University of Nebraska Omaha

University of Nebraska at Omaha
DigitalCommons@UNO

Student Work

10-1-2000

Development of an Abstract Graph Partitioning Model Using the Maple V Computer Algebra System.

Christopher J. Augeri

Follow this and additional works at: https://digitalcommons.unomaha.edu/studentwork

Recommended Citation

Augeri, Christopher J., "Development of an Abstract Graph Partitioning Model Using the Maple V Computer Algebra System." (2000). *Student Work*. 3539.
https://digitalcommons.unomaha.edu/studentwork/3539

This Thesis is brought to you for free and open access by DigitalCommons@UNO. It has been accepted for inclusion in Student Work by an authorized administrator of DigitalCommons@UNO. For more information, please contact unodigitalcommons@unomaha.edu.

Development of an Abstract Graph Partitioning Model Using the Maple V Computer Algebra System

A Thesis

Presented to the

Department of Computer Science

and the

Faculty of the Graduate College

University of Nebraska

In Partial Fulfillment

of the Requirements for the Degree

Master of Arts

University of Nebraska at Omaha

by

Christopher J. Augeri

October 2000

UMI Number: EP74737

All rights reserved

INFORMATION TO ALL USERS
The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript
and there are missing pages, these will be noted. Also, if material had to be removed,
a note will indicate the deletion.

UMI EP74737

Published by ProQuest LLC (2015). Copyright in the Dissertation held by the Author.

Microform Edition © ProQuest LLC.
All rights reserved. This work is protected against
unauthorized copying under Title 17, United States Code

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 - 1346

THESIS ACCEPTANCE

Acceptance for the faculty of the Graduate College,

University of Nebraska, in partial fulfillment of the

requirements for the degree Master of Arts, University

of Nebraska at Omaha.

Committee

Name	Department/School
Hesham Ali	Computer Science
[signature]	[signature]
Azad Azadmanesh	computer science

Chairperson [signature]

Date 7/27/00

Abstract

Graph partitioning involves decomposing a relational graph into smaller graphs, subject to domain-specific parameters and constraints. There are a number of application areas to include database querying, map coloring, job allocation, VLSI design, and parallel processing. The primary goal is to unify a portion of these concepts, allowing discussion and execution at a more common and abstract level.

There are many facets to graph partitioning. Typical areas include the number of partitions, the size of a partition, number of inter-partition connections, and the amount of replication involved. These and many other factors must be considered when generating a partitioning algorithm. We propose an abstract graph partitioning model, the *AGPM.*

This model includes structures for modeling the parameters, constraints and goals when partitioning. From this abstract model, we develop two new graph-partitioning algorithms. The first is a *k*-way bounded partitioning using genetic techniques. The other is a hierarchical method based on graph centers. We compare both of these algorithms against domain-specific methods from VLSI Design and Parallel-Processing Scheduling.

For analysis, we use a broad cross-section of graphs used current research. We also demonstrate applications with other domains and graphs. All research was conducted within the C/C++, Tcl/Tk and Maple V development environments. Primary development was done with Maple V, demonstrating its usefulness as a simulation and experimental research tool across both the mathematics and computer science fields.

Acknowledgment

To Mom and Dad, whose love of learning gave me the heart to complete this work. For Roxanne, Andrew and Joshua, whose motivation made writing this work bearable, patience made it enjoyable, and support made it a reality. Without them, the early mornings and late nights would have been a tunnel with no end.

My adviser, Dr. Hesham H. Ali, tolerated my wandering over various aspects of partitioning and implementation, in addition to providing direction and scope. Dr. Azad Azadmanesh, in addition to asking necessary questions, continually checked on when I'd ever finish. Dr. G. Griffith Elder provided some hope of other potential uses, and brought a level of formality I would not of otherwise obtained or necessarily attempted.

Also deserving of recognition is Dr. Margaret P. Gessaman, my analysis instructor and unofficial adviser over the last few years. Thanks go to Dr. Hossein Saiedian, who pointed out when I was missing the forest for the trees. Kirby Bohling provided assistance regarding C++ templates and other points of the language. Andrew Stalcup provided the edge data used in the United States map coloring, along with supportive interest.

My most heartfelt gratitude goes to the people of MEDICIS at the Polytechnique, France, whose resources made the level of experimentation we achieved possible. A special thanks goes to Teresa Gomez-Diaz and Joel Marchand of MEDICIS, whose support assisted completion of our computations.

This work is also offered in memoriam to Eugene L. Lawler, whose 1967 paper inspired my efforts. For those not mentioned, I offer my apologies and sincere thanks.

Table of Contents

1. Introduction 1

1.1. Research Objective 3

1.2. Previous Research 11

2. Fundamental Terms and Concepts 12

2.1. Graph Theory Fundamentals 13

2.2. Partitioning Specific Concepts 17

2.3. Problem Definition 20

2.4. Problem Complexity 22

2.5. Input, Control and Output Metrics 25

3. Background Research 29

3.1. Overview 29

3.2. Problem Categories and Scope 30

3.3. Prior to mid-1970s 32

3.4. Partitioning in VLSI Design 35

3.5. Partitioning in Parallel Processing 40

4. The AGPM and Developed Algorithms 44

4.1. Abstract Graph Partitioning Model 44

4.2. GEA: Genetic and Evolutionary Algorithm 46

4.3. CTR: Center-Based Algorithm 60

4.4. RAN: Random Iteration Algorithm 65

5. Results Analysis 66

5.1. Overview 66

5.2. Graph Families 71

5.3. MEDICIS 83

5.4. Sample Partitioning Results 84

5.5. Space and Time 92

5.6. Replication (|V|), Components (K), and Partitions (k) 94

5.7. Cross-Algorithm Category Analysis 95

5.8. CTR Variants Analysis 101

5.9. GEA Variants Analysis 107

5.10. AGPM Sample 113

6. Conclusions 114

6.1. Future Directions 114

6.2. Additional Graph Sources 115

6.3. Alternative Applications 116

6.4. Final Thoughts 118

7. References 119

A. Visual Depiction of Thesis Structure 124

B. Project & Reference Research System (PRRS) 125

C. GraphPar Functions 128

1. Introduction

I first became aware of graph partitioning in a VLSI (Very Large Scale Integration) Circuit design course during the spring semester of 1998 [37]. Within the VLSI design field, partitioning is used when a specified circuit is too large for the available hardware. This partitioning can occur at the chip, board and system levels. For instance, within a modem, the encryption, decryption and compression circuits may be too large to fit on a single chip. Thus the designers may manually decompose these onto two separate chips. Designers may also have to allocate chips to boards, or at a higher level, boards to systems. There are many design factors to consider and automated partitioning allows in-depth consideration of these factors.

Limitations when decomposing a circuit include the available space per device (e.g. a chip), the number of pins on the device and the delay encountered when going between two physical devices. All of these factors must be taken into account when mapping a circuit to the available technology, and a hand-designed solution will not suffice, especially as the circuit size increases.

Perhaps the most significant material learned during this time was an intense emphasis by both the text and instructor on abstracting the graph from the circuit, at least for purposes of partitioning and other operations performed when designing circuits. In other words, at times the technology took a back seat to the graph and mathematical theoretic concepts in order to develop successful techniques of developing the circuit. For our purposes, we

treated the circuit as a graph and essentially, "ignored" that we were working with and modeling a circuit. My specific project for this course involved developing a working version of the Best Predecessor algorithm, later modified and used in this research.

An algorithm is a plain-language version of a routine used to process a set of data, "...especially an established, [finite] recursive computational procedure..." [53]. It is essentially the computer scientist's version of a how-to manual. This initial implementation made me aware of fundamental issues regarding partitioning and demonstrated limitations of this algorithm. The Best Predecessor algorithm also incorporates the concept of replication, wherein it is faster to do the work on more than one device in order to minimize the completion time.

Algorithms eventually lead to actual computer programs. These programs will often have several, perhaps thousands or more, of related tasks to complete in order to reach the end of the program. In many cases, some of these steps may be performed concurrently, e.g. one person can work on the roof while another puts in floorboards. This type of concurrency is what drives the demand for parallel processing. However, some of these tasks must be executed sequentially, e.g. the frame must be built before the floor or roof can be laid.

I became aware of other partitioning applications in parallel processing during an advanced computer architecture course [5]. In parallel processing, partitioning is used to decompose a program task graph and allocate subtasks to multiple processing units.

Primary concerns are the direct path adherence, or linearity, of the partitioning and the inter-processor communication delay. It is necessary to ensure the schedule maintains the original graphs' dependencies.

In elementary terms, partitioning involves the subdivision of related tasks or objects into distinct groupings with a minimum of overlap, and communication. It is the scientific study of divide-and-conquer techniques. This research shall enable mapping of the studied domain onto graphs so the problem may be solved in an abstract form.

Other domains, besides those already mentioned include allocating fixed amounts of money, materials or other resources to various individuals or places. A delivery company manager would be interested in maximizing the number of cartons per truck and minimizing delivery times. Map coloring, to be discussed later, may also be worded as a partitioning problem [4: 250-268, 15]. Knowledge and database researchers have also expressed interest in partitioning [39, 29]. Any time a goal involves grouping a set of items, partitioning is a useful tool.

1.1. Research Objective

The primary objective is to form a collective body of knowledge across a broad spectrum of domains requiring partitioning functions, known as the *Abstract Graph Partitioning Model*, or *AGPM*. At this time, we are unaware of a cohesive effort to unify concepts found in different areas of application. Additionally, two general-purpose heuristic techniques, CTR and GEA, shall be developed and compared against two existing domain-specific algorithms, BPD and DSC. These algorithms are all used to partition

graphs of varied structure and size. The partitions are controlled by a number of values for several control parameters.

Also developed during this research was a script-based GUI front-end, *GraphOp*, to join together various aspects of this research: graph generation, graph partitioning and graph visualization. A prototype graph visualization tool was also experimented with, based on Ousterhout's sample script [26]. Also included is a discussion of three graph visualization tools used in this research. The final contribution of this research is the *Project & Reference Research System*, *PRRS*, used to track the various readings cited in the references. The final product is a well-developed partitioning code base, *GraphPar*, developed in the *Maple V* mathematical research package.

1.1.1. Abstract Graph Partitioning Model (AGPM)

Partitioning is usually presented in the context of domain-specific. As in many human endeavors, it is often possible to classify certain approaches based on one or more criteria. This allows for both broader application of the techniques, along with presenting a unifying framework and providing a natural discussion and education pattern. The primary goal is to provide a graph partitioning model allowing researchers to discuss graph partitioning on common ground. It is assumed the domain may be suitably represented via a directed acyclic graph. A directed acyclic graph is one such that once arriving at a certain point there is no path linking back to it. This is desirable, as it greatly simplifies both the problem definition and algorithm development.

A domain is defined as a collective body of knowledge in a specific field of study. For instance, VLSI design, parallel processing, product delivery and education may all be considered individual domains. In general, there are often specific factors that characterize a domain when a need for partitioning arises. The goal is to expose areas of overlap between these domains, shedding new light on current problems.

The net effect of such a model is to drive cross-field discussion of graph partitioning and to permit analysis and comparison of various algorithms. With sufficient future development, it allows classification by properties of the partitioning rather than the domain. It further allows us freedom to see areas of partitioning needing future research and transfer knowledge across domains. The *AGPM* is designed to provide a robust tailoring for domain-specific partitioning. This model shall provide the framework that allows problem formulation, algorithmic formulation and results analysis across multiple domains.

1.1.2. Developed Algorithms

Two new general-purpose approaches for graph partitioning, based on the *AGPM*, are presented. The first method, GEA, is based on genetic and evolutionary modeling techniques. While these techniques exist and have been used for partitioning, a new encoding scheme was developed, providing greater freedom of exploration and ease of analysis. Genetic techniques use a string of data to represent a problem solution, also

known as an individual. A set of individuals is referred to as a population and an iteration a genetic algorithm is known as a generation.

Via a set of genetic operators, strings are manipulated to permit a controlled, random exploration of the potential solution space. The search space is formally defined as the set of all potential solutions. With problems such as graph partitioning, this search space is too large to exhaustively examine in a reasonable period of time. Any selected solution must be evaluated for both feasibility and quality. In some genetic algorithms, evaluation of feasibility and quality may overlap.

The second algorithm, CTR, is a multi-phase algorithm incorporating existing ideas in graph centers. Each vertex is measured according to its distance from the vertices. These values are recorded and used to assign other vertices to a set of subgraphs, where each subgraph is initially assigned a single vertex. All k-vertex combinations are tested to determine the true graph centers.

Intuitively, it is desired to group items that are "close" in order to minimize the number of inter-partition edges, thus minimizing the delay. There are a number of center types and techniques for determining them. One aspect of the research involved determining suitable center metrics to determine the seed vertices for a partition.

1.1.3. Comparative Analysis

The performance of the developed algorithms is compared against the performance of two existing algorithms. The algorithms chosen for comparison are the dominant sequence clustering algorithm, DSC, from Gerasoulis and Yang [3], used in parallel processing; and the best predecessor algorithm, BPD, from Wong and Rajaraman [11], used in VLSI design. All algorithms are tested in both the parallel processing scheduling and VLSI design domains.

DSC involves zeroing edge delays along the current critical path via a top-down trace [3]. Working bottom-up, the BPD algorithm works begins with the primary outputs, replicating vertices if there is predecessors not present in a pre-determined subgraph. These subgraphs are determined via a pre-processing step. This approach may require a post-processing step for node reduction; however, it does provide an optimal solution for a single constraint [11].

The comparative analysis involves a set of 124 unique graphs, categorized by structure and the number of vertices. Each algorithm run was controlled by a combination of two partitioning domains, two inter-partition delay values, and three partition sizes. Additionally, a control algorithm, RAN, executed the root mean square processing time of the other algorithms. While some combinations proved computationally infeasible, over 11,000 data points were obtained.

The results analysis for these computations includes a comparison of the constraints and domains an algorithm is best designed for, its time and space requirements, the maximum delay observed, and the level of replication required. Additionally, these results are used to compare not only algorithm families but also variants of both the GEA and CTR algorithms. All results were computed on the heterogeneous server farm operated by the French research lab, MEDICIS [33].

1.1.4. Support Tools

The primary support tool was the early development of the *Project and Reference Research System*. This is a Relational Database Management System, or RDBMS, allowing the tracking of people, projects and references. It allows direct links to personal contact information, personal notes on papers or other references, along with project tracking, to include a project journal. All of this information is cross-referenced, allowing a person to find persons and all items or projects worked on by that person. All interaction is via user forms and implemented in Microsoft Access 95/97. Screen shots of this research tool are available on pages A:7–9.

Graph visualization, an important area in its own right, was necessary to get a better picture of certain results. An initial attempt was made towards developing an automated graph visualization tool. However, development time proved excessive and unnecessary. The decision was made to use existing products. Examples and discussion of the selected graph visualization tools are provided. All the systems considered provide at least one means of automatic graph layout, and in some cases, visual partitioning support. Battista

and his co-authors describe many of the algorithms used by these layout tools in their outstanding text, *Graph Drawing* [22].

The automated computational geometry systems used are *GraphLet,* from the University of Passau, Germany [36]; *GraphViz*, developed at AT & T Bell Labs Research [47]; and a demonstration application based on the software library of Tom Sawyer Software [54]. These systems respectively represent a non-profit educational institution, a commercial research laboratory and a commercial software company. They provided a broad sample of available graph visualization tools and demonstrate the importance of their use in the study of graph partitioning.

1.1.5. Software Development Lifecycle

Historically, this research evolved through several rapid application development iterations. The first attempt involved the existing work from the original project incorporating one partitioning algorithm and a rudimentary graph generator developed in C. The file format used in this version was too simple to allow in-depth research. Furthermore, the graph generator only controlled edge density, weight and node weights. It had no provision for the generation of specific graph classes.

The generator was first converted to a pure C++ version. This was done to facilitate having a robust class structure, capable of handling a large category of graphs. This was originally intended to encompass multi-layer graphs and replication, both internally and externally to a specific partition. A large portion of this implementation was done,

however, the development overhead for a rapid application development environment was too high.

During this time, experimental progress was made with a multi-functional Tcl/Tk applet, *GraphOp*. This included interfaces to directly modify text files, launch graph visualization and partitioning tools and interact with a random graph generator. An attempt was also made to develop a graph visualization tool (GVT), based on Tcl/Tk. However, GraphOp and this other GVT are currently considered prototypes, given our final development choice, *Maple V*.

Maple V is a symbolic computer algebra system [32]. Although performance of at least 1-2 orders of magnitude was sacrificed, gained was an existing graph abstract data type and environment more suited to algorithm development and experimentation. Furthermore, using *Maple V* forced the author to think in mathematical terms versus application implementation.

While some execution-time difficulties occurred with *Maple V*, it proved a worthwhile learning experience and demonstrates the usefulness of general-purpose mathematics computing software in a research environment. Another graph theoretic study based on the doctoral thesis research of several authors is presented in *Network Reliability: Experiments with a Symbolic Algebra Environment* [12]. A visual depiction of the thesis structure and its development is available in Appendix A.

1.2. Previous Research

Examination of the current literature base suggests the need for a general graph partitioning model. The original objective was to develop a fairly complete taxonomy of graph partitioning algorithms and derive the model after this taxonomy was completed. Realizing this was too large an undertaking, the goal was scaled back to analyze a smaller set of literature, develop an initial model based on key partitioning characteristics, and leave the taxonomy for further research.

Most authors capably illustrate the domain of interest for their partitioning, however, lesser effort has been made to present an abstract partitioning model. There are at least three reasons for developing such a model:

1. Understand when to use a specific partitioning method.
2. Provide a unifying framework for practitioners in different domains.
3. Provide a backdrop against which more comprehensive analysis can be performed.

This model provides both the foundation and objective of this research. It is based on examination of graph partitioning efforts in the domains of VLSI design and parallel processing. Additional discussion is also made in workshop job allocation, map coloring, database querying, and knowledge base systems. This literature spans approximately 40 years, from 1960 - 2000.

2. Fundamental Terms and Concepts

Why graphs? Graphs are a tool that can represent any group of items and their relationships (or lack thereof). Though perhaps not so obvious, graphs are everywhere in daily life. A subway map, a biological dichotomous key and a genealogical family tree are all examples of graphs. A graph is a way of identifying unique objects and the relationships amongst them. Dependent on the domain, both objects and relationships may have a number of attributes associated with them.

In today's increasingly connected world, graphs are gaining in both their role and importance. As society becomes increasingly information-centric, there is a desire, regardless of the domain, to divide, or partition this information. Graphs are a natural tool to model this information. Thus drives the need for dividing, cutting and decomposing these graphs into partitions.

Of course, mathematicians are very precise people, and have formal definitions for all elements and properties of graphs. Furthermore, the operations performed on graphs may also be formally described. Armed with these formal definitions, properties and operations, it is possible to define both new problems and new operations.

Although not a comprehensive treatment, the following section should facilitate comprehension of the research. This material is available in any combinatorics [41], graph theory [25] or discrete math [28] textbook, amongst others. Basic definitions and operations are defined, followed by terms and concepts necessary for graph partitioning.

For those familiar with graph theoretic terms and concepts, the remainder of this chapter is unnecessary, although sections 2.2 – 2.5 cover terms and concepts directly related to partitioning.

2.1. Graph Theory Fundamentals

A graph, $G = (V, E)$, consists of V, a nonempty set of vertices, and E, a set of unordered pairs of distinct elements of V, called edges. In the example, $V = \{1,2,3\}$ and $E = \{(1, 2), (2, 3), (1,3)\}$.

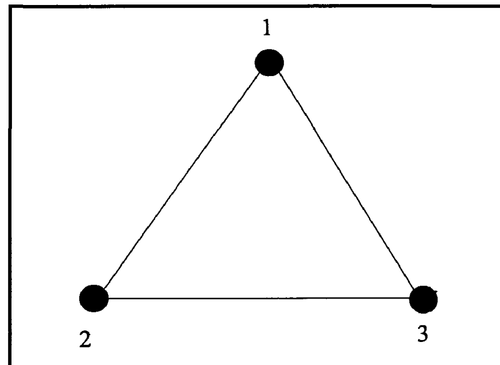


Figure 2-1

The elements of V and E may have assigned values. For instance, each vertex in V may represent a town, and the value its' population, while the value assigned to each edge in E represents the distance between the vertices in V. Such values or attributes create weighted vertices and weighted edges. Creating some values, $V = \{(1, 5000). (2, 7500), (3, 50000)\}$ and $E = \{(2, 1, 3.5), (2, 3, 6.0), (1, 3, 1.7)\}$. Note drawings are not to scale.

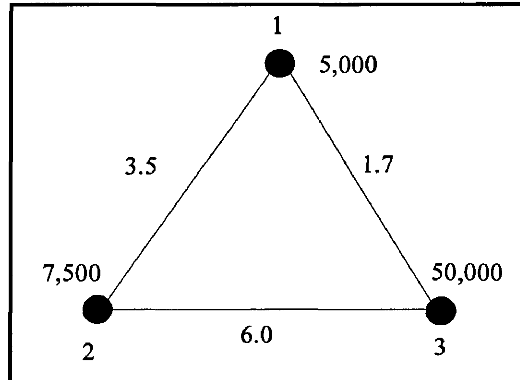


Figure 2-2

Graphs without directed edges are also known as simple graphs. A directed graph, or digraph, is similar, except E is a set of ordered pairs of distinct elements of V called edges. When drawing a digraph, the edges will have an arrow pointing towards the pair's second element. This is indicated below, where $V = \{(1, 5000). (2, 7500), (3, 50000)\}$ and $E = \{(2, 1, 3.5), (1, 3, 1.7), (3, 2, 6.0)\}$.

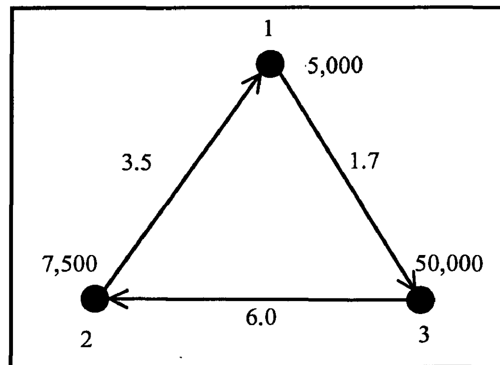


Figure 2-3

Figure 2-3, contains what is known as a cycle. In other words, can endlessly traverse Towns #1, 2 and 3 without reaching a final stopping point. A cycle, in both the undirected and directed forms, permits the return to a previously visited vertex via the existing edges. A graph without any cycles is called an acyclic graph. Directed acyclic graphs, or DAGs, simplify many of the operations in graph theory. In our research, although the

underlying simple graph may have cycles, the digraph shall not. Removal of (3, 2, 6.0) renders figure 2-3 acyclic, as seen below:

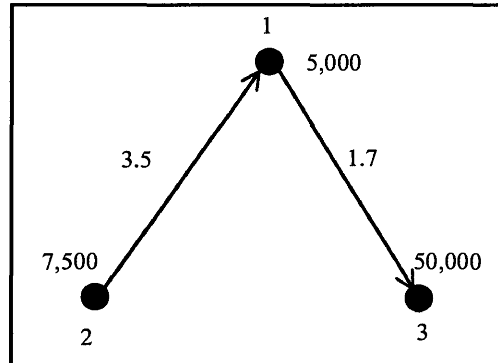


Figure 2-4

The directed, acyclic condition may also be worded as: for all x in E, $x = (u, v)$, $u <> v$ and if (u, v) is in E then (v, u) is not in E; if there is a directed path in E from u to v there is not a directed path from v to u.

Another area related to graph simplification is transitivity. If one may reach a vertex via another vertex, it is redundant to have a direct connection to it. Increased transitivity is needed when redundant connections are necessary amongst vertices, while reduced or no transitivity when cost or simplicity is a concern. All graphs used in this research shall be made intransitive. Removing the edge (2, 3, 6.0) from the left image below becomes the intransitive graph on the right.

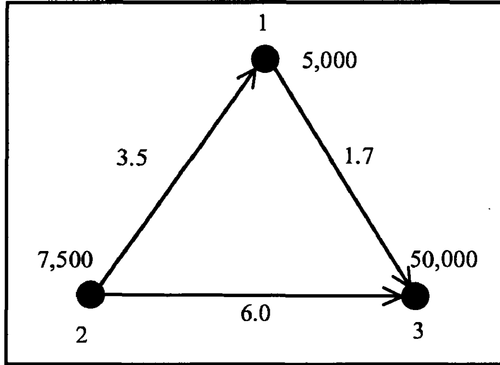


Figure 2-5

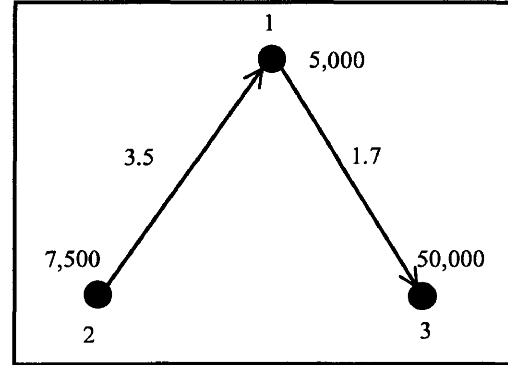


Figure 2-6

A path is a distinct ordering of directed edges in *E*, indicating the edges necessary to follow from the starting node, *u*, to the ending node, *v*. If there exists at least one path for any vertex *v* from at least one primary input to at least one primary output, and the underlying simple graph has a path between every pair (*u*, *v*), then the graph is connected. If there exists any two nodes *u* and *v* and no path exists between them, the graph is independent. In our research, we work solely with intransitive, connected, directed acyclic graphs.

The number of vertices in *V* is represented by $|V|$. The degree of a node in an undirected graph is the number of edges connected to it. In Figure 2-1, each vertex has a degree of 2. The number of edges in *E*, shown by $|E|$, is exactly one-half of the total degree. Thus ½(2 + 2 + 2) = 3. This is easily observed, as each edge has two vertices.

In a directed graph, the in-degree is the number of edges leading to the node and the out-degree is the number of edges leading away from a node. The vertex degree is the sum of the in-degree and the out-degree. A vertex with an in-degree of 0 is considered a primary input, while those with out-degrees of 0 are considered primary outputs. In Figure 2-6,

vertex #2 is a primary input, while the destination; vertex #3 is a primary output. Vertex #1 is considered an internal vertex and in this route has an in-degree and out-degree equal to 1, with a degree of 2.

A subgraph is considered a subset of G, G_i, containing V_i and E_i. A subgraph is a graph by definition, and all terms, definitions and operators may be applied to it. For Figure 2-6, two potential subgraphs are $G = (V=\{1, 2\}, E = \{(2, 1, 3.5)\})$ or $G = (V=\{3\}, E = \{\})$.

2.2. Partitioning Specific Concepts

This section defines graph-theoretic terms and concepts directly related to partitioning. These partitioning qualities and metrics are used in both determining partitioning parameters and objectives, as well as analyzing the results. All readers are encouraged to review this section, as some definitions may be unique to this research.

2.2.1. Definitions

A partition is the complete allocation of a graph, G to a set, P containing k sub-graphs of G. If G_i and G_j are sub-graphs of G and G_i is equivalent to G_j, then $i = j$. Furthermore, P also contains a set E_P, where E_P, contains those edges outside of any partition. An example partition is $P = \{(V_1 = \{1, 2\}, E_1 = \{(2, 1)\}), (V_2 = \{3\}, E_2 = \{\}), E_P = \{(1, 3)\}\}$.

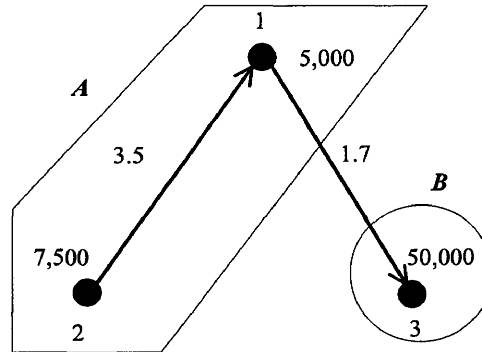


Figure 2-7

In this example, all of V_i and E_i, along with E_P, are disjoint. This type of partition is said to be non-replicated. When replication is allowed, the sub-graphs and E_P may not be disjoint. It is assumed a vertex may only be represented within each sub-graph a single time. Thus, if there are M partitions, a vertex may only exist M times. Edge replication may be considerably higher, however, it is only necessary to ensure all dependencies are observed.

The fields most thoroughly analyzed were circuit decomposition and parallel processing. In circuit decomposition, a circuit is decomposed into M sub-circuits, each with a specified component and connector limit, with an emphasis on minimizing the maximum delay at any one primary output. Within the field of parallel processing, a task graph is decomposed into M graphs, with no vertex or edge limits, such that each CPU, M_i, is kept occupied and the total delay is minimized. This distinction shall become clearer, especially when discussing linearity.

As an alternative problem, when creating a map coloring, the objective is to divide the map into M partitions, where M is the number of available colors, such that the amount of

each color in area is balanced and the total length of adjacent borders *of the same color* is minimized. It has been previously proven by Appel and Haken four colors are sufficient, however, finding such a coloring can be difficult [25]. A sample map-coloring problem shall be discussed later.

Granularity is a metric measuring the work accomplished in a partition relative to communication cost. If granularity decreases, work is being allocated to subgraphs in smaller amounts. As granularity increases, larger subgraphs are being created, however, the potential for parallelism decreases. Depending on the domain, one may request a certain level of granularity [3].

Closely related to granularity is the concept of linearity. A subgraph may be considered of the following types: linear or non-linear. Linearity may have several methods of expression, however, it essentially measures whether a subgraph contains a set of dependent vertices forming a single path or a set of independent vertices forming several paths. For example, the graph on the left is linear, while the graph on the right is both non-linear and disconnected [3].

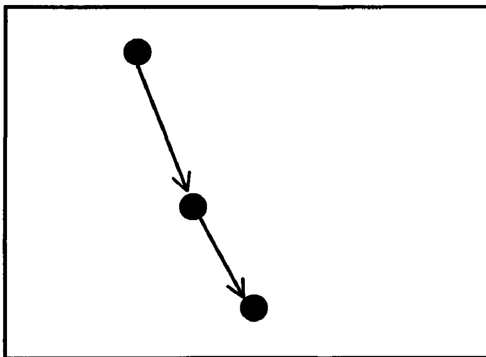


Figure 2-8

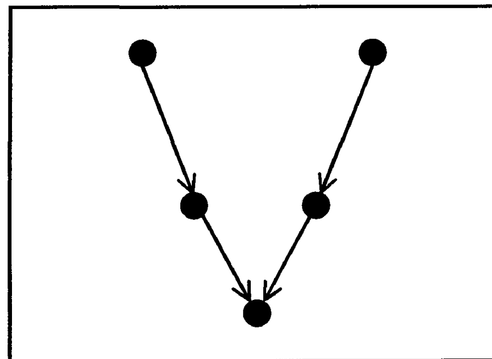


Figure 2-9

Granularity and linearity identify important differences between VLSI design and parallel processing. We use them here simply to demonstrate that their end objectives are different and why we chose them as our two primary domains of interest.

We did not directly measure granularity or linearity in our analysis. However, analysis is done with respect to the structural family a graph belongs to and graph densities. It was also necessary to form linear partitions as a post-processing step when analyzing partitions within the parallel processing domain. For simplicity, we assume there are no fan-in or fan-out limitations. In other words, an input or output can handle any number of connections necessary to it.

2.3. Problem Definition

Although we have described basic graph theory and partitioning concepts, we must have a problem we are studying. However, we must be cognizant of both the algorithms we are studying and the AGPM we are developing. Thus, there are several variants of partitioning we shall consider. The basic partitioning problem, often referred to as k-way balanced partitioning, involves the division of a graph into k subgraphs. The only constraint is the number of vertices, $|V|$, must be divided as equitably as possible, with a maximum difference of one [51].

There is no consideration given to the number of edges between these subgraphs, although there is a common objective of minimizing delay at the primary outputs. A modification to this basic problem include the special case of $k = 2$, often referred to as bi-partitioning. Certain approaches use bi-partitioning techniques to achieve a k-way, or

multi-way partitioning. Additional modifications to the basic partitioning problem include:

1. Delay Minimization
    a. maximum delay
    b. total delay
    c. average delay
2. Cut ($|E|$) limits
    a. k-way limits
    b. pre-specified limits
3. Vertex ($|V|$) limits
    a. non-balanced
    b. weight limits
4. $k = \infty$
5. Replication

The minimization objective is an optimization goal. The domain of interest will drive the optimization goal and thus the delay metric selected. Additionally, certain domains, e.g. VLSI Design, may have cut limitations for the subgraphs. This will reduce the feasible solution space, although some algorithms may not have a mechanism to directly incorporate this. Modifications of the vertex limits including allowing non-balanced partitions, e.g. $|V_i| > |V| / k$. Alternatively, rather than assuming each vertex has a weight of one, each vertex may have an independent weight that is used to control the number of vertices in a subgraph.

Loosening of the problem constraints includes unrestrained partitioning, or $k = \infty$. The complete extension of this would place every vertex in its own subgraph. Thus, all edges would become inter-partition edges, subject to the inter-partition delay. Additional loosening of constraints also allows for vertex and/or edge replication. While this may help achieve delay minimization, it must be deemed suitable for the domain of interest.

These various problem definitions are presented to demonstrate several concepts. One, there are obviously extensive variations of the partitioning problem. There are one or more domains that may have their partitioning needs met by each of these variants. Furthermore, it helps demonstrate the necessity of the AGPM, so that we successfully describe the problem, the constraints and optimizations goals, and thus select the appropriate algorithm. The algorithms, and the subsequent analysis, match up these problem definitions with each other.

2.4. Problem Complexity

Defining the problem brings with it the question of how complex is the problem. The reason graph partitioning algorithms are so critical is its' inherent complexity. Complexity is often analyzed from two perspectives, space and time. It is an attempt to measure how much memory or how long, respectively, an algorithm requires to arrive at a final solution. More specifically, we are concerned with the total number of solutions for a given problem. As it turns out, graph partitioning problems have an extremely large number of potential solutions. In addition, evaluating the quality of these solutions is a time-consuming process.

Problems are generally divided into two categories, P and NP. P presents those problems and algorithms considered to be polynomial on a traditional computing device. For instance, sorting algorithms are typically considered to be within P. If we designed a sorting algorithm that examined every permutation to determine if a specific solution was

sorted, it would be considered NP. NP refers to those problems that are non-polynomial on traditional computing devices.

P, or polynomial, problems and algorithms are generally considered feasibly computable on current computing devices (for reasonably sized data sets). Graph partitioning belongs to the special class of problems referred to as NP. The reason is we are attempting to optimize for the delay through the graph. Furthermore, partitioning belongs to a special category referred to as NP-hard. Specifically, evaluation of a particular solution is computable in time P. However, since the number of solutions is in NP, while evaluation is in P, partitioning is considered NP-complete. Since we are optimizing for delay, we must concern ourselves with the NP aspect of partitioning. At this point we transition from NP-complete to NP-hard [37, 51]. Although we can describe a machine that can solve a NP problem in time P, none have yet to be constructed. Quantum computing is considered a step in this direction [21].

Bui and Moon report that even the bi-partitioning problem for general and bipartite graphs is also NP-hard (bi-partitioning is considered to be simpler than k-way) [51]. Bipartite graphs are those that may have the vertices placed two groups, such that no members of either group are adjacent to members of the same group. In the figure below, and used earlier, note city #1 has had its shading removed. It is only adjacent to cities #2 and #3, neither of which is adjacent to each other.

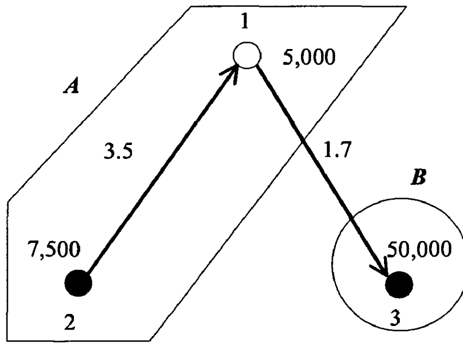


Figure 2-10

Bui and Moon also report that even finding approximate solutions for general graphs and planar graphs are NP-hard [51]. Planar graphs are those graphs that can be drawn without any edges overlapping each other. For instance, although the complete graph of four vertices is normally as a non-planar graph, it may be drawn as a planar graph. The complete graph of six vertices is not planar, however. The following images clarify this.

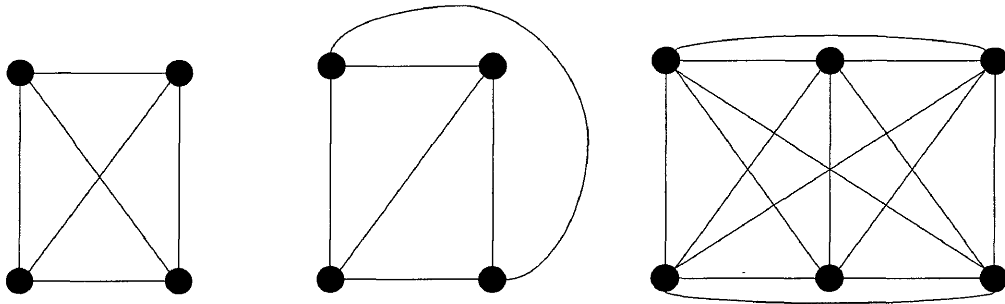


Figure 2-11 (non-planar) Figure 2-12 (planar) Figure 2-13 (non-planar)

There are several ways to approach the NP-hard complexity of graph partitioning. In some cases, it becomes feasible to loosen the restrictions, e.g. not to limit the problem to k-way partitions or not require balanced partitions. Other scenarios may allow special conditions, e.g. vertex replication. This essentially increases the number of potential solutions. However, it also increases the number of ways to reach an optimum [30].

2.5. Input, Control and Output Metrics

The following metrics outline the ones forming the *AGPM*'s foundation.

Category	Metric	Description
Input Graph Metrics	G_f	Input Graph
	$\sum(deg(G))$	|roots|, (primary inputs & outputs)
	$\min((a,\ldots,b)_{dt})$	min delay along any path
	$\max((a,\ldots,b)_{dt})$	max delay along any path
	$\|E\|$	number of edges (including replicas)
	$\sum E_{wt}$	sum of edge weight || delays
	$\|V\|$	number of vertices (including replicas)
	$\sum V_{dt}$	sum of vertex delays
	$\sum V_{wt}$	sum of vertex weights
Execution Control Parameters	P_f	Partition Family (e.g. Circuit or Schedule)
	P_a	Partition Algorithm (e.g. BPD or DSC)
	C	max cut/partition (including |root vertices|)
	M	max vertex weight/partition
	K	number of desired partitions
	D	inter-partition delay (constant)
	D_{max}	max delay along any path (worst case)
Output Partitioning Metrics	$\kappa(G)$	number of components in partitioned graph
	$\max(deg(P))$	max cut/partition (including roots)
	$\sum(deg(P))$	sum of partition cuts
	$\min((a,\ldots,b)_{dt})$	min delay along any path
	$\max((a,\ldots,b)_{dt})$	max delay along any path
	$\|E\|$	number of edges (including replicas)
	$\sum E_{wt}$	sum of edge weight || delays
	$\|P\|$	number of partitions
	$\|V\|$	number of vertices (including replicas)
	$\sum V_{dt}$	sum of vertex delays
	$\sum V_{wt}$	sum of vertex weights
	$\sigma(V_{wt})$	standard deviation of partition vertex weight sums
	$\max(\sum V_{wt})$	max of partition vertex weight sums
	$O_s(n)$ & $O_t(n)$	processing space (bytes & seconds)

The previous table defines three basic categories of graph metrics: input graph metrics, execution control parameters and output partitioning metrics. These categories are used to facilitate both the development of the *AGPM* and a black-box approach for analysis. Within each category we find several key metrics and parameters critical to graph partitioning research. Additionally, this table synthesizes the mathematical symbols used in the multiple readings we have perused. We shall now entertain a brief discussion of each.

Within the input graph metrics category, there are nine values used to distinguish between graphs. Before analysis, each graph was given a unique name, G_f. We then determined the number of primary inputs and outputs, $\sum(\deg(G))$, where a primary input has no inputs and a primary output has no outputs. We then determined the minimum and maximum delay from any primary input to any primary output, $\min((a,\ldots,b)_{dt})$ and $\max((a,\ldots,b)_{dt})$, respectively. The total number of edges, $|E|$, and the sum of their delays, $\sum E_{wt}$, completed the edge analysis. Similarly, we found the total number of vertices, $|V|$, and the sum of the delays and weights, $\sum V_{dt}$ and $\sum V_{wt}$, respectively.

There are seven execution control parameters. The first two contain the application domain, P_f, and the algorithm, P_a, used to partition the graph. C specifies the maximum subgraph cut, including primary inputs and outputs. M is the maximum vertex weight per subgraph. If the vertex weights are equal to one, M is also equal to $|V|$. K is the number of desired partitions, as in k-way partitioning. In our implementation, k may determine C and M or vice versa, depending on the problem specification. D is the inter-subgraph

delay when connecting vertices in two distinct subgraphs. D_{max} is a worst-case metric that assumes all edges are inter-subgraph edges. Thus, D_{max} is equal to $|E| * D + \sum V_{dt}$.

The output partitioning metrics cover a wide range of measurements and are also the most plentiful. There are a total of fifteen metrics, some of which are new; others are partitioning-specific versions of input and execution metrics. The number of components, $\kappa(G)$, in the resulting partition identifies the number of distinct, independent subgraphs in the final solution. The maximum partition cut, $\max(\deg(P))$, identifies if an algorithm observed the limit imposed by the execution control parameter, C. The minimum and maximum path delays, $\min((a,\ldots,b)_{dt})$ and $\max((a,\ldots,b)_{dt})$, respectively, are the optimization goals of graph partitioning.

The number of edges, $|E|$, and its related metric, $\sum E_{wt}$, or edge weight sums, are provided to measure edge results independently. The number of partitions, $|P|$, is used to determine if an algorithm observed any k-way constraints imposed on it. The vertex metrics are some of the most valuable, as almost every partitioning variant has some vertex limit. The number of vertices (per partition), $|V|$, includes any replicated vertices. The summed vertex delay and weight values, $\sum V_{dt}$ and $\sum V_{wt}$, are also on a per partition basis. All of these may used in variants limiting the number or weight of vertices per subgraph. The standard deviation of vertex weights, $\sigma(V_{wt})$, is useful for those domains needing balanced subgraphs. The maximum vertex weight sum was used most extensively in our own analysis compared to other vertex limits.

The final two metrics are raw measurements of the processing space and time, $O_s(n)$ and $O_t(n)$, used by a given algorithm during a partitioning run. The space is measure in bytes of data and is considered as the total amount of data processed. It is not an indication of any data concurrency requirements. Time is measured in seconds and is considered as user time only, not any machine activities to support the algorithm's execution. Both are measured for the duration the algorithm executes, not including any graph pre-processing or metric collection post-processing phases.

3. Background Research

This review of existing literature was originally intended to be a significant portion of the results. Due to the quantity of literature available, this was scaled back to a level sufficient to permit the development of and demonstrate the need for the abstract graph partitioning model, or *AGPM*. Thus, instead of a formal taxonomy, various domains and algorithms are reviewed and used to develop the *AGPM*.

3.1. Overview

The objective of the *AGPM* is to be able to characterize an algorithm by a number of characteristics. Most authors capably illustrate the domain of interest for their partitioning algorithm, however, lesser effort has been made to present an abstract partitioning model. This model will provide the foundation of the research. After developing this model, it is possible to analyze existing approaches and develop new ones based on the *AGPM*. Recalling the three primary reasons for developing the model, it helps to…

1. understand when to use a specific partitioning method.
2. provide a unifying framework for practitioners in different domains.
3. provide a backdrop against which more comprehensive analysis can be performed.

The background research is divided into two main eras. Prior to the mid-1970s, many solutions focused on integer-programming techniques. Much of this early work was on a smaller scale, perhaps even performed by hand, and had an emphasis on circuit decomposition. After this time, until the early 1990s, research is aimed more at problem-specific, solution building, refining techniques. This is also during the time when large

parallel processing systems began to be developed, removing the need for hand-solved solutions and permitting study of larger problems.

In recent years, interest seems to be moving towards a hybridization of these approaches, e.g. a random search algortithm combined with a deterministic computation component. As of this writing, a comprehensive literature base of journal articles, doctoral theses, texts and research reports have been reviewed.

3.2. Problem Categories and Scope

From Wilson and Watkins' text on introductory graph theory, a categorization of graph-theoretic problems is obtained [25]. They suggest the following groupings: existence, construction, enumeration and optimization problems. An example of each of these follows:

1. **Existence:** is it possible to partition this graph with balanced, linear partitions?
2. **Construction:** assuming it exists, how can such a partition be constructed?
3. **Enumeration:** how many such partitions are there?
4. **Optimization:** of these, which have the minimum number of partition interconnections?

When developing an algorithm, each of these categories must be considered at some point. Although an algorithm may be developed, it is wise to determine if the development effort is warranted by the difficulty of the problem. To this end, the following formulae are provided to show the staggering number of solutions a combinatoric problem such as graph partitioning may have. These formulae hold assuming vertex and edge replication is not allowed. Two derivations of this formula are

provided. The first is based on the multinomial theorem as presented in Grimaldi's text, *Discrete and Combinatorial Mathematics*, [41]:

$$P = \frac{n!}{\prod_{i=1}^{k}\left(\left(m_{(i,x)}!\right)^{m_{(i,y)}} \cdot m_{(i,y)}!\right)}, \sum_{j=1}^{k}\left(m_{(1,x)} \cdot m_{(1,y)}\right) = n$$

Figure 3-1

The second formula assumes the k-way partitioning problem is being solved, wherein there are a fixed number of balanced partitions. It was developed Russo and his colleagues while employed at IBM's Thomas J. Watson Research Center [44, 5]. For cases where $n \bmod k \neq 0$, n is incremented until $n \bmod k = 0$.

$$P = \frac{n!}{(n/k)!^{m} \cdot k!}$$

Figure 3-2

Using either formula, if $n = 11$, $k = 4$ and $m = \{(3,3), (2,1)\}$, there 15,400 unique partitioning solutions (n is increased to 12 for Russo's formula, $m = \max(m_{i,x})$.

$$\begin{aligned} P &= \frac{n!}{\prod_{i=1}^{k}\left(\left(m_{(i,x)}!\right)^{m_{(i,y)}} \cdot m_{(i,y)}!\right)} \\ &= \frac{11!}{\left((3!)^{3} \cdot 3!\right) \cdot \left((2!)^{1} \cdot 1!\right)} \\ &= \frac{39{,}916{,}800}{\left((6)^{3} \cdot 6\right) \cdot \left((2)^{1} \cdot 1\right)} \\ &= \frac{39{,}916{,}800}{1{,}296 \cdot 2} \\ &= 15{,}400 \end{aligned}$$

Figure 3-3 (Grimaldi)

$$\begin{aligned} P &= \frac{n!}{(n/k)!^{m} \cdot k!} \\ &= \frac{12!}{(12/4)!^{3} \cdot 4!} \\ &= \frac{479{,}001{,}600}{6^{3} \cdot 24} \\ &= \frac{479{,}001{,}600}{216 \cdot 24} \\ &= 15{,}400 \end{aligned}$$

Figure 3-4 (Russo)

The number of bi-partitions of a 100 vertex graph is on the order of 10^{29}, while a 1,000 vertex graph has 10^{300} solutions. A 4-way partition increases these to 10^{56} and 10^{597} solutions, respectively. As discussed earlier, although partitioning is NP-hard, these numbers give an example as to how fast the solution set grows [50, 55, 8, 52, 24].

3.3. Prior to mid-1970s

We begin the background reference with Breuer's comprehensive survey of computer design automation [35]. He presents seven categories of design automation, with the area of interest being "physical implementation". Within this category, several specific problems are listed, with suggested solutions for many. The ones of interest are:

1. The partitioning of the network into portions, each of which can be implemented on at least one standard circuit card type.
2. The assignment of each portion of the network to an available circuit type on a card.
3. The assignment of circuit card types to backboards (motherboards).

Breuer prefaces all mentioned algorithms by identifying each as either exhaustive search or approximate solution algorithms. The single optimal algorithm he identified is Lawler's, although it is deemed infeasible for many problems as the problem size is often, [now], in the thousands and millions of vertices.

An approximation heuristic used by one algorithm Breuer cites uses integer programming techniques, however, it does not fully examine the search space. Specifically, once a component is allocated to a partition, it is not moved. Alternatively stated, the solution is monotonically non-decreasing. This serves to limit the search space. Some of these

algorithms use multiple runs, varying input orders via some criteria or random order as their guiding heuristic.

A key point is the absence of graph-theoretic problem treatment. This becomes important as the problem sizes continue to grow [37, 44, 35]. Lawler, Levitt and Turner's algorithm, based on creation of all possible single-output rooted subgraphs, published in 1969, indicates the beginning of a paradigm shift [17]. As the designs and algorithms become more complex, a more abstract structure, notably graphs, becomes important. This is evident not only in contrast to Lawler's earlier work, but also as compared to contemporary researchers of the late 1960s and 1970s [18].

Kernighan and Lin's classic bi-partitioning technique dominates the 1970s. This technique further defines the transition from an engineering focus to a graph-theoretic emphasis, as evidenced in their title, *An Efficient Heuristic Procedure for Partitioning Graphs* [56]. The K & L algorithm randomly forms a bi-partition, and then iteratively swaps previously unmoved vertices until no delay reduction may be observed.

Additionally, as the K & L algorithm begins with a random partition, it further marks a departure from deterministic to probabilistic algorithms. Fiduccia and Mattheyses, and then Krishnamurthy presented significant improvements to the K & L algorithm [7, 4]. The K & L algorithm, along with its variants, is often used in bi-partitioning, or 2-way partitioning, and *k*-way partitioning. Sherwani indicates bi-partitioning is used recursively for multi-way partitioning [37: 171].

Around the mid-1980s, additional stochastic techniques begin to appear. Stochastic techniques have a random operation as their core operating heuristic. Sherwani provides solid exposition on many of these techniques, to include simulated annealing and simulated evolution. Simulated annealing finds solutions based on an algorithm that imitates the process used to purify metals and other substances. Simulated evolution algorithms are designed to imitate life processes to guide itself to a solution. Both of these processes mimic their natural counterparts in a guided, yet random, manner. He concludes by saying they both tend to produce a better solution than other approaches. However, they respectively take more time and space to do so [37: 171].

Sherwani's writing also mentions the concept of vertex, or component replication, as presented in Kring's efforts [9]. However, he indicates Kring's algorithm is the first to use replication. Russo and his colleagues demonstrated the usefulness of vertex replication as early as 1971 [44, 5]. Furthermore, Eugene Lawler's efforts of the 1960s also incorporate replication [18, 17].

Russo's proposed algorithm is a two-phase algorithm. During the first phase, it increases the granularity of the graph by grouping closely attached vertices. Then, the modified graph is incrementally allocated to a set of subgraphs, based on their distance and the partitioning constraints. In some cases, replication is permitted. Thus, the partitioning is grown around topologically distant vertices. An analogy may be made to watching salt crystallize during super-saturated cooling. However, the algorithm is non-improving,

therefore, if an incomplete partition is formed, it is suggested to adjust the critical net selection of the first phase.

While we have presented a summary introduction to graph partitioning, we shall now focus our attention on the two domains and reference algorithms chosen for comparison purposes. As mentioned earlier, we shall examine both Lawler et al's original algorithm and the more recent version we used. Additionally, the parallel processing algorithm, dominant sequence clustering, shall be discussed in the subsequent section.

3.4. Partitioning in VLSI Design

In VLSI design, partitioning is used to subdivide a circuit into multiple circuits. This is done when a circuit is too large for a single device. A device may be a module, IC (integrated circuit), PCB (printed circuit board) or other comparable object. A circuit, for these purposes, is equivalent to a graph, and more specifically a directed, acyclic graph, or DAG. The resulting sub-circuits are considered a partition. A gate is equivalent to a vertex and a net is used to refer to an edge(s).

A key concept to note is independent circuits located on the same device do not have to wait for other independent circuits located in the same device to execute prior to their execution. In other words, independent circuits may execute in parallel. Therefore, non-linear sub-graphs would be of more interest in this domain. Additionally, in contrast to many others, the algorithm we have chosen directly handles k-way partitioning, rather than a recursive bi-partition.

The work done by Eugene Lawler and his colleagues during the 1960s developed what is known as the Unit Delay Model, or UDM. In this model, delay within a subgraph of a partition is considered negligible. Additionally, the delay through any gate is considered to be zero and the delay between subgraphs, or clusters, is set at a unit, hence the model's name. This algorithm, Best Predecessor (BPD), and its successors, consist of a three-phase routine [11, 18, 17].

First, the delay is identified between a gate and all its ancestors. Based on this delay, each vertex is taken as a root of a subgraph. Then, the closest predecessors, in terms of delay, are added to the subgraph until subgraph constraints are violated. There are $|V|$ optimal subgraphs formed, one for each vertex. In the final phase, beginning with the primary output with the longest delay, direct ancestors of the current subgraph are added from this pre-computed set of subgraphs.

Each subgraph may only be added once, however, an individual vertex may be present in more than one subgraph, thus allowing replication to occur. Vertex replication may be reduced during post-processing. The algorithm generates an optimal partitioning solution (with replication), given there is only a single constraint, e.g. vertex weight limits.

The general delay model, or GDM, developed by Sangiovanni-Vincentelli, Murgai and Brayton, expands upon the UDM. They set forth three rules that define the GDM [40]:

1. Each gate may have a unique delay, V_{dt}.
2. No delay exists between gates in the same subgraph.
3. A constant delay, D, exists between gates in different subgraphs.

The GDM allows for the modeling of more realistic circuits and equipment. For instance, a long wire length within a device may be a delay factor. This may be modeled by inserting dummy vertices with the delay value of the wire length. Additionally, this model can be tailored to handle the circuit design abstraction we are modeling, be it chip, board or device level. This model is used by Rajaraman and Wong to extend Lawler's algorithm. The core remains, and the UDM may be implemented by the GDM [11].

However, if we examine the typical results of the BPD algorithm, two items may be noted. One, there is a significant amount of vertex replication involved. Additionally, the algorithm successively adds linear branches to a subgraph until its limits are violated, e.g. the number of gates. This may be used as input to the *AGPM,* i.e. one can specify whether replication or linearity is allowed in the final partitioning.

The following images depict a sample graph, ISCAS C17, and its optimum 4-way partitioning. ISCAS C17 was provided to attendees of the 1985 International Symposium on Circuits and Systems [19, 13]. The graph contains 11 vertices and 10 edges, with a maximum delay of 6, along paths {CFHK, CFIK, EIK}. It contains five primary inputs, {A, B, C, D, E} and two primary outputs, {J, K}. ISCAS C17 is a bipartite, planar graph and was discussed in Rajamaran and Wong's paper on BPD using the GDM [11]. We will use it to introduce each of the algorithms used in this research.

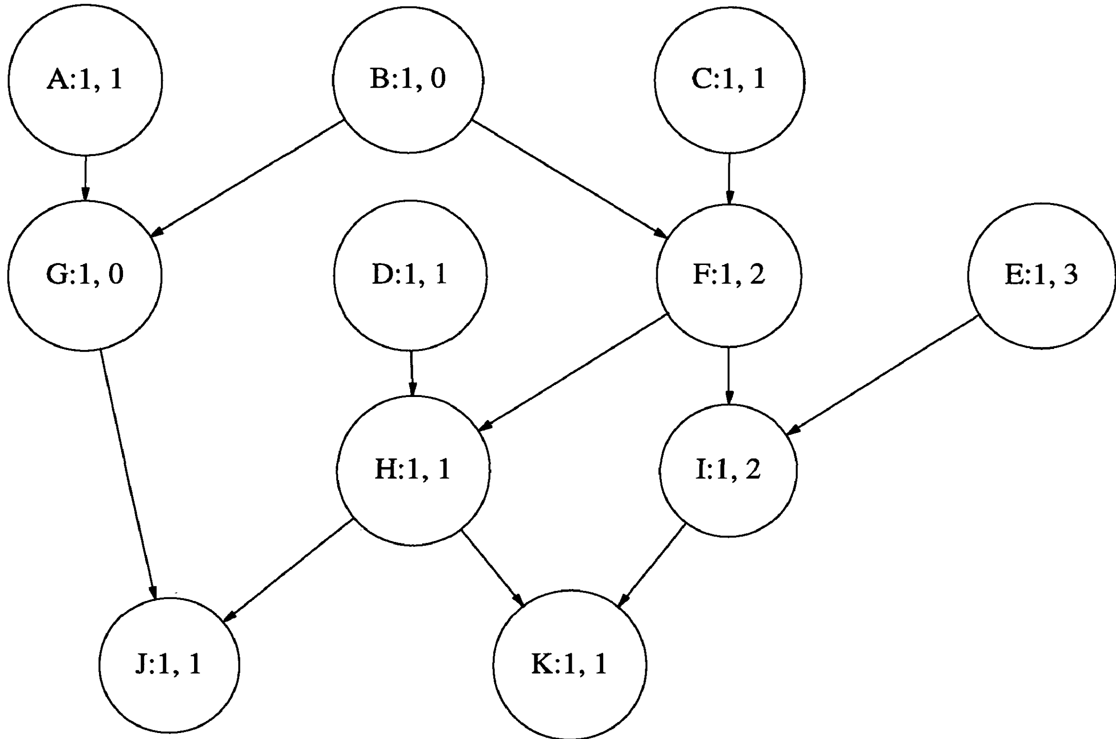


Figure 3-5 (ISCAS C17, GraphViz)

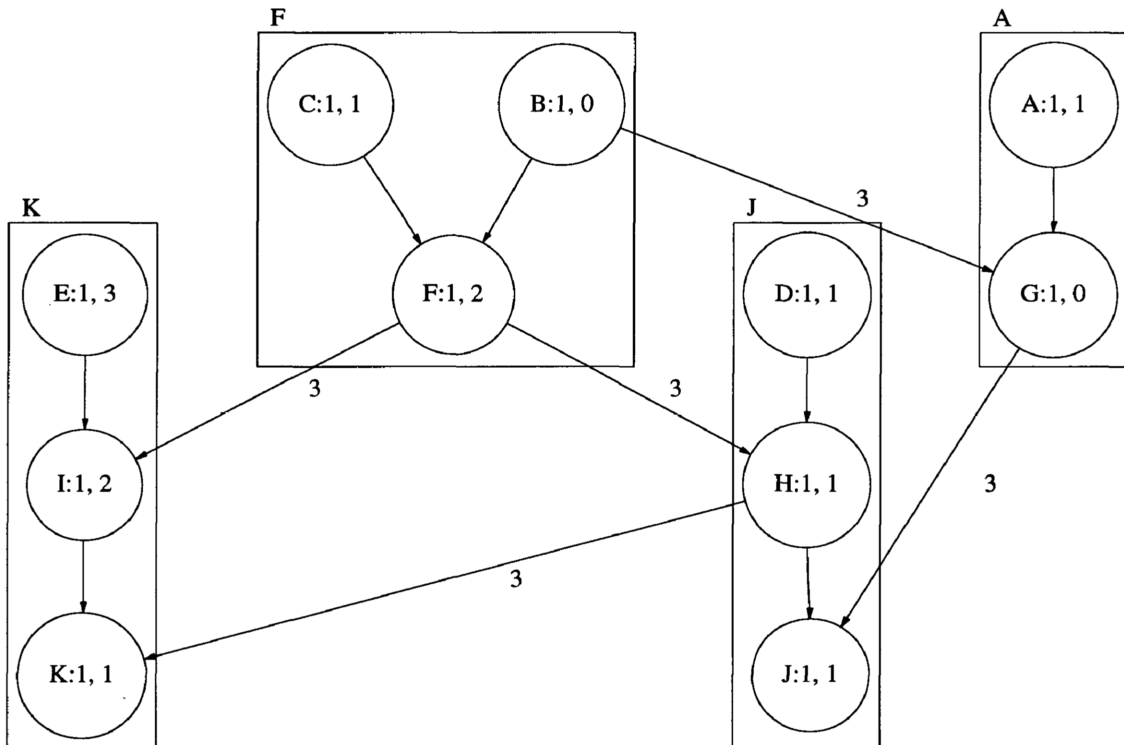


Figure 3-6 (Optimal C17 Partitioning, GraphViz)

The following images show the BPD results for |V| = 3 and D = 3, using the ISCAS C17 circuit. The intermediary stage of BPD, showing all the subgraphs generated prior to their final selection, is also provided. BPD's selection order of the pre-determined subgraphs was [J, K, F, D, G, E]. The subgraphs {A, B, C, G, H, I} were not used in the resulting partition. Note that vertices {A, B, G} are present in multiple subgraphs. This serves to reduce the delay, at the expense of circuit space. BPD achieves its delay minimization by ignoring the k-way limits and allowing replication, although it will observe $|V|$ and $|E|$ limits of a k-way partitioning.

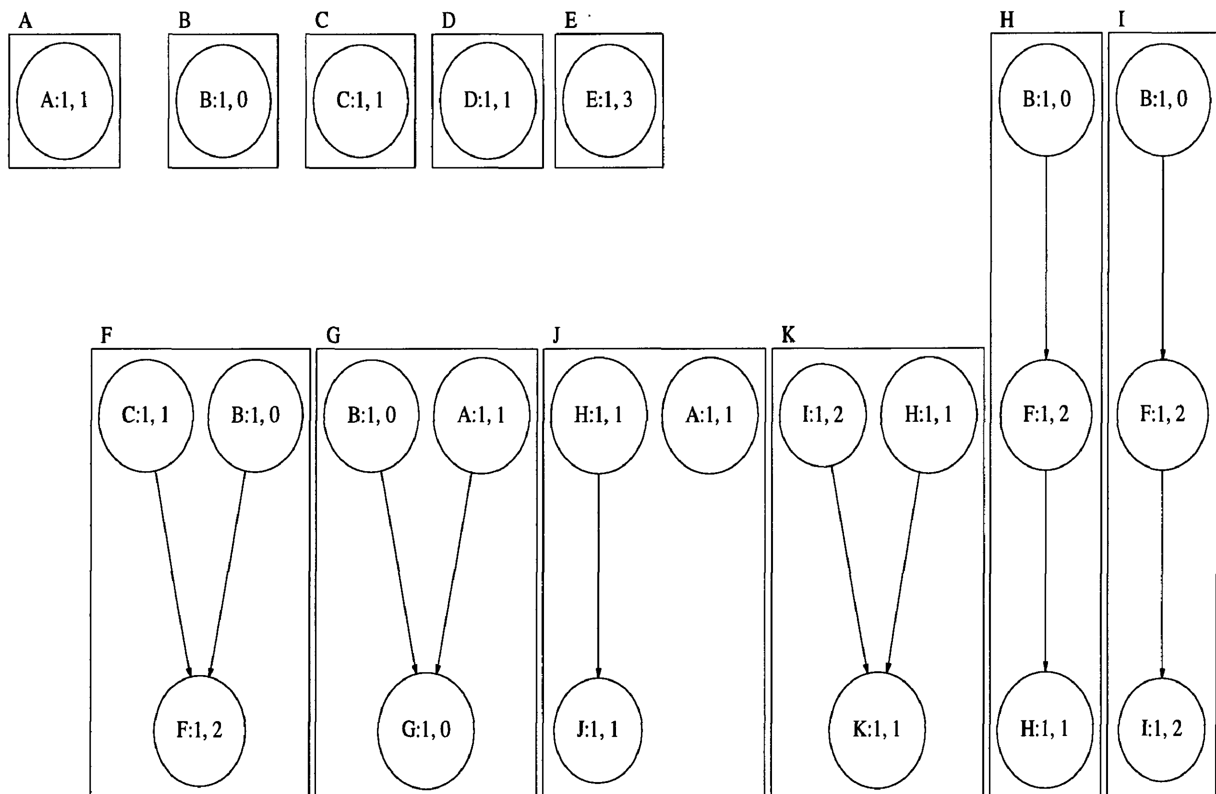


Figure 3-7 (BPD: Intermediate Solution)

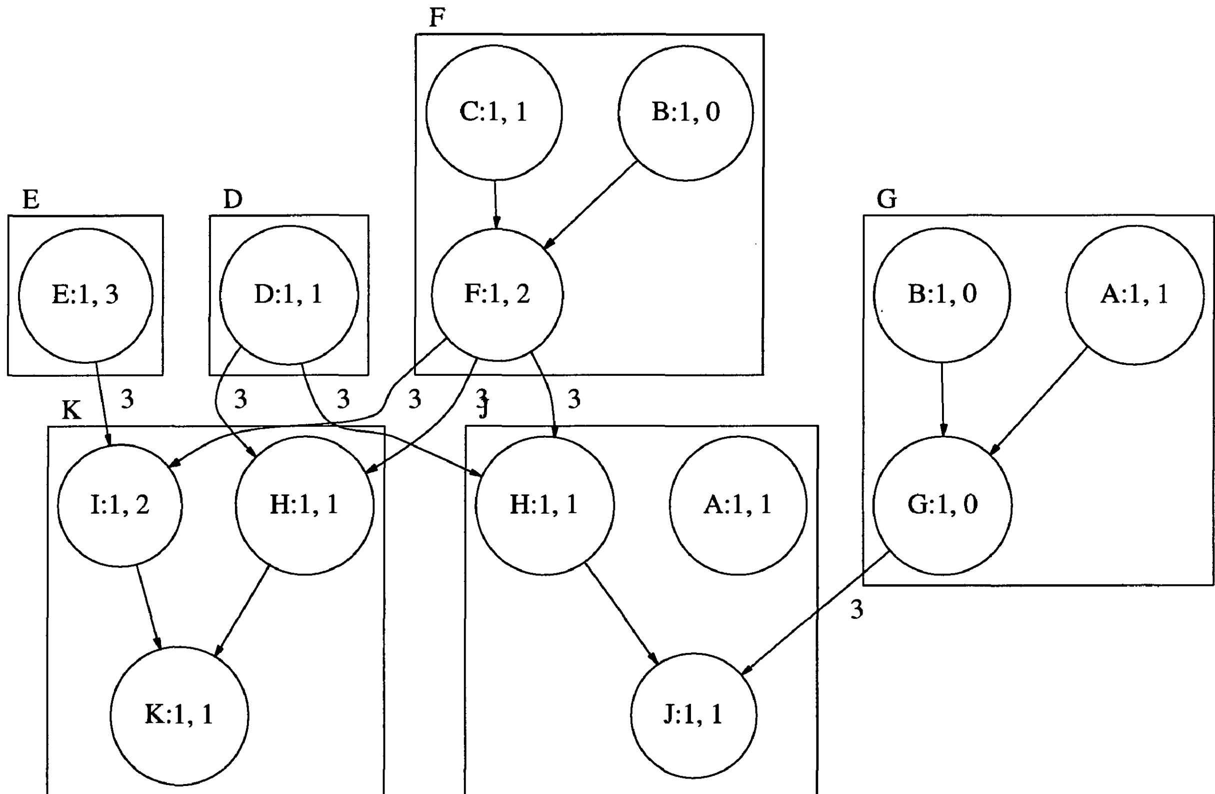


Figure 3-8 (BPD: Final Solution)

3.5. Partitioning in Parallel Processing

In the parallel processing domain, the mapping of terminology is similar to that of VLSI design. A task graph is a graph of the tasks and their dependencies. Tasks are equivalent to nodes and edges are synonymous with dependencies. The term grain is often used instead of sub-graph. The size of a grain is often compared to the amount of inter-processor communication. Fine-grained graphs tend to involve a greater amount of communication, while large grained graphs, at the risk of decreased parallelism, uses less communication. Granularity is given excellent treatment in the paper by Lewis and Kruatrachue, and in Kruatrachue's doctoral dissertation [49, 6].

Another significant point of Kruatrachue's research is that execution time partitioning may yield better results than an engineered decomposition by the programmer(s). A dynamic approach to increasing the granularity of tasks is presented by the 1991 paper of Mohr and colleagues [16]. They also advocate the automated approach over the programmer-engineered approach. However, they require the designer or programmer to *explicitly* determine what may be decomposed, while leaving the technique of the decomposition to the run-time environment. Their research also identifies various pitfalls of current dynamic methods.

Our selected algorithm, dominant sequence clustering, places heavy emphasis on both linearity and granularity of the partitioning. Recall that linearity dictates to what extent sequential tasks are being performed on an individual computational device. Non-linear, or independent circuits, may still be used in parallel processing. However, the sum of their execution must be less than or equal to the communication cost of waiting for another node to perform the work in parallel.

In other words, if the execution cost of two dependent tasks is 5 time units (TU), each and the communication cost is 8 TU, it would make sense to cluster these tasks together. If, however, the communications cost is 4 TU, it would be more appropriate to assign these tasks to different processors to maximize parallelism. The dominant sequence algorithm accommodates this by assuming there is an unlimited number of processing devices to allocate work to [3].

DSC functions in the following top-down manner. It assigns the computation that would suffer the maximum delay to the current CPU. Based on this assignment, it recomputes the new dominant sequence. Additionally, a free list of nodes whose ancestors are assigned, and a partially free list of nodes who have at least one ancestor assigned are updated with their new delay times. The algorithm then chooses the best node to assign from the free list to a processor.

This process continues, where nodes from the free list are iteratively assigned until all vertices are assigned to a subgraph (processor). When vertex allocation to the current processor would increase computation time, a new subgraph is created, as the algorithm is assigning these computations to a new CPU. However, DSC currently has no provision to replicate computations on a CPU. Furthermore, the algorithm is not guaranteed to partition the graph, as it may determine the best schedule is to have *one* processor perform all the computations. The following image shows the DSC results for the ISCAS C17 graph with D = 3.

The order of processing by DSC for this graph was [C, B, F, E, I, D, H, K, A, G, J]. Although not stated earlier, DSC's vertex processing sequence must be a topological sort. However, DSC may add a vertex to different subgraphs during each iteration. Note that the DSC algorithm does not observe $|V|$ limits. This is due to DSC's assumption that there are an infinite number of processors.

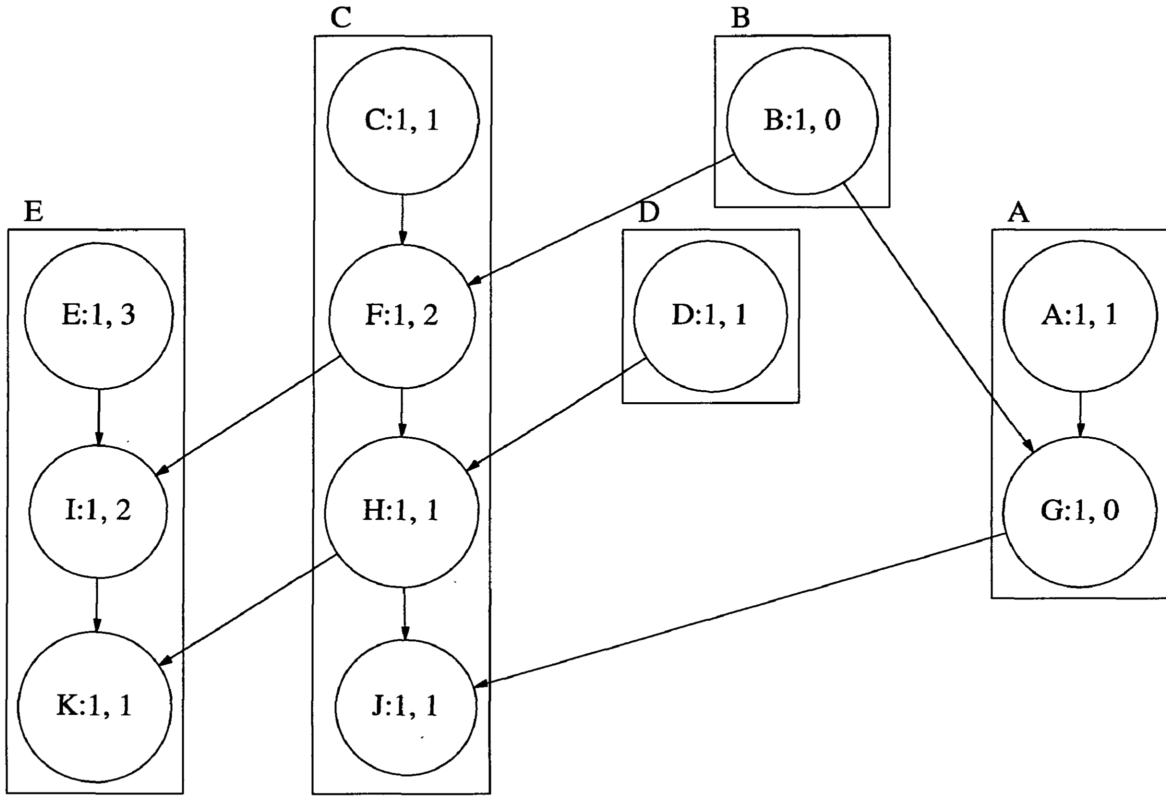


Figure 3-9 (DSC)

4. The AGPM and Developed Algorithms

Our historical development focuses on general graph partitioning principles, such as linearity and granularity, to demonstrate the key differences between our chosen domains of interest, VLSI circuit design and parallel processing scheduling. This chapter introduces the key concepts of the *AGPM* in light of this historical discussion. In addition, we propose two new algorithms, GEA, a genetic and evolutionary algorithm; and CTR, a graph-center determining algorithm.

4.1. Abstract Graph Partitioning Model

During his conclusion, Breuer states, "Most of the problems associated with the physical implementation of logic have been solved, though refinements of these solutions continue to be made". Given the more than 200 papers being produced in this arena as of Sherwani's 1995 text, there were a lot of refinements necessary, or not all the problems were solved. It is for this reason we propose the *AGPM*. The *AGPM*'s intent is to provide a framework for partitioning algorithm selection regardless of the application domain. We present both mathematical and language specifications that allow this decision-making process to occur.

Reasons of justification for the *AGPM* include the nature of graphs, precedents in other graph theoretic fields, and precedents within partitioning. The definition of a graph, in applied language may be defined as a set of objects and the relationships, or absence

thereof, between them. Thus, in the applied sense, a graph itself is an abstract model. It then follows that manipulators of an abstraction are themselves abstract [25: 1-29].

As for precedents in other graph theoretic fields, the recent text on automated graph drawing presents a means to classify graph drawing parameters and techniques [22]. Furthermore, the development of graph-theoretic approaches to circuit design is one of the primary factors leading to the developmental explosion of recent years. In partitioning literature, we observe several models, two of which prompted the creation of the *AGPM*, the UDM (Unit Delay Model) and GDM (General Delay Model), from the BPD algorithm [11, 18, 17, 40].

The UDM and GDM, however, are both designed for the VLSI design domain. The objective of the *AGPM* is to step back and define a model that may then be applied and extended for a particular domain. The *AGPM* is not an algorithm, but rather a framework to determine an appropriate algorithm. Sherwani suggests five key criteria when partitioning circuits [37: 148-150]:

1. Partition Number: number of partitions
2. Partition Size: maximum number of vertices/partition
3. Delay: delay between partition subgraphs
4. Partition Interconnections: number of edges between partition subgraphs
5. Terminal Count: area available for interconnections/device

All of these issues become metrics in the broader scope of the *AGPM*. The *AGPM* defines three broad classifications that define its framework:

1. Input Graph Metrics
2. Algorithm Control Parameters
3. Output Partitioning Metrics

Sherwani's criteria predominantly center on the latter two, with no development of the first classification we propose. Sherwani's criteria, along with the additional metrics shown in section 2.2.2, provide the *AGPM* framework. These categories allow full adjustment to all aspects of the graph partitioning process.

The *AGPM* is not self-limiting, but rather is considered an open architecture model. This allows inclusion of new metrics and exclusion or modification of existing metrics within the model. The metrics and control values of these classifications may be either real-numbered or Boolean values. For instance, we may classify an algorithm as "allows" or "does not allow" replication to occur. However, other metrics, such as the graph density, or $|E| / |V|$ (the number of edges relative to the number of vertices), results in a real-numbered value.

4.2. GEA: Genetic and Evolutionary Algorithm

Genetic algorithms are a considered to be a combination "...of directed [guided or controlled] and stochastic [random] search." [57: 16]. Genetic algorithms are considered extremely useful and robust in large, complex search spaces. [57: 16, 42: 18].

Application domains that have successfully used genetic and evolutionary algorithms include: knowledge base partitioning, game playing, scheduling, wire routing, map coloring and others [42]. The first successful genetic algorithm was used to model gas pipeline distribution control [42]. Genetic algorithm methods and terms are based on the biological world they imitate.

Genetic algorithms typically consist of the five components identified by Michalewicz. They are:

1. A genetic representation for potential problem solutions
2. Parameter values, e.g. population size and operator usage probabilities
3. A method to form the initial population (population seeding or creation)
4. Genetic operators that form a population's offspring
5. A fitness function, simulating the environment

These five components form the core of the genetic algorithm. The bulk our discussion involves the development of variants of each of these components. This discussion will also further illustrate how genetic algorithms simulate the biological life cycle.

The term genetic, or evolutionary, algorithm encompasses a broad range of stochastic biological-modeling algorithms. Typically, a solution instance is referred to as a chromosome, and each generation will contain a population of at least one chromosome. Classification of a chromosome is done via a schema that, during analysis is able to group various potential solutions. The initial population is generated either via a simpler heuristic method or random assignment. Operators in a genetic algorithm are then used to manipulate a population and form a new generation. Standard operators include mutation and crossover. A costing function is formulated to evaluate an individual based on various criteria.

Since genetic algorithms are designed to model the processes of life, we define an individual as a single problem solution. An initial pool of individuals, or population, is formed. This population forms a generation. Each generation is allowed to simulate life via a set of operators. These operators allow offspring to form via a variety of methods.

During each generation, a selection rule(s) is used to select the next generation's population. Individuals who are not selected are disposed, although most algorithms do not prevent a rebirth of these individuals via a future operator computation.

A biological species is considered to have a unique genetic makeup. This includes the number of chromosomes, the size of these chromosomes and the mapping of traits to various positions on these chromosomes. It is now well known that chromosomes consist of the arrangement of four nucleic acids in a sequential string pair. This protein encoding is then translated by cellular biology to control the organism's traits. In simple organisms, only a single chromosome is present.

This single chromosome model shall be used for our genetic algorithm. The encoding technique must be chosen as to suitable represent the problem, while being robust enough to allow operators to modify it in such a way as to create new and feasible individuals. Encoding techniques are typically a fixed length array of values. These values may be of various data formats, e.g. binary, integer, floating point, alphabetic or others. In addition, the order, distance and relative values of these array elements are important in various domains, depending on the encoding scheme. For instance, a scheduling algorithm would typically be concerned with the order of the array elements, as would the traveling salesman problem.

Historically, two encodings are typically used in partitioning: group number encoding and permutation with separators encoding. With group number encoding, the value of each

array position indicates the subgraph that vertex belongs to. From the following example, we have P = [AAB]. Vertices #1 and #2 are assigned to partition A, while vertex #3 is assigned to partition B. Permutation with separators encoding would encode this as [21_3] or [12_3].

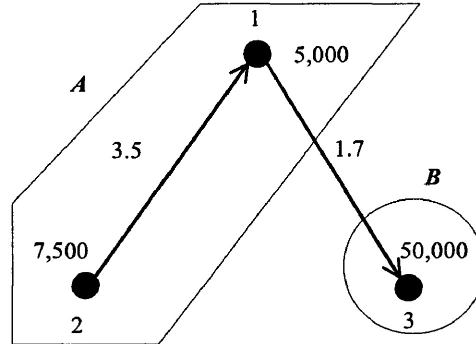


Figure 4-1

The separator used here, '_', could be any character, perhaps even the name of the subgraph, e.g. [12A3B]. Note that the order within the array range positions for a subgraph is irrelevant.

Our encoding scheme is a combination of both of these encoding schemes, called graph and group separator permutation encoding. We maintain a pair of arrays. The first array contains a permutation of the graph's vertices. The second array contains a list indicating the relative separator positions. By using the relative, rather than the absolute positions, we are able to use the same genetic operators on both arrays. For Figure 4-1, the encoding might be {[1,2,3], [2,1]}, {[2,1,3], [2,1]} or {[3,1,2], [1,2]}. This encoding allows easier implementation of partitioning constraints and operator flexibility.

Furthermore, the encoding allows us tremendous flexibility with k-partitioning variations. We may set ranges on the upper and lower limits for these $|V|$ limits of a subgraph, as long as the sum of these values is equal to $|V|$ of the input graph. Partition types include:

1. k-balanced: min = max - 1, max = $\lceil |V| / k \rceil$
2. k-min/max: (to force some usage) min = 1, max = $+\infty$
3. k-unlimited: (may be better not to partition) min = 0, max = $+\infty$

For instance, with the group number encoding method, we may have to scan the entire array to determine the vertices in a specific subgraph. Additionally, certain operators could corrupt an individual by violating the number of vertices in a specific subgraph. With the permutation with separators encoding, if one needs to extract the information for a specific subgraph, a scan of the array must be conducted, as the separator positions are unknown. Also, as with the group number encoding, special checks have to be incorporated into the operators, lest they violate partitioning constraints.

The optimal ISCAS 17 partitioning better illustrates the different encoding methods:

ISCAS 17 Vertex List	[ABCDEFGHIJK]
Group Number Encoding	[12234213434]
Permutation w/Separators	[AG;BCF;DHJ;EIK;]
Graph & Group Separator Permutation	([AGBCFDHJEIK], [2333])

We can see how group number encoding places the corresponding sub-graph indicator in the respective vertex position. Permutation with separators reorders the vertices and then adds separators in between subgraphs. The Graph and Group Separator Permutation is similar, however, it places the sizes of the subgraphs in a separate array.

Although the permutations show the vertices ordered within a subgraph, this is not necessary. It is only done to facilitate comparing the encodings. As we can see, to determine the vertices in the third subgraph we would have to do the following:

Group Number Encoding:	Scan entire array for the value '3'
Permutation w/Separators:	Scan array for 3rd ';' separator
Graph & Group Separator Perm.:	Values before 3rd position of 2nd array

Furthermore, if we randomly change a value (e.g. mutation) the following might occur:

Group Number Encoding:	Only swap, as might violate $\lvert V \rvert$ limit
Permutation w/Separators:	Separators swap may corrupt array
Graph & Group Separator Perm.:	Can swap any values in either array

These limitations make it difficult to manipulate the vertices in a specific subgraph, even within the $\lvert V \rvert$ constraints for the specific *k*-way partitioning being solved. Graph and Group Separator Permutation encoding does so, with less time and space requirements.

Now that a suitable encoding scheme has been established, it is necessary to form a population. Various methods have been suggested, however, random creation of individuals has turned out to be the most useful [51, 34]. With group number encoding, a value within the subgraph range is randomly assigned, assuming the $\lvert V \rvert$ limit has not been violated. For permutation with separators, a random vertex permutation is obtained and separators are inserted at legal positions within the array.

A similar operation is used for our encoding technique, where the vertex list is a permutation of the sorted node name list and the M-values are stored in a second, independent array. Additionally, one individual in the genesis population was a topological sort of the input graph. A topological sort of a DAG ensures that all ancestors

in a graph are listed before all predecessors. For ISCAS C17, example topological sorts include [ABCDEFGHIJK] and [BCDEAFGHIJK]. As seen here, there *may* be multiple, permissible topological sorts for a given DAG.

One parameter in genetic algorithms is the population size. The population size affects a number of issues. These include the number of offspring and the number of generations. In our algorithm, population size was set based on two factors, $|V|$ and k. This allowed the number of vertices and partitions to both affect the population in a controlled, monotonically increasing manner. The following growth plot shows $P = \log_2(|V|) * \log_2(k)$, where $1 \le |V| \le 1024$ and $2 \le k \le 16$:

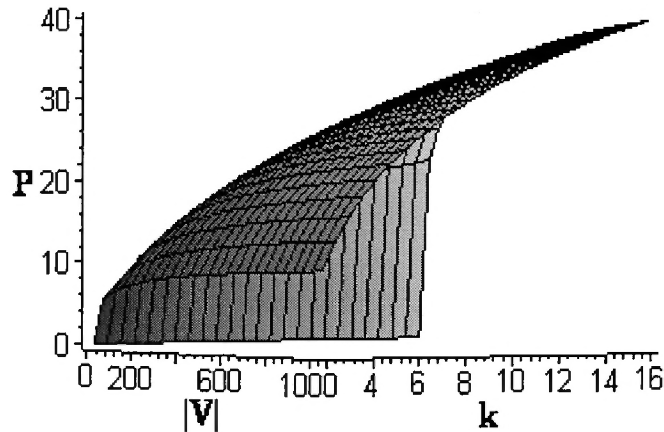


Figure 4-2 (Image by Maple V)

Considering that the number of unique combinations for a 1,024 element object divided into 16 objects is somewhere around 90 digits, this is a small population, yet is still able to generate reasonable solutions. We wrote a routine to iterate over all possibilities, however, given the combinatorial explosion in solutions, it is only good for $2 \le |V| \le 20$. This routine was used to generate the optimal ISCAS C17 solution discussed earlier.

Once an initial population is created, some form of generation controls must be used to limit how long the algorithm will execute. This control is typically either a measurement of CPU or the number of generations actually created. In our algorithm, $G_n = \log_2(|V|) * \log_2(|E|)$, where $4 \leq |V| <= 1024$ and $(|V| - 1) \leq |E| \leq (|V| * (|V| - 1) / 2)$. The following plot depicts the number generations based on a graph's |V| and |E| metrics.

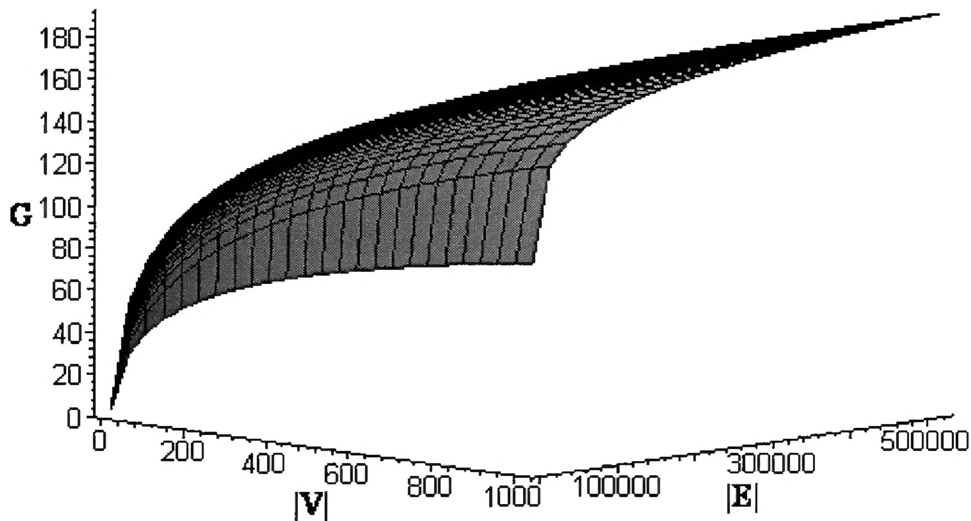


Figure 4-3 (Image by Maple V)

Essentially, we desired to base the number of algorithm iterations on the density of the graph, while keeping the computation time to a reasonable level, especially as $|V|$ and $|E|$ become large. We also allowed a consecutive generation non-improvement stopping value, but for our experiments, set it equal to the maximum number of generations.

Now that the encoding method, population size and generation sizes have been established, the algorithm will need techniques to create offspring from the population. These techniques are formally referred to as operators. Operators are designed to manipulate one or more individuals to generate one or more offspring. Genetic operators

include various crossover routines and mutations. We also introduce migration as a unique operator.

Operators are also rated on an intelligence, or domain awareness scale. A low-awareness operator will simply compute a random operation, i.e. base its change on a pseudo-random number generator. More intelligent operators will incorporate the constraints, parameters and goals of the final solution. They may include macro and micro-levels of information. Furthermore, the incorporation of this domain-awareness may be explicit or implicit. [57, 42, 23].

The mutation operator used, PairSwas, had low domain-awareness knowledge. PairSwap allows specification of two mutation rates, V_r and k_r, on a single parent and generates a single offspring. V_r controls the number of random vertex swaps obtained. K_r was specifically created for the new encoding method. It allows specification of the number of random subgraph size swaps. Once any pair of elements, vertices or subgraph sizes, were swapped, they are eliminated from further mutation. This limits the mutation rate to 50%, although we used much lower than this. An example mutation, with $V_r = k_r = 0.50$, is:

P:	([BCDEAFGHIJK],	[3323])	(Topological sort for ISCAS 17)
C:	([BCDEAFGHIJK],	[3323])	Pool: [A**B**CDEFGH**I**JK]
C:	([**I**CDEAFGH**B**JK],	[3323])	Pool: [AC**DE**FGHJK]
C:	([IC**ED**AFGHBJK],	[3323])	Pool: [**A**CF**G**HJK]
C:	([ICED**G**F**A**HBJK],	[3323])	Pool: [C**F**H**J**K]
C:	([ICEDG**J**AHB**F**K],	[3323])	Pool: [**CH**K]
C:	([I**H**EDGJA**C**BFK],	[3323])	Pool: [K], Done w/vertex mutations
C:	([IHEDGJACBFK],	[3323])	Pool: [**3**3**2**3]
C:	([IHEDGJACBFK],	[**2**3**3**3])	Pool: [**33**]
C:	([IHEDGJACBFK],	[2**3**3**3**])	Pool: [], Done with $\|V\|$ mutations
C:	([IHEDGJACBFK],	[2**3**3**3**])	(Final offspring)

As the example shows, if a pair is used in a mutation, it is removed from the mutation pool, thus allowing each vertex to move only once in a generation.

The intent of mutations is to fine-tune an existing good solution, while the disruption caused by this example mutation is rather large. Crossover or other operators normally handle large disruptions in genetic material. The primary intent of crossover is to merge pieces of existing high-quality solutions, with the hope they will produce good, "healthy" offspring.

We selected two existing crossover methods to use with our encoding technique. Both crossover operators have been used with graph partitioning using other encoding methods [57:258, 42:110-111]. The two crossover operators selected were partially mapped crossover (PMX), and order crossover (OX). These particular crossover operators incorporate the desired disruptiveness sought after with crossover operators, while generating feasible offspring.

Both PMX and OX use two parents to generate offspring. Both operators then select two random cut points. Data between these two cut points is then inserted in the same position into the empty offspring arrays. The PMX operator then adds any vertices in P_2 not already in C_1. Remaining conflicting vertices are then added based on their mappings between C_1, P_1, and P_2. The M-values for the subgraphs were assigned based on the parent generating more genetic material to a child. The following example should clarify this procedure.

```
P1 = ([ABCDEFGHIJK], [3323])
P2 = ([BCDEAFGHIJK], [3233])

Cutpoint1 = 3, Cutpoint2 = 8
Cutsize = 5, |V| = 11, thus Ci[M] = Pi[M]

C1 = ([xxxDEFGHxxx], [3233])
C2 = ([xxxEAFGHxxx], [3323])

C1 = ([BCxDEFGHIJK], [3233])
C2 = ([xBCEAFGHIJK], [3323])

C1 = ([BCADEFGHIJK], [3233]), D -> E, E -> A
C2 = ([DBCEAFGHIJK], [3323]), A -> E, E -> D
```

The OX operator is similar in operation, however, as its name implies, it desires to preserve the order of vertices from the second parent. After exchanging the data between the cut points, starting to the right of the cut point, data is copied, looping around when the end of the chromosome is reached. Vertices already present in the offspring are skipped. A similar example, for the OX operator and using the same parent strings, follows:

```
P1 = ([ABCDEFGHIJK], [3323])
P2 = ([BCDEAFGHIJK], [3233])

Cutpoint1 = 3, Cutpoint2 = 8
Cutsize = 5, |V| = 11, thus Ci[M] = Pi[M]

C1 = ([xxxDEFGHxxx], [3233])
C2 = ([xxxEAFGHxxx], [3323])

C1 = ([BCADEFGHIJK], [3233])
C2 = ([BCDEAFGHIJK], [3323])
```

Comparing the two operators, we see it is possible for them to generate the same answer.

The low-rate mutation operator, coupled with the partial order preservation characteristics of the crossover operators, can lead to population stagnation, where any of these operators tend to only create offspring that are relatively close in nature. This problem can be compared to the issues raised with inbreeding of a particular species strain. To alleviate this stagnation and solution localization, we introduce the "migration" operator.

Recalling how the original population is created, we decided it was best to simply randomly generate a set of individuals. Within each generation, we simulate individuals migrating to the existing population by simply creating new ones. They are created in the same random manner as the original population. This process may actually be reworded as a worst-case 100% mutation. In other words, it may be viewed as generating a permutation of any individual. As with mutation and crossover, a ratio operator relative to the population size also controls migration.

We now have the capability to encode a solution, create a population, control the number of generations and generate these generators via operators. We must know establish a method of evaluating these individuals. This method is normally called a costing or fitness function. It determines, based on a defined environment, the fitness of a particular individual. In other words, if we had two fish, one freshwater and one saltwater, they would score different fitness values based on the environment's salinity.

Some genetic algorithms evaluate fitness explicitly and some evaluate it implicitly. An explicit fitness function would not allow the freshwater fish to be included in the next

generation; if it indicated it could not handle the salinity level of the current environment. It would not allow any accommodation for how much salinity it could handle. An implicit fitness function, on the other hand, would allow it to survive, however, it would rate it poorly.

The GEA algorithm uses a 3-valued implicit fitness function based on the edge cut, maximum delay and subgraph vertex weights. For each of these values, a maximum and contribution factor value was determined. The sum of the three contribution factors must equal one. Furthermore, if the actual divided by the maximum was greater than one, this value was added to the function, otherwise the product of the ratio and contribution factor was added. The formal equation is:

$$G_c = \begin{cases} C_a/C_m > 1, C_a/C_m \\ C_a/C_m \le 1, C_a/C_m * C_f \end{cases} + \begin{cases} D_a/D_m > 1, D_a/D_m \\ D_a/D_m \le 1, D_a/D_m * D_f \end{cases} + \begin{cases} W_a/W_m > 1, W_a/W_m \\ W_a/W_m \le 1, W_a/W_m * W_f \end{cases},$$

$$C_f + D_f + W_f = 1$$

Assuming an individual is fully feasible and meets the environmental or domain constrains, the value of $G_c <= 1$, with smaller values indicative of better solutions. This cost function allowed us to experiment with VLSI Design, Scheduling, Equal Piles, and Map Coloring problems. The equal piles version was used to determine experiment distributions on different machines, based on the number of vertices in the graph.

After evaluating a population, or at least new offspring formed during a generation, we must have some method of selecting a new population. One of the primary selection methods is elitism. Based on elitism, we select a certain number of individuals based on

the allowed population size. The remaining individuals are selected via a number of methods. In GEA, we also specify a parental selection rate that allows us to force a number of parents selected, with the remaining individuals chosen from offspring not selected during the elitism phase.

The following table and drawing depict the solution obtained by GEA for the ISCAS C17 circuit. You may observe the differences of the three encoding schemes rather readily. Although the new encoding scheme takes up more memory, if an array data structures is used, information on the partition is much more readily accessible. Additionally, no conversions are necessary to identify what vertex is in what partition. It is also not necessary to scan the entire array to determine a partition's members.

ISCAS 17 Vertex List	[ABCDEFGHIJK]
Group Number Encoding	[12234313434]
Permutation w/Separators	[AG;BCF;DHJ;EIK;]
Graph & Group Separator Permutation	([AGBCFDHJEIK], [2333])

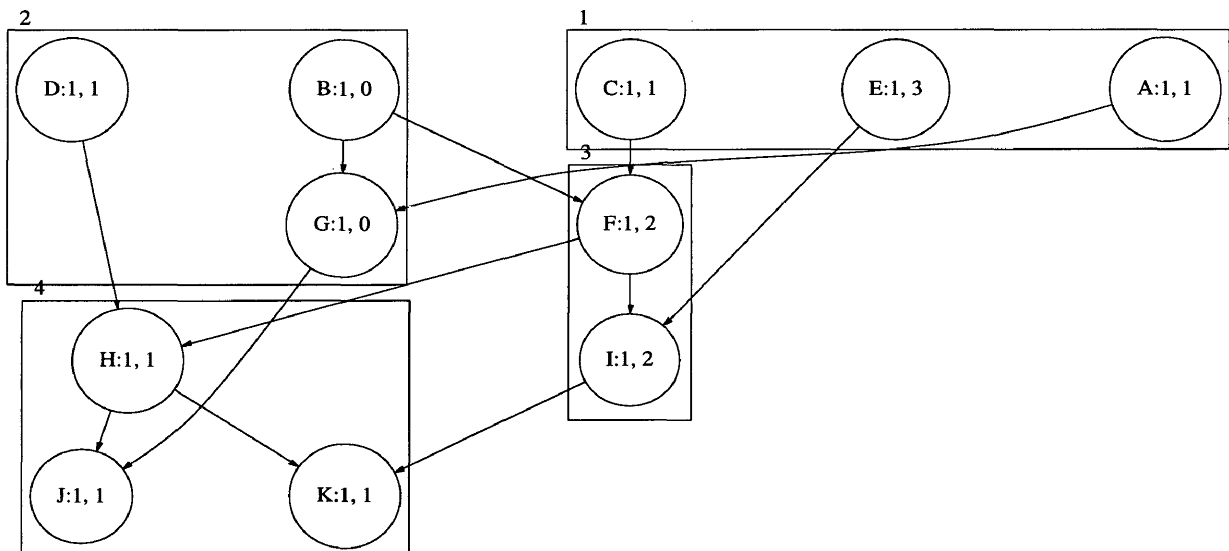


Figure 4-4 (GEA Partitioning of ISCAS C17)

4.3. CTR: Center-Based Algorithm

Our second algorithm, CTR, is based on graph center metrics. The proposed algorithm was a recursive algorithm involving determining a graph center, extract a subgraph based on this center, and repeat this process until all vertices were allocated to a subgraph. However, after our historical literature review, we decided to extend the efforts of F. Harary and F. Buckley [20] and C. Lenart [10]. To understand the approach, we shall first define several center-related concepts.

The intuitive idea in CTR is suppose a graph is a pictorial representation of some physical object and the vertices are potential grip points. A balanced k-center is a set of vertices in this object, which if physically grasped at these k-center grip points, balances the load. For example, a 4-center of a large box might be the four corners of the box. Similarly, if the vertices in four subgraphs of a partitioning were all "close" to a pre-determined vertex in their respective subgraphs, the partitioning would be a good partitioning.

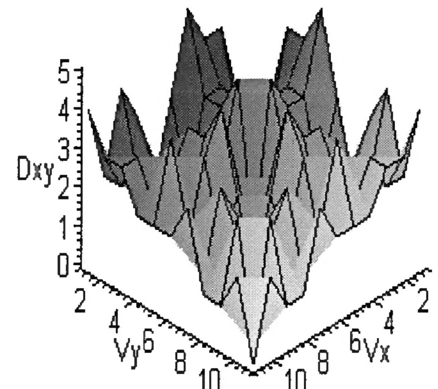
Center distances may be measured in a several means. We shall concern ourselves with two specific measurements, eccentricity and status. Eccentricity of a vertex is defined as the longest shortest path from the vertex, v, to all other vertices in the graph. The eccentricity is often provided in tabular format. Each row and column intersection defines the distance between that pair of vertices. The status is the sum of the eccentricities for a given vertex, v.

A k-center is defined as the set(s) of vertices whose number of elements is k, and whose eccentricity is equal to the minimum eccentricity, or radius of the graph. For instance, a 3-center is the set of all sets of combinations of 3 vertices with the lowest eccentricity. A 1-center is the set of all nodes with the lowest eccentricity. Groupings based on status are compared to the summed eccentricity of the subgraphs. Theoretically, by grouping nodes "closest" to each other together, the delay will remain lower. Putting the steps together:

1. Set edge values to the sum of the end vertex delays and the inter-partition delay
2. Generate all-pairs shortest path of underlying simple (undirected) graph
3. For each combination *S* of C (V, k) do
    a. For each V_n in *S* create a subgraph populated by V_n
    b. For V_i in V_n / *S* do
        i. Find minimum distance to all members of *S*
        ii. Add V_i to subgraph (G_n) with min (d_t) & min ($|V|$)
    c. Verify k-constraints for each G_n (optional)
        i. M ($|V|$)
        ii. C ($|E|$ = sum (inputs, outputs, & edges))
    d. Add inter-partition edges

For example, the following would be a result of finding the 4-center of the ISCAS 17 Graph, assuming M = 3, C = 5, and D = 3. The minimum eccentricity for a max (4-center) is 19. This is the first value for which at least 4 vertices have an eccentricity that low.

	A	B	C	D	E	F	G	H	I	J	K	$\sum d_t$	RMS
F	12	5	6	11	15		8	6	7	11	11	15	9.7
B	7		11	16	2	5	3	11	12	7	16	20	10.1
H	13	11	12	5	19	6	9		11	5	5	19	10.5
J	8	7	17	1	24	11	4	5	16		1	24	11.8
K	18	16	17	1	14	11	14	5	6	1		18	12.0
G	4	3	14	14	23	8		9	15	4	14	23	12.3
I	19	12	13	16	8	7	15	11		16	6	19	13.0
D	18	16	17		24	11	14	5	16	1	1	24	14.3
C	18	11		17	21	6	14	12	13	17	17	21	15.1
A		7	18	18	27	12	4	13	19	8	18	27	15.8
E	27	2	21	24		15	23	19	8	24	14	27	19.2



Note [A = 1, B = 2, ... K = 11]. The following pages are results for ISCAS C17 using the described CTR variants (RMS represents Root Mean Square, or the square root of the average of the squared data values):

Eccentricity Metric	**Name**	**Abbreviation**
min ($\sum d_t$)	Summed Minimum	**SIN**
max ($\sum d_t$)	Summed Maximum	**SAX**
min (RMS (d_t))	RMS Minimum	**RIN**
max (RMS (d_t))	RMS Maximum	**RAX**

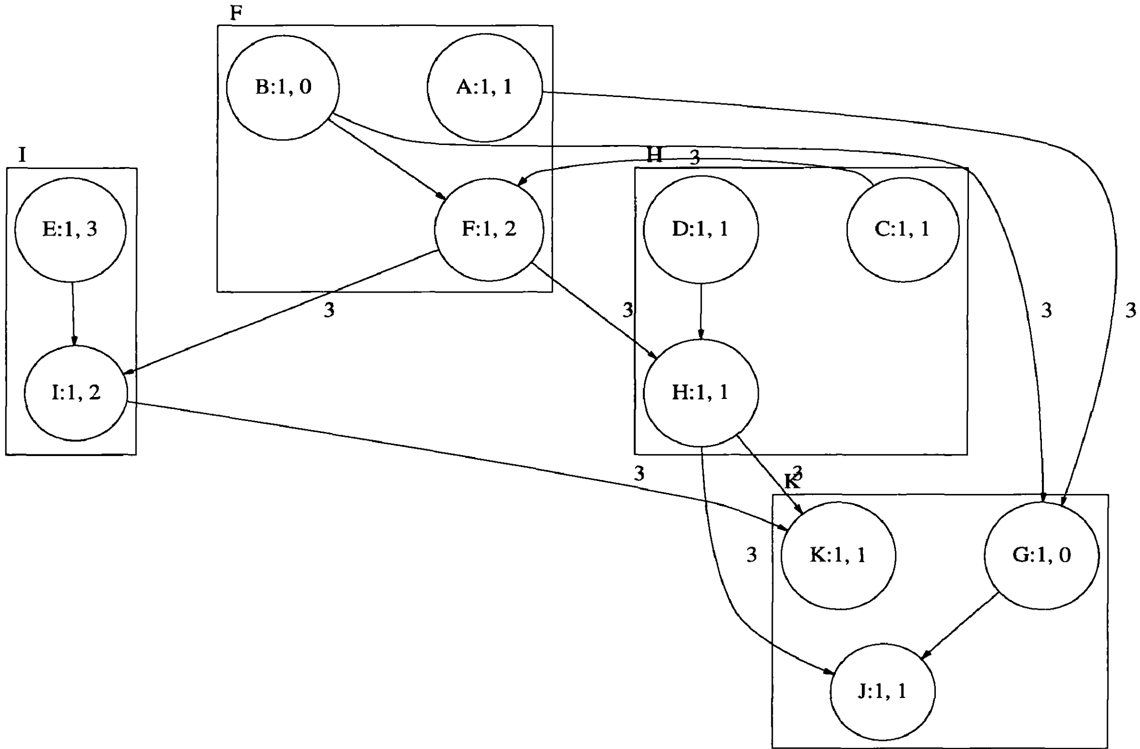


Figure 4-5 (SIN: Summed Minimum Eccentricity)

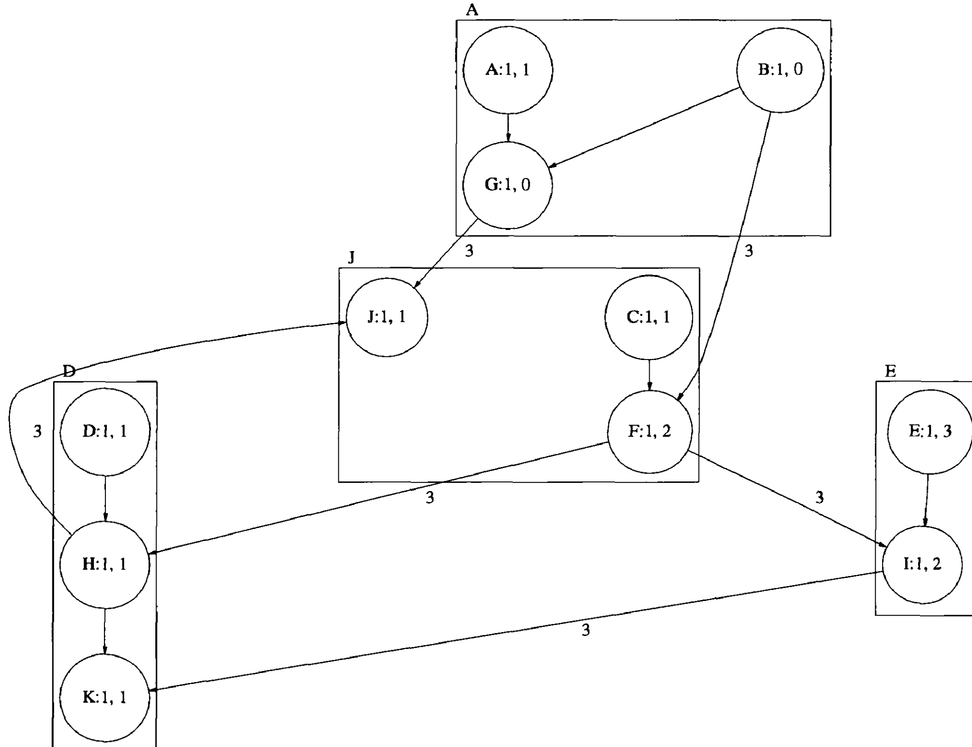


Figure 4-6 (SAX: Summed Maximum Eccentricity)

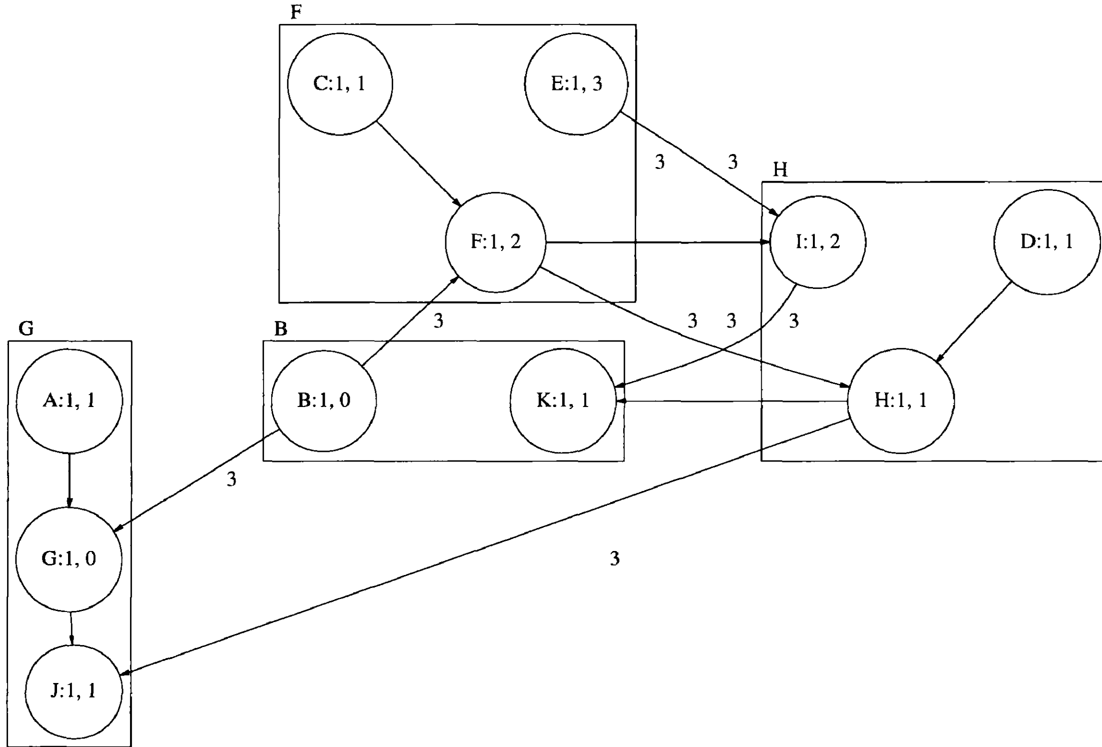


Figure 4-7 (RIN: Root Mean Square (RMS) Minimum Eccentricity)

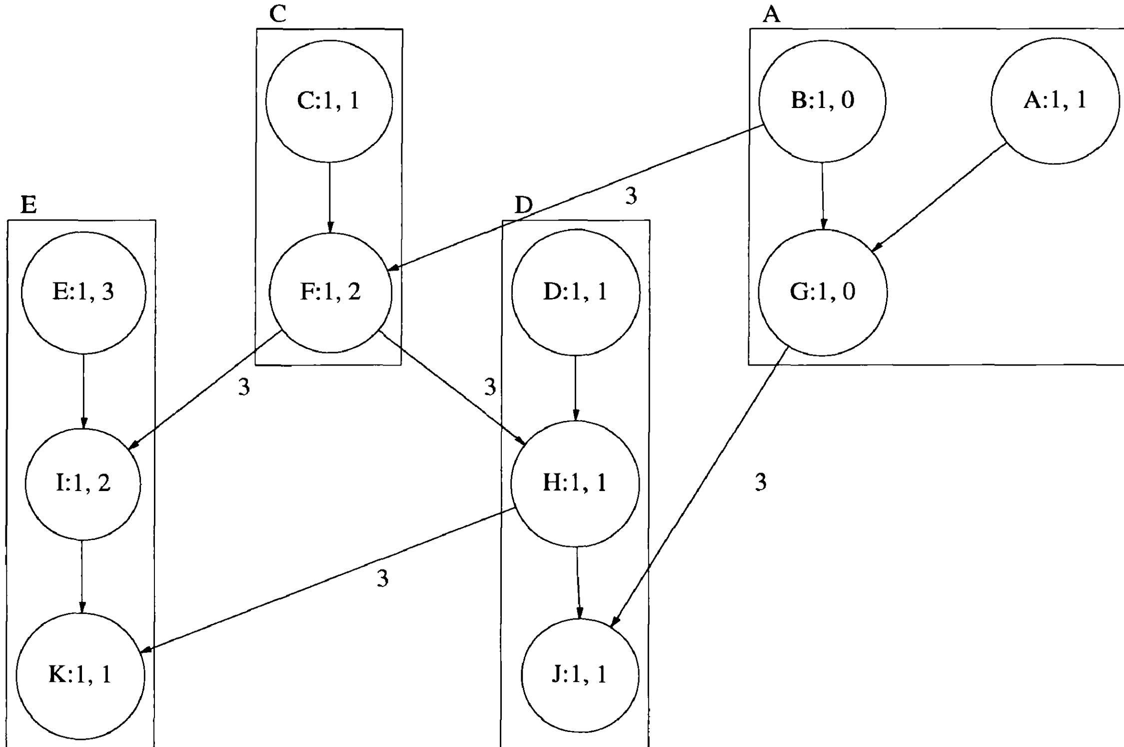


Figure 4-8 (RAX: Root Mean Square (RMS) Minimum Eccentricity)

4.4. RAN: Random Iteration Algorithm

The random iteration algorithm is rather simple. Its' function is to serve as a control algorithm for the other algorithms under study. The encoding scheme we created for the GEA algorithm is used as the underlying solution format. However, instead of maintaining a large population and generating offspring, a random permutation string is selected every iteration, to include the initial string. This is compared to the current best solution found. The iterations continue until either a specific number of iterations have been achieved or a time limit observed.

After conducting the runs of all the comparison algorithms under study, RAN is set to execute under a given scenario for the root mean square (RMS) execution time of the other algorithms. Other time limits are possible, however, this seemed the most suitable. A minimum of a single iteration will always be conducted.

As a side note, the RAN algorithm is essentially GEA with a generational time limit, population size of one migration of one, mutation and crossover offspring rate of zero and elitism rate of 1.0. RAN also uses the same evaluation criteria as GEA, and thus has similar flexibility when it comes to potential application domains. Samples of RAN output are provided in the results analysis.

5. Results Analysis

5.1. Overview

Our results encompass a survey of graph visualization tools, along with in-depth discussion of the experimental graphs used and the partitioning results. During the original work done on the Best Predecessor algorithm, other than manually drawing the results, we had no means of visually depicting the results obtained. Additionally, experiments were conducted on a large cross-section of related graph families, representing a number of domains. Primary study was focused on graph families representative of the VLSI design and parallel processing domains.

During the intermediate stages of our research, we had attempted to develop a Tcl/Tk GUI applet. The intent behind this applet was to link the graph generators, text editing of the input and output files, and a graph visualization tool, or GVT. While the first two objectives were obtained, development of a GVT was well beyond the scope of this research. However, a need still remained to directly view the resulting partitions.

We focused considerable attention on utilization of existing GVTs. Several GVTs were considered, to include on-line Java, UNIX, Macintosh, and DOS-based applications. In the end, we settled on three specific GVTs. These three applications are:

1. *GraphViz*: industry research automated graph drawing tool [47]
2. *GraphLet*: university research Tcl/Tk application [36]
3. *Tom Sawyer Software*: industry commercial product [54]

These tools greatly facilitated our understanding of both the input graphs and the resulting partitions. Additionally, they helped guide intuitive insights in development of the AGPM.

These tools represent a cross-section of graph visualization and graph theory. *GraphViz* provides both command-line and GUI modes. The GUI is built on a proprietary, but extensible scripting language. It is made freely available by a commercial research laboratory. Although the GUI required a 3-button pointing device, e.g. mouse for full functionality, it still offered the poorest user interface. However, the input file format was extremely robust, although it did not permit pre-determined vertex positions. If a desired graph was obtained, it had to be saved in either postscript printer or screen bitmap formats. *GraphViz* is the only application to provide native partitioning support, while offering reasonable performance in both speed and the size of graphs it is able to handle. It became our tool of choice for all post-partitioning analysis.

GraphLet, like *GraphOp*, is a Tcl/Tk applet. It combines compiled applications with the graphic scripting capability of Scriptics' Tcl/Tk. This tool offered the broadest number of automated graph drawing algorithms, along with a number of other graph metric and analysis tools. It is developed under a continuing project at the University of Passau, Germany. The researchers at this institution are also assisting in the development of a standard graph description language. However, this tool offered the slowest performance and no post-partitioning drawing capability.

The commercial package, *Tom Sawyer*, was not a pre-packaged application but rather a set of Application Programming Interface, or API, calls, offered with both a C/C++ and Java Software Development Kits (SDK). This enables extensive end-user customization. Budget restrictions for our research limited our evaluation of Tom Sawyer to a demonstration developed with these APIs by Tom Sawyer Software Company. The GUI was complete and functional, while the layout algorithms were the fastest observed. However, partitions were only indirectly supported, while graph files were several times larger relative to the others.

Our primary development environment, *Maple V*, did provide rudimentary graph drawing support. It provided a choice of a single concentric arrangement and layered concentric or linear arrangements. However, the automation level was low, as it was necessary for us to specify what vertices would be in what layer and their order of appearance within each layer.

The images below show the ISCAS C17 graph as drawn by *Maple V* and the three GVTs. Although difficult to see, *Maple V* provides a fair amount of control over the resulting image, a large portion of it must be done via user commands, e.g. the rotate command was used to obtain the image in the *Maple V* sample. The differences in the capabilities of the other GVTs are readily apparent. Although *GraphViz* was our final choice, the interface requires a high degree of user knowledge. For example, one must be cognizant of the node property names when working with *GraphViz*, while *GraphLet* and *Tom Sawyer* provide property templates.

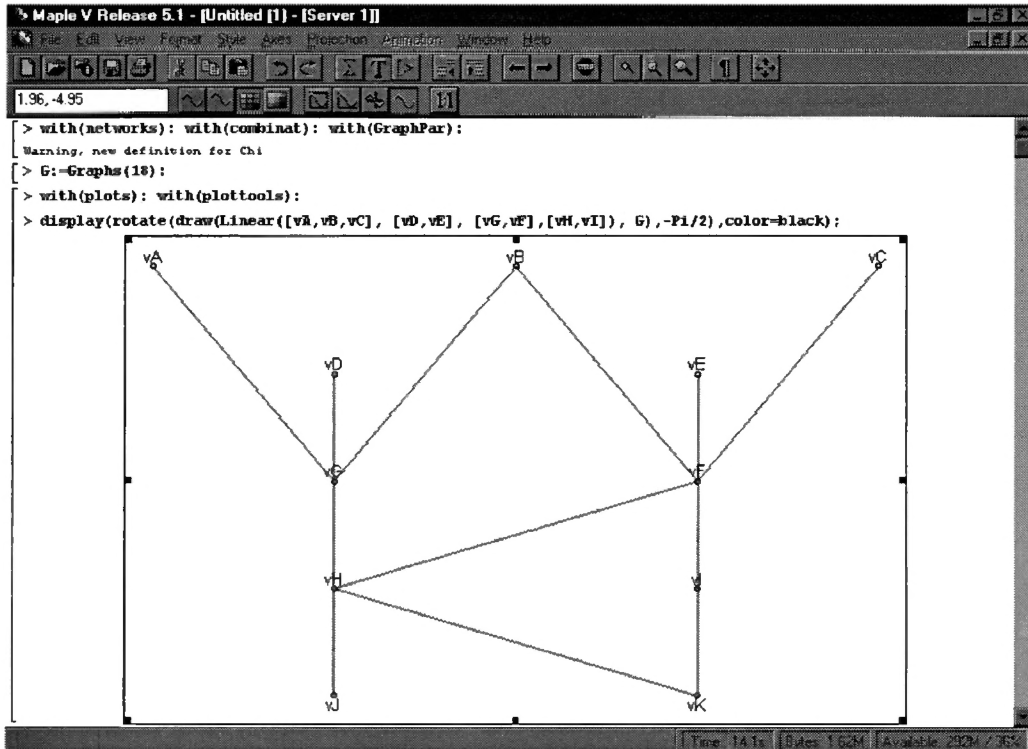


Figure 5-1 (MapleV)

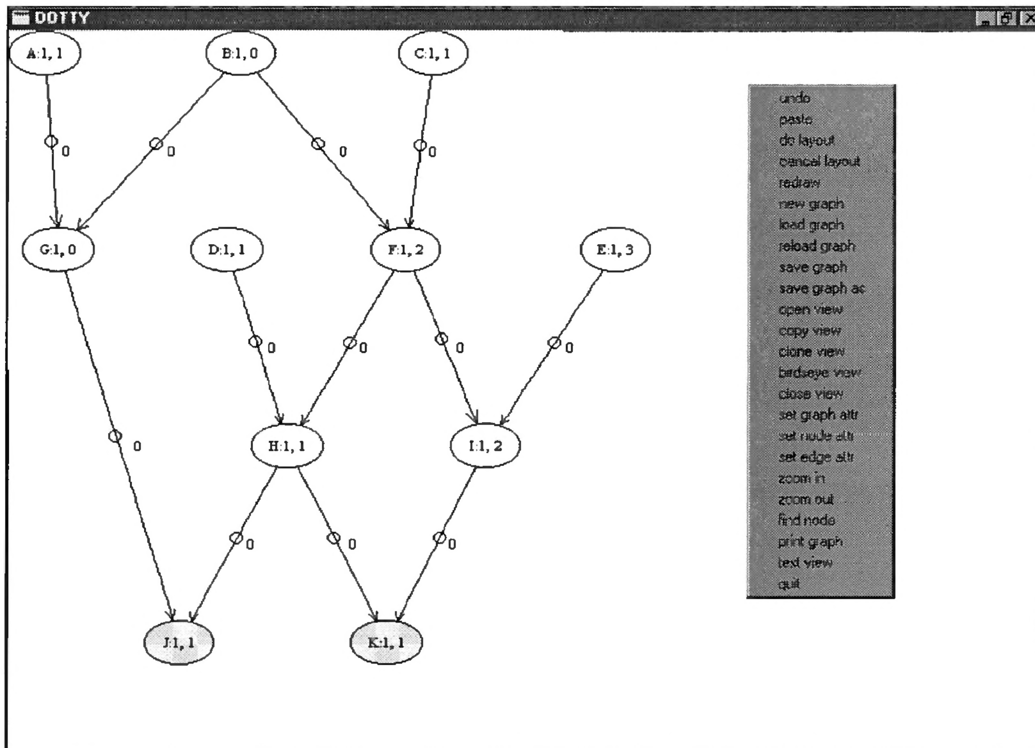


Figure 5-2 (GraphViz)

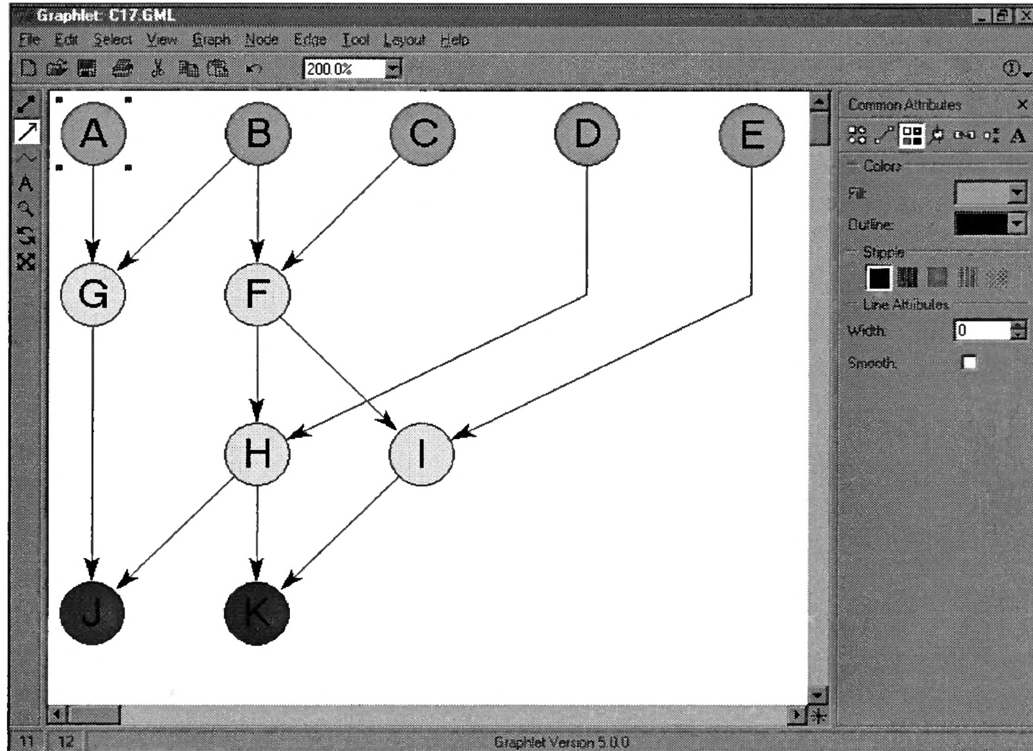


Figure 5-3 (GraphLet)

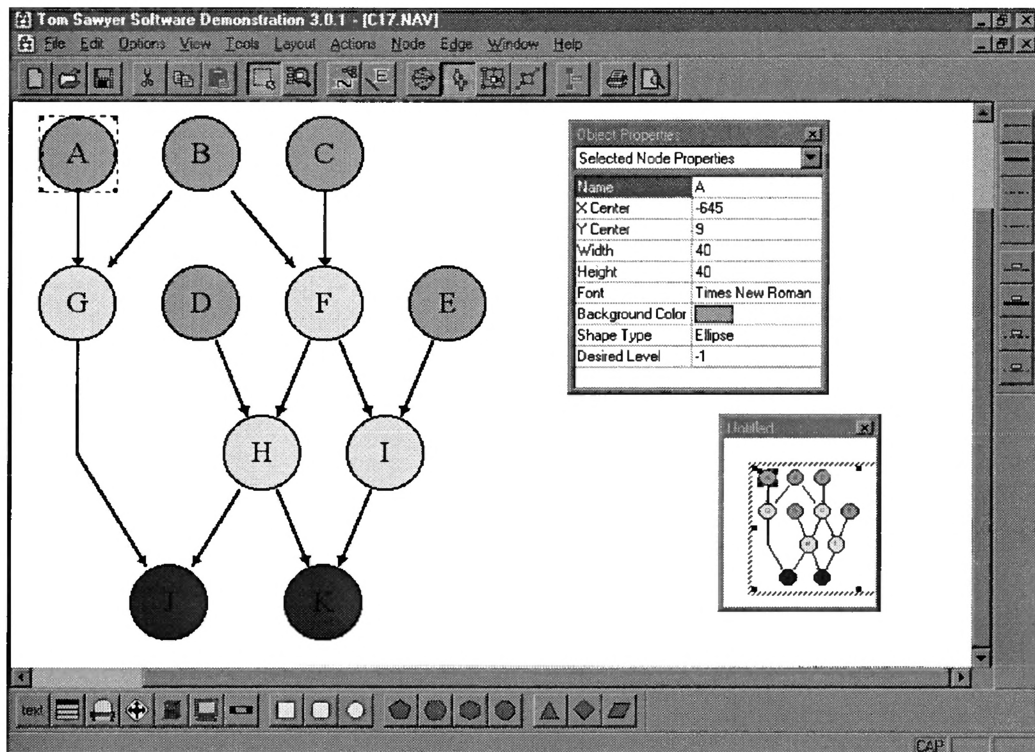


Figure 5-4 (Tom Sawyer)

5.2. Graph Families

Our experimental graphs were based on previous research. There are three families:

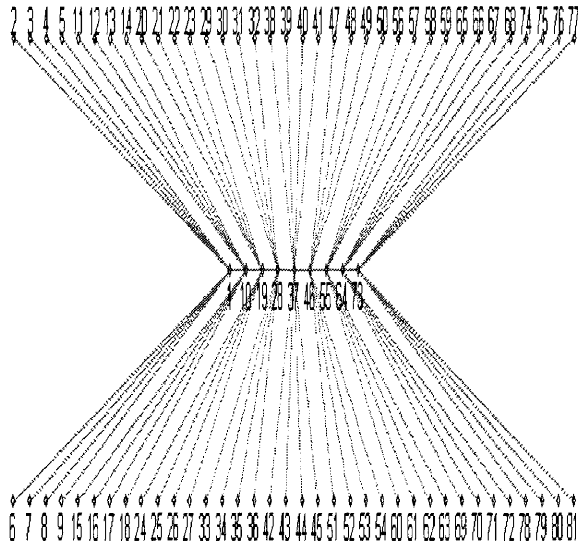
1. Caterpillar: line of n vertices with m inputs and outputs [51]
2. Geometric Grid: equally spaced square of $\sqrt{n}$ * $\sqrt{n}$ vertices [23, 48]
3. Random Geometric: n random vertices, connected if within t units [23, 51]

The caterpillar family consists of a set of nodes along a central path, along with an equal number of primary inputs and outputs attached to each. The geometric grid family consists of a specified number of vertices spaced equally apart on a 1 by 1 unit square. They are connected based on a pre-determined edge pattern and geometric adjacency. The random geometric family consists of a specified number of vertices randomly placed in the same 1 by 1 unit square. However, these vertices are connected if within a distance t. In other words, for each vertex, all vertices located within a circle of radius t are connected.

Precedence, or head → tail determination is based on the vertex numbers, where a lower-numbered vertex is always the head and a higher-numbered vertex is always the tail. This eliminates the potential for cycles. After each graph is constructed, a transitive reduction is performed, where any direct edges between a head and tail vertex are removed if the tail vertex has an inherited relationship to the head vertex.

After construction, but before the experiment runs are conducted, each graph has its node names randomly permuted. This helps eliminate any algorithmic bias for graphs where only a lower-numbered node may be a head vertex in a given relationship. The following pages provide samples of 81-vertex Caterpillar, Geometric Grid-S and Grid-X graphs.

Caterpillar



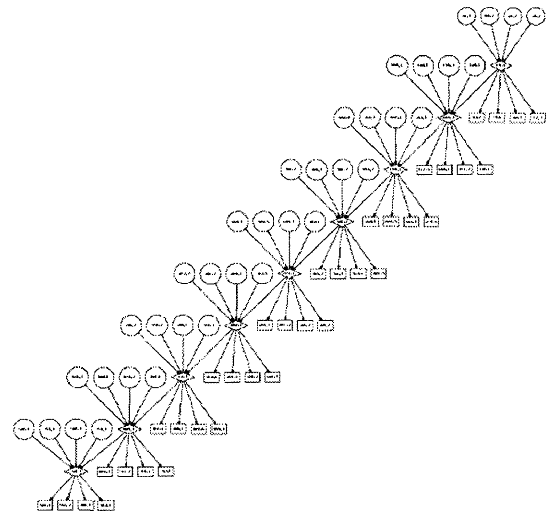
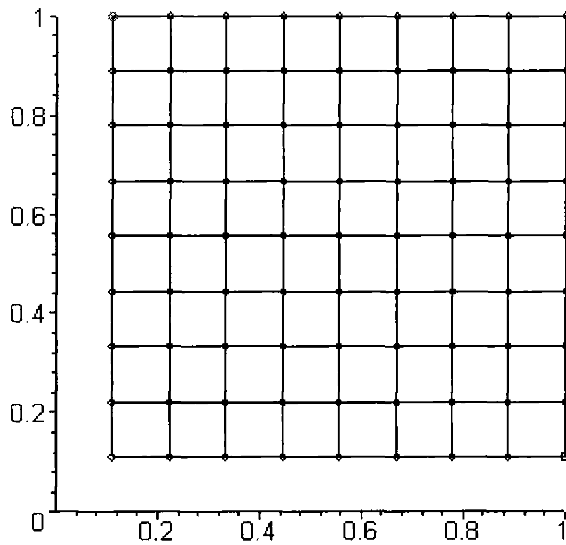


Figure 5-5 (Maple V)

Figure 5-6 (GraphViz)

Geometric Grid-S



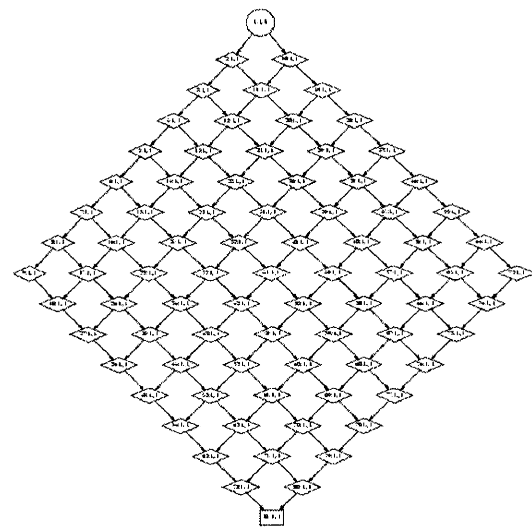


Figure 5-7 (Maple V)

Figure 5-8 (GraphViz)

Geometric Grid-X, Original

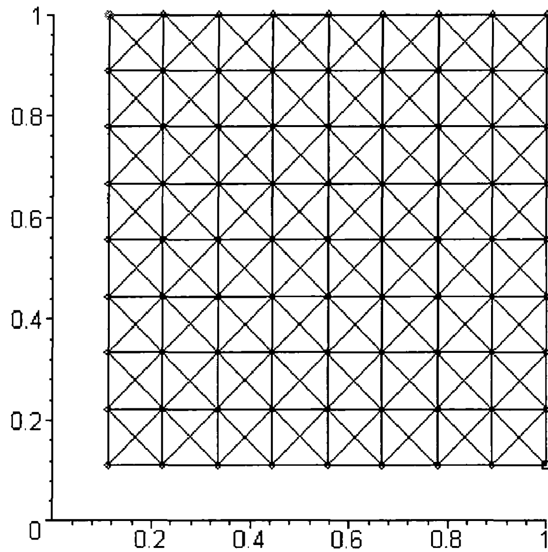


Figure 5-9 (Maple V)

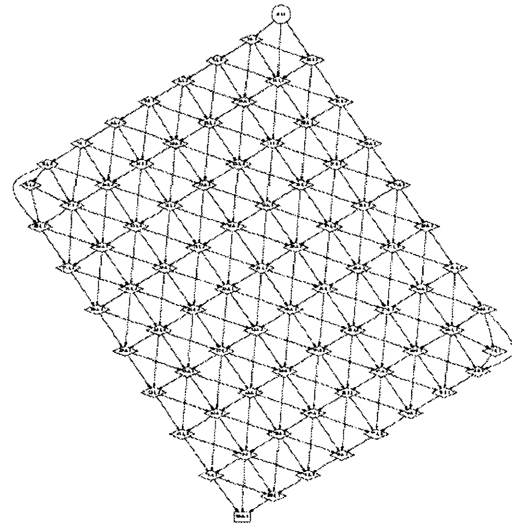


Figure 5-10 (GraphViz)

Geometric Grid-X, Transitively Reduced

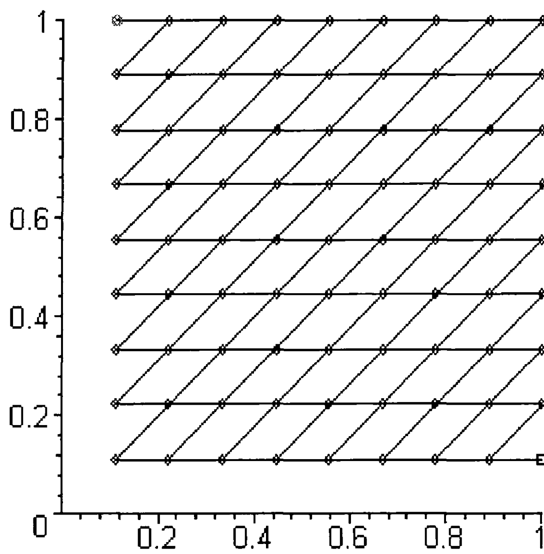


Figure 5-11 (Maple V)

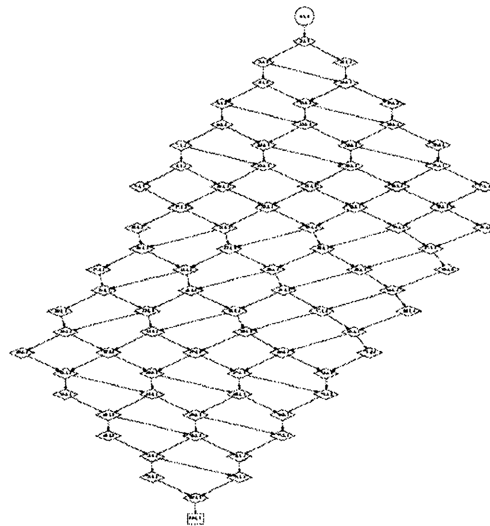
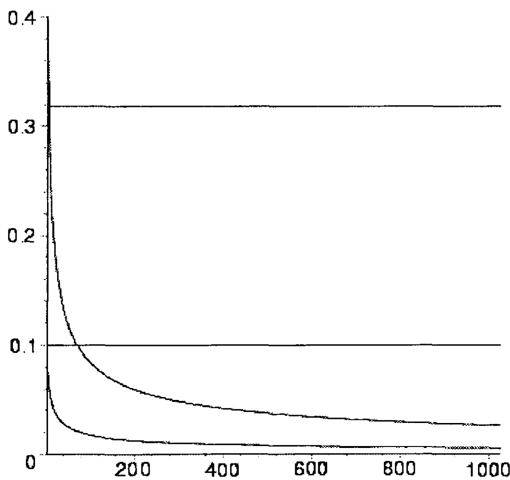


Figure 5-12 (GraphViz)

For the random geometric graphs, there were four variants we used. They are all based on the formula d = Π * |*V*| * t^2, where *t* = radius of the circle from the vertex, *v*, the |*V*| is the number of vertices in the graph, and d is the number edges per vertex. The following table summarizes the values used for these variants.

	t	d	Comments
1	0.1	$\pi(0.1)^2\lvert V\rvert$	constant distance, variable density
2	$= \sqrt{\frac{(0.1)\lvert V\rvert}{\pi\lvert V\rvert}} = \sqrt{(0.1)/\pi}$	$(0.1)\lvert V\rvert$	variable distance, constant density percentage
3	$= \sqrt{\frac{\pi\sqrt{2}}{\pi\lvert V\rvert}} = \sqrt{\sqrt{2}/\lvert V\rvert}$	$\pi\sqrt{2}$	variable distance, constant density value
4	$1/\pi$	$= \pi(1/\pi)^2\lvert V\rvert = \lvert V\rvert/\pi$	constant distance variable density, for $\lvert V\rvert = 1, d = t = 1/\pi$

The below charts demonstrate the t and d values relative to v for each of the four variants.



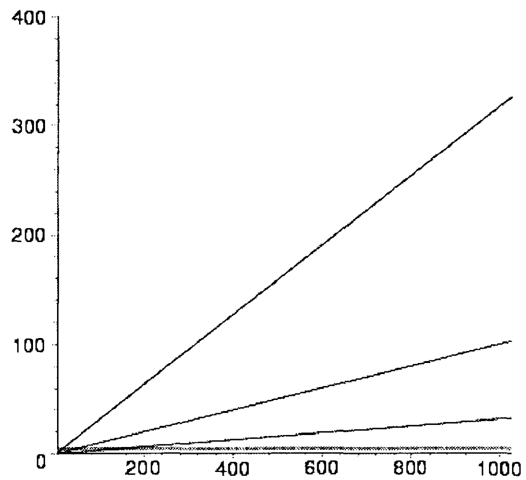


Figure 5-13 (Maple V, |*V*| vs. t)

Figure 5-14 (Maple V, |*V*| vs. d)

The next images shows the radius circles for an 81-vertex graph, assuming t = 0.1. A cross symbol indicates the center of each circle.

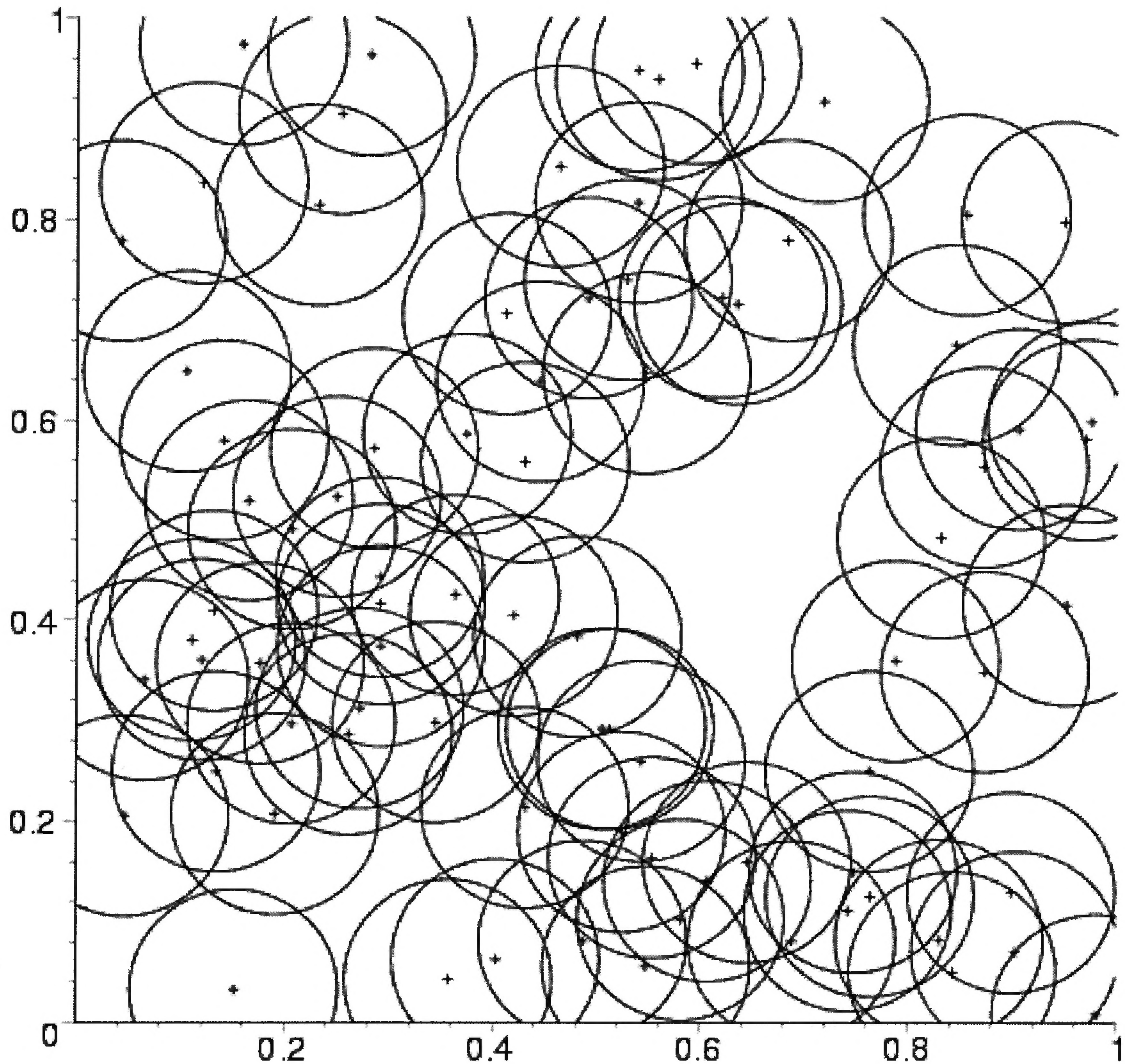


Figure 5-15 (Maple V, T-1)

Note in the bottom left of the graph, there is a circle with no other vertices within its each. This problem led to the creation of the geometric touch modification for the random geometric graph family. Under the geometric touch routine, individual components are

attached to their nearest neighbor after all vertices with distance *t* are connected. In essence, for these vertices, we minimally increase *t* to obtain a single-component graph.

The following two images use the same vertex positions as the previous radius graph, but demonstrate how the touch algorithm would randomly connect the closest vertex, assuming t = 0, or no edges were initially connected.

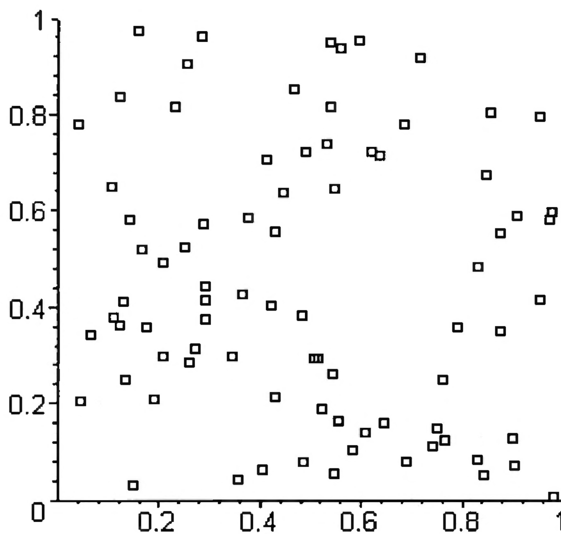


Figure 5-16 (Maple V, T-3)

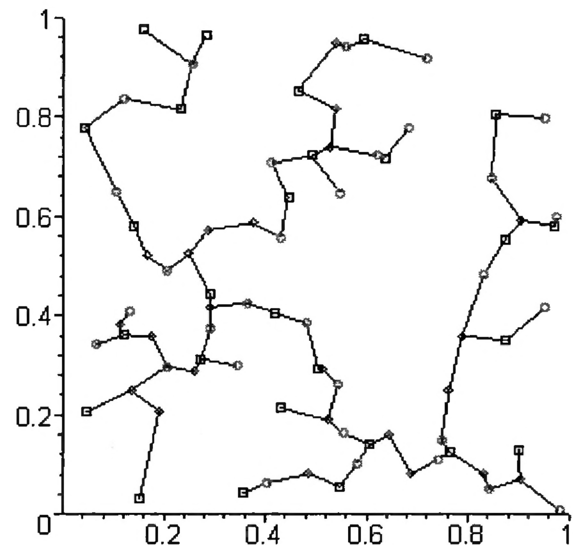


Figure 5-17 (Maple V, T-4)

The next set of figures show how the geometric touch graphs are created.

1. Connect all vertices within distance t
2. Minimally increase t to connect individual graph components
3. Compute the transitive reduction, eliminating unnecessary edges

Each page contains six images, with the left hand showing the respective stage as drawn by our Maple V routines. On the right, the same graph is shown as drawn by *GraphViz*. Note that *GraphViz* only depicts the relationships between the vertices, not their original unit square positions. All four geometric touch variants for $|V| = 81$ are shown.

Geometric Touch Graph (T-1: $t = 0.10$, $d = \sim 2.54$)

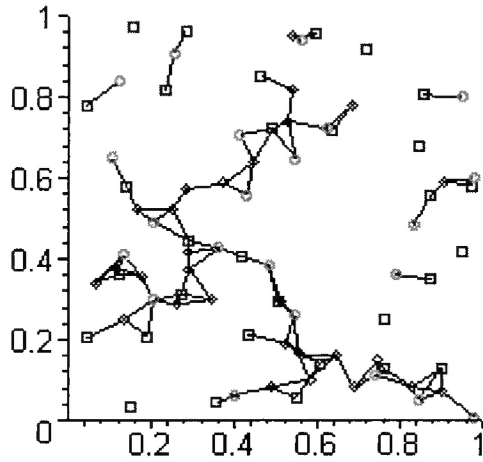


Figure 5-18 (Maple V)

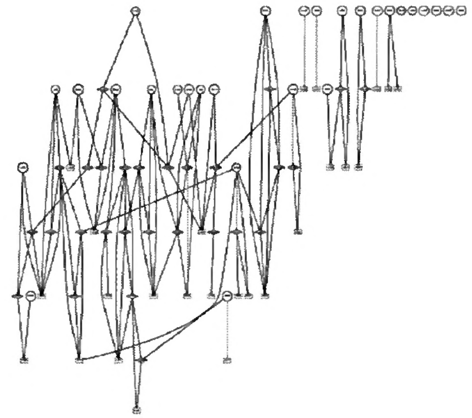


Figure 5-19 (GraphViz)

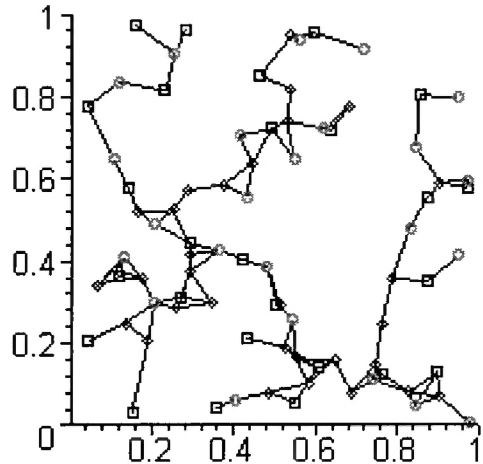


Figure 5-20 (Maple V)

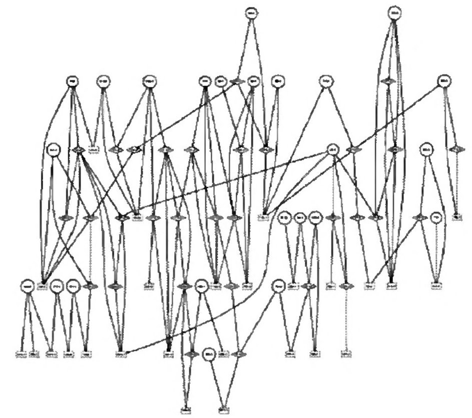


Figure 5-21 (GraphViz)

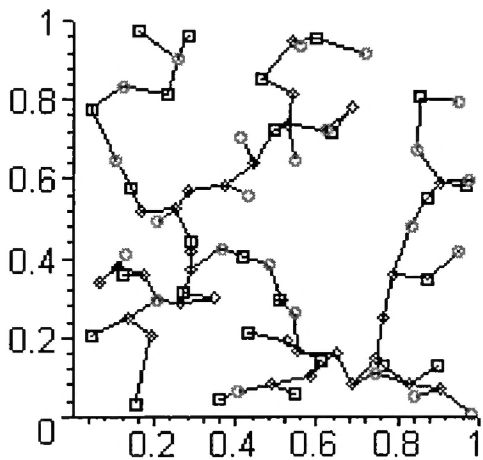


Figure 5-22 (Maple V)

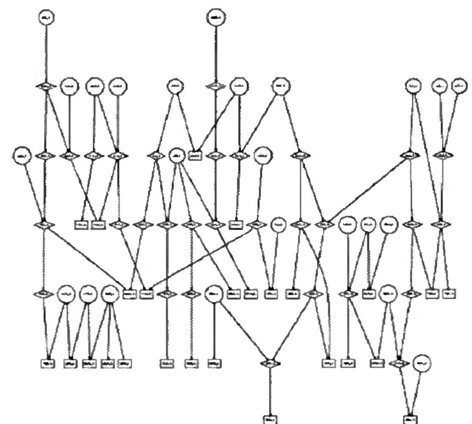


Figure 5-23 (GraphViz)

Geometric Touch Graph (T-2: t = ~0.18, d = ~8.10)

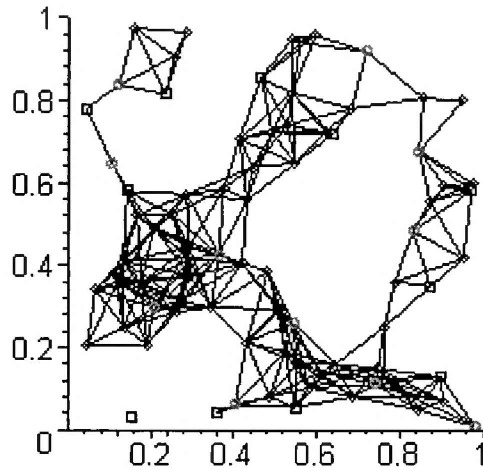


Figure 5-24 (Maple V)

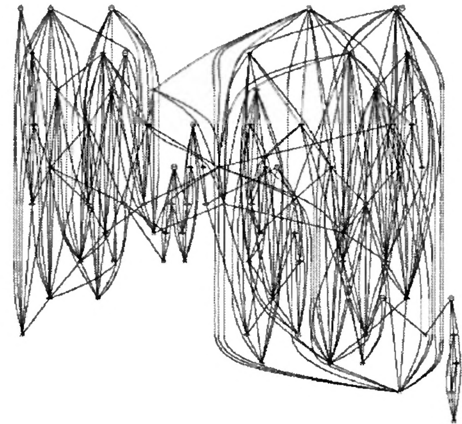


Figure 5-25 (GraphViz)

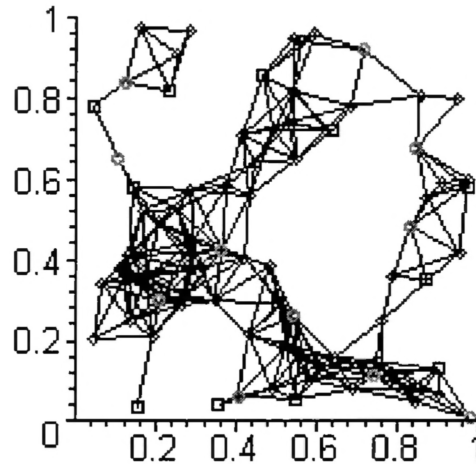


Figure 5-26 (Maple V)

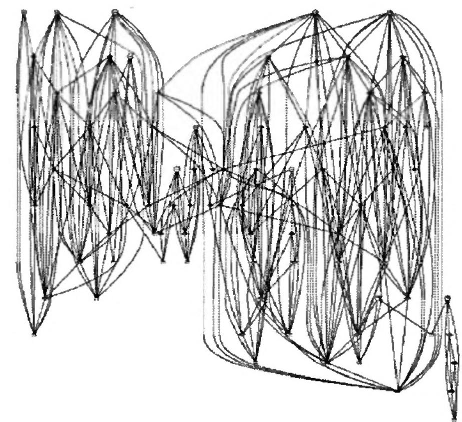


Figure 5-27 (GraphViz)

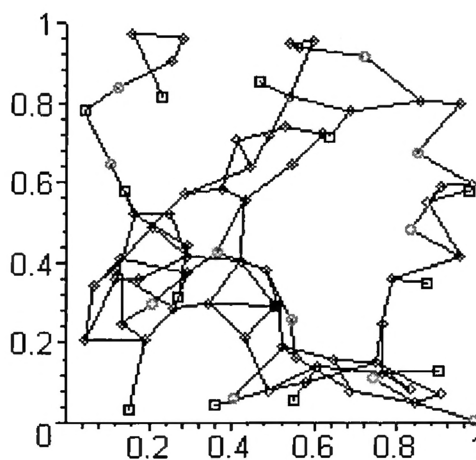


Figure 5-28 (Maple V)

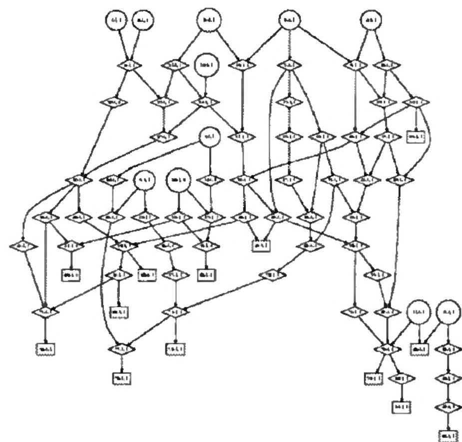


Figure 5-29 (GraphViz)

Geometric Touch Graph (T-3: $t = 0.13$, $d = \sim 4.44$)

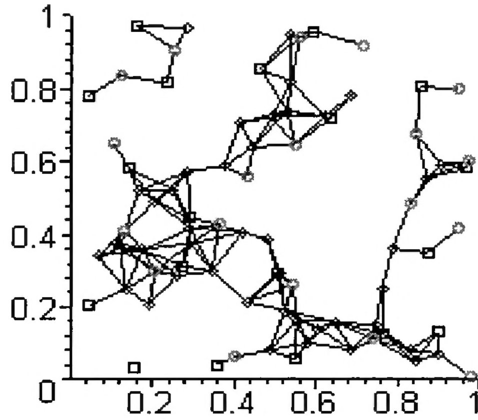


Figure 5-30 (Maple V)

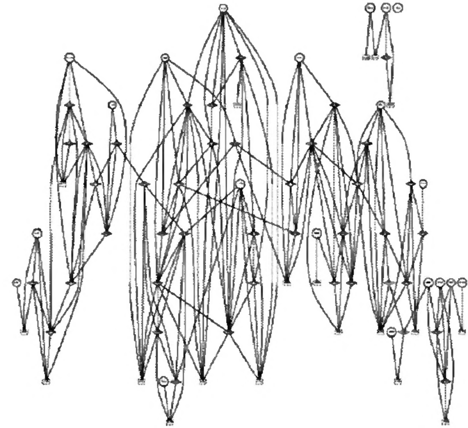


Figure 5-31 (GraphViz)

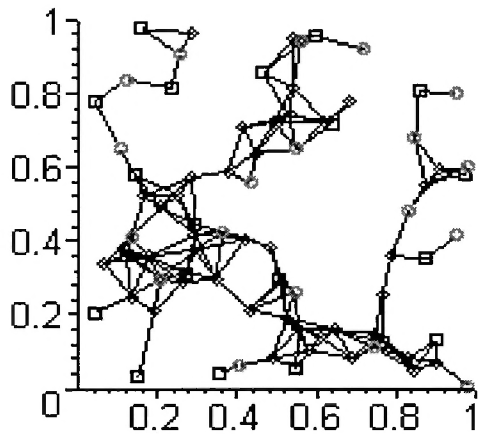


Figure 5-32 (Maple V)

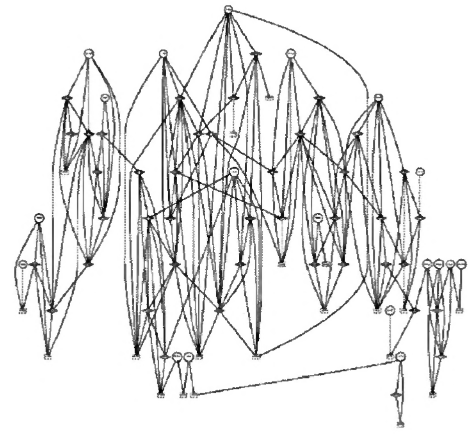


Figure 5-33 (GraphViz)

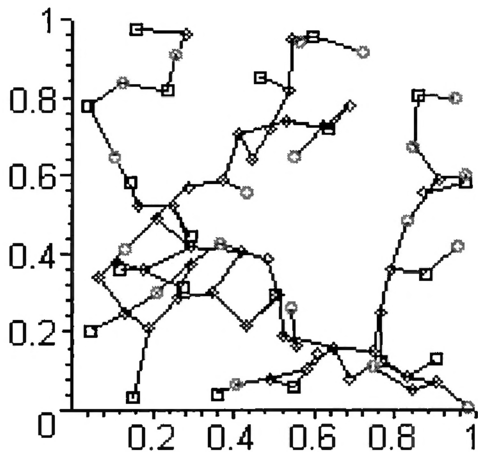


Figure 5-34 (Maple V)

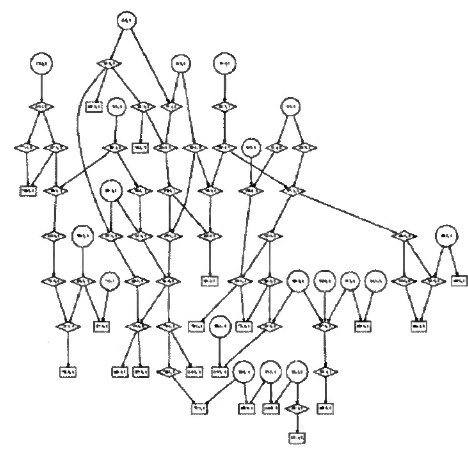


Figure 5-35 (GraphViz)

Geometric Touch Graph (T-4: t = ~0.32, d = ~25.78)

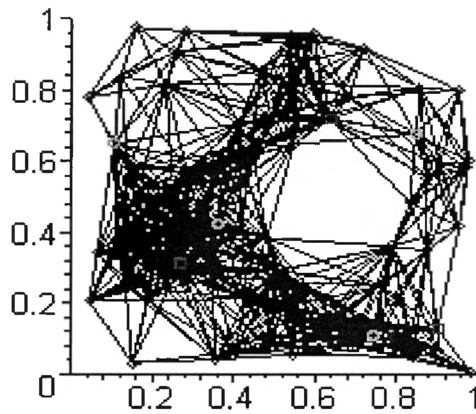


Figure 5-36 (Maple V)

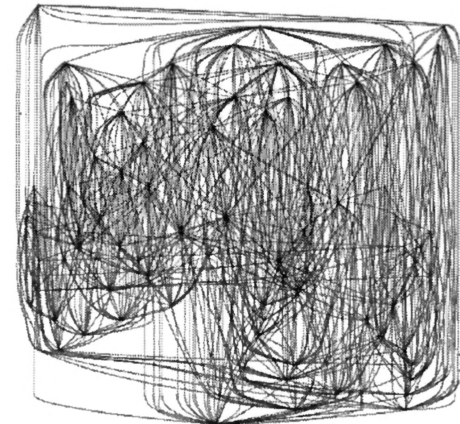


Figure 5-37 (GraphViz)

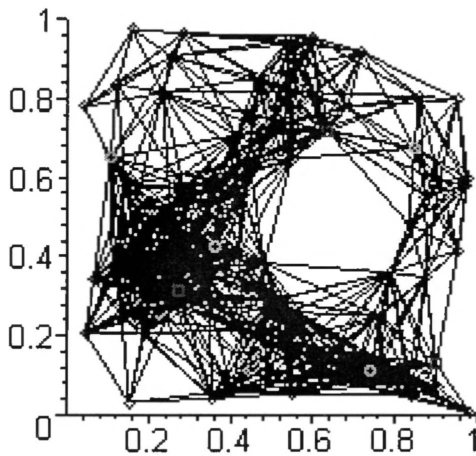


Figure 5-38 (Maple V)

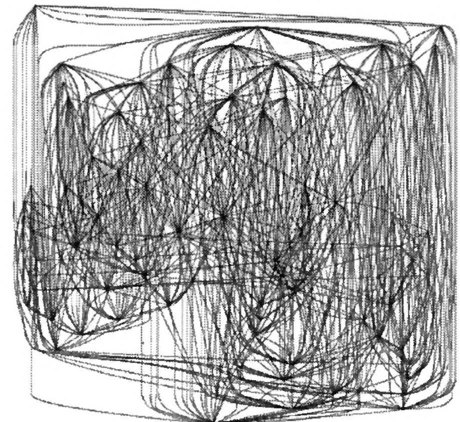


Figure 5-39 (GraphViz)

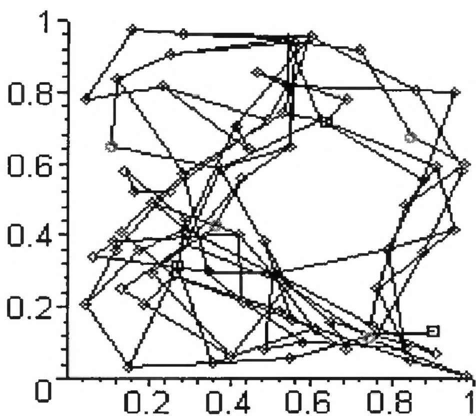


Figure 5-40 (Maple V)

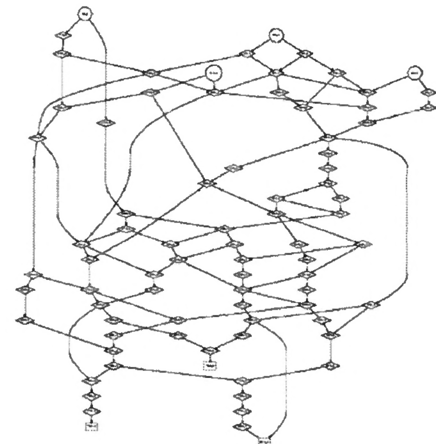
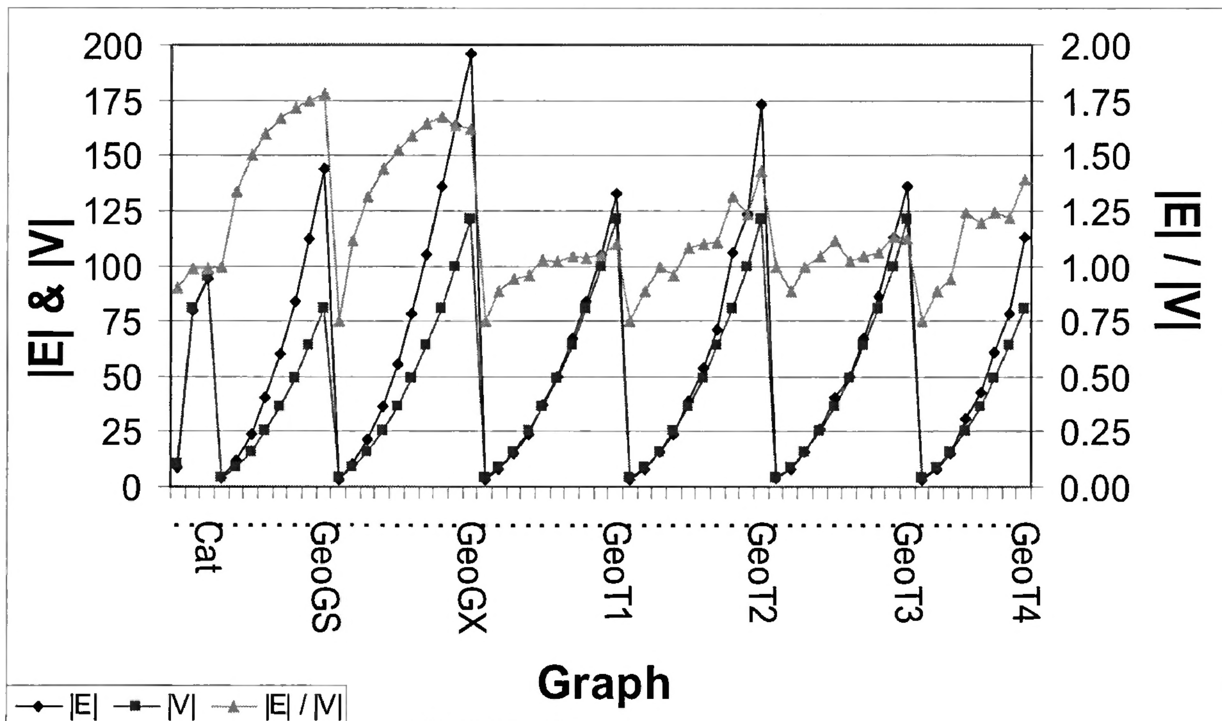


Figure 5-41 (GraphViz)

The following chart summarizes the data for all graphs used in our experiments. The |V|, |E| and density, or |E| / |V|, metrics are included in this chart. There were 59 graphs used for these experiments, spread over the 7 graph families. They were constructed such that $4 \leq |V| \leq 121$.



We decided on four control parameters for our partitioning experiments:

1. D: inter-partition delay value
2. Domain: source application of the graph
3. |V|: vertex limit in a partition
4. |E|: edge limit in a partition

The D parameter specifies the delay experienced when going between subgraphs of a partition. We used two values, D = 10 and D = 1000, testing for both a small and large inter-subgraph delay respectively. Our two domains, of course are VLSI Design and

Parallel Processing. For VLSI Design, we measured the maximum delay within the partitioning, while for parallel processing we measured the total delay within the partitioning.

The |V| and |E| values were to used to limit the total weight of the vertices and the total cut within a subgraph of the partitioning. We used a total of five combinations for |V| and |E|. Three of these were bi-partitioning variants, while the other two were to simulate technology mapping of the resulting partitions onto two existing FPGA families. Field Programmable Gate Arrays, or FPGAs, are arrays of transistors provided by various manufacturers in pre-specified sizes. Our two were chosen based on the small maximum |V| sizes of our input graphs. The selected FPGAs are Xilinx's Spartan (|E| = 77, |V| = 238) [46, 45, 2] and Lucent's ORCA 2 series (|E| = 160, |V| = 400) [38]. The number of vertices and edges in our test graphs drove our FPGA choices.

The twenty combined possibilities of these input controls are enumerated below:

Domain	**D**	**|E|**	**|V|**
Circuit	10	1	2
Circuit	10	2	1
Circuit	10	2	2
Circuit	10	77	238
Circuit	10	160	400
Circuit	1000	1	2
Circuit	1000	2	1
Circuit	1000	2	2
Circuit	1000	77	238
Circuit	1000	160	400

Domain	**D**	**|E|**	**|V|**
Schedule	10	1	2
Schedule	10	2	1
Schedule	10	2	2
Schedule	10	77	238
Schedule	10	160	400
Schedule	1000	1	2
Schedule	1000	2	1
Schedule	1000	2	2
Schedule	1000	77	238
Schedule	1000	160	400

The combined graph and partitioning possibilities gave 828 individual partitionings across the 11 algorithm categories (including category variants), for a total of 9,108 data

points. There were several thousand other data points obtained, but not used in this analysis.

For the partitionings not producing a schedule, a simple scheduling heuristic was used, based on the resulting partitioning. For each partition in the solution, a topological sort, based on the vertex delays is generated. For vertices in the same partition with no edge between them, where one precedes the other in the delay-based topological sort, an edge is added.

5.3. MEDICIS

Success in this research may be largely attributed to the cluster located at the École Polytechnique in France, called MEDICIS. This is a heterogeneous collection of machines, operating systems and software available to those performing mathematics-oriented academic research. This cluster is managed by LSF from Platform Computing [31]. The primary computing machines used were (6) Compaq Alpha-based XP 1000 servers and (6) dual-Intel Pentium II servers. Development work was primarily performed on Windows 95 & NT workstations, along with test partitionings. Additionally, an Alpha-based server, Apollo, at the University of Nebraska at Omaha, provided testing and execution of the CTR and RAN algorithms.

5.4. Sample Partitioning Results

The next set of images displays the resulting partition obtained for each of the $|V| = 81$ graph variants and for each partitioning family. The upper left image on each page is the original graph. GraphViz generated all the images of these sample results. These pages provide a guide on how the resulting partitionings aesthetically appear for a given graph family.

There are many facets of each algorithm apparent. The BPD algorithm generates a large amount of vertex replication. The assumption of DSC that there are an unlimited number of subgraphs is readily apparent, along with an occasional decision that no partitioning is necessary. CTR, largely as a side-effect of its construction method, often creates aesthetically pleasing results. GEA and RAN both tend to be rather messy in their appearance, but as will be seen later, do occasionally find smaller delays than the other algorithms.

Caterpillar Graph (C9_x_4)

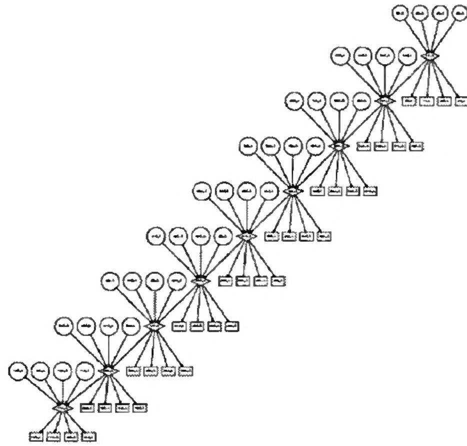


Figure 5-42 (Original Graph)

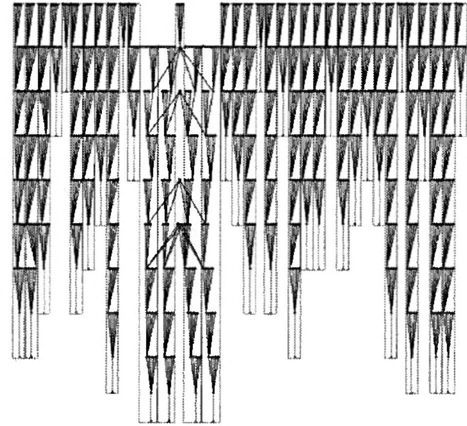


Figure 5-43 (BPD)

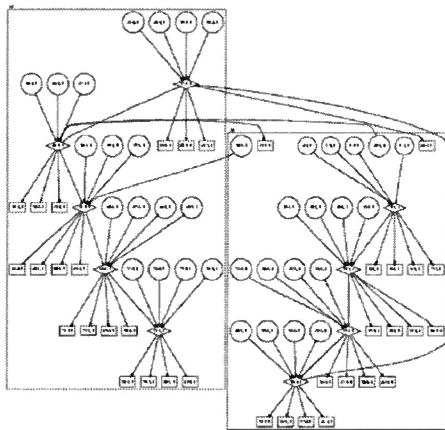


Figure 5-44 (CTR)

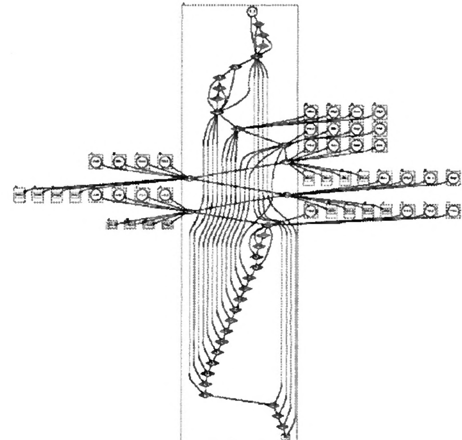


Figure 5-45 (DSC)

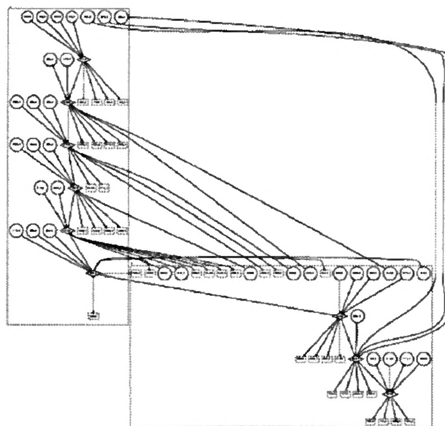


Figure 5-46 (GEA)

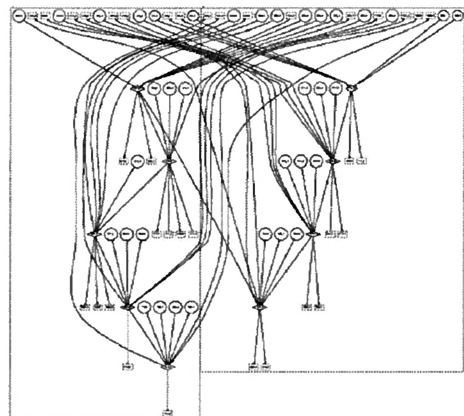


Figure 5-47 (RAN)

Geometric Grid Square Graph (9 x 9)

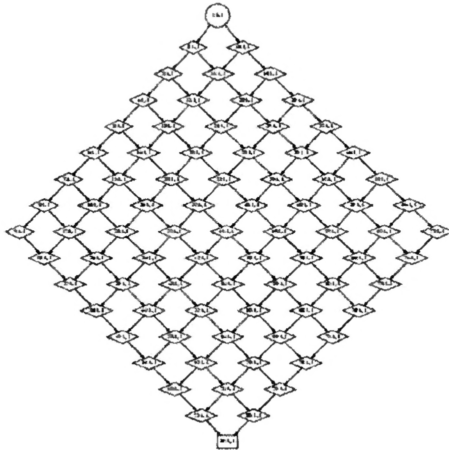


Figure 5-48 (Original Graph)

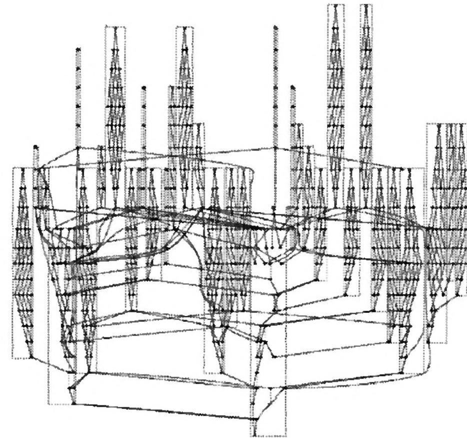


Figure 5-49 (BPD)

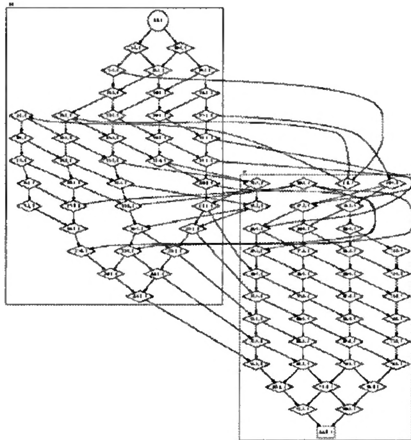


Figure 5-50 (CTR)

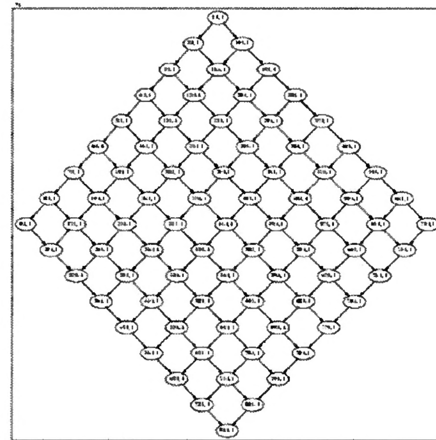


Figure 5-51 (DSC)

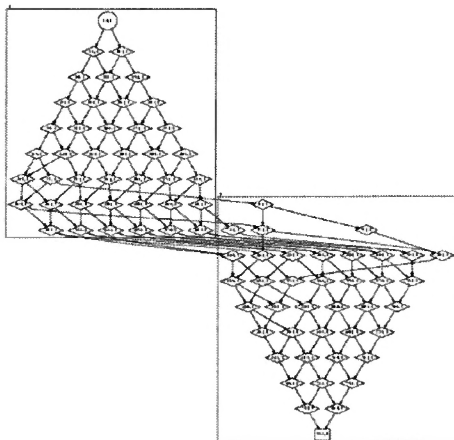


Figure 5-52 (GEA)

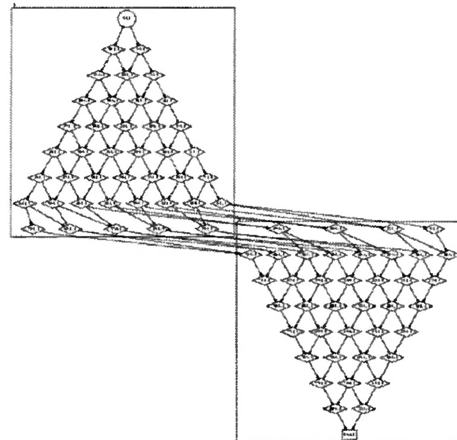


Figure 5-53 (RAN)

Geometric Grid Square-X Graph (9 x 9)

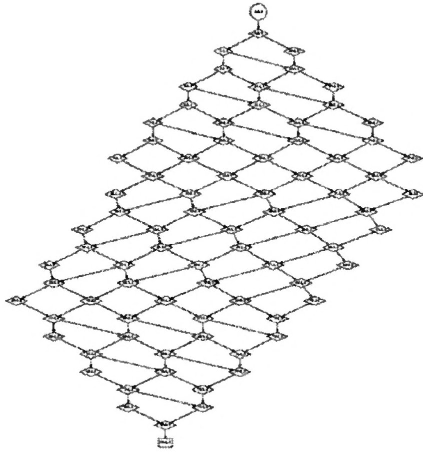


Figure 5-54 (Original Graph)

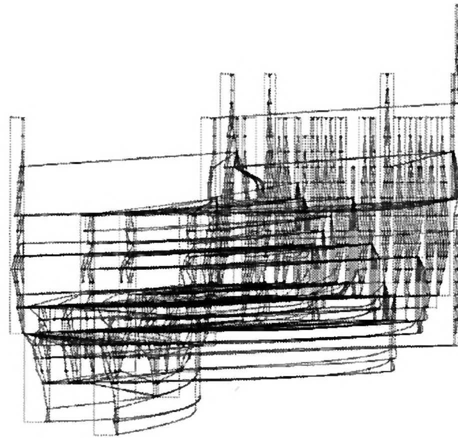


Figure 5-55 (BPD)

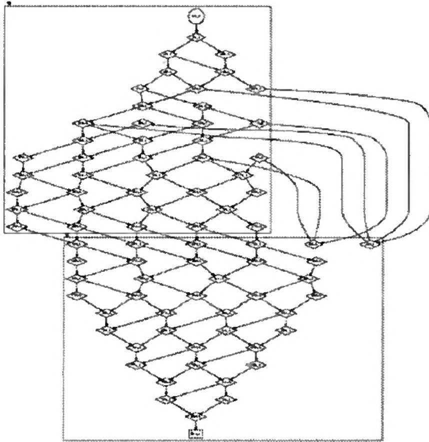


Figure 5-56 (CTR)

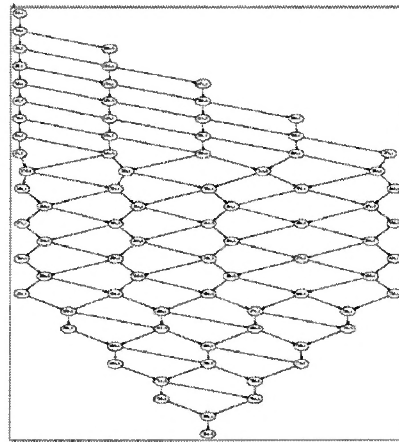


Figure 5-57 (DSC)

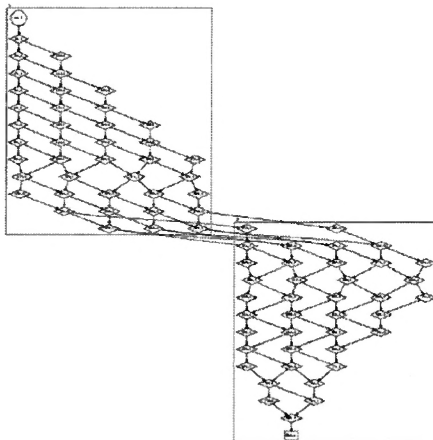


Figure 5-58 (GEA)

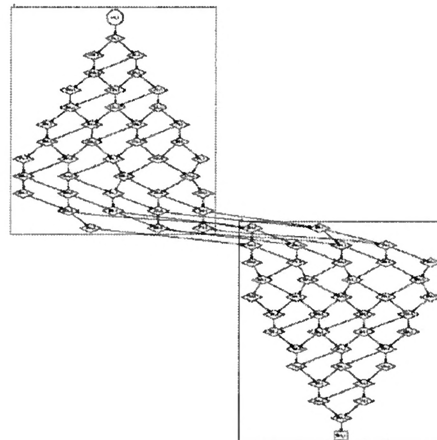


Figure 5-59 (RAN)

Geometric Touch Graph (T-1: $t = 0.10$, $d = \sim 2.54$)

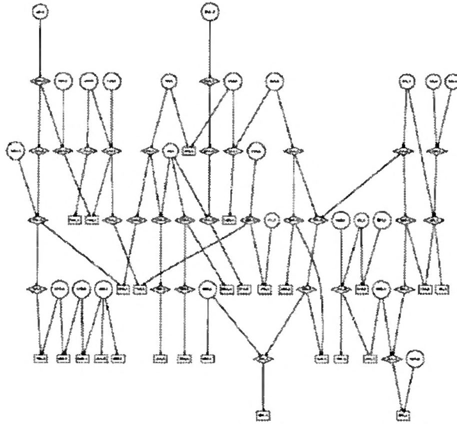


Figure 5-60 (Original Graph)

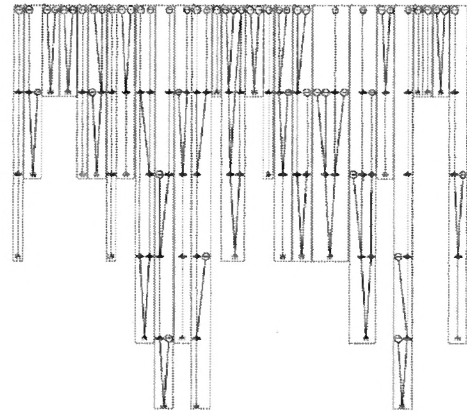


Figure 5-61 (BPD)

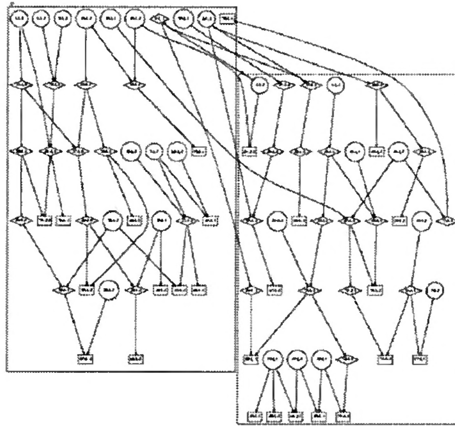


Figure 5-62 (CTR)

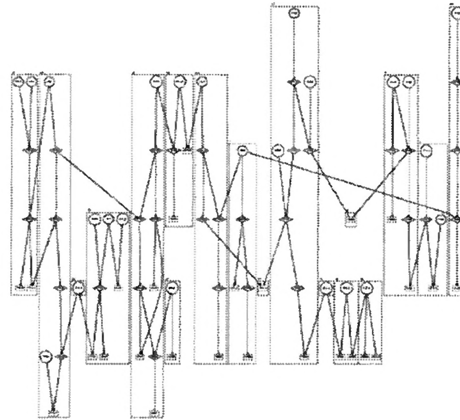


Figure 5-63 (DSC)

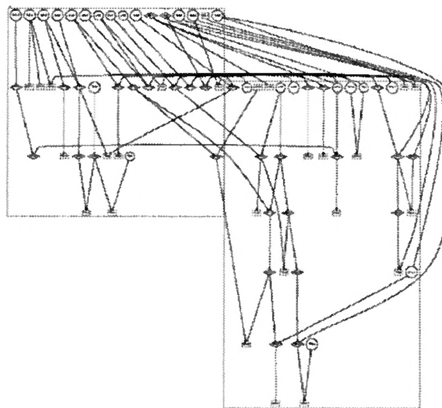


Figure 5-64 (GEA)

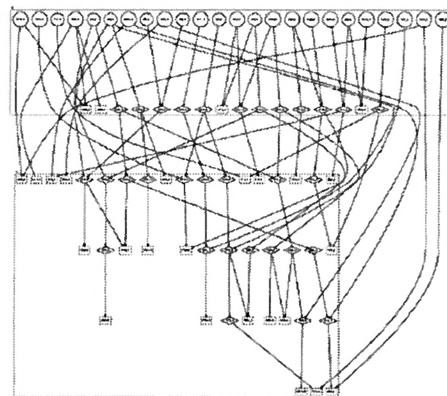


Figure 5-65 (RAN)

Geometric Touch Graph (T-2: t =~0.18, d = ~8.10)

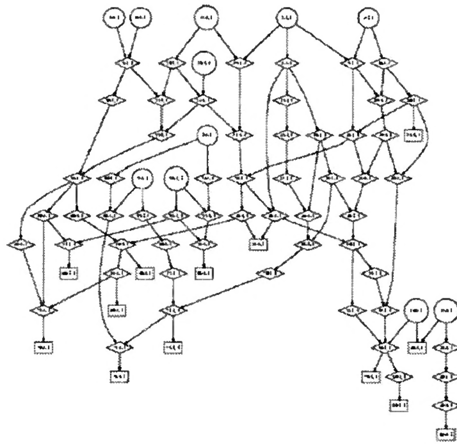


Figure 5-66 (Original Graph)

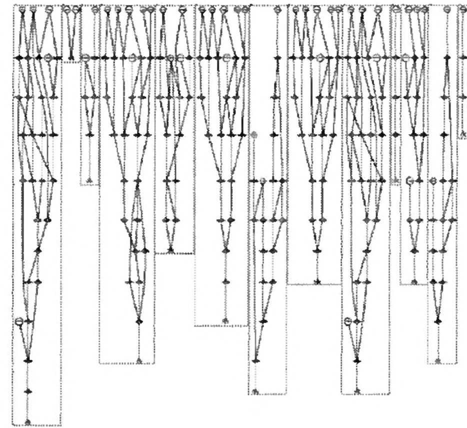


Figure 5-67 (BPD)

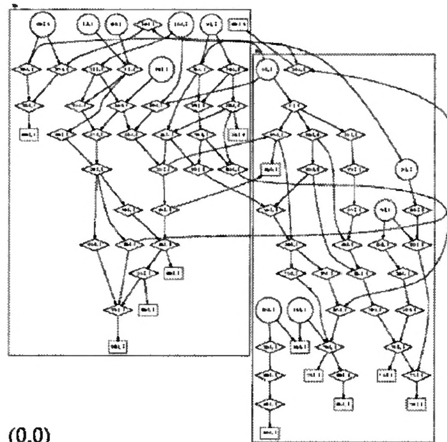


Figure 5-68 (CTR)

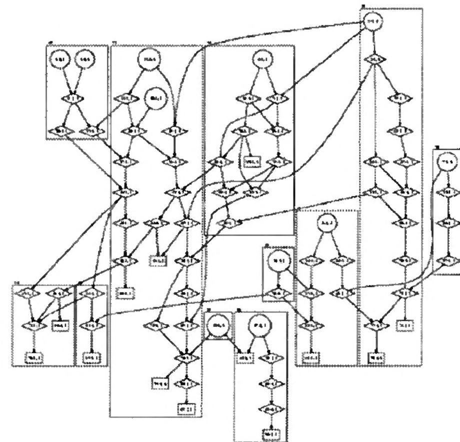


Figure 5-69 (DSC)

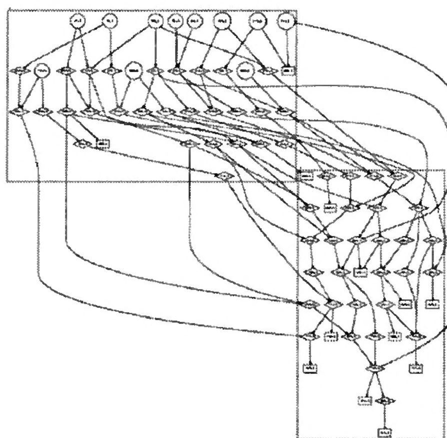


Figure 5-70 (GEA)

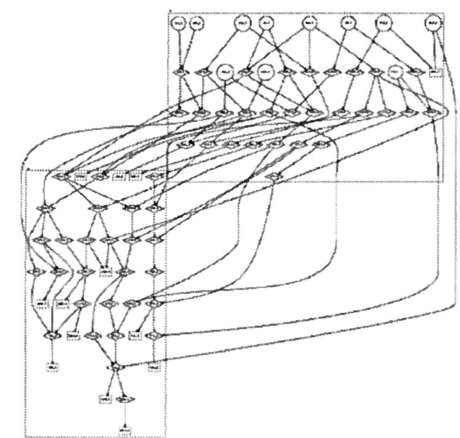


Figure 5-71 (RAN)

Geometric Touch Graph (T-3: $t = 0.13$, $d = \sim 4.44$)

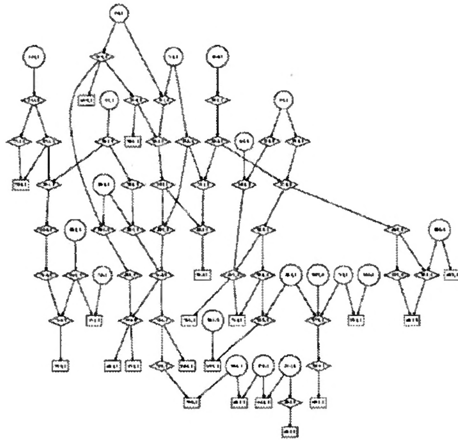


Figure 5-72 (Original Graph)

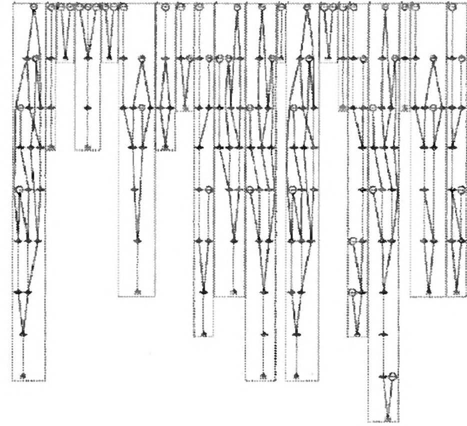


Figure 5-73 (BPD)

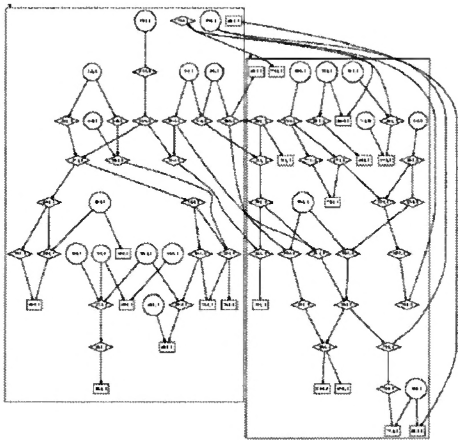


Figure 5-74 (CTR)

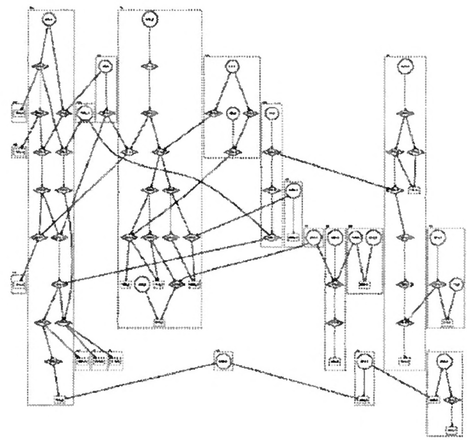


Figure 5-75 (DSC)

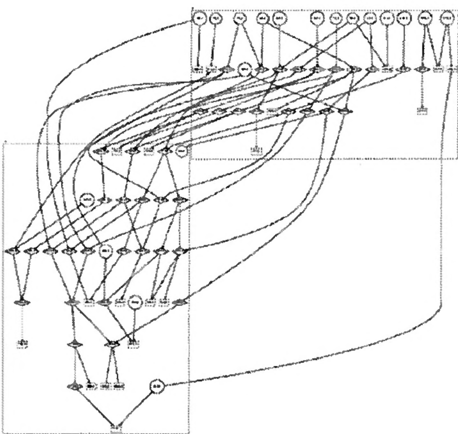


Figure 5-76 (GEA)

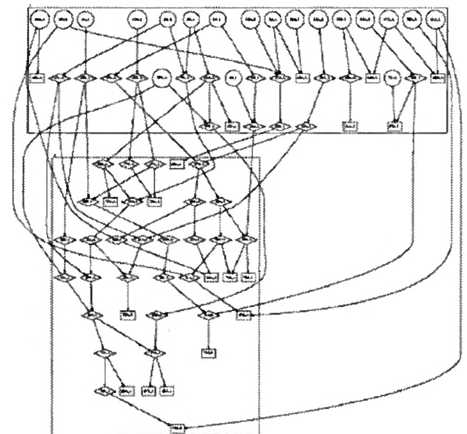


Figure 5-77 (RAN)

Geometric Touch Graph (T-4: $t = \sim 0.32$, $d = \sim 25.78$)

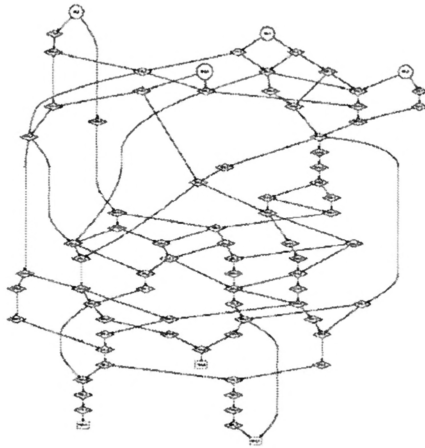


Figure 5-78 (Original Graph)

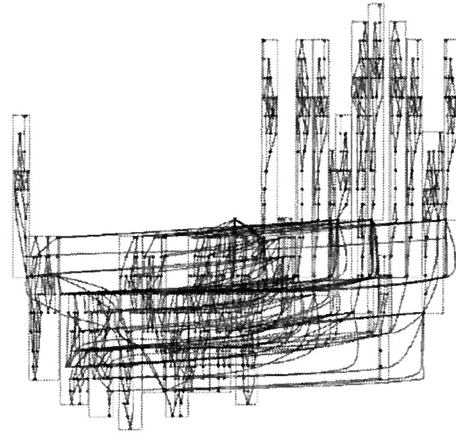


Figure 5-79 (BPD)

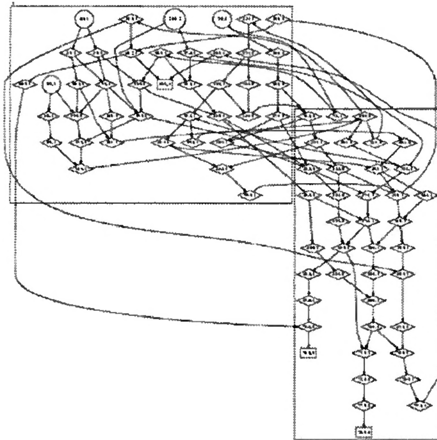


Figure 5-80 (CTR)

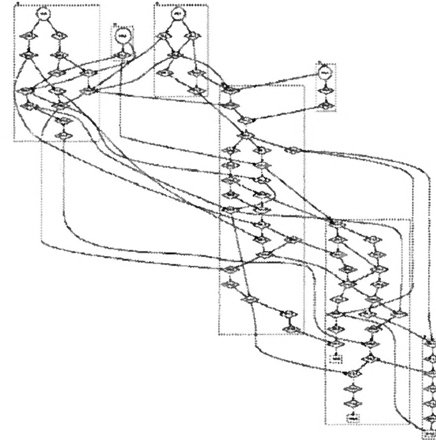


Figure 5-81 (DSC)

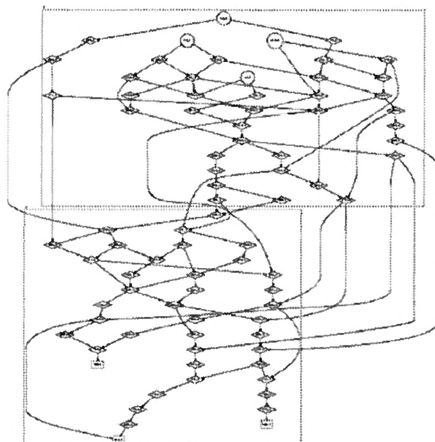


Figure 5-82 (GEA)

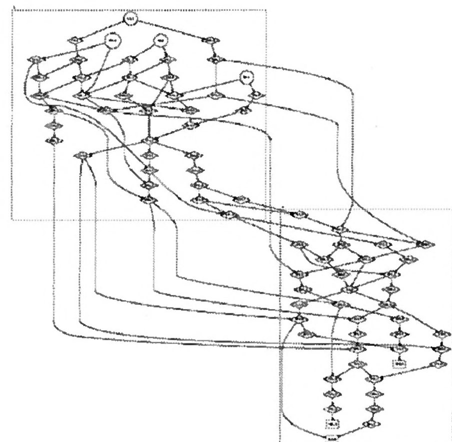


Figure 5-83 (RAN)

5.5. Space and Time

The graph below illustrates the relationship between the number of vertices in a source graph to the root mean square (RMS) time of all algorithms, except RAN. This time was used as the execution time available for the RAN algorithm. While close, a definite order is detectable: [GeoGS, GeoGX, GeoT4, GeoT2, GeoT3, GeoT1, Cat].

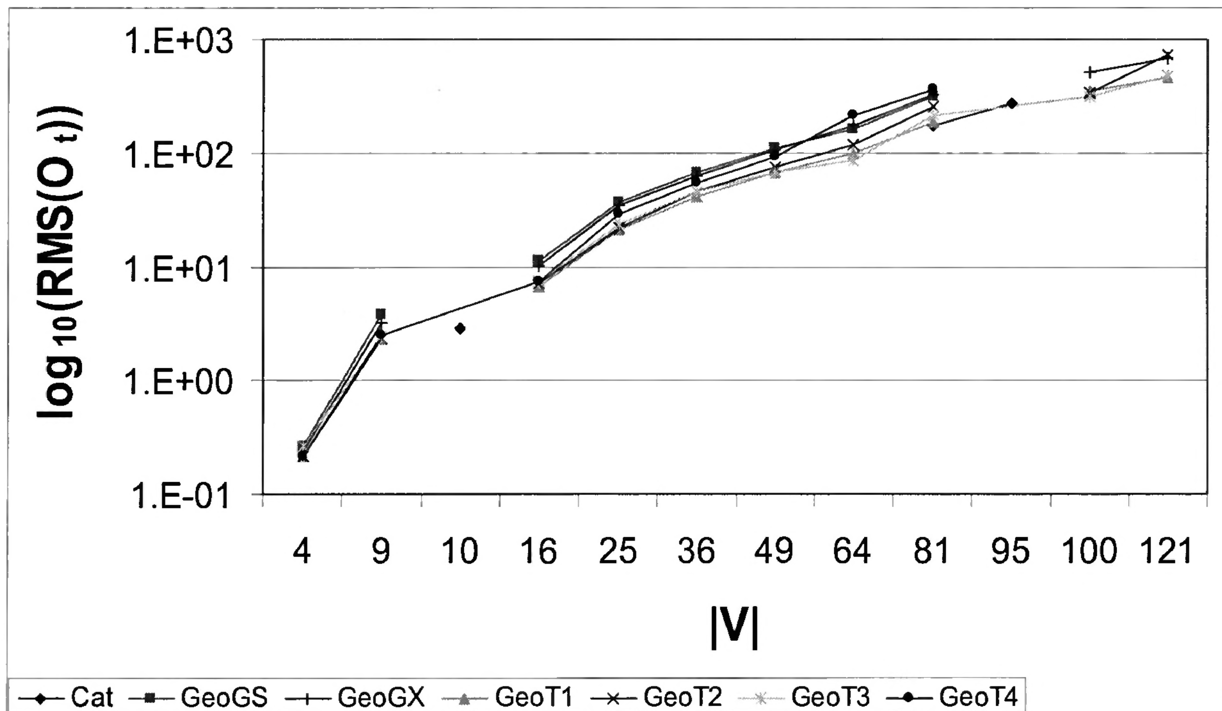


Figure 5-84 (|V| vs. $\log_{10}(RMS(O_t))$, seconds)

The space and time graphs are roughly comparable, but on a different order of magnitude. Essentially, DSC, CTR, GEA and RAN use roughly the same of amount of space and time regardless of the graph structure. However, BPD is highly variable, oscillating between DSC and RAN. BPD and CTR have a 50-50 chance of performing better than each other.

This variableness may be attributed to the |V| limitations on each subgraph and the amount of replication created. Both of these space and time graphs measure the total amount used over the course of determining the partitioning.

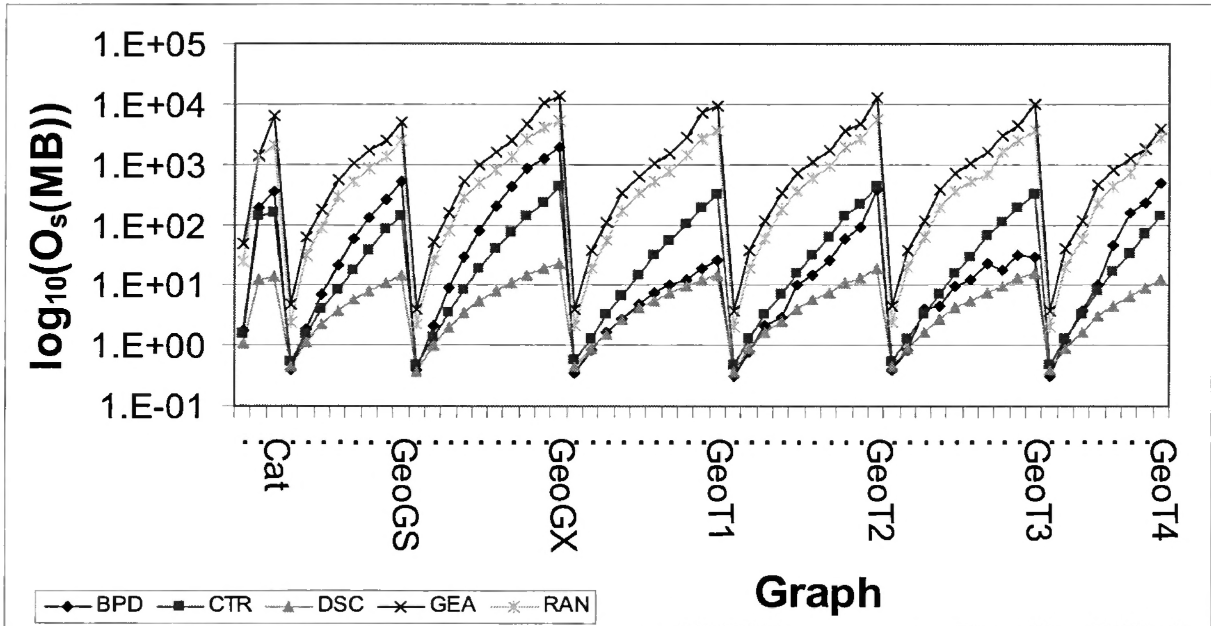


Figure 5-85 (Graph vs. $\log_{10}(O_s)$, MB)

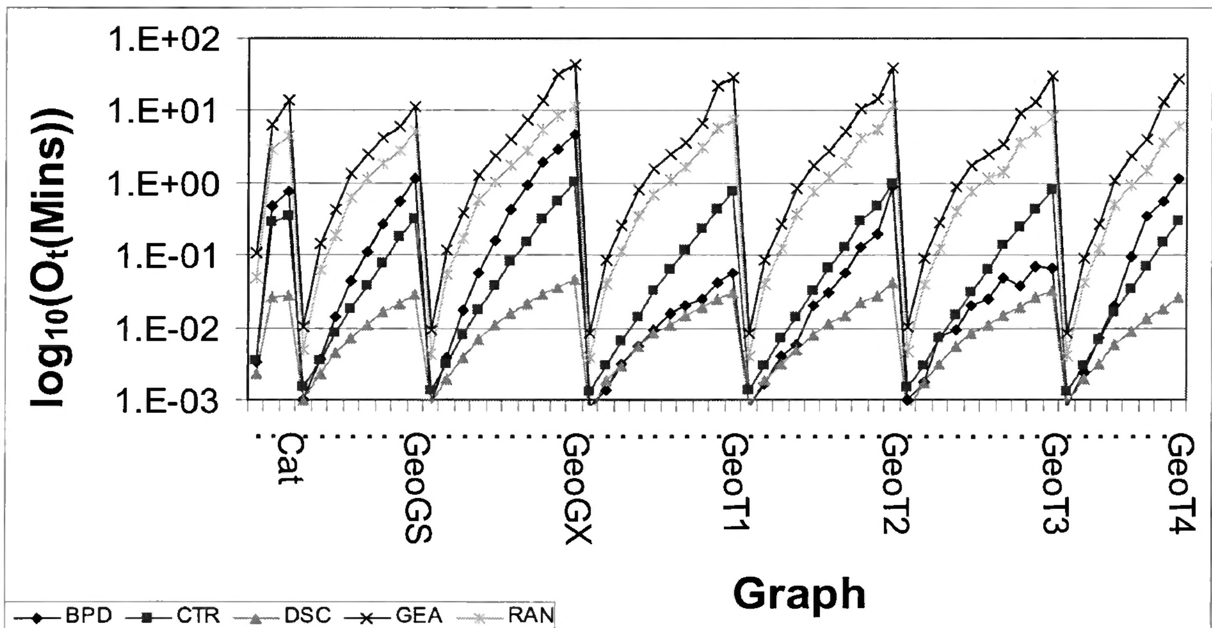


Figure 5-86 (Graph vs. $\log_{10}(O_t)$, Minutes)

5.6. Replication (|V|), Components (K), and Partitions (k)

This 3D graph, perhaps more than any other, illustrates the differences between the existing and new algorithm categories. It shows the values of three metrics, replication, components, and partitions, relative to their expected values. For example, the graph reveals for the graphs tested, Best Predecessor caused a replication factor of four compared to the original number of vertices. We are also able to observe it created a high number of individual components and partitions. DSC also created a high number of partitions relative to the total desired number, as it currently has no method of limiting itself to a specific number of partitions. CTR, GEA and RAN all obtain values of one for each of these metrics. These metrics, |V|, K and k, should be included in the *AGPM* in order to successfully classify a specific algorithm.

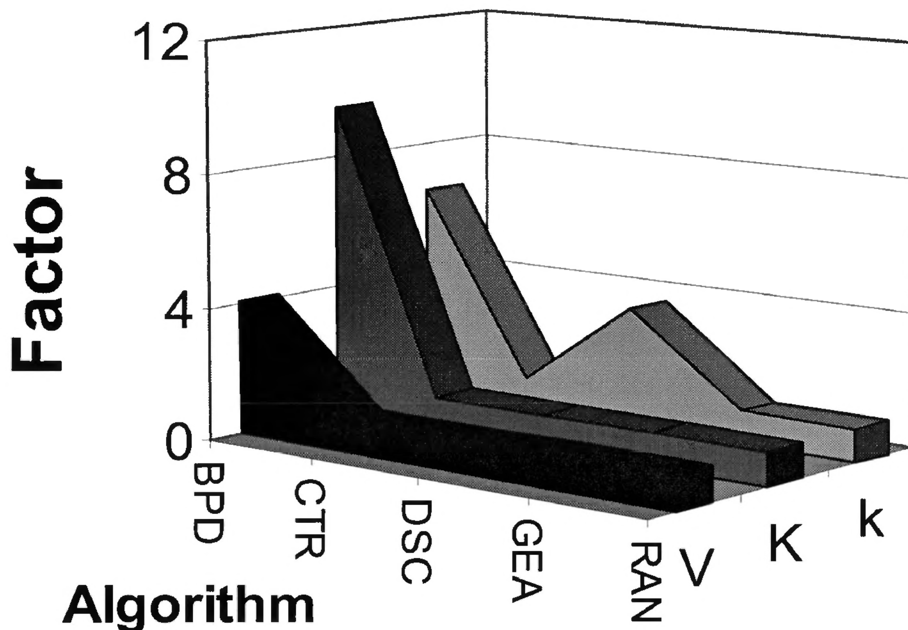


Figure 5-87 (Algorithm vs. V, K and k)

5.7. Cross-Algorithm Category Analysis

Based on the problem definition we were examining, coupled with the nature of the domains and algorithms compared, we selected five key comparisons. These key comparisons are each measured relative to the percentage of minimum delays for the respective algorithm category. If more than one result for a specific category was available, the minimum result for that sample was used. Variants of the CTR and GEA algorithms are examined using the same comparisons in subsequent two sections, respectively. The five key comparisons used are:

1. Graph Family
2. D: 10 & 1,000
3. Domain: Circuit & Schedule
4. |E| and |V| Partition Limits
5. Density: |E| / |V|

Each graph provided summarizes the 828 possible partitioning for the algorithms used. Each pair represents the total and unique percentages for each algorithm. Although not always significantly variant, they are provided for completeness and to indicate how often an algorithm generates a unique minimum. The resulting data was first converted from a text file to a *Microsoft Access* database. Specific queries were then executed, and the results transferred to *Microsoft Excel*. These imported query results were then used to generate the graphs shown on the following pages. A short discussion is included on each page, summarizing the visual depiction of the data shown on that page. There are many other metrics obtained and possible, however, they are not all included here.

Graph Family vs. % Min D_t

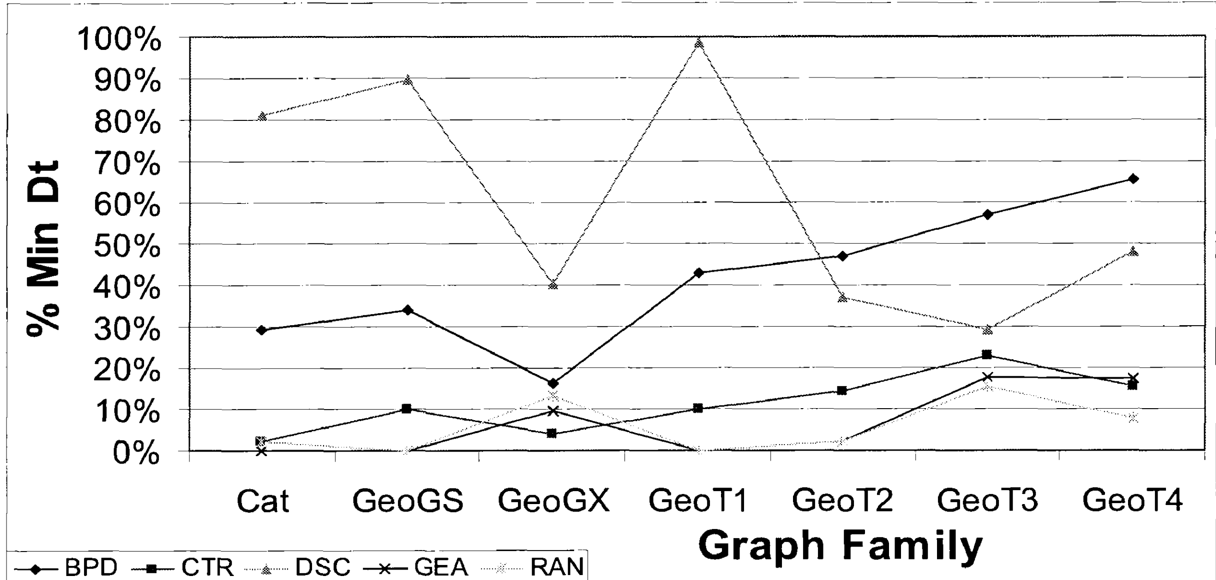


Figure 5-88 (Total)

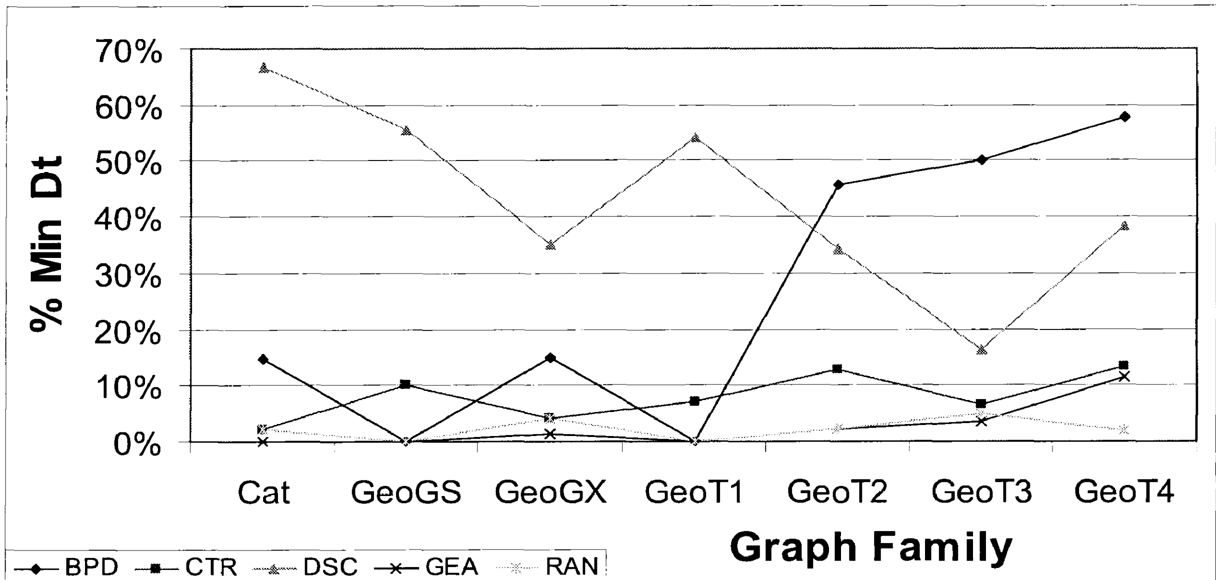


Figure 5-89 (Unique)

The graph family proves to be one of the most distinguishing metrics when classifying partitioning algorithms. Notably, BPD outperforms DSC on circuit-simulation graphs. CTR averages approximately 10% of the findings, with GEA and RAN each contributing a negligible amount. Additionally, BPD finds 30% of the total minimums for the Geometric Grid–Square graphs, yet none of the unique minimums. Note that DSC and BPD do not observe |V| and |E| limits.

Algorithm Category vs. % Min D_t (Circuit & Schedule)

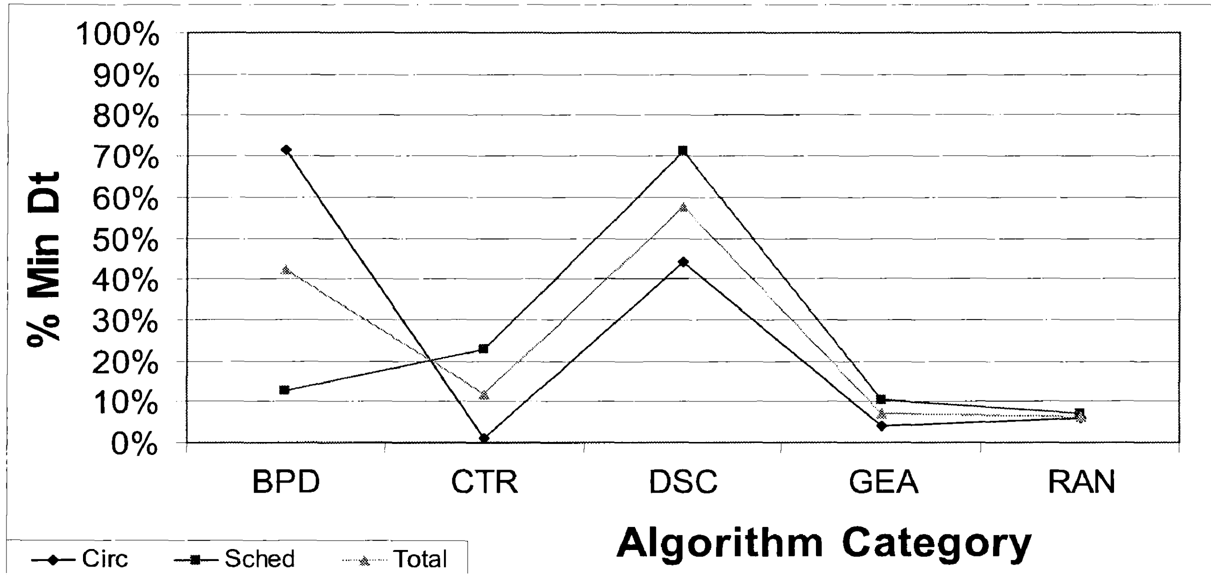


Figure 5-90 (Total)

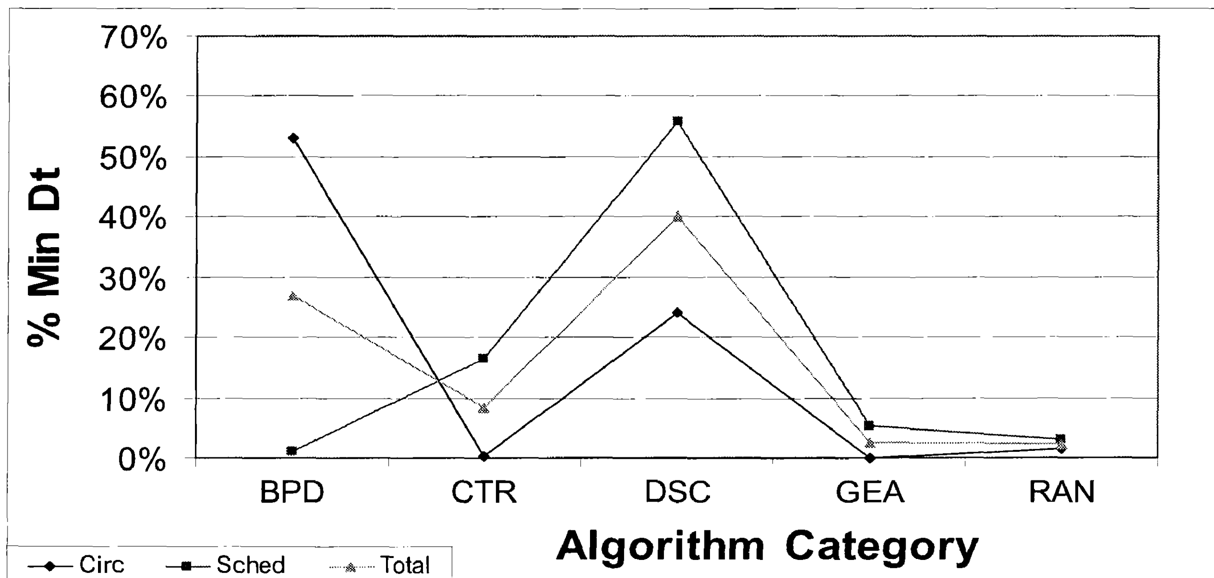


Figure 5-91 (Unique)

As is observed, DSC reports the best results. However, BPD performs exceptionally well when evaluating a graph and its results partitioning as a circuit. This was expected, as it was designed for circuit partitioning. CTR appears best for schedule determinations; however, we expected it to perform better with circuit partitionings. GEA and RAN essentially tie for last place. A subsequent paper might compare based on the graph's granularity and linearity metrics.

Algorithm Category vs. % Min D_t (D = 10 & D = 1,000)

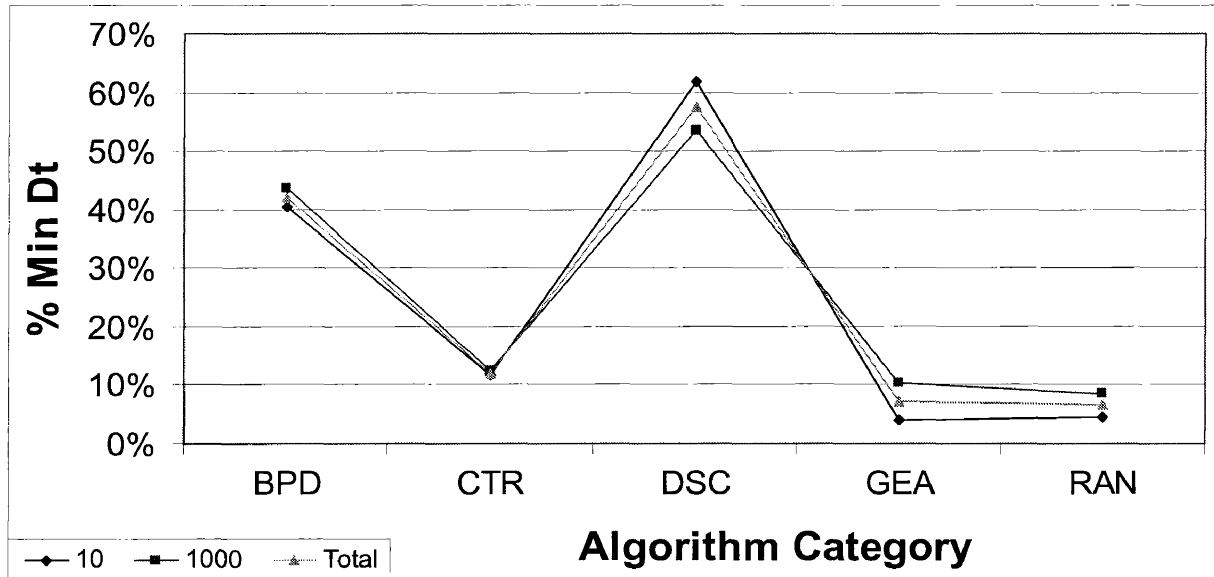


Figure 5-92 (Total)

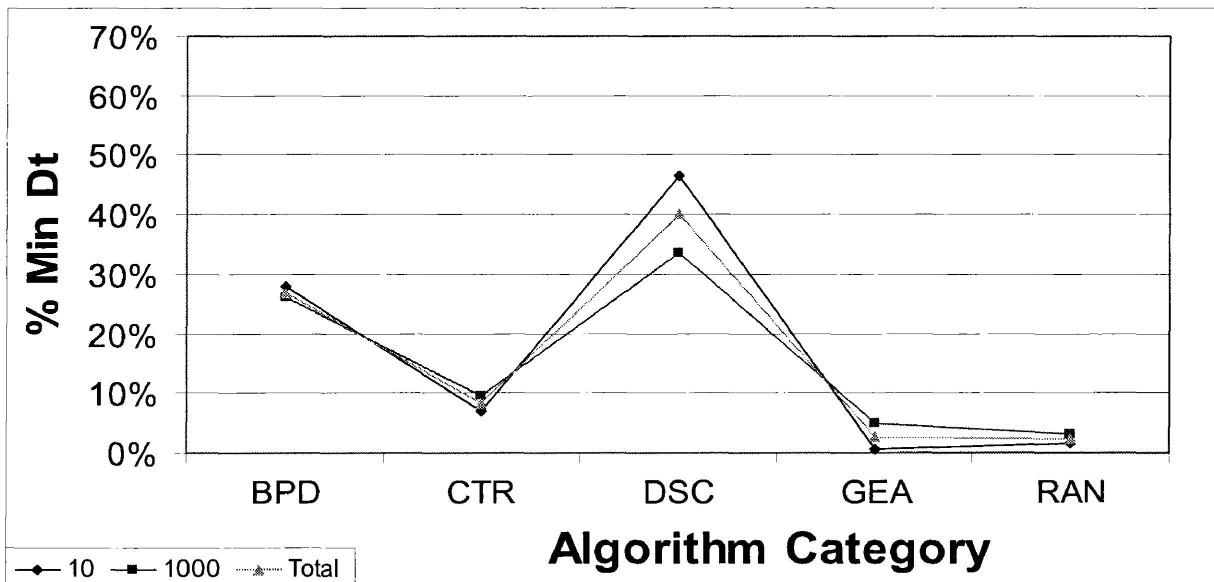


Figure 5-93 (Unique)

These graphs depict the best performance of each algorithm for the two inter-partition delay variants, D = 10 and D = 1,000. GEA and RAN both appear to be affected to a minor extent when D = 10 (representing a small inter-partition delay). However, none of our other data provides any indication of a reason for this difference. The significant difference in DSC is most likely attributable to its assumption of an infinite number of clusters, creating more delay paths.

Algorithm Category vs. % Min D_t (E-Split & V-Split)

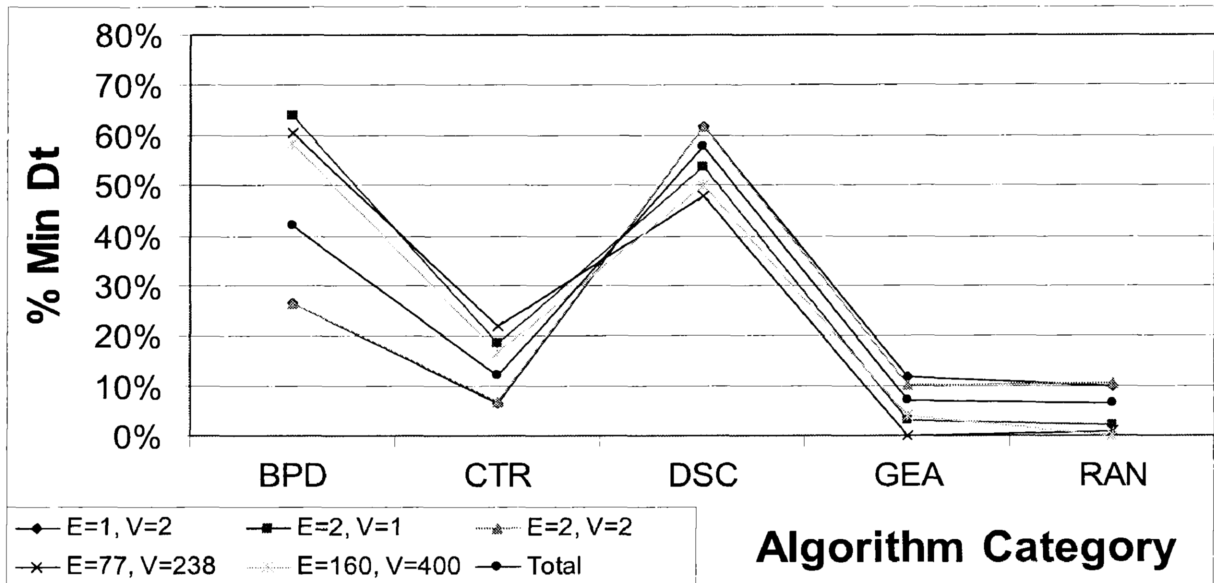


Figure 5-94 (Total)

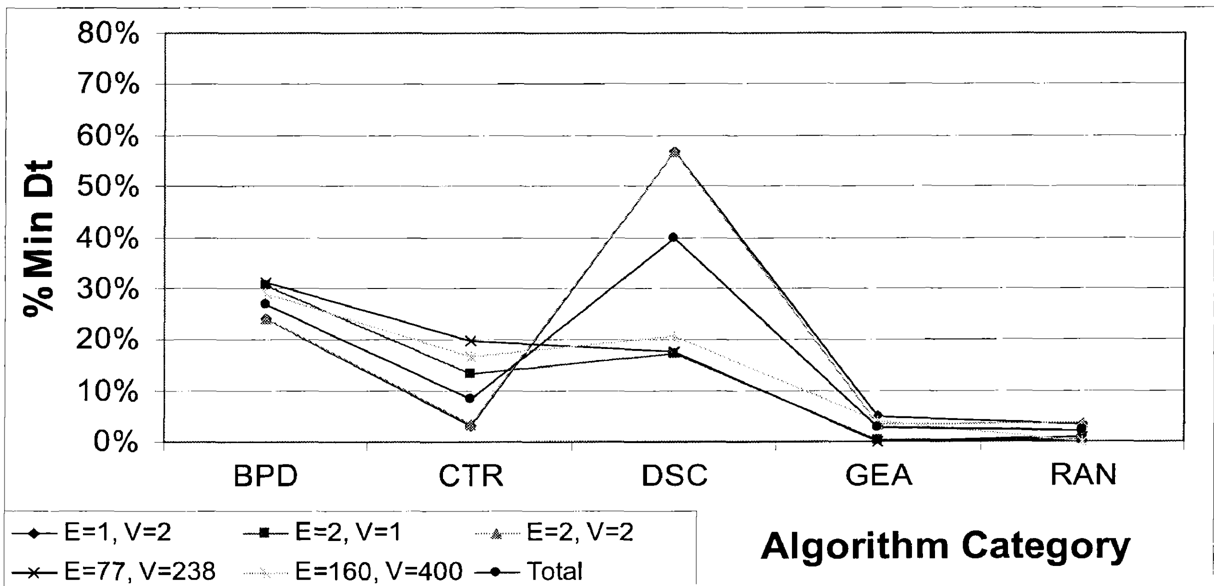


Figure 5-95 (Unique)

E and V represent the |E| and |V| limits for a partition. When E = 77 and V =238, we are simulating mapping to Xilinx's Spartan FPGA family. For E = 160 and V = 400, we are mapping to Lucent's Orca 2 FPGA family. Overall, E = 2, V = 2; E = 160, V = 400; E = 2, V = 1; E = 77, V = 238; and E = 1, V = 2, are the easiest to hardest limits to partition. DSC comes in second, and third, for some E and V limits. GEA and RAN are relatively unaffected by the partition limits.

Algorithm Category vs. % Min D_t (|E| / |V|)

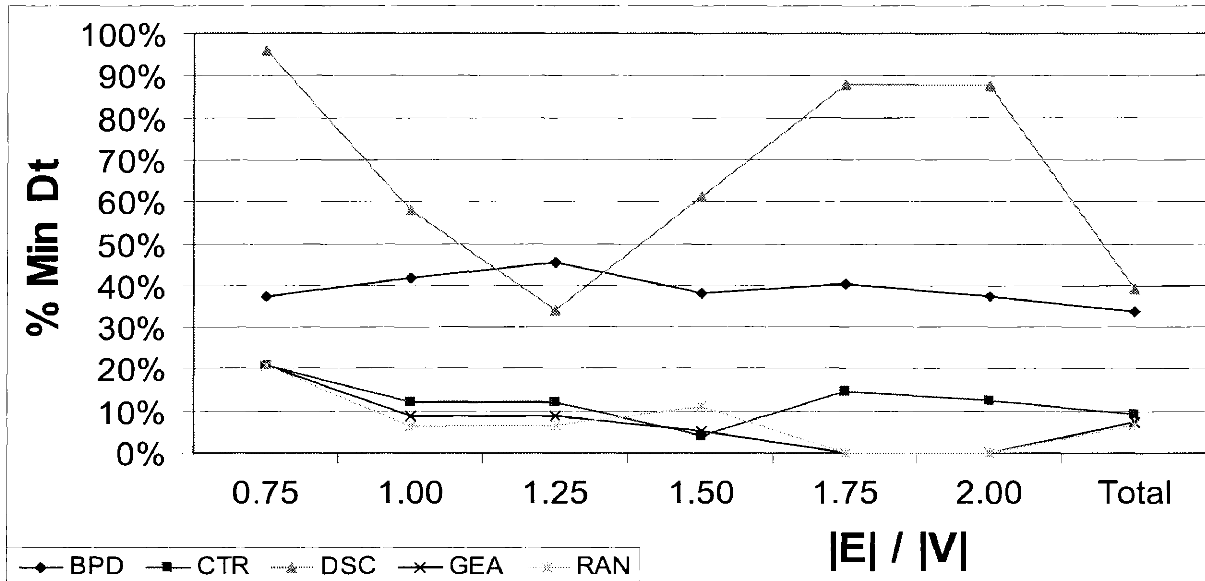


Figure 5-96 (Total)

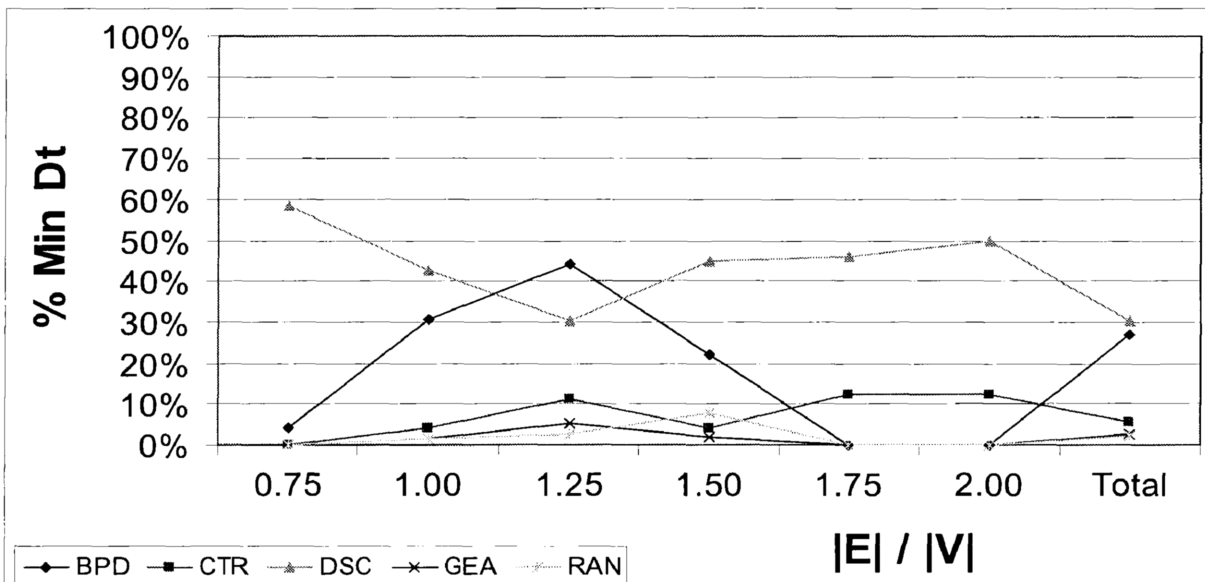


Figure 5-97 (Unique)

The clear differentiation between various density levels should come as no surprise, as the graph family also had significant impact on an algorithm's performance. We have another reversal, when the density is 1.75 – 2.00, as BPD drops from 40% of the total minimums to 0% of the unique minimums. Additionally, the CTR, GEA and RAN algorithms offer relatively similar total and unique performance. It is important to recall CTR took less execution time than RAN and GEA.

5.8. CTR Variants Analysis

The results of CTR used in the previous section were the best results for each particular graph and control parameter limits combination. The CTR algorithm had four variants used to obtain these results. These variants determined what the first vertex in each sub-graph would be. All variants used the same eccentricity data. The variants are:

1. RAX: Root Mean Square (RMS) Maximum Eccentricity
2. RIN: Root Mean Square (RMS) Minimum Eccentricity
3. SAX: Summed Maximum Eccentricity
4. SIN: Summed Minimum Eccentricity

Although the CTR algorithm has no explicit |E| limits currently, it was extremely successful in adhering to them with the selected test graphs. Overall, of the 3,312 test cases obtained by the CTR algorithm variants, only 1.2% (40) of them failed to observe the |E| limits desired for the test. More importantly, the most successful of the four variants, RIN, had no violations of the |E| limits. RAX failed both by itself and in conjunction with SAX and SIN. However, SAX and SIN never failed in the same scenario. In order of increasing violations, the variants had the following |E| limit violations: RIN (0), SAX (4), SIN (12), RAX (24).

Additionally, the only cases where the CTR variants failed to observe the |E| limits was in the dual bi-partitioning case (E-split = 2, V-split = 2). Eight graphs were affected by this: GeoGX_16, GeoT1_100, GeoT2_16, GeoT2_9, GeoT3_16, GeoT4_25, GeoT4_36 and GeoT4_64. Note that no Caterpillar or Geometric Grid-Square variants had any violations of |E| limits by the CTR algorithm. The following reflects the same analysis as before .

Graph Family vs. % Min D_t

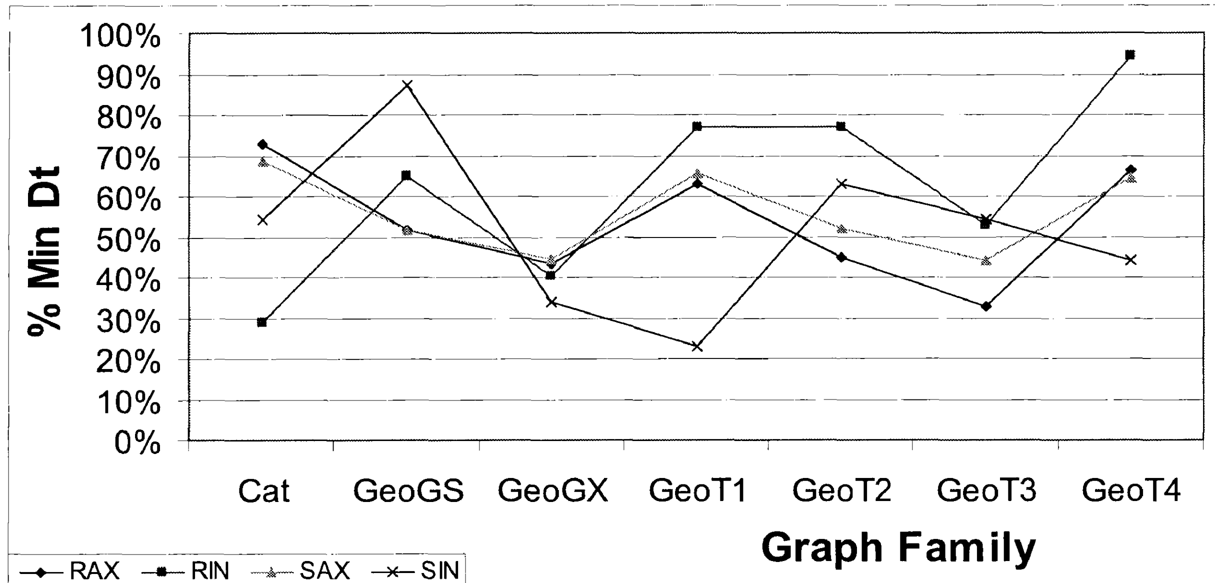


Figure 5-98 (Total)

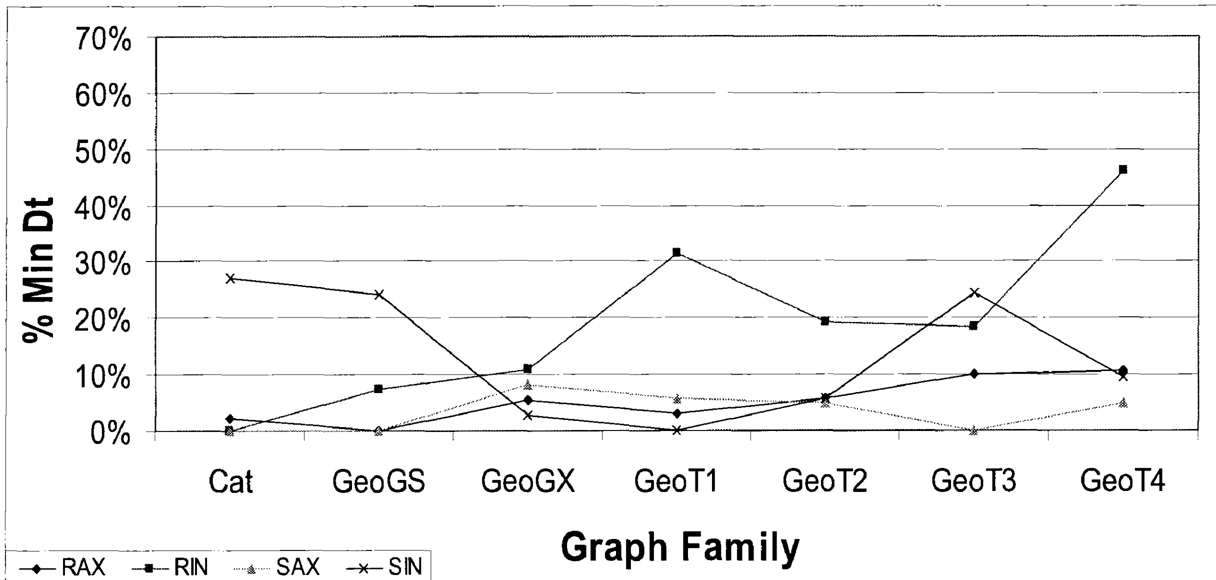


Figure 5-99 (Unique)

From this graph, we are able to observe that SIN is the preferred variant for the Caterpillar and Geometric Grid-Square and Geometric Touch-3 graph families. The RIN variant dominates in the Geometric Grid-X and Geometric Touch-1/2/4 graph families. Both SAX and RAX, average 10% or below (unique). They are only close to outperforming RIN and SIN in the Caterpillar (total) and Geometric Grid-X (unique) scenarios.

Algorithm Category vs. % Min D_t (Circuit & Schedule)

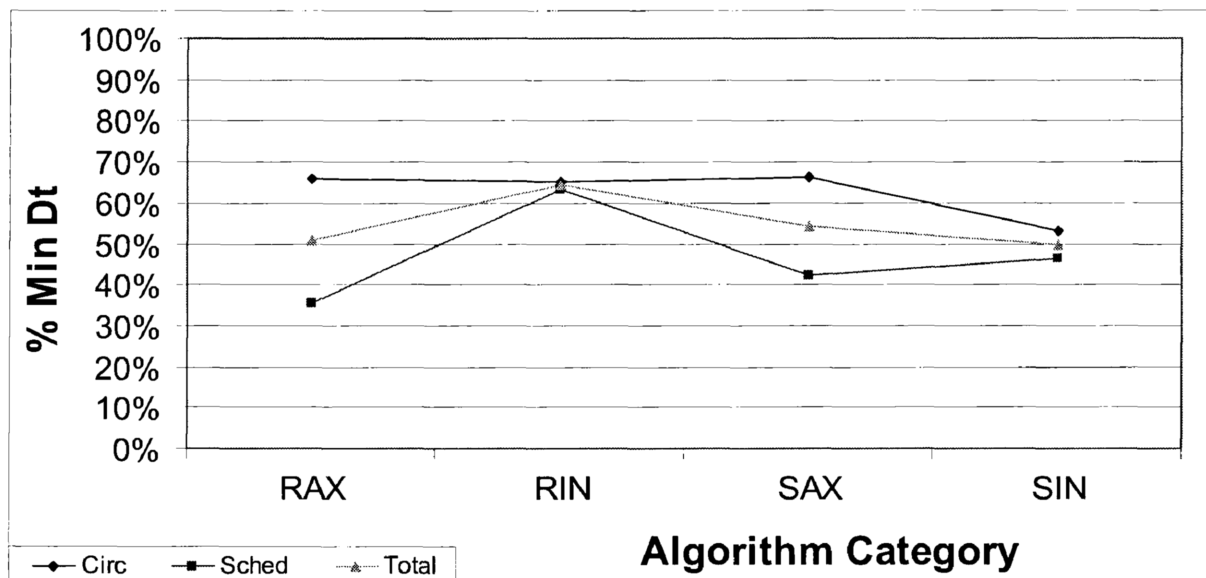


Figure 5-100 (Total)

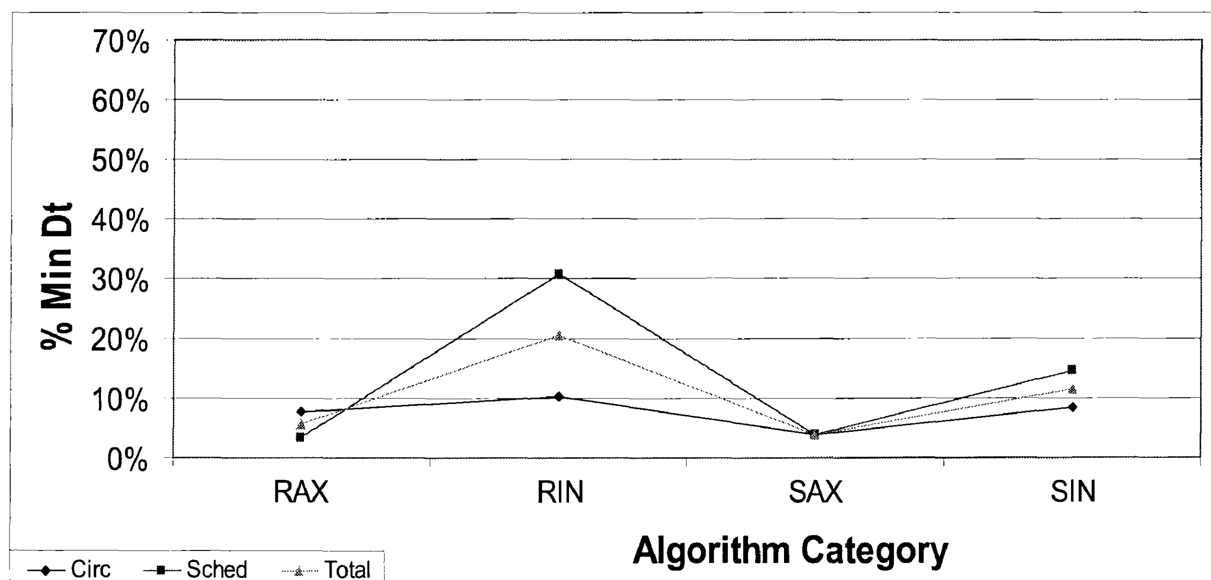


Figure 5-101 (Unique)

For the total number of minimums, all four variants are better with circuit results than schedule results. However, in terms of the unique results, they each find more unique results for schedule than circuit domains, except for RAX. However, CTR performed better on the schedule domain in the cross-algorithm category of the previous section. For the total number of results, SAX is better with circuit graphs and RIN with schedule graphs. RIN is preferred for unique results.

Algorithm Category vs. % Min D_t (D=10 & D=1,000)

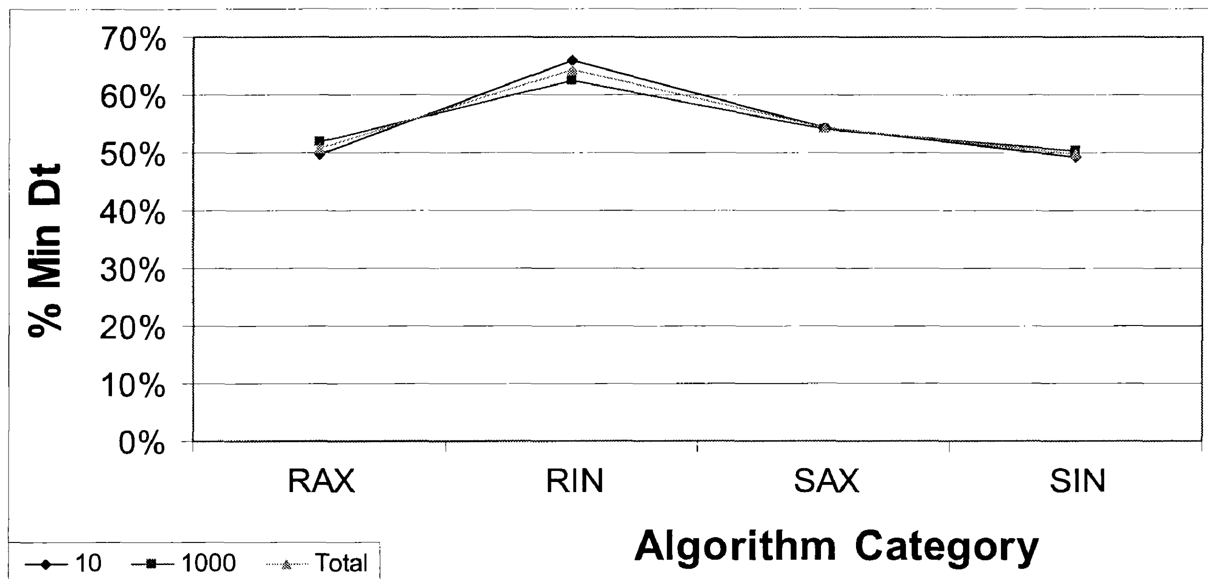


Figure 5-102 (Total)

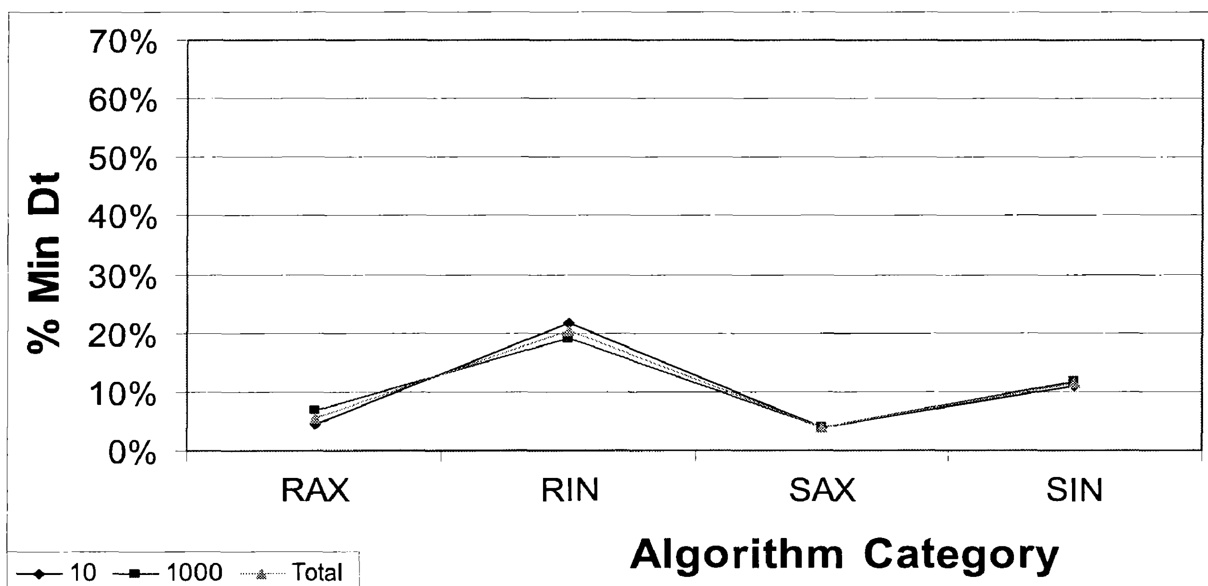


Figure 5-103 (Unique)

All four variants obtain tightly clustered results, regardless of the relative inter-partition delay value. RIN has nearly a 10% lead , followed by SIN, RAX and SAX. Essentially, the inter-partition delay value does not have an impact on the variant's partitioning results.

Algorithm Category vs. % Min D_t (E-Split & V-Split)

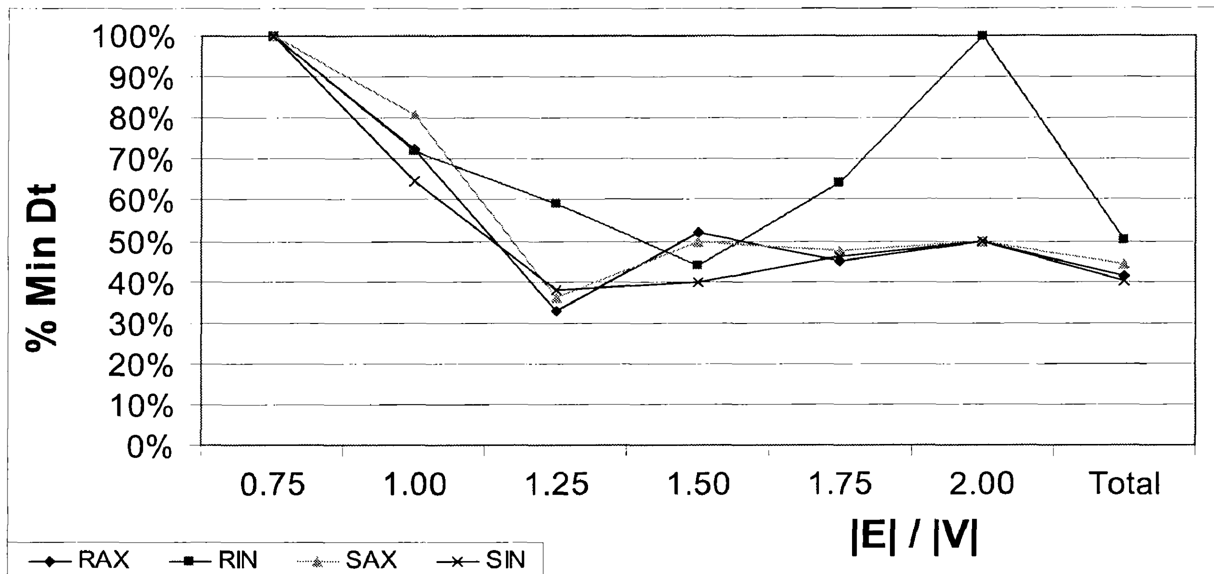


Figure 5-104 (Total)

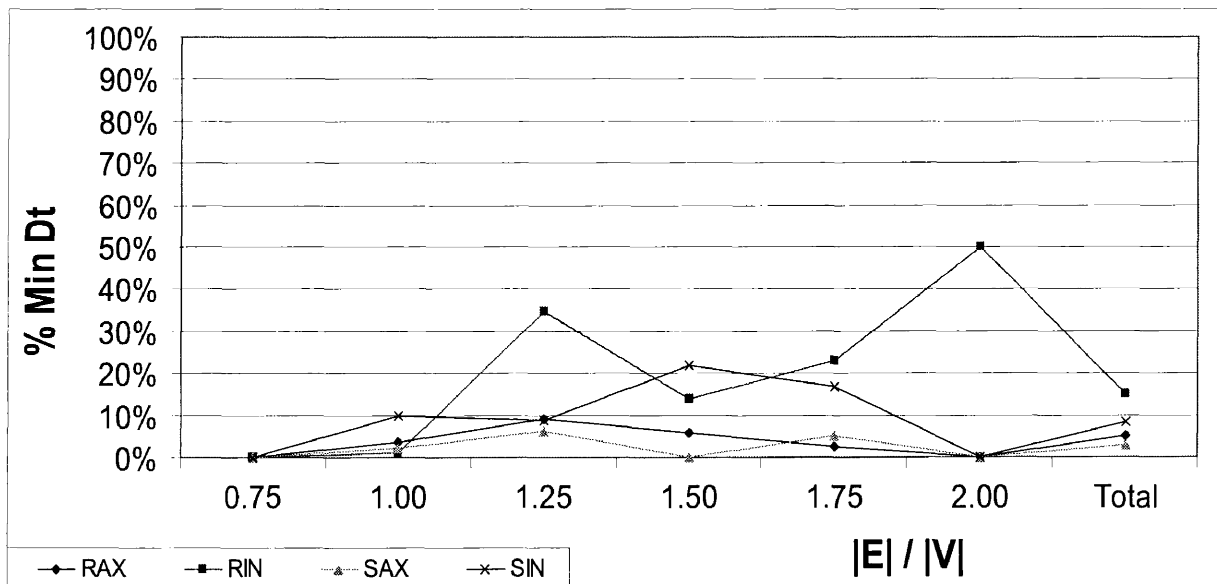


Figure 5-105 (Unique)

As in the cross-algorithm category analysis, the density factor has an impact on the performance of the CTR variants. It has a relation to the graph family comparison, although it is different. RIN and SIN are the clear leaders for the {1.25, 1.75, 2.00} and {0.75, 1.00. 1.50} density values, respectively. This is true for both the total and unique results values except for when the density is approximately 1.00.

Algorithm Category vs. % Min D_t (|E| / |V|)

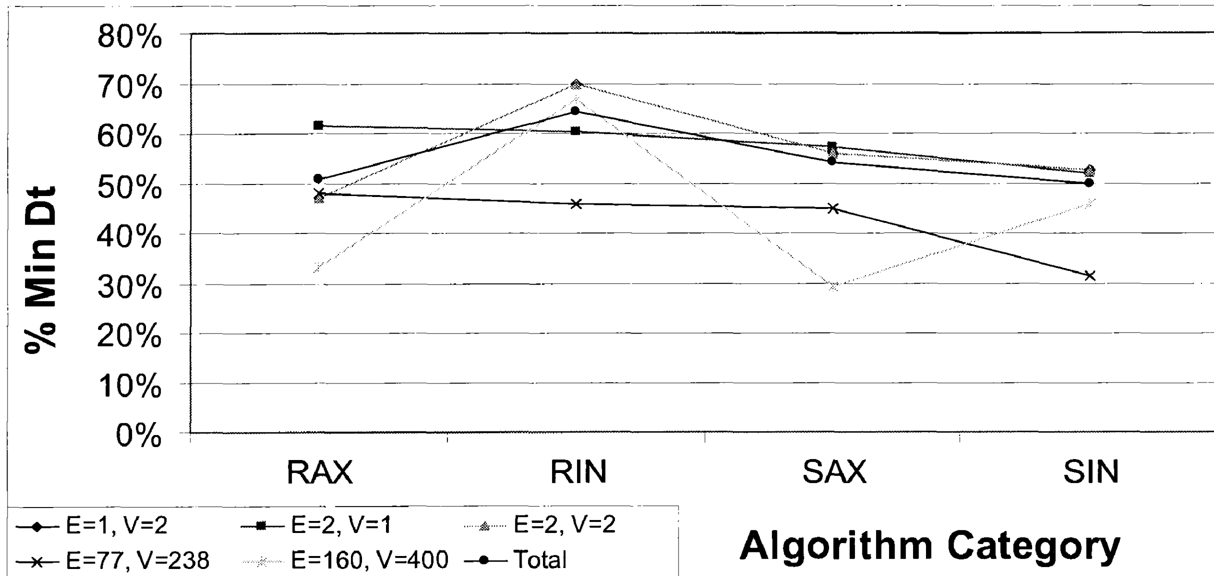


Figure 5-106 (Total)

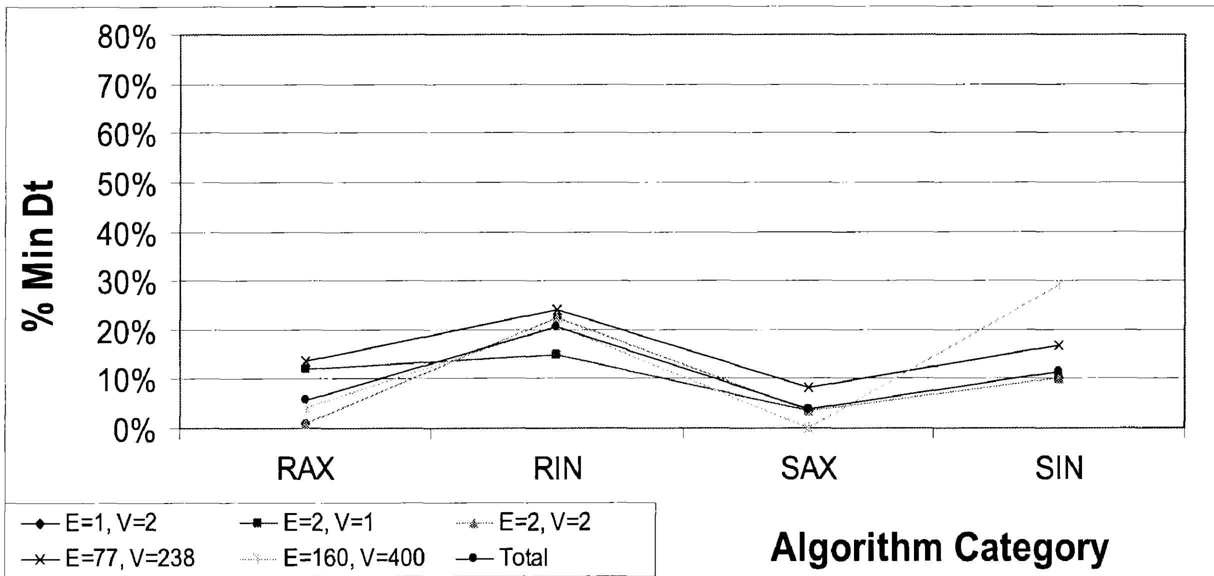


Figure 5-107 (Unique)

These results are somewhat unusual. For the total percentage minimum D_t values, RIN is the leading variant, followed by RAX, SAX and SIN. However, for the unique percentage minimum D_t values, RIN, followed by SIN, RAX and SAX is the order. Notably, the SIN variant outperforms RIN when |E| = 77 and |V| = 238.

5.9. GEA Variants Analysis

As with CTR, the GEA algorithm also was tested with four different variants. However, while the CTR variants affected the seed vertex for a partition, essentially a pre-processing stage, the GEA variants where primary algorithm variants. Although there were a number of parameters to experiment with, the two selected were the crossover operator used and the level of elitism. For each parameters, there are two values, providing four variants.

The crossover operators used were the PMX (permutation) and OX (order) operators. The two elitism values used were E = 0.50 and E = 0.75. Thus, for the current generation, parents and offspring, 50% and 75% of the best individuals, respectively, were selected. For both variants, 25% were randomly selected from the remaining parent individuals. When E = 0.50, the other 25% were selected from the remaining offspring of the current generation. The combined variants are:

1. oxE50 : OX / E = 0.50
2. oxE75 : OX / E = 0.75
3. pmxE50 : PMX / E = 0.50
4. pmxE75 : PMX / E = 0.75

As we have seen, the GEA algorithm experienced relatively poor performance relative to the others. However, it is also the only algorithm to evaluate complete solutions every iteration and observe the |E| and |V| limits. It is also the one most likely to observe a relative significant speed improvement if converted to a compiled language. Additionally, it is flexible, as we were able to use its' encoding scheme for the RAN algorithm, a load-balancing solution for our experiments and the map-coloring sample provided later.

Graph Family vs. % Min D_t

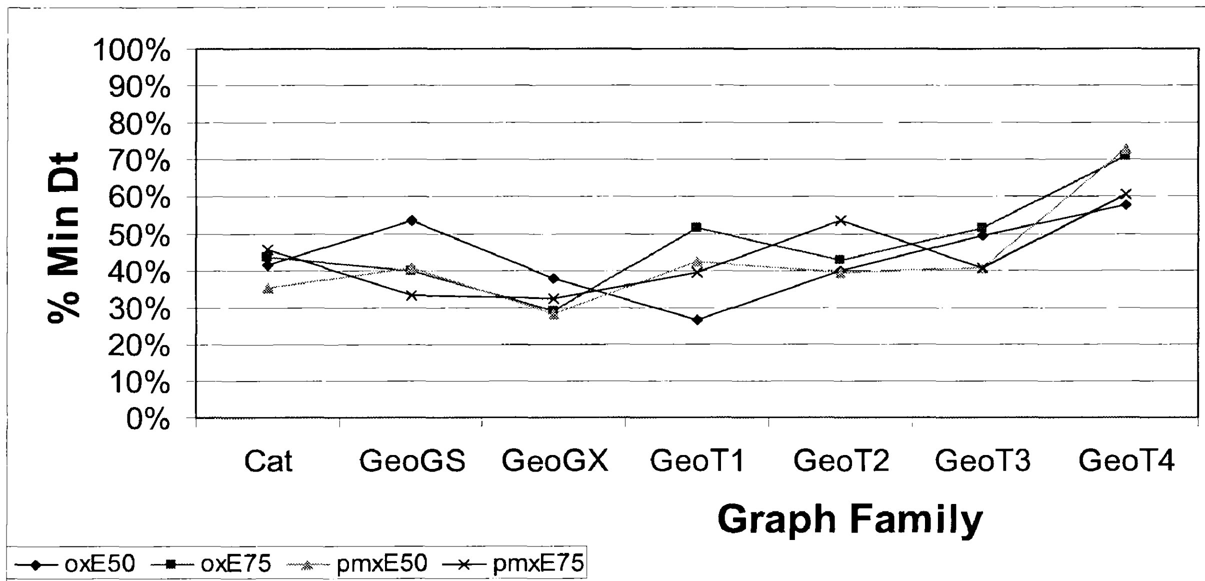


Figure 5-108 (Total)

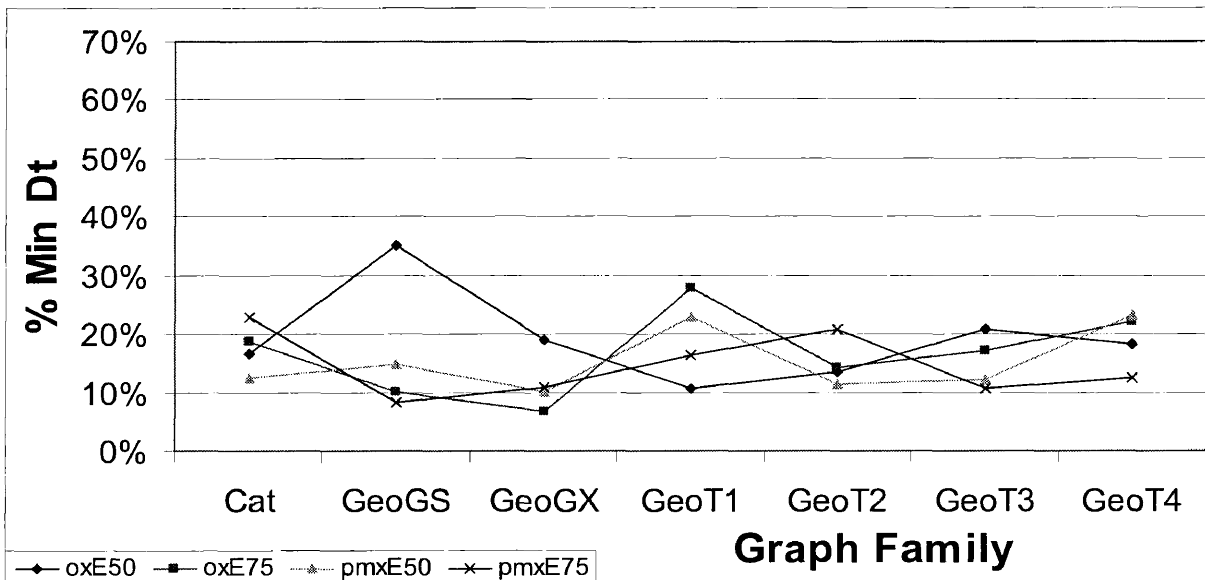


Figure 5-109 (Unique)

As with the other comparisons, graph family is a significant predictor of an algorithm's performance. The graph family's and their best variants are: Cat:pmxE75, GeoGS:oxE50, GeoGX:oxE50, GeoT1:oxE75, GeoT2:pmxE75, GeoT3:oxE50, GeoT4:pmxE50. The OX operator dominance is relevant through the remainder of this analysis, along with the slight edge of the E = 0.75 parameter.

Algorithm Category vs. % Min D_t (Circuit & Schedule)

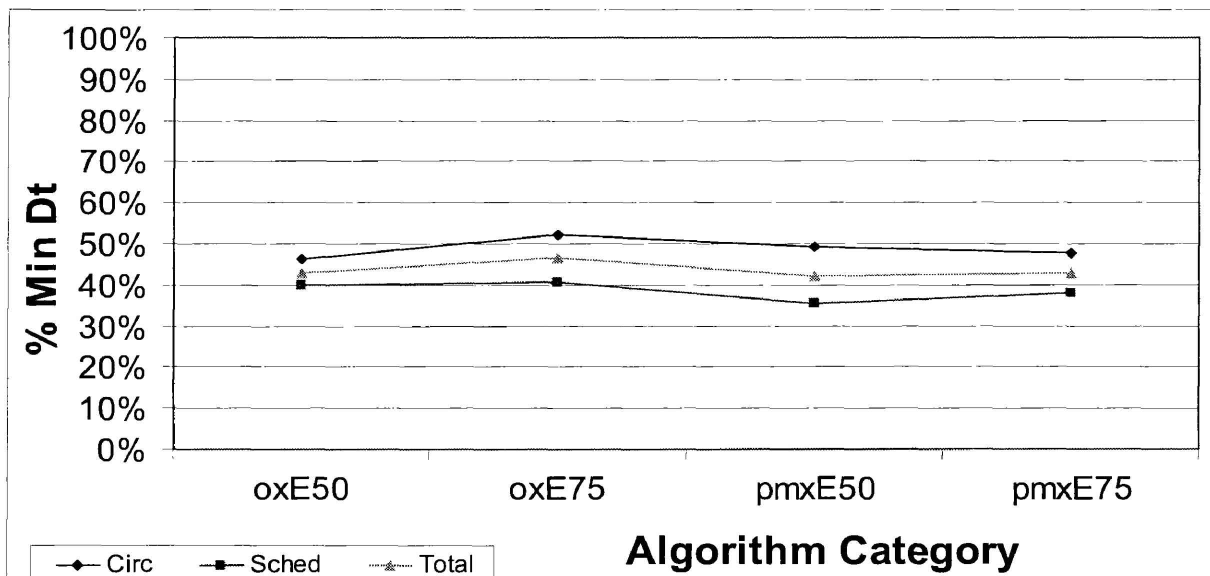


Figure 5-110 (Total)

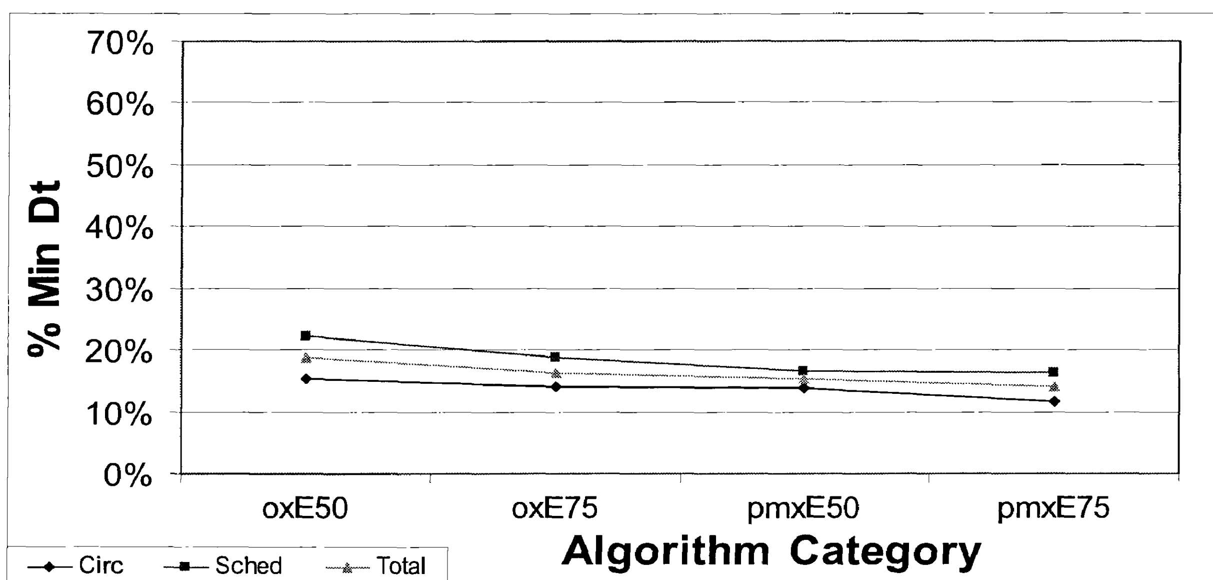


Figure 5-111 (Unique)

GEA's better performance on circuit vs. schedule is due to the fact that position information is not directly used in the evaluation. In other words, a vertex can still be in the same subgraph in two different array positions, but the position has no direct impact on the algorithm execution or individual evaluation. However, complete reversal occurs with the unique values, where schedule percentages are higher than circuit percentages. Again, OX / E = 0.75 have the leading results.

Algorithm Category vs. % Min D_t (D=10 & D=1,000)

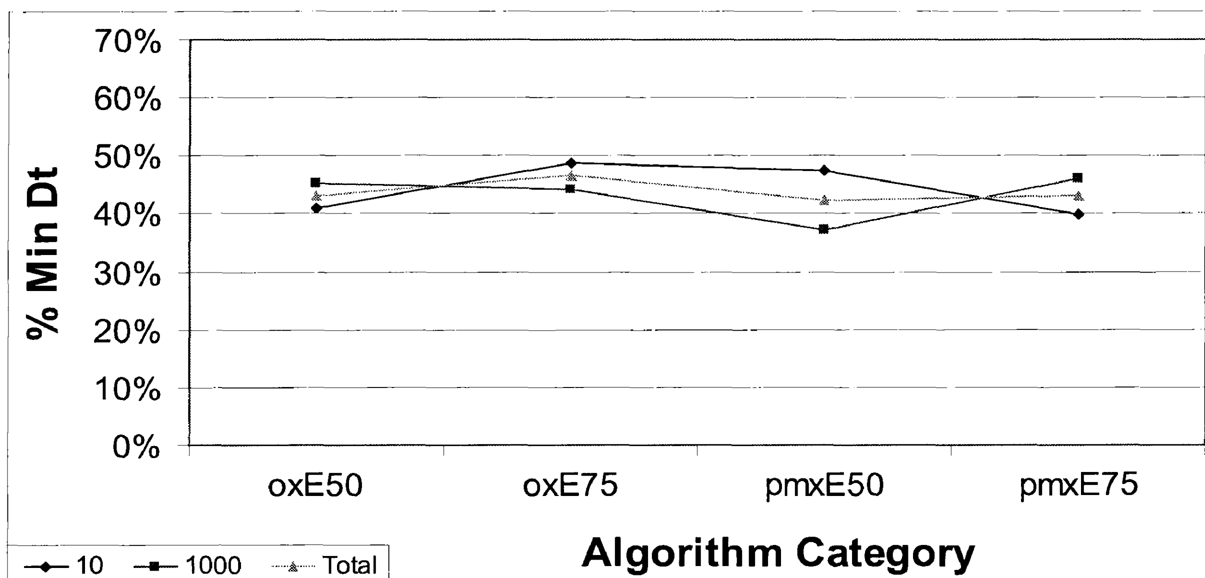


Figure 5-112 (Total)

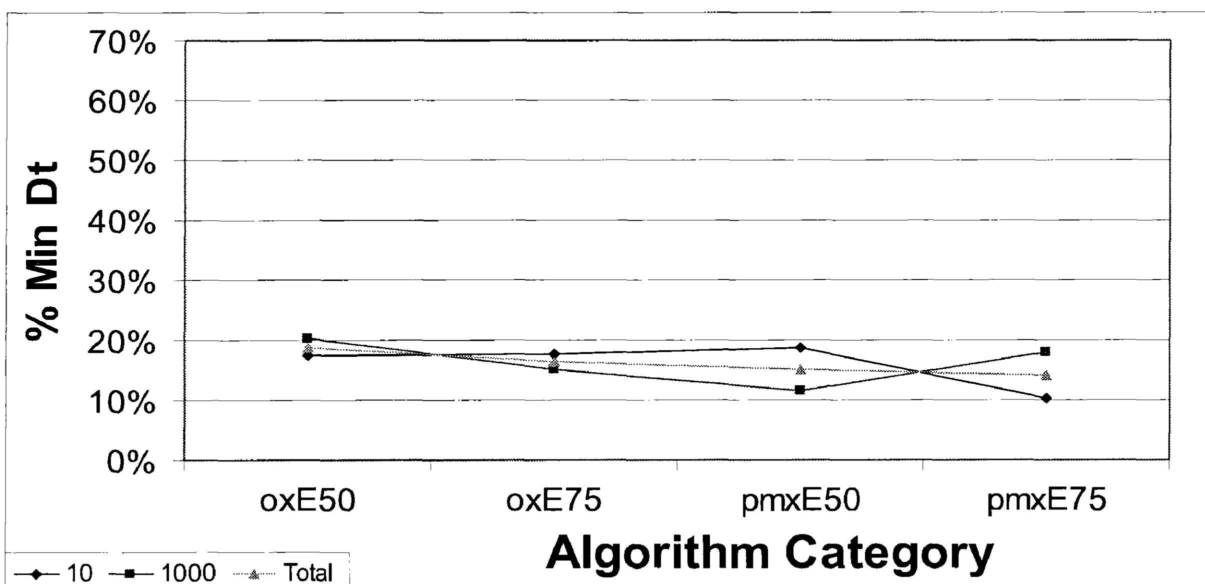


Figure 5-113 (Unique)

The OX and E = 0.75 parameter again take the lead. However, we have an interesting performance results. For D = 10, OX / E = 0.75 outperforms OX / E = 0.50 and PMX / E = 0.50 outperforms PMX / E = 0.75. The exact opposite is true for D = 1,000. This holds for both the total and unique results, with no reversals.

Algorithm Category vs. % Min D_t (D=10 & D=1,000)

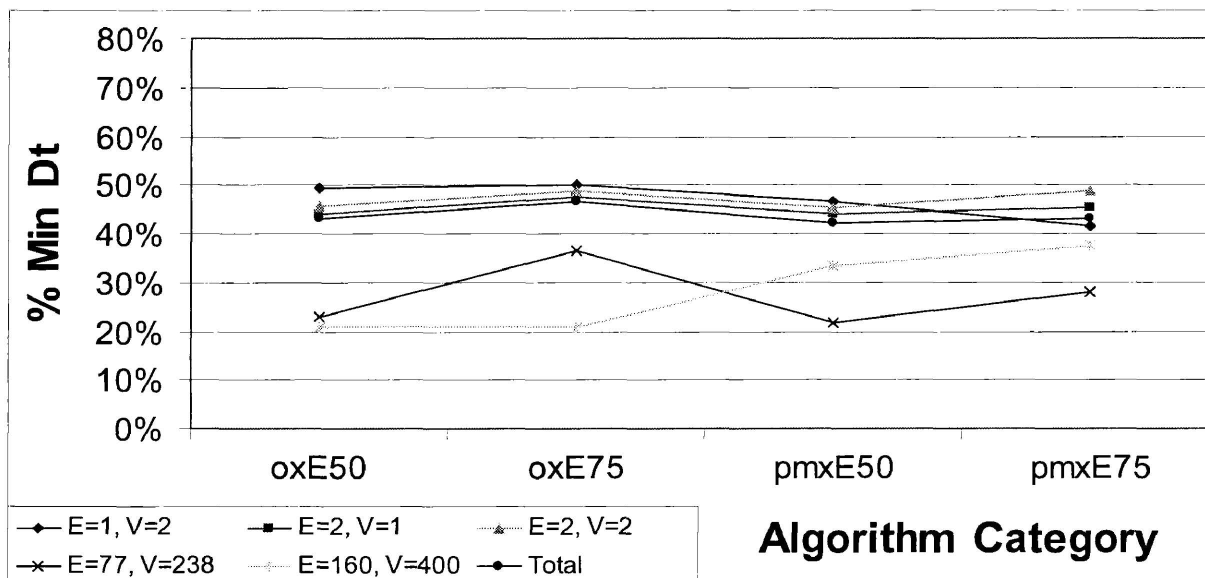


Figure 5-114 (Total)

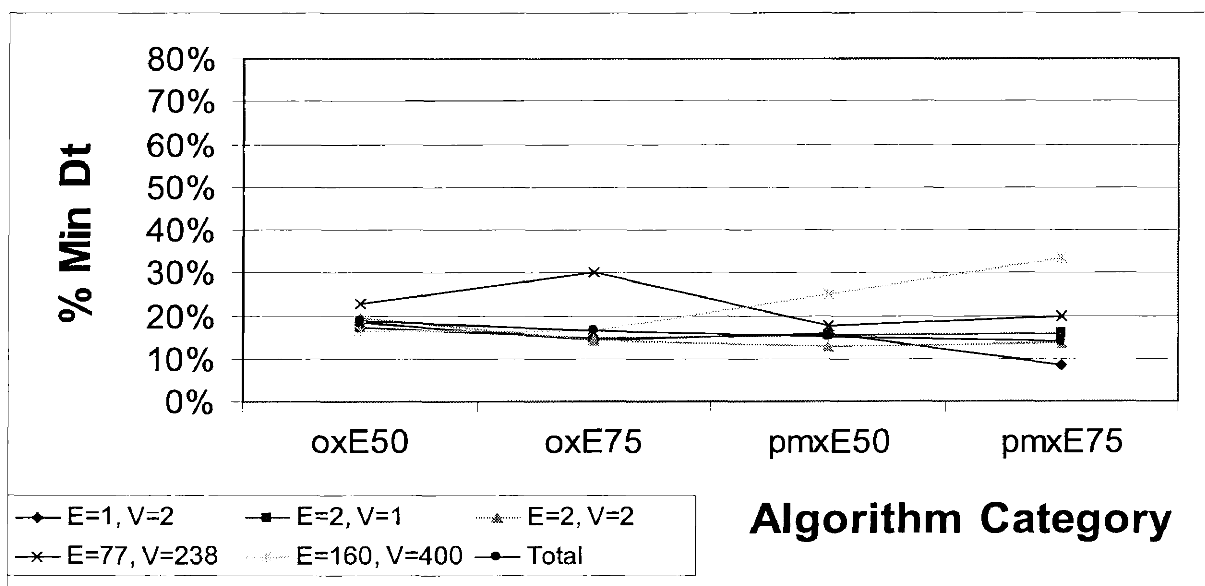


Figure 5-115 (Unique)

For the most part, OX / E = 0.75 again have the leading results. However, the two FPGA samples we used have a significantly different results, where the OX operator is better for Xilinx's Spartan family, while Lucent's Orca 2 family is handled well by the PMX operator. This result is observable on both the total and unique graphs.

Algorithm Category vs. % Min D_t (|E| / |V|)

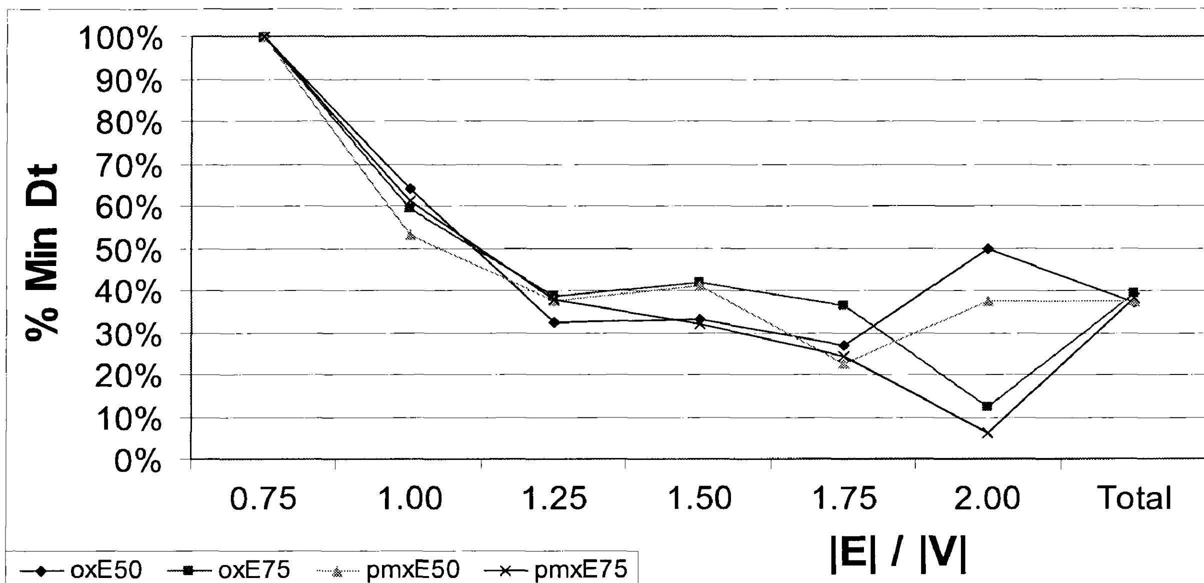


Figure 5-116 (Total)

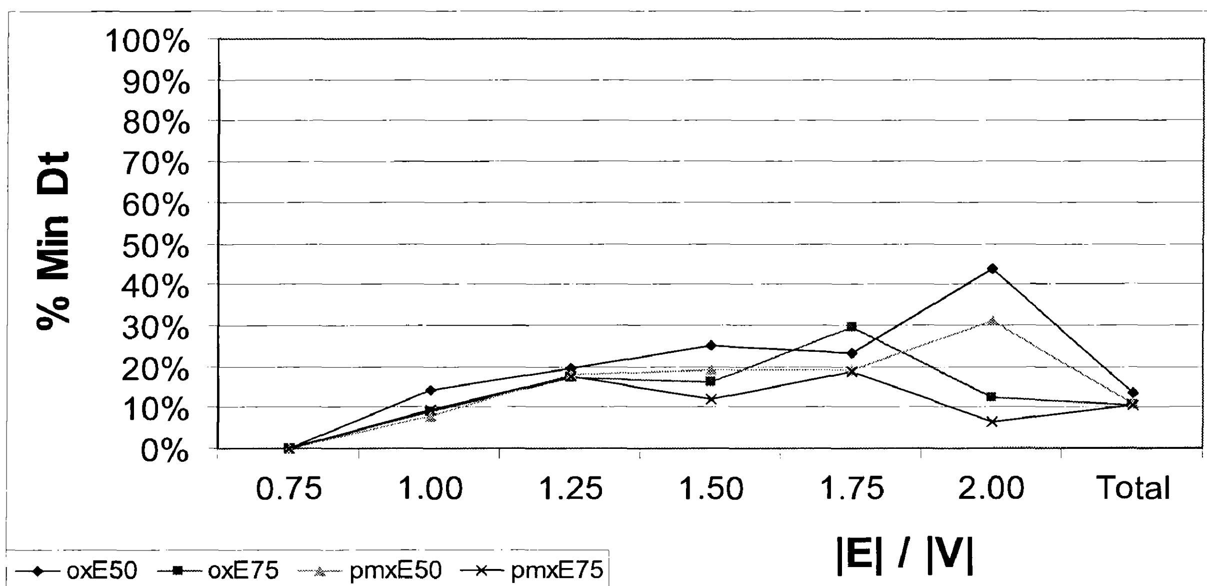


Figure 5-117 (Unique)

The significant difference between these two graphs can be partially explained by the fact that the lower density graphs also have a lower number of vertices and edges. These graphs are thus easier to partition. However, the remaining values, |E| / |V| > 1.25 offer good discussion. We can see the OX and E = 0.75 dominance, although the PMX operator has a strong showing. This is further evident from the tight clustering of the average totals of all variants on both graphs.

5.10. AGPM Sample

Here we attempt to synthesize the results presented in this chapter, coupled with the long-term objectives of the *AGPM*, to provide a sample of how it would appear and be used. Additionally, factors as to whether replication or the number of available partitions (k) would also be available. This data would be accessible based on any subset of these metrics. Note that only critical metrics, based on our analysis, are included here.

Graph Structure	**Density (|E| / |V|)**	**Domain**	**|E|-limit**	**|V|-limit**	**Algorithm**
Caterpillar	≤ 1.25	Circuit	1	1	BPD
Geometric Grid	> 1.25	Schedule	2	2	CTR
Geometric Grid-X			Lucent Orca 2	Lucent Orca 2	DSC
Geometric Touch			Xilink	Xilink	GEA
			Spartan	Spartan	RAN

For instance, the relative inter-partition delay was not determined to be significant. Most notably, the original graph structure and density, coupled with the application domain, were most significant. As mentioned earlier, future research would directly benefit from some measure of the linearity and/or granularity of both the original graph(s) and the resulting partition(s). These metrics would then be used to determine which algorithm to use, based on the input and the desired output controls.

For example, when working with Caterpillar graphs and in the Circuit domain, BPD is the optimum algorithm. However, if replication is not permitted, then the RAX and RIN variants of CTR, or perhaps the OX/E = 0.75 and PMX/E = 0.75 variants of GEA are best suited. If we are afforded an unlimited number of devices (e.g. raw speed is the only objective), then DSC is the algorithm of choice.

6. Conclusions

In summary, we have presented the abstract graph partitioning model, or *AGPM*. This model was used to compare two domain-specific algorithms. Additionally, the *AGPM* was successfully used to demonstrate its ability to motivate the development of our two proposed algorithms. We also provided a comparison of graph visualization tools. Along with the comparison, development was begun on a research database, *PRRS* [Appendix B], and a unifying GUI, *GraphOp*. Future plans would call for the conversion of *GraphPar* to a compiled language [Appendix D].

6.1. Future Directions

Future directions of research are significant with any number of possible avenues. There is of course, the continued development of the *AGPM*. Although we offer only a sample here, compilation of both existing and future data would serve to further determine which metrics are useful when distinguishing between domains and algorithmic results. Eventually, the final product would be a multi-dimensional matrix or a decision tree leading users to the desired algorithm, while providing an explanation of the domains and graphs the algorithm is best suited for.

We also suggest further experimentation with any of the discussed algorithms: BPD, DSC, GEA, CTR, and RAN. BPD would benefit from replication limits, versus having it as a post-processing step (if required). DSC, while extremely efficient, could be mapped to domains requiring a k-way partition limit. GEA warrants further comparison,

specifically with the encoding schemes that we used to develop ours. The CTR algorithm would benefit from other metrics suitable for determining a center and classification of when each metric is useful.

Our plan to develop a cohesive and robust graph ADT, coupled with a similarly capable GUI did not materialize. This application would entail two levels, akin to Richard Parris' Peanut Software, *Windisc*, where one can view the results only or step through an algorithm [43]. Furthermore, it would be extensible, in the file formats, GUI capabilities and algorithms supported. A command-line mode, as with *GraphViz* or *Maple V*, would reduce CPU load when the GUI is not needed.

6.2. Additional Graph Sources

On a different note, *The Stanford GraphBase*, by Donald E. Knuth, was written to provide a robust and common platform for graph-theoretic computing [14]. In addition to having a robust data structure, standard graphs are available at <ftp://labrea.stanford.edu/pub/sgb/>. We mention this resource as a useful background text and point of further research. We became aware of it in the latter stages of the own implementation, but did not incorporate it into our work. The primary thread of discussion demonstrates that while theoretical results are important, how algorithms behave with real-world data is often more useful.

Also available are a large number of sample graphs from several sources. We would be remiss if we failed to mention the *C17* circuit used in our examples. This circuit, along

with 10 others was distributed to attendees of the 1985 International Symposium of Circuits and Systems. It, along with other benchmark graphs, is available at <http://zodiac.cbl.ncsu.edu/> [19, 13]. One of the authors of *Graph Drawing*, Giuseppe Di Battista, has a collection of directed and undirected graphs available for download at <http://www.inf.uniroma3.it/people/gdb/wp12/LOG.html> [22]. Finally, our primary graph drawing tool, *GraphViz*, has a number of graphs available at <http://www.research.att.com/sw/tools/graphviz/refs.html> [47].

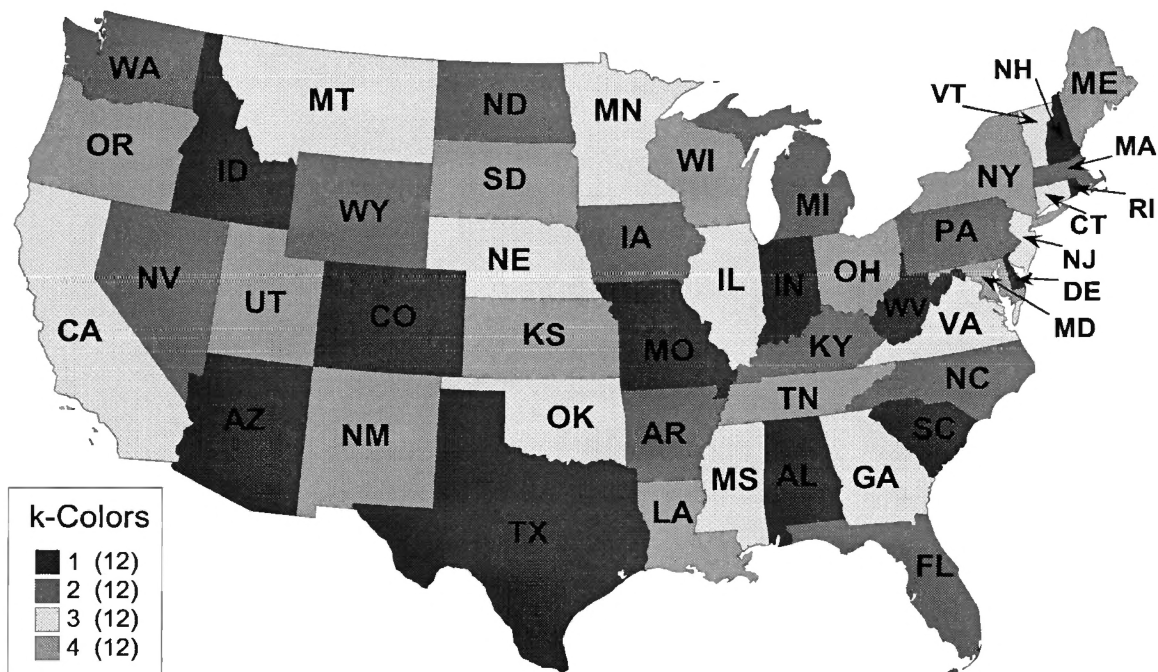
6.3. Alternative Applications

In addition to the primary domains studied during this research, the *AGPM* is extendible to additional domains. For instance, the map-coloring problem may be viewed as a graph partitioning problem. The minimum number of colors needed to color a map is four, however, finding a four-color solution is often difficult [25, 15]. Jones and Beltramo also studied U.S. map colorings [15].

The problem is formulated as follows: with only four colors, color twelve states with each color in the continental U.S., while also equalizing the geographic area for each color and minimizing the adjacent edge distance with the other colors. Geographic data was collected from the Rand McNally Atlas [1], courtesy of Andrew Stalcup. Using the GEA algorithm, UNO's server, Apollo, gave the following solution in ten hours.

Color	States	Coverage (mi^2)	Adjacent Distance (mi)
C_1	AL, AZ, CO, DE, ID, IN, MO, NH, RI, SC, TX, WV	783,722	0
C_2	AR, FL, IA, KY, MA, MI, NC, ND, NV, PA, WA, WY	702,351	0
C_3	CA, CT, GA, IL, MN, MS, MT, NE, NJ, OK, VA, VT	748,240	0
C_4	KS, LA, MD, ME, NM, NY, OH, OR, SD, TN, UT, WI	725,174	0

A visual depiction of this data is provided below. When reviewing this image, note the four-corner border of Colorado (CO), New Mexico (NM), Arizona (AZ) and Utah (UT) meets the zero-length distance requirement. However, no explicit provision was made for zero-length borders.



We were also able to load-balance our experiments via the genetic algorithm. This was done based on the number of vertices in a specific graph, using the genetic algorithm. This same variant could be used to allocate a set of unrelated tasks to a set of individuals or machines.

6.4. Final Thoughts

We view this research as a stepping-stone for further work, both for other researchers and ourselves. The AGPM, existing algorithm variants, new algorithms variants and supporting utilities all offer a rich number of directions for additional in-depth research. We have enjoyed our experiments and hope they are of use both the academic and industrial communities.

7. References

1. 1995 Rand McNally Atlas. Rand McNally Company. 1995.

2. André DeHon. The Density Advantage of Configurable Computing. IEEE Computer. April 2000. pgs. 41-49.

3. Apostolos Gerasoulis & Tao Yang. *On the Granularity and Clustering of Directed Acyclic Graphs, Scheduling and Load Balancing in Parallel and Distributed Systems.* IEEE Computer Society Press, 1995. pg. 143-202.

4. Balkrishnan Krishnamurthy. *An Improved Min-Cut Algorithm for Partitioning VLSI Networks*. IEEE Transactions on Computers, Vol. C-33 (5), 1984. pg.438-446.

5. Bernard S. Landman and Roy L. Russo. *On a Pin Versus Block Relationship For Partitions of Logic Graphs*. IEEE Transactions on Computers, Vol. C-20 (12), Dec. 1971, pg. 1469-1479.

6. Boontee Kruatrachue. *Static Task Scheduling and Grain Packing in Parallel Processing Systems*. PhD Thesis, Oregon State University, 1988.

7. C. M. Fiduccia and R. M. Mattheyses. *A Linear-Time Heuristic for Improving Network Partitions*. Proceedings of the 19th Design Automation Conference, 1982. pg. 175-181.

8. Christos H. Papadimitriou and Mihalis Yannakakis. *Towards an Architecture-Independent Analysis of Parallel Algorithms*. SIAM Journal on Computing, Vol. 19 (2), April 1990, pg. 322-328.

9. Chuck Kring and A. Richard Newton. *A Cell-Replication Approach to Mincut-Based Circuit Partitioning*. Proceedings of the IEEE International Conference on Computer-Aided Design, Nov. 1991. pg 2-5.

10. Cristian Lenart. *A Generalized Distance in Graphs and Centered Partitions*. Siam Journal of Discrete Mathematics. Vol. 11 (2), May 1998. pgs. 293-304.

11. D. F. Wong & Rajmohan Rajamaran. *Optimal Clustering for Delay Minimization*, 30th ACM/IEEE Design Automation Conference (309-314), ACM, 1993.

12. Daryl D. Harms et al. *Network Reliability: Experiments with a Symbolic Algebra Environment*. CRC Press, Inc. Boca Raton, 1995.

13. David Bryan. The ISCAS '85 Benchmark Circuits and Netlist Format. MCNC <http://zodiac.cbl.ncsu.edu/>.

14. Donald Knuth. *The Stanford GraphBase: A Platform for Combinatorial Computing.* ACM Press, 1993.

15. Donald R. Jones and Mark A. Beltramo. *Solving Partitioning Problems with Genetic Algorithms*. Proceedings of the Fourth International Conference on Genetic Algorithms, 1991. pg. 442-449.

16. Eric Mohr, David A. Kranz, and Robert H. Halstead, Jr.. *Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs*. IEEE Transactions on Parallel and Distributed Systems, Vol. 2, no. 3, July 1991, pg. 264-280.

17. Eugene L. Lawler, Karl N. Levitt, and James Turner. *Module Clustering to Minimize Delay in Digital Networks*. IEEE Transactions on Computers. Vol C-18 (1), Jan. 1969. pg. 47-57.

18. Eugene L. Lawler. *Electrical Assemblies with a Minimum Number of Interconnections*. IRE Transactions on Electronic Computers. Feb. 1962, pg. 86-88.

19. F. Brglez and H. Fujiwara. A Neural Netlist of 10 Combinational Benchmark Circuits and a Target Translator in FORTRAN. Int. Symposium on Circuits and Systems, Jun. 1985.

20. Fred Buckley and Frank Harary. *Distance in Graphs*. Addison-Wesley Publishing Company, 1990.

21. Gerard J. Milburn. The Feynman Processor: Quantum Entanglement and the Computing Revolution. Perseus Books, Reading. 1998.

22. Giuseppe Battista et al. *Graph Drawing: Algorithms for the Visualization of Graphs*, Prentice-Hall, Inc., Upper Saddle River, 1999.

23. Gregor von Laszewski. *Intelligent Structural Operators for the k-way Graph Partitioning Problem*. Proceedings of the Fourth International Conference on Genetic Algorithms. Morgan Kaufmann, 1991. pgs. 45-52.

24. Hesham El-Rewini, Theodore G. Lewis, and Hesham H. Ali. *Task Scheduling in Parallel and Distributed Systems*. Prentice Hall, Englewood Cliffs, 1994.

25. John J. Watkins & Robin J. Wilson. *Graphs: An Introductory Approach*, John Wiley & Sons, Inc., New York, 1990.

26. John K. Ousterhout. *Tcl and the Tk Toolkit*, Addison-Wesley Professional Computing, Reading, 1994.

27. Kai Hwang. *Advanced Computer Architecture*, McGraw-Hill, St. Louis, 1993.

28. Kenneth H. Rosen. *Discrete Mathematics and Its Applications*, McGraw-Hill, Inc., St. Louis, 1995.

29. Keshav Dev and C. Siva Ram Murthy. *A Genetic Algorithm for the Knowledge Base Partitioning Problem*. Pattern Recognition Letters, Vol. 16 (8), 1995. pg. 873-879.

30. L. J. Stockmeyer. *Handbooks in Operations Research and Management Science: Computing*. Eds. E. G. Coffman, J. K. Lenstra, and A. H. G. Rinnoy Kan. Vol. 3, New York, 1992. pg. 480-517.

31. Load Sharing Facility (LSF). The Platform Company. <http://www.platform.com/>.

32. Maple V Release 5.1. Waterloo Maple, Inc. November 5, 1998. <http://daisy.uwaterloo.ca/>.

33. Marc Giusti. *MEDICIS*. 2000. <http://www.medicis.polytechnique.fr/index-eng.html>.

34. Martin Hulin. Circuit Partitioning with Genetic Algorithms Using a Coding Scheme to Preserve the Structure of a Circuit. Lecture Notes in Computer Science, vol. 496, 1990. Springer-Verlag, New York. Pgs. 75-9.

35. Melvin A. Breuer. *General Survey of Design Automation of Digital Computers*. Proceedings of the IEEE, Vol. 54 (12), Dec. 1966. pg. 1708-1721.

36. Michael Himsolt. *GraphLet*, University of Passau, Germany. July 1999, <http://www.infosun.fmi.uni-passau.de/Graphlet/>.

37. Naveed Sherwani. *Algorithms for VLSI Physical Design Automation*. Kluwer Academic Publishers, Norwell, MA, 1997.

38. *ORCA Series 2 Data Sheet*. Lucent Technologies, Jun. 1999. <http://www.lucent.com/micro/fpga/>.

39. Paramartha Dutta. *An Evolutionary Heuristic for Knowledge Base Partitioning Problem*. IEEE International Conference on Evolutionary Computation, 1997. pg. 657-662.

40. R. Murgai, R. K. Brayton and A. Sangiovanni-Vincentelli. *On Clustering for Minimum Delay/Area*. IEEE International Conference on Computer-Aided Design (ICCAD), 1991. pg. 6-9.

41. Ralph P. Grimaldi. *Discrete and Combinatorial Mathematics*. Addison-Wesley Publishing Company, New York, 3rd ed., 1994.

42. Randy L. Haupt and Sue E. Haupt. *Practical Genetic Algorithms*. John Wiley & Sons, New York, 1998.

43. Richard Parris. 2000. <http://math.exeter.edu/rparris/windisc.html>.

44. Roy L. Russo, Peter H. Oden, and Peter K. Wolff, Sr. *A Heuristic Procedure for the Partitioning and Mapping of Computer Logic Graphs*. IEEE Transactions on Computers, Vol. C-20 (12), Dec. 1971, pg. 1455-1461.

45. Scott Hauck. Multi-FPGA Systems. University of Washington, PhD Thesis, 1995.

46. *Spartan and Spartan XL Series Data Sheet*. Xilinx, Jan. 1999. <http://www.xilinx.com/products/spartan.htm>.

47. Stephen C. North. *GraphViz*, AT&T Labs – Research, Florham Park, NJ. September 1999, <http://www.research.att.com/~north/graphviz/>.

48. Sun Y. Kung. *VLSI Array Processors*. Prentice Hall, Englewood Cliffs, 1988. pgs. 374-83.

49. Ted Lewis and Boontee Kruatrachue. *Grain Size Determination for Parallel Processing*. Scheduling and Load Balancing in Parallel and Distributed Systems. IEEE Computer Society Press, 1995. pg. 143-202.

50. Thang N. Bin and Andrew Peck. *Partitioning Planar Graphs*. SIAM Journal on Computing, Vol. 21 (2), April 1992, pg. 203-215.

51. Thang N. Bui and Byung R. Moon. *Genetic Algorithm and Graph Partitioning*. IEEE Transactions on Computers, vol. 45, no. 7, July 1996, pg. 841-55.

52. Thang N. Bui and Curt Jones. *Finding Good Approximate Vertex and Edge Partitions is NP-Hard*. Information Processing Letters, Vol. 42 (3), May 1992, pg. 153-159.

53. *The American Heritage® Dictionary of the English Language, Third Edition*. Houghton Mifflin Company, 1992. Microsoft Corporation, 1996. Electronic version licensed from InfoSoft International, Inc.

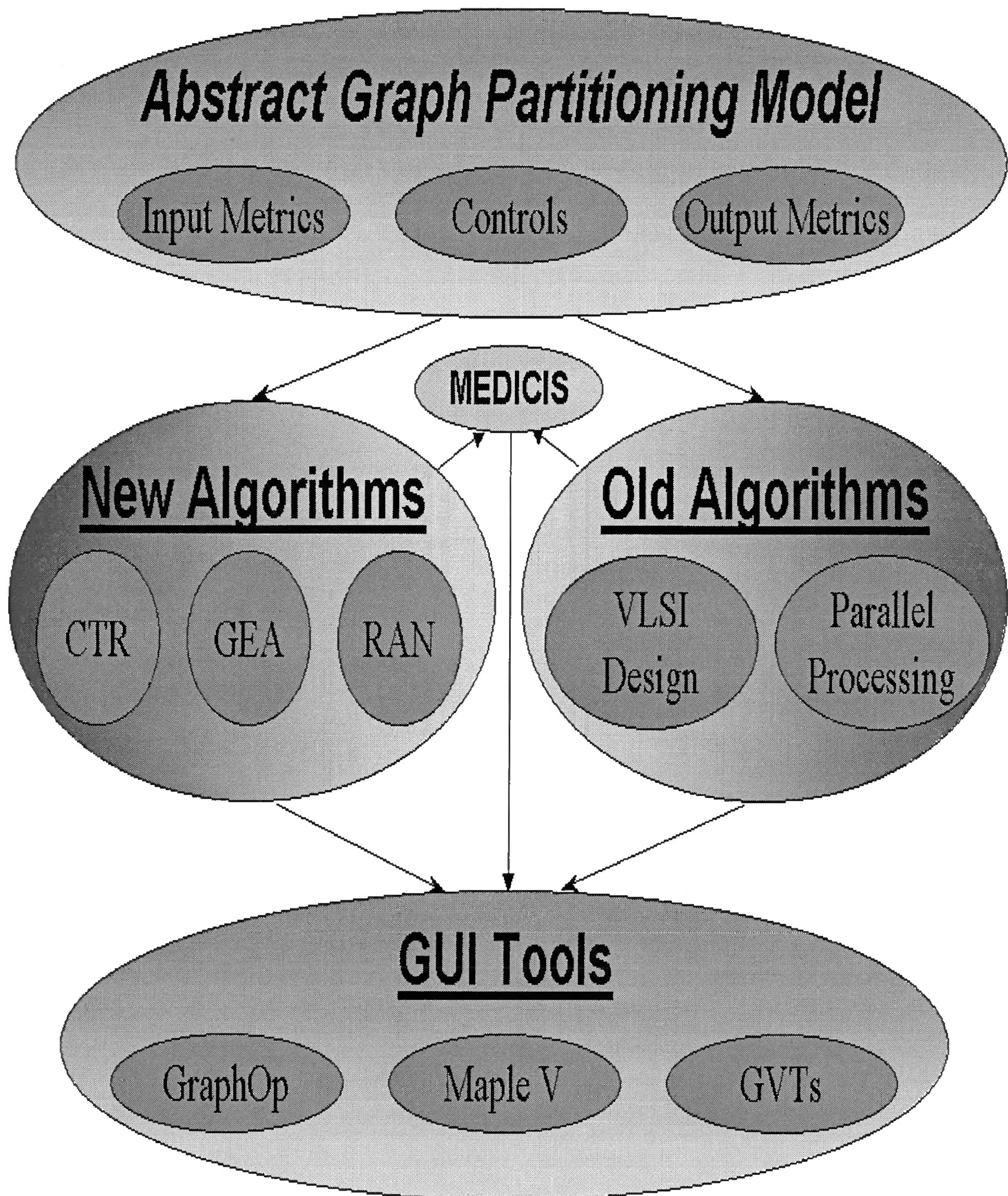
54. *Tom Sawyer Demo*. Tom Sawyer Software. 1999. <http://www.tomsawyer.com/download-soft.html>.

55. Vijay V. Raghavan and Brijesh Agarwal. *Optimal Determination of User-Oriented Clusters: An Application for the Reproductive Plan*. Proceedings of the Second International Conference on Genetic Algorithms. Lawrence Erlbaum Associates, Hillsdale, July 1987. pg. 241-246.

56. W. Kernighan and S. Lin. *An Efficient Heuristic Procedure for Partitioning Graphs*. Bell System Technical Journal. Vol. 49, 1970. pg. 291-307.

57. Zbigniew Micalewicz. *Genetic Algorithms + Data Structures = Evolution*. Springer-Verlag, New York, 1994.

A. Visual Depiction of Thesis Structure



B. Project & Reference Research System (PRRS)

PRRS Projects Form

Project & Reference Research System - [Projects]

File Edit Insert Records Window Help

Close this Form!

Project Name: A Sample Project

Purpose: To demonstrate the Project and Reference System

Done For!: A Sample Agency

Primary DIRECTORY: Your primary on-line storage location for the this project

Journal! Starting Date: 1/1/99 Status!: Completed Ending Date: 1/1/00

Notes and Comments: This is a sample project established to allow you, the end-user the opportunity to view the cross-referencing capabilities of this system. This is version 1.1

References for this Project

View This Reference!

A Sample Title

Record: 1 of 1

Open References!

Project Members

View Member's Info! | View Positions Listing!

Christopher J. Augeri | Team Leader

Sample Person | Team Member

Record: 2 of 2

Open People!

Record: 1 of 6

PRRS References Form

Project & Reference Research System - [References]

File Edit Insert Records Window Help

Close this Form!

Ref. Title: **A Sample Title**

Bibliography Info: Publisher, City & Year of Publication, Vol & Issue, Page #s used, etc.

Local Reference Location: The on-line or hard-copy location of this reference

Original Reference Location: Where you acquired this reference (i.e. library & call #, ftp site & directory, http address, person, etc.)

Sources!: Journal

Receipt Methods!: ftp

Projects Using This Reference

View this Project!

A Sample Project

Record: 1

Open Projects!

Authors of this Reference

View Author's Info!

Christopher J. Augeri

Record: 1

Open People!

Copy Ref. Code!

R[2]

Date Entered: 12/23/98

Reference Comments, Notes and Thoughts!

Once this reference is initially entered, this field may be used to record thoughts, ideas, or other items of interest for this reference (e.g. follow-up research).

Record: 1 of 1 (Filtered)

PRRS Contacts Form

Project & Reference Research System - [People]

File Edit Insert Records Window Help

Close this Form!

Person's Name	Sample Person
Home Address	
Line #2	
Line #3	
Line #4	
Home Phone!	
Home Fax	
Home E-mail!	
Home WWW!	
Pager!	
Cell Phone!	
Company Name	
Work Address	
Line #2	
Line #3	
Line #4	
Department!	
Work Phone!	
Work Fax	
Work E-mail!	
Work WWW!	

Person's Picture!

Notes and Comments

Project(s) Person Worked On...

Article(s) Person Authored...

Open Projects!

Open References!

Record: 1 of 1 (Filtered)

C. GraphPar Functions

An electronic version of the source is available by contacting the author, currently via caugeri@ieee.org and is also available on-line at <http://nothinbut.net/~caugeri>. The source code includes the following commands, loaded by via "with(GraphPar):"

I. Input/Output (I/O) Functions

1. GraphVizOut: output file in GraphViz format
2. MapleGraphIn: read file in our Maple V graph format
3. MapleGraphOut: write file in our Maple V graph format
4. MapleTableOut: output file in table format for input later

II. Graph Generator Functions

1. Caterpillar: generate caterpillar graph and drawing
2. GeoDraw: draw geometric graphs
3. GeoGrid: generate geometric grid & grid-X graphs
4. GeoMetric: generate random geometric graphs
5. GeoTouch: generate random geometric touch graphs
6. Graphs: generate previous programmed graphs (e.g. ISCAS17)
7. Piles: generate independent weighted graphs

III. Metric Functions

1. gStats: determine input graph metrics
2. pStats: determine output partition metrics
3. mCut: measure cut of output partition
4. mDelay: measure delay of output partition
5. mWeights: measure vertice weights of output partition
6. GraphCost: determine max cut, delay & weights, measure output metrics

IV. Support Functions

1. ConSubGphs: generate all connected subgraphs of input graph
2. CreateParting: generate graph reflecting provided partition data
3. Delay: return delay values of vertex/vertices
4. GraphPerm: generate random permutation of graph's vertex labeling
5. GraphRoots: return primary input and output roots of graph
6. Hasse: return list of vertices in appropriate levels
7. Height: return table of heights

8. InduceCopy: copy undirected or directed graph
9. InputGraph: set vertex delays & weights to immediate # predecessors

V. Support Functions (continued)

1. InterPart: return list of inter-partition edges
2. MaxDT_ALL: return max delays to all, including itself
3. MaxDT_OO: return max delays to predecessors only
4. Max: more robust max(x, y) procedure (e.g. strings)
5. Min: more robust min(x, y) procedure (e.g. strings)
6. Orient: orient a graph, based on vertex label names
7. PartPerm: return resulting partition to un-permuted graph state
8. QuickSched: add edges to a graph such that TopSort will also be a schedule list
9. TablePerm: permute table based on pre-determined list
10. Tabulate: return frequency of items in [sorted] input list
11. TopSort: return topological sort of the graph
12. TopSortD: return topological sort based on vertex delays of the graph
13. TopSortR: return reverse topological sort of the graph
14. TransReduce: perform a transitive reduction of the input graph

VI. Partitioning Algorithms

1. BPD: Best Predecessor Algorithm
2. CTR: Eccentricity-Based Algorithm
3. DSC: Dominant Sequence Clustering Algorithm
   a. DSRW: internal to DSC, implements constraint DSRW
   b. MinDSC: internal to DSC, minimum free vertex
4. GEA: Genetic Algorithm
   a. Cross-Over Operators
      i. PMX: Permutation Cross-Over
      ii. OX: Order Cross-Over
   b. Mutation Operators
      i. DecodeKlist: translate from absolute to relative separator positions
      ii. EncodeKlist: translate from relative to absolute separator positions
      iii. RandKlist: generate initial partition separator positions
      iv. PairSwap: generate mutated offspring
5. RAN: Random Solution Algorithm