downward branches of the tree, as yet unassigned (Refer to Figure 3.5). Once this node is created, the program reads through the edges-to-add list to see if any edges can be completed without crossings. Any such edges are completed by updating the vertex value in the TreeNode to the number of the edge end vertex. The node is then used as a template to create a PathNode to be added to the Path List, described below. At the root level, the path to be added to the Path List as a sublist consists of a single PathNode. Any completed edges are removed from the edges-to-add list, and the program checks whether edges remain to be added. At the root level, there will always be additional edges to add since no crossings have occurred yet, so we proceed by adding the current region to a restricted region list and dynamically allocating memory for TreeNodes at the next level. TreeNode pointers point to new branches for each exit edge from the region, since all exits are legal from the root region.

At the next tree level, Level 1, a node is created for each new TreeNode pointer, as seen in Figure 3.5. These are initialized with the edge that was crossed leaving the starting region, the new region that was entered, an initial vertex number of 0,

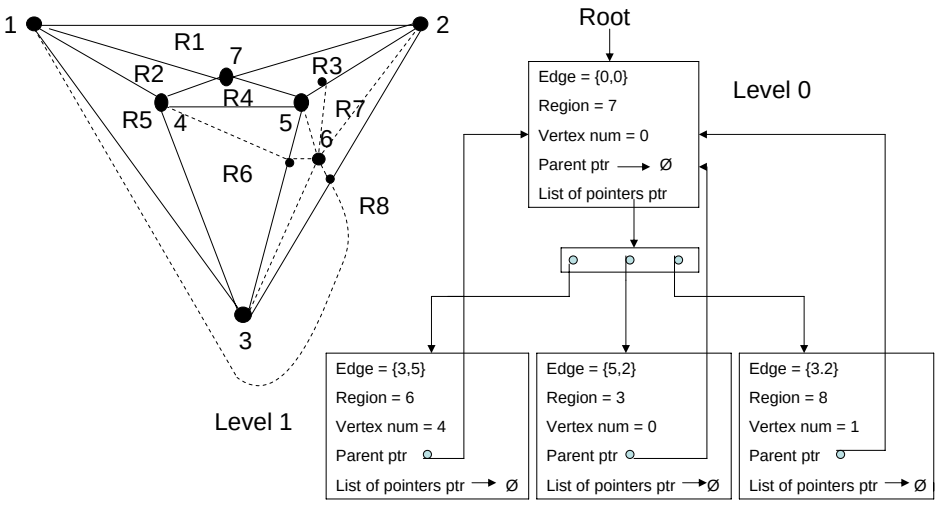


Figure 3.5: Snapshot of a K_5 to K_6 Distance Tree After Creating Level 1

a parent pointer pointing to the root node, and a list-of-pointers pointer as before. Once again we check for edges that can be completed, moving from node to node at the new level and scanning the edges-to-add list. One important difference, beginning with this level and continuing with all others, is that edges are not removed from the edges-to-add list until all nodes on the level have been visited. This is because all shortest paths to the vertices must be obtained, and all paths on the same level will have the same number of crossings, so no node further across on the level should be prevented from making a connection to a terminal vertex just because an earlier one already connected to it. If an edge can be completed, we create a path to attach to the Path List consisting of two PathNodes linked together; one based on the TreeNode at this level, and one based on its parent, obtained by following the parent pointer. At the end of the level, when all nodes have been visited, the edges-to-add list is updated. If the edges-to-add list is not empty, all the regions on the current level are added to the restricted region list, and nodes are dynamically allocated at the next level. New nodes are created for all edges that can be legally crossed from each current region, which are the ones that lead to regions not found on the restricted regions list.

This process continues, with new levels and nodes being created, until the edges-to-add list is empty, and all paths have been transferred to the Path List. A final point is that the restricted region list is not updated until after each level is complete, since it is both possible and legal for two edges to enter the same region at the same level from different directions. A potential problem arises if these edges cross before exiting the region, which will be dealt with when the graphs are constructed. Restricting paths from entering regions that have been entered at a higher level of the tree does not result in loss of any minimal paths, since all possible exits from that region have already been explored at the higher level; repeating those paths further down the tree only leads to longer paths with more crossings. Distance Tree region restriction can thus be defined as follows:

Definition 16 *Given a Distance Tree as described, a* ***Distance Tree Region Restriction*** *states that once one complete level of a Distance Tree is constructed, no region on that level may be visited by any branch of the Distance Tree at any later level.*

which then leads us to a new theorem:

Theorem 6 *In generating a Distance Tree for minimal K_n from minimal K_{n-1} as described, the use of Distance Tree Region Restriction does not eliminate any possible minimal graphs from being generated.*

The minimal crossing number for this graph is calculated starting with $\nu(\mathrm{K}_{n-1})$, which is passed to the program as a parameter. Each time a path is completed through the tree and traced back for transfer to the Path List, the number of levels is counted and cumulatively added to the starting crossing number (for n - 1), since each level represents one crossing. When all paths have been created, $\nu(\mathrm{K}_n)$ is the cumulative result. Note that with respect to crossing number, it doesn't matter how many times a vertex is found on a given level; each vertex only has a path to it counted once, and after being removed from the edges-to-add list, will never be sought at a lower level.

3.3.3 Path List

As mentioned, a Path List is composed of PathNodes, which are created based on corresponding TreeNodes as the path is traced from leaf to root. The Path List is actually a list of lists, where each individual sublist represents one path from the new vertex to an original native vertex.

A PathNode class object consists of three data members: the edge crossed by this segment of the path, the region crossed by this segment of the path, and the terminal vertex, if any, found in this region. All PathNodes in a single path will have a terminal vertex of 0 except the last one. When linked together into a sublist, this provides the entire path for the edge from the new vertex n to an original vertex i.

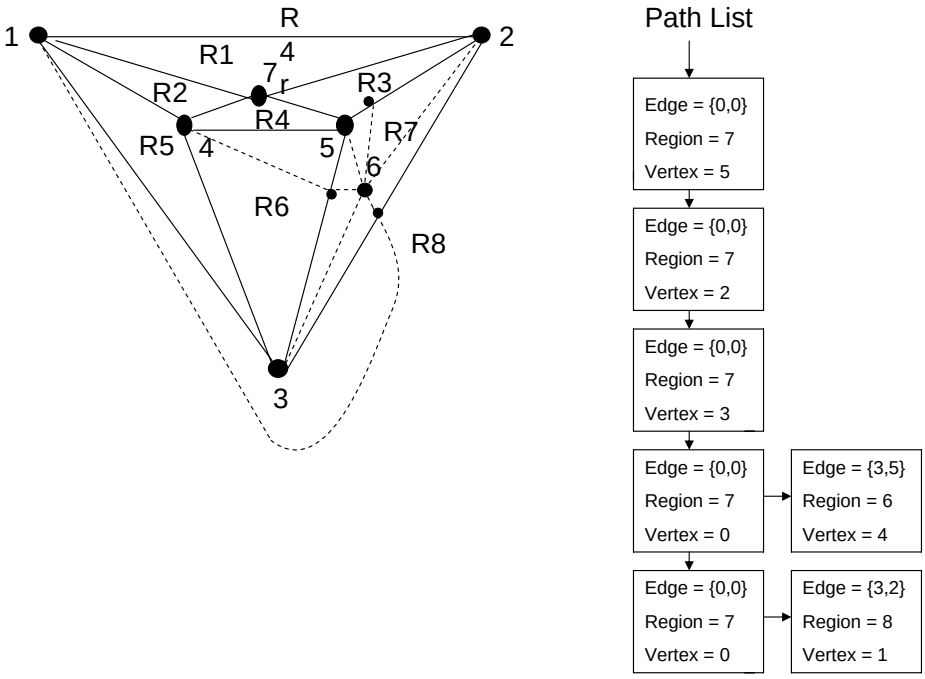


Figure 3.6: A K_5 Graph with Completed Path List

When complete, the Path List contains all of the sublists for all of the paths from n to i (Figure 3.6). Prior to saving them to a file, the sublists are sorted so that all the paths to a single vertex, i, are listed together, and the sublists proceed from $i = 1$ to $i = n - 1$.

3.3.4 K_n Graph

A K_n Graph is not a unique data structure, but simply an instance of a Region List. It is created by starting with the input K_{n-1} Region List and inserting a set of new paths from the Path List, one for each original vertex. The appearance of the new paths radiating out from the newly added vertex to various points of the graph in a star configuration is what led to the Star Analysis moniker (Figure 3.7). Graphs

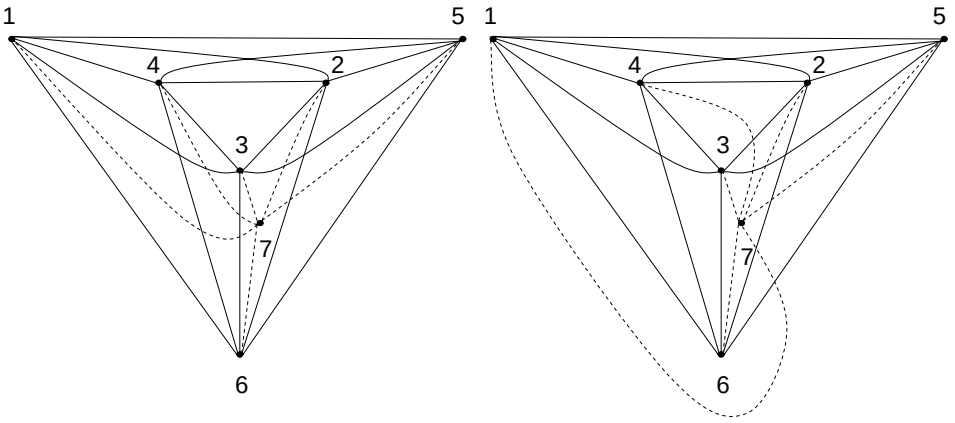


Figure 3.7: K_7 Path Overlays on Minimal K_6 Demonstrating Star Configuration

can be built from all combinations of minimal paths or from some other subset of the Path List.

Paths are added one by one in the reverse order from which they were created, starting from the terminal vertex and proceeding back to the newly inserted vertex. Edge segments are laid down sequentially in the order they occur in the Path List, with the graph regions updated after each addition. Each update requires that the crossed region be split into two new regions by the newly laid segment, and the graph is amended to reflect this. Since these region updates create a partial graph that may not match the Path List of paths that are added subsequently, two look-up tables are maintained to associate old edges and old regions with new ones.

When one path from each of the $n-1$ vertices to the *nth* has been added, the resulting graph depicts K_n with crossing vertices still in place to keep it planar. For

purposes of drawing the actual K_n Graph, the crossing vertex numbers are removed. Completed graphs are saved to a file for further processing into isomorphic families.

We mentioned earlier one potential problem that can arise when paths are laid down totally independent of one another. That is the situation where two independent edges enter a region and make an illegal crossing midregion before exiting in different directions (Figure 3.8).

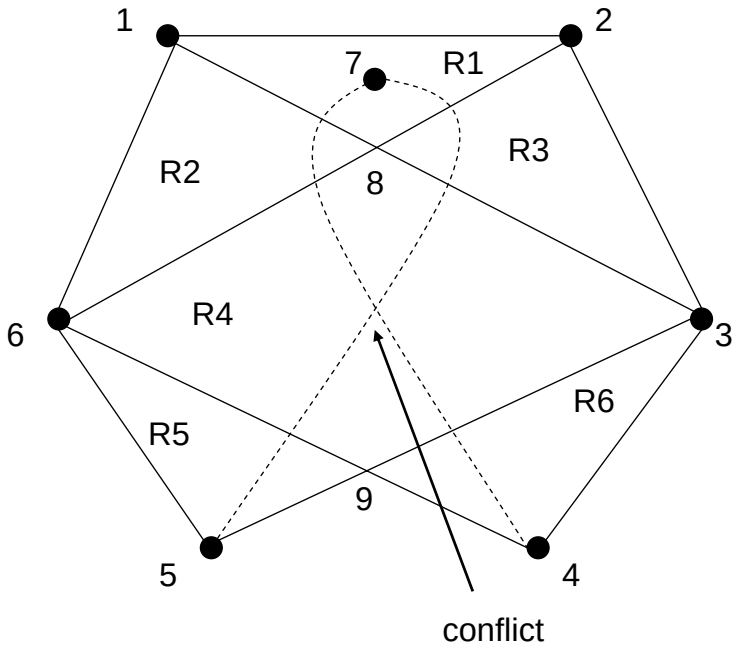
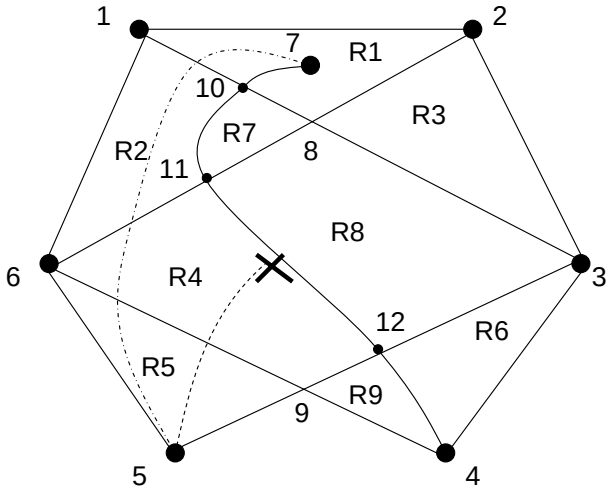


Figure 3.8: Demonstration of a Conflict Between Two Independently Constructed Paths

The region and edge look-up tables prevent these graphs from being created, as seen in the following example. When edge $\{4, 7\}$ is created (Figure 3.9, solid line), the graph's Region List is updated to reflect new regions R7, R8, and R9. The region look-up table adds R7 as a subset of R2, R8 as a subset of R4 and R9 as a subset of R6. Similarly, the edge look-up table adds the entries shown to keep track of the new legal subsets of edges that were crossed. When the program attempts to lay down edge $\{5, 7\}$ by the second path (Figure 3.9, dashed line), it encounters edge {12,11}, which is not a legal subset of the expected edge in its PathNode, {3,8}. Therefore,

the graph is thrown out as an illegal graph. The other alternative path from 5 to 7 (Figure 3.9, dash-dot line), is legal because it finds edge {11,6} as a subset of {8,6} and edge {1,10} as a subset of {1,8}. For the purpose of defining an edge, the order in which the vertices are specified is not critical. In this example, the legal edge passes through regions whose names designations have not changed, so the region look-up table is not needed.



Look-up Tables
(after first new path laid down)

Edge	Legal Alternatives
{1,8}	{1,10}, {10,8}
{8,6}	{8,11}, {11,6}
{9,3}	{9,12}, {12,3}

Region	Legal Alternatives
R2	R7
R4	R8
R6	R9

Figure 3.9: Resolution of Path Conflict

3.4 Example: K_6 to K_7

We will now step through the process of building a K_7 graph from a minimal K_6 to illustrate the entire algorithm. We begin with an instance of a minimal K_6 graph with a new vertex 7 inserted into region R9 (Figure 3.10). This graph, encoded as a Region List, is passed to the Star Analysis program, along with an edges-to-add list, and the identity of the region where the new vertex should be placed, R9. The root TreeNode, Level 0, is created as shown in Figure 3.10, with values of {0,0} for the edge variable, since no edge was crossed; 9 for the region; 0 for the vertex initially; and the pointers.

The first step is to connect any vertices on the edges-to-add list that exist in the

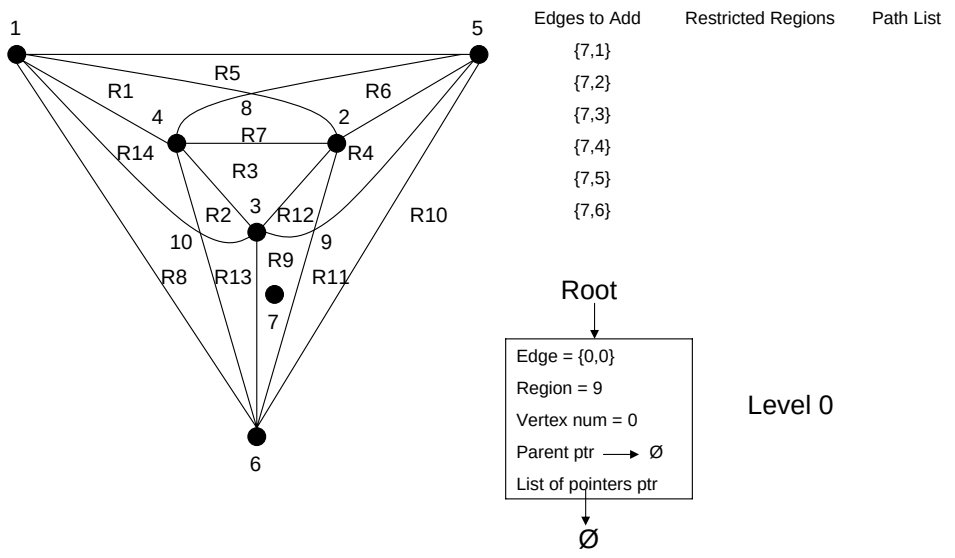


Figure 3.10: Starting to Grow K_7 From K_6, With Vertex 7 Placed and Edges-to-Add List

current region. Scanning the edges-to-add list, we find {7,3}. The vertex value in the TreeNode is updated to 3. A PathNode is created and added to the Path List, as a path of one step, containing the following values (copied from the TreeNode): {0, 0} for edge, 9 for region, and 3 for vertex (See Figure 3.11). The TreeNode vertex value is returned to 0, and we resume scanning the edges-to-add list, where we find {7,6}. The process of creating a PathNode is repeated, and it is added to the PathList (Figure 3.11). No further vertices are found on the edges-to-add list, so we remove edges {7,3} and {7,6}, and then allocate memory for TreeNodes at the next level, Level 1 (Figure 3.12). We are done with this level, so we place R9 on the restricted region list.

Region R9 has three edges on its boundary. Since there are no regions other than R9 on the restricted region list, we create nodes that cross each of these edges. The nodes are seen on Level 1 of Figure 3.12, with data values assigned. We inspect the remaining edges-to-add for any that can be added directly in any of the three regions. We find vertex 2 in R12 first, and update the vertex value to 2 in our TreeNode. We create a new path on the Path List, consisting of two PathNodes

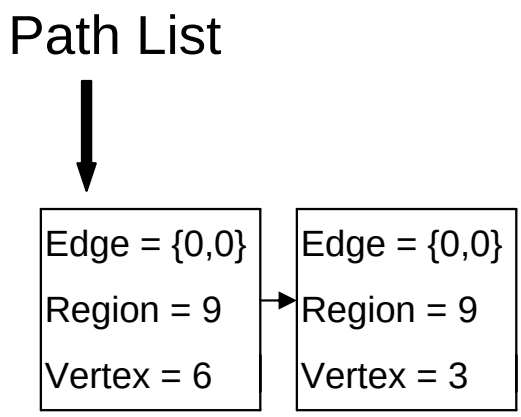


Figure 3.11: Path List at the End of Level 0

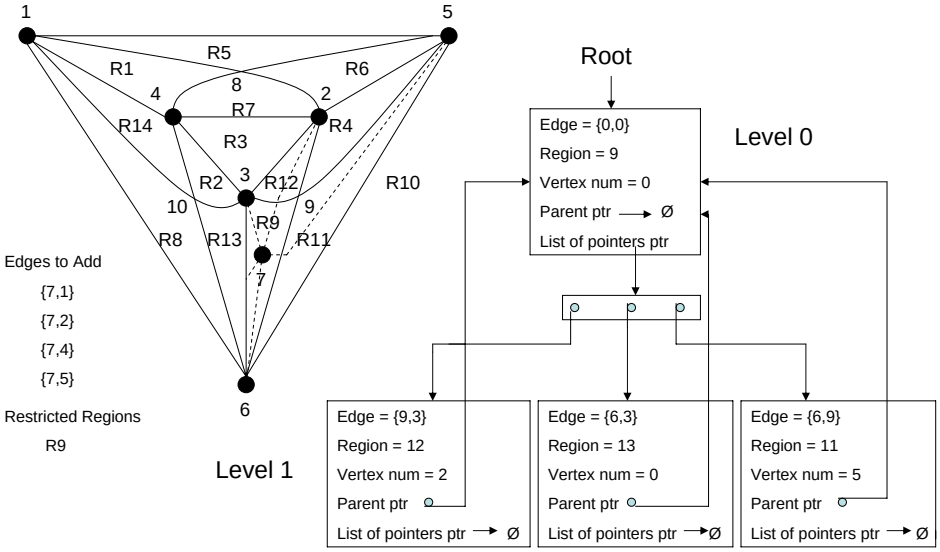


Figure 3.12: Continuing to Grow K_7 From K_6

as shown in Figure 3.13 by tracing backwards using the parent pointer. Moving on across the level, we find vertex 5 in R11. Another path is created (Figure 3.13). No further paths can be completed on this level, so we remove {7,5} and {7,2} from our edges-to-add list and add regions R11, R12, and R13 to the restricted regions list.

The next level of nodes, Level 2, is created, shown in Figure 3.14. For clarity, only TreeNodes that result in vertex connections are shown. Since all remaining vertices are connected on this level, none of the TreeNodes not shown here lead on further

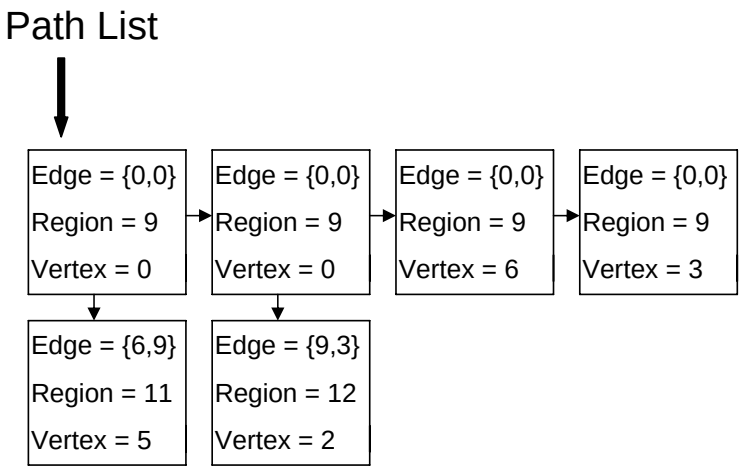


Figure 3.13: Path List at the End of Level 1

down the tree.

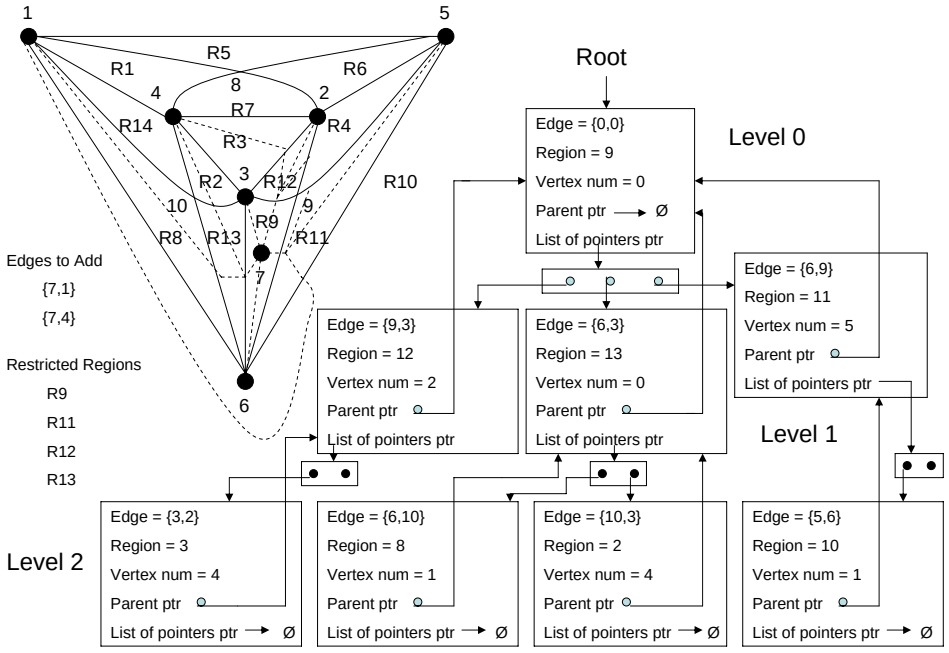


Figure 3.14: Completed Distance Tree for Vertex Placement in Region 9

Scanning across the row, we see that vertices 1 and 4 are both encountered on this level via two different paths each. PathNodes are created and the paths are traced back for all four of these terminations. At the end of the level, edges {7,1}

and {7,4} are removed from the edges-to-add list, which is now empty. The final step is to sort the Path List so that the vertices are in order, shown in Figure 3.15, and save the Path List to a file.

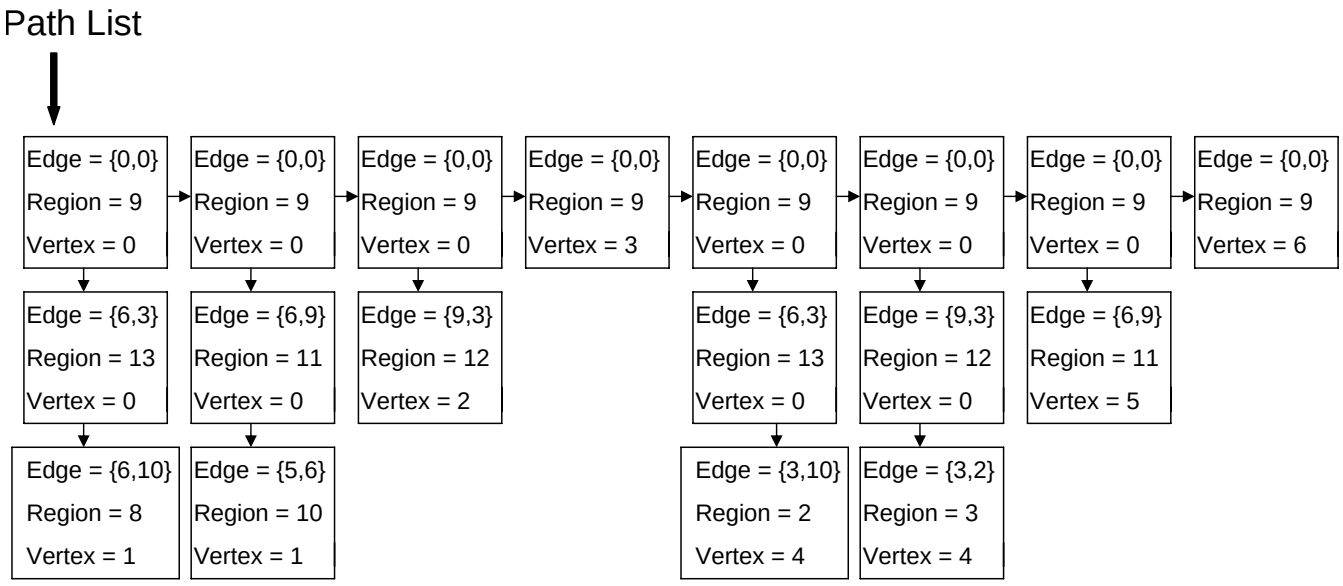


Figure 3.15: Completed Path List for New Vertex Placement in Region 9

After all the input graphs have been evaluated, one for each region assignment of the new vertex from each isofamily, and all the Path Lists have been saved, we can begin to build graphs. Let's assume we have reopened the Path List generated above, along with the starting K_6 graph from Figure 3.10. Our first graph will be created by selecting one edge for each terminating vertex, in this case the first, third, fourth, fifth, seventh and eighth paths. The program is currently written to cycle through all combinations of shortest paths to create a graph for each combination, but for this example, one will suffice. Each edge back to vertex 7 is created, segment by segment. After each addition, the Region List representing the Graph is updated to reflect the new region boundaries, and the look-up tables are updated to reflect the region and edge divisions. When all edges have been added, the Region List/Graph looks like Figure 3.16. The crossing vertices are left numbered to keep the graph planar and provide region boundaries. It is again saved, to be sent onwards to *NAUTY* for

isomorphic family assignment.

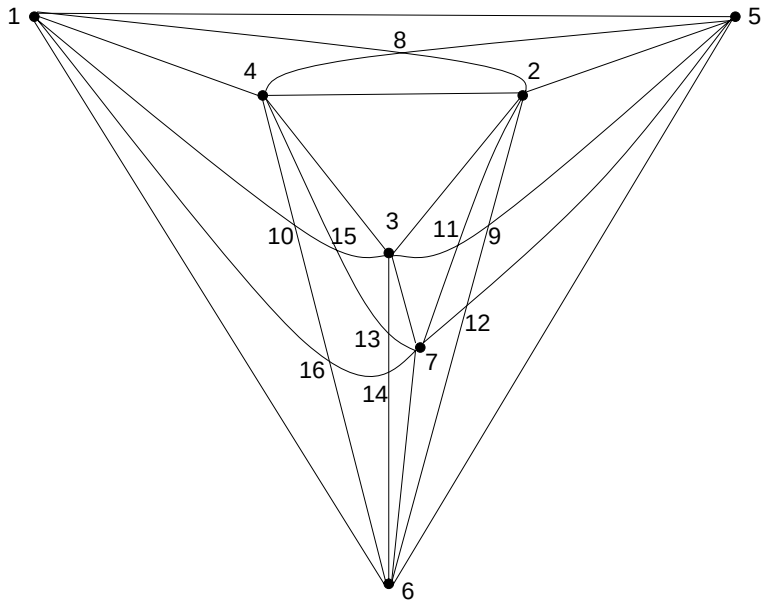


Figure 3.16: Completed K_7 Graph With Crossing Vertices Labeled

Chapter 4

Results

4.1 Complete Graphs

The Star Analysis process represents a total departure from the exhaustive search method. Beginning with Harris and Harris and progressing up to this point, each improvement built upon the work that had gone before without changing the underlying algorithm, apart from optimizing the search space. With Star Analysis, we started from the ground up, so our first concern was assuring that the process worked correctly. Therefore, we began our Star Analysis test runs on the complete K_n graphs generated previously by the processes described in Chapter 2. Recall that graphs up to $n = 9$ were generated by the parallel processing method with basic region restriction. K_{10} graphs were generated with radical region restriction, the most optimized system prior to Star Analysis. By repeating the process using the Star Analysis module, we would be able to confirm that the same number of total graphs, regions per graph, and isomorphic families were being generated. We also of course confirmed that the same crossing numbers were obtained (Table 4.1). "Region Restriction" in the table refers to either basic or radical, depending on data availability.

All values in Table 4.1 are exact, except for the value for K_{11} total graphs, which was estimated as follows. All possible legal graphs were built for the first ten isofamilies, then the number of resulting graphs was compared with the number of possible path combinations from which they were created. Recall that not all path combinations lead to legal graphs, and the number of illegal paths goes up rapidly as the

n	5	6	7	8	9	10	11
Total Graphs							
Region Restriction	12	4	76	20	4,590	56,618	7.1B (est)
Star	12	4	76	20	4,590	56,658	*
Iso Families							
Region Restriction	1	1	5	3	1,453	5,679	*
Star	1	1	5	3	1,453	5,679	*
Number of Regions							
Region Restriction	8	14	25	40	65	97	*
Star	8	14	25	40	65	97	146
Crossing Number							
Region Restriction	1	3	9	18	36	60	*
Star	1	3	9	18	36	60	100

Table 4.1: Comparison of Results with Region Restriction and Star Analysis for K_n (* = did not complete running)

graph size increases. At the K_{11} level, approximately 72% of path combinations are legal. The total number of path combinations for all isofamilies was then determined. The total number of graphs was estimated to be the same percentage (72%) of the total path combinations as was found for the first ten isofamilies.

A minor discrepency occurred at K_{10}; the same number of isomorphic families and regions were found, but there was a slightly greater number of total graphs generated with Star Analysis than with Parallel/Region Restriction. This discrepency remains unresolved, but we are reviewing the graphs generated by Star Analysis to see where the extra graphs occur. This issue does not affect the progression from K_{10} to K_{11}, since all isomorphic families are available to provide feeder graphs, but should nonetheless be resolved.

Table 4.2 shows the timing results of these test runs as compared with the results from parallel processing/radical region restricton from [5]. Time approximations for Star Analysis are based on the processes of Path List generation, crossing number determination, and building all possible minimal grafts, with the exception of K_{11}, where graph building is not included. Radical Region Restriction times do not include running through isomorphic testing, so that portion of the time is excluded from Star

Analysis as well. Also bear in mind that Star Analysis is running on a single processor, whereas Radical Region Restriction is running in parallel on multiple processors.

n	8	9	10	11
Radical	5 min	8 min	1 week	*
Star	$<$ 1 sec	27 sec	108 min	62 hr

Table 4.2: Runtimes to Generate Minimal K_n
(* = did not complete running)

For the larger values of n, $NAUTY$ became the rate-limiting step in the Star Analysis process. We were able to generate the Path Lists and obtain the value for minimal crossing number relatively quickly. Table 4.3 shows the approximate times for all the processes in Table 4.2 plus isomorphic testing by $NAUTY$. $NAUTY$ is reputed to be the fastest iso-testing program available, but improvements in processing time here would be most welcome.

n	8	9	10	11
Runtime	3 sec	12 min	62 hr	pending

Table 4.3: Star Analysis Runtimes With Isomorphic Testing By $NAUTY$

We discovered one interesting and, to our knowledge, not previously reported phenomenon as we progressed from $n = 5$ to $n = 11$. In all cases when a K_{n-odd} graph was grown from the previous $K_{(n-1)-even}$, a minimal graph could be obtained with new vertex placement in every region. This was not the case for any instance of $K_{(n-1)-odd}$ to K_{n-even}. In other words, there were one or more regions in all $K_{(n-1)-odd}$ graphs that did not grow any minimal K_{n-even} graphs when the new vertex n was placed in them. We don't have an explanation for this phenomenon, but feel it merits further investigation.

Finally, we present the results from a test of the crossing number of a K_{11} graph. The crossing, $\nu(K_{11})$, of 100 verifies Guy's Conjecture for $n = 11$. A single instance of a minimal K_{11} graph is shown in Figure 4.1.

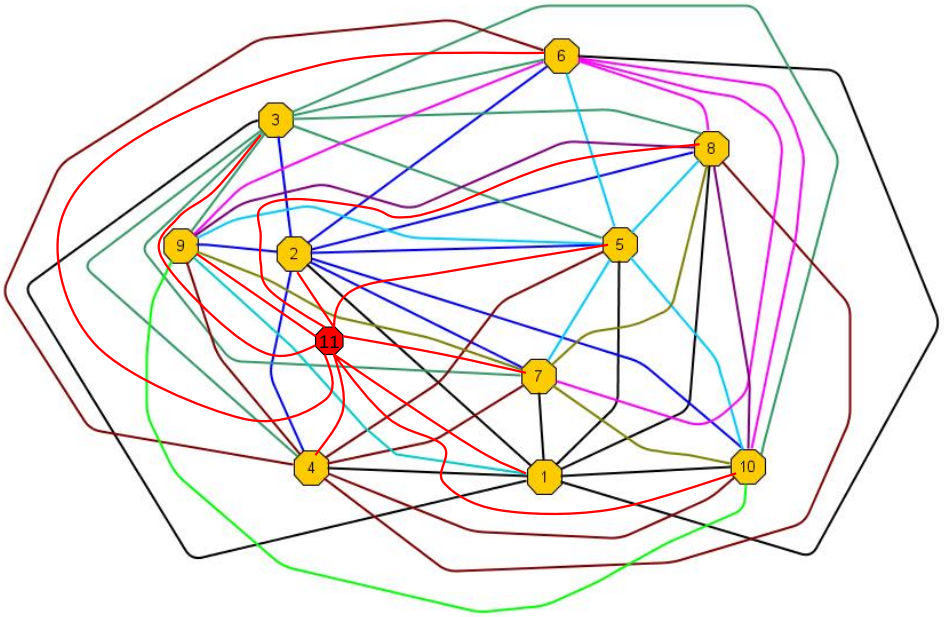


Figure 4.1: One Instance of a Minimal K_{11} Graph [J. Fredrickson [5]]

4.2 Complete Bipartite Graphs

One of our objectives in building the Star Analysis module was to create a system that could be generally applied to multiple graph families. From the perspective of Star Analysis, it is merely creating paths based on the Path List delivered to it. It neither knows nor cares whether those paths are generating a complete graph, a less-than-complete graph, or a graph of an entirely different type. In order to demonstrate proof of concept, our second series of tests were run to grow complete bipartite graphs and determine their crossing numbers.

One adjustment that was required to switch from complete graphs to complete bipartite graphs was that the program-calling scripts had to be revised in order to create, for example, edges-to-add lists that were specific to the new format. The process was run first for a family of bipartite graphs, $K_{3,3}$ through $K_{3,10}$. The starting $K_{3,2}$ graph was generated by hand and the starting input file was created manually. Thereafter, the scripts took over and ran each family member in succession, progress-

ing from $n = 4$ to $n = 10$. No modification of the actual Star Analysis program was required. In order to minimize running time for this trial, we did not generate all isomorphic families above K_8.

The results of the first test are shown in Table 4.4. All crossing numbers match those previously reported, as well as confirming Zarankiewicz' conjecture.

n	3	4	5	6	7	8	9	10
Crossing Number	1	2	4	6	9	12	16	20

Table 4.4: Crossing Numbers Found by Star Analysis for $K_{3,n}$

Interestingly, the phenomenon described for K_{n-even} growing to $K_{n+1-odd}$ graphs, in which minimal graphs were found with new vertex placement in every region, was also seen growing bipartite graphs. In the bipartite case, the situation was a bit more complicated. Given $K_{m,n}$, minimal graphs could be created when progressing from n-odd to n-even when m was held constant, irrespective of whether m was odd or even. In other words, for a family of graphs such as the $K_{3,n}$ family seen in Table 4.4, every other generation, specifically when n was odd, consisted of graphs in which new vertex placement in each region resulted in minimal graphs in the next generation.

Chapter 5

Conclusions and Future Work

5.1 Summary and Conclusions

We have presented a new approach to the generation of minimal graphs based on analysis of independent paths between vertices. Our process grows minimal K_n graphs from minimal K_{n-1} with a process we have entitled Star Analysis. Using minimal K_{n-1} as a template, independent shortest paths are created from newly added vertex n to each of the original $n-1$ vertices. Uncoupling the process of shortest path creation from graph drawing efficiently determines the minimal crossing number of the graph without the overhead of lengthy procedures to lay down edge segments. Graphs to be drawn can then be selected from the resulting Path List, and only desired graphs are drawn, as opposed to previous methods where all possible graphs were drawn until they exceeded the minimum. The result of eliminating the unnecessary partial graph drawing is a substantial reduction in processing time. All other optimizations with respect to restriction of regions through which paths are drawn are maintained in the current system and contribute to the improved results.

Prior to this work, the largest confirmed minimal crossing number was for complete graphs of order 10. We have confirmed that the minimal crossing number for K_{11} is 100, which satisfies Guy's conjecture. We are eager to pursue graphs of higher order to see if the conjecture holds for $n > 11$.

We have also shown that our Star Analysis process can be applied to other families of graphs. Specifically, we have used it to generate minimal bipartite graphs.

As in the complete graph family, these graphs are grown from minimal $K_{m,n-1}$ to minimal $K_{m,n}$. We have been able to confirm Zarankiewicz' conjecture for graphs as large as $K_{7,5}$ to date.

5.2 Future Work

We have come a long way since the original algorithm for finding the minimal crossing number of a complete graph was devised [11], but much work remains. Despite all optimizations, creating minimal graphs of large n remains a formidable challenge. Further optimization of the process is clearly required to progress on into uncharted territory.

The first obvious optimization is to return to parallel processing. We believe that with the current method, we can eliminate the queuing system and all its message passing overhead. The reason for this is that the Star Analysis module is well suited to static partitioning. Previous search methods either statically divide the search space among processors, resulting in unbalanced search trees and their corresponding unbalanced processor loads, or split creation of individual graphs among processors, resulting in huge overhead. Star Analysis allows Path Lists to be created for individual graphs independent of each other. Each processor can be assigned a single graph from one isomorphic family to process, and the processing time should be roughly the same. Using multiple processors should improve processing time sufficiently to progress to higher order graphs, $n > 11$.

A second optimization is suggested by the finding we described in Chapter 4 that when growing minimal K_n for odd values of n, minimal graphs can be created with new vertex placement in every region of K_{n-1}, whereas that is not the case creating minimal n even graphs. Further analysis of this phenomenon, if it persists with increasing n, might reveal a way to eliminate regions from consideration in placement of the new vertex, further reducing the search space. It is also possible that other features of graph substructure might lend themselves to similar optimizations.

Another avenue of exploration suggested by the above involves the case of graphs

that are locally but not globally minimal, in other words, graphs that are minimal based upon the chosen region for vertex placement, but that do not generate the global minimal crossing number. Further analysis of these graphs could provide insight into the substructural arrangements of minimal versus non-minimal graphs, perhaps providing further opportunities for search space reduction.

Finally, the question remains whether it is possible to generate minimal K_n from non-minimal K_{n-1}. All the minimal K_n graphs created by Star Analysis are grown from minimal K_{n-1}. It is theoretically possible that some minimal graphs are therefore not being generated by our system. One approach to this problem is to take all the minimal-plus-one K_{n-1} paths created by Star Analysis, use them to generate graphs, and then create a K_n Path List. We can then discover if any of the paths so generated achieve the previously determined minimal crossing number. It is our belief that no such graphs will be found, but this remains to be proven.

Bibliography

[1] G. Di Battista, R. Eades, P. Tamassia, and I.G. Tollis. *Graph Drawing: Algorithms for Visualization of Graphs*. Prentice Hall, Upper Saddle river, NJ, 1st edition, 1999.

[2] G. Chartrand and L. Lesniak. *Graphs and Digraphs*. Chapman & Hall/CRC, Boca Raton, FL, 3nd. edition, 1996.

[3] J. Edmonds. A combinatorial representation for polyhedral surfaces. *Notices of the American Mathematical Society*, **7**:646, 1960.

[4] Scott Fortin. The graph isomorphism problem. Technical Report TR96-20, University of Alberta, July 1996. www.cs.ualberta.ca/research/techreports /1996/TR96-20.php.

[5] Judith R. Fredrickson. *On the Crossing Number of Complete Graphs: Growing Minimal K_n From Minimal K_{n-1}*. PhD thesis, University of Nevada, Reno, NV, 2006.

[6] Judith R. Fredrickson, Bei Yuan, and Frederick C. Harris, Jr. A time saving region restriction for calculating the crossing number of K_n. *Congressus Numerantium*, **168**:145–158, May 2004.

[7] M.R. Garey and D.S. Johnson. Crossing number is NP-complete. *SIAM J. of Alg. Disc. Meth.*, **4**:312–316, 1983.

[8] Richard K. Guy. A combinatorial problem. *Nabla (Bulletin of the Malayan Mathematical Society)*, **7**:68–72, 1960.

[9] Richard K. Guy. The decline and fall of Zarankiewicz's theorem. In Frank Harary, editor, *Proof Techniques in Graph Theory (Proc. Second Ann Arbor Graph Theory Conf., Ann Arbor, Mich., 1968)*, pages 63–69. University of Michigan, 1969.

[10] Frank Harary and Anthony Hill. On the number of crossings in a complete graph. In *Proceedings of the Edinburgh Mathematical Society*, volume **13** 2nd Series, pages 333–338, London, 1963. Edinburgh Mathematical Society.

[11] Frederick C. Harris, Jr. and Cynthia R. Harris. A proposed algorithm for calculating the minimum crossing number of a graph. In Yousef Alavi, Allen J. Schwenk, and Ronald L. Graham, editors, *Proceedings of the Eighth International Conference on Graph Theory, Combinatorics, Algorithms, and Applications*, volume **2**, pages 469–478. Western Michigan University, June 1998.

[12] Daniel J. Kleitman. The crossing number of $K_{5,n}$. *Journal of Combinatorial Theory*, **9**:315–323, 1970.

[13] F. T. Leighton. *Complexity Issues in VLSI*. MIT Press, 1983.

[14] F. T. Leighton. New lower bound techniques for VLSI. *Math. Systems Theory*, **17**:47–70, 1984.

[15] Brendan D. McKay. Practical graph isomorphism. *Congressus Numerantium*, **30**:45–87, 1981.

[16] R. B. Richter and C. Thomassen. Relations between crossing numbers of complete and complete bipartite graphs. *American Mathematical Monthly*, **104**:131–137, Feb. 1997.

[17] Farhad Shahrokhi, Ondrej Sykora, Laszlo A. Szekely, and Vrt'o Imrich. A new lower bound for the bipartite crossing number with applications. *Theoretical Computer Science*, **245**:281–294, August 2000.

[18] Umid Tadjiev. Parallel computation and graphical visualization of the minimum crossing number of a graph. Master's thesis, University of Nevada, Reno, NV, 1998.

[19] Umid Tadjiev and Frederick C. Harris, Jr. Parallel computation of the minimum crossing number of a graph. *Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing*, 1997.

[20] D. R. Woodall. Cyclic-order graphs and Zarankiewicz's crossing-number conjecture. *Journal of Graph Theory*, **17**(6):137–145, 1993.

[21] Bei Yuan. A generic queueing system and time saving region restrictions for calculating the crossing number of K_n. Master's thesis, University of Nevada, Reno, NV, 2004.

[22] Bei Yuan, Sean C. Martin, Judith R. Fredrickson, and Frederick C. Harris, Jr. A generic queuing system for computationally intensive problems. *Congressus Numerantium*, **171**:193–206, May 2004.

[23] K. Zarankiewicz. On a problem of P. Turán concerning graphs. *Fundamenta Mathematicae*, **41**:137–145, 1954.