Theoretical and Algorithmic Approaches to Field-Programmable Gate Array Partitioning

A Dissertation
Presented for the
Doctor of Philosophy Degree
The University of Tennessee, Knoxville

Barbara C. Plaut
December 1999

Acknowledgements

First and foremost, I thank my advisor, Prof. Michael Langston. I thank him for his guidance and unwavering support, and for the many, many things I have learned under his advisorship.

I am grateful to the Navy for the Augmentation Award for Science and Engineering Technology (AASERT), and to Prof. Langston for awarding me the privilege of being its recipient.

I thank Profs. Berry, Bouldin and Raghavan for serving on my graduate committee, and their efforts in that regard.

I thank all the faculty and staff in the Computer Science Department who have helped me in so many ways.

I thank my family members who have cheered my accomplishments and buoyed me up during the low times.

Last, but not least, I thank my mentor, best friend and husband, Conrad, for having so much faith in me.

Abstract

Many practical problems dealing with the design of Very Large Scale Integrated (VLSI) circuits can be modeled as graphs in which vertices represent components of the circuit and edges represent a relationship between these components. When expressed as graphs, these problems can then often be solved using graph theoretic methods. Unfortunately, many such problems are $\mathcal{NP}$-complete, hence no practical exact solutions are known to exist.

In this dissertation, we study $\mathcal{NP}$-complete problems taken from the realm of partitioning for Field-Programmable Gate Arrays (FPGAs). We adopt a two-pronged approach of theory and practice, developing practical heuristics driven by theoretical study.

The theoretical approach is motivated by well-quasi-order (WQO) theory, which can be used to show, among other things, that when some hard problems have fixed parameters, polynomial-time solutions exist. This is of significance in the area of FPGA partitioning, in which practical problems are often characterized by fixed-parameter instances. WQO techniques are not generally practical, however, and we develop new methods to solve several important problems in VLSI that are not even amenable to WQO techniques.

We begin with a representative partitioning problem, Min Degree Graph Partition (MDGP), the fixed-parameter version of which is closed under the immersion order. We show that the obstruction set (set of immersion minimal elements) for this problem is computable; we prove both upper and lower bounds on the obstruction set size; and we completely characterize all fixed-parameter MDGP simple tree obstructions.

WQO theory tells us only that fixed-parameter MDGP is solvable in (high-degree) polynomial time. We attack the problem using what we refer to as *kd-candidate subsets*, culminating in linear-time decision and search algorithms. The *kd-candidate subset* method also paves the way for an efficient heuristic for the FPGA Minimization problem.

We then broaden our scope to incorporate delay minimization into FPGA partitioning. We develop, analyze and test a novel method called *critical path compression*, inspired in part by compiler optimization techniques.

We then look at a variety of generalizations of MDGP. Some of these problems are not immersion closed; others are not even defined in a way that WQO theory applies. However, almost all of them are efficiently solvable via the *kd-candidate subset* approach.

Interspersed in these results are many refinements of what is known about the complexity of these problems. We also discuss a few other solution strategies, and present many open problems.

Contents

List of Figures

List of Tables

Chapter 1

Introduction and Background

The technology of VLSI circuit design has progressed rapidly in recent years. The process of transforming an abstract circuit design into a physical entity has become increasingly complex. In order to handle this complexity, the process is broken down into a series of tasks, each of which can be handled relatively independently. Some of these tasks are behavior modelling, functional and logic minimization, logic fitting and simulation, partitioning, placement, routing and fabrication ([SY]). In this work, we focus on the partitioning stage.

Another consideration in circuit design is that of the physical layout style, some of which are full-custom, gate-array, standard-cell, macro-cell, programmable logic array (PLA) and field-programmable gate attay (FPGA) ([SY]). Our interest is in partitioning for FPGA layouts.

This particular aspect of partitioning is itself a broad problem, with numerous specific formulations, many of which have been extensively studied ([AK]). These problems, when translated into graphical terms, are usually $\mathcal{NP}$-complete, and ultimately tackled by approximation and heuristic algorithms. However, theoretical results of Robertson and Seymour ([RS1], [RS2], [RS3], [RS4]) can often be used to show that in many cases fixed-parameter versions of the problem are in fact solvable in polynomial time. This is of significance when dealing with FPGA partitioning problems, which are inherently confined to instances with bounded parameters. Unfortunately, even theoretically efficient algorithms are often not practical. Nevertheless, the theoretical study often paves the way to new and better heuristics.

In this chapter, we give an introduction to the topic, some definitions and background information.

In the second chapter, a representative and fundamental partitioning problem is defined and studied from a theoretical perspective. The fixed-parameter versions of this problem are relevant to FPGA partitioning. They are known to be solvable in (high-degree) polynomial time, because of their closure under the immersion order. In this work, we show that the obstruction sets for these graph families are computable by demonstrating upper bounds on the obstruction set sizes. We also determine lower bounds, and completely characterize the simple tree obstructions. We then show that both the search and decision versions of the problem are solvable in linear time. While this problem does not capture all of the issues important in FPGA partitioning, it serves as a useful starting point for further study.

The fundamental problem is extended to consider FPGA Minimization in the third chapter. Here we strive to partition a circuit into as few chips as possible, in order to minimize cost, and to enable realization of a circuit on a specific system. We delve deeper into the complexity of the problem, and derive a practical heuristic driven by theoretical results.

In the fourth chapter, the fundamental problem is extended in yet another direction. The circuit system is considered in its dynamic state, with electrical current flowing through it. We seek to minimize the maximum time for a signal to flow from any input to any output. Here we must broaden our graphical representation of the circuit from undirected to directed graphs, and the theoretical picture changes significantly. In this chapter, we develop and study a new method for minimizing delay in a partitioned circuit.

The fifth chapter deals with many variations of the fundamental partitioning problem. Although many of these problems are no longer amenable to WQO techniques, we find that most of them can still be solved efficiently using the techniques of the second chapter. We conclude with some ideas for future study, from both a theoretical and a practical standpoint.

Table 1.1 summarizes the main theoretical and practical results of this work.

1.1 Definitions and Mathematical Preliminaries

For our purposes, an undirected graph $G = (V_G, E_G)$ consists of finite sets of vertices V_G and undirected edges E_G. Multiple copies of edges are allowed, but self-loops are ignored, because they have no consequence in any of the algorithms that we develop.

A directed graph $G = (V_G, E_G)$ is defined similarly, except each edge pair has an ordering.

The simplified notation $G = (V, E)$ is used when G is the only graph under consideration.

If v is a vertex in G, the *degree* of v (denoted $\delta_G(v)$) is the number of edges in G that are incident on v. When there is no ambiguity about the graph, we simply use $\delta(v)$ to denote the degree of v. The notation is extended to denote the degree of a set of vertices as follows: for $S \subseteq V_G, \delta_G(S)$ is the number of edges in G that have exactly one endpoint in S.

Two vertices $u, v \in V$ are *adjacent* or *neighbors* if $uv \in E$. $N(u) = \{v : uv \in E\}$ denote the set of *immediate* neighbors of u. Note that $\delta(u) > |N(u)|$ if multiple copies of an edge adjacent to u exist.

The notation K_n is used to signify a graph containing n vertices, in which every pair of vertices is connected by a single edge.

A subgraph of G induced by some $V' \subseteq V_G$ consists of vertex set V' and edge set $\{uv | uv \in E_G, u \in V', v \in V'\}$.

Table 1.1: Summary of main results

MDGP: in $\mathcal{P}$ when restricted to simple trees; **Fixed-parameter MDGP:** solvable in linear time, obstruction set computable, upper and lower bounds on obstruction set size, complete characterization of simple tree obstructions
FPGA Minimization: $\mathcal{NP}$-complete for many classes of graphs, development of theoretically-based heuristic; **Fixed-parameter FPGA Minimization:** exponential obstruction set size
Delay Minimization: $\mathcal{NP}$-complete for many classes of graphs, development of critical path compression heuristic; **Fixed-k, d Delay Minimization:** $\mathcal{NP}$-complete
Fixed-parameter Hypergraph MDGP: in $\mathcal{P}$
Heterogeneous MDGP: in $\mathcal{P}$ when restricted to simple trees; **Fixed-parameter Heterogeneous MDGP:** decision and search solvable in linear time; obstruction set computable
Balanced MDGP(k,d,0): $\mathcal{NP}$-complete
MDC(d): exponential obstruction set size
Sixteen WQO closure-preserving operators: whether closed or not

If, for every two vertices $x, y \in V_G$, there exists a series of edges from x to y, we say that G is *connected.* Each maximally connected subgraph of a graph is referred to as a *component.* Two vertices $u, v \in V$ are *n-edge-connected* if a minimum of n edges must be deleted to disconnect G in such a way that u and v lie in different components.

An *n-path* is a connected, acyclic graph containing $n > 1$ vertices, each vertex of which has either 1 or 2 neighbors.

The following two definitions are from [H]. A shortest $u - v$ path is called a *geodesic.* The *diameter* of a connected graph is the length of any longest geodesic.

A *tree* is a connected, acyclic graph. A *simple tree* is a tree in which there is at most one copy of each edge. A *forest* (*simple forest*) is a graph whose components are all trees (simple trees).

Two graphs H and G are said to be *isomorphic* if there is a bijection $f : V_H \rightarrow V_G$ such that $uv \in E_H \Leftrightarrow f(u)f(v) \in E_G$.

A *tree-decomposition* of $G = (V, E)$ is a pair $(T = (V_T, E_T), f)$ where T is a tree and f is a function mapping V_T into a set of subgraphs of G, with f satisfying the following properties:

1. $\cup_{t \in V_T} f(t) = G$; and

2. for $s, t \in V_T$, if u is on the path from s to t in T then $f(s) \cap f(t) \subseteq f(u)$.

The width of a tree-decomposition (T, f) is $max_{t \in V_T}|f(t)| - 1$. The *treewidth* of G is the minimum treewidth of all tree-decompositions of G. Figure 1.1 shows a graph and a corresponding tree-decomposition of width two.

It is evident that every tree has treewidth 1. Therefore, the family of all trees is of bounded treewidth. As an example of a family of graphs with unbounded treewidth, consider the family of all $w \times w$ grids, for all w, each of which has treewidth w ([RS1]).

Given graphs H and G, we say that $H \leq_i G$, meaning H is contained in G under the *immersion* order, if and only if a graph isomorphic to H can be obtained from

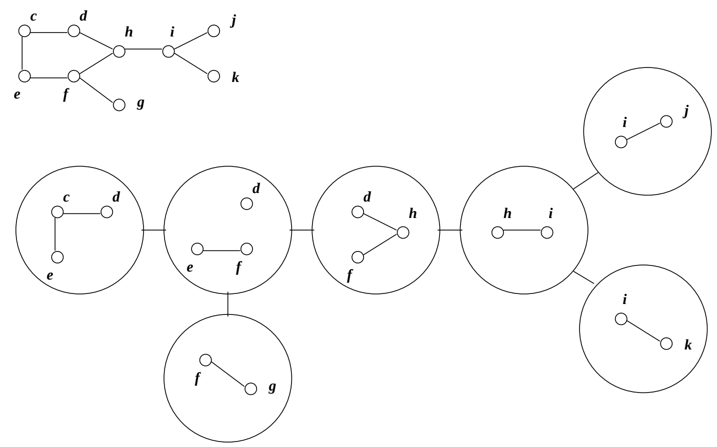


Figure 1.1: A graph and its tree-decomposition of width two

G by a series of the following two operations: taking a subgraph, or lifting a pair of adjacent edges. A pair of adjacent edges uv, vw, with $u \neq v \neq w$ is lifted by removing uv and vw and adding uw. Figure 1.2 illustrates that C_4 is immersed in $K_1 + 2K_2$ ([La1]).

The immersion order can also be viewed in terms of edge-disjoint paths: H is immersed in G if and only if there exists an injection from V_H to V_G for which the images of adjacent elements of V_H are connected in G by edge-disjoint paths.

A family F of graphs is said to be *immersion closed* if $G \in F, H \leq_i G \rightarrow H \in F$. The *obstruction set* for a family F of graphs is the set of graphs in the complement of F that are minimal in the immersion ordering. Therefore, if F is immersion closed, it has the following characterization: G is in F if and only if there is no H in the obstruction set for F such that $H \leq_i G$.

This tells us that there exists a membership algorithm for any immersion-closed family F: simply test for the presence of any immersed obstruction. This will succeed if the obstruction set is finite, which, as we shall soon see, is always the case.

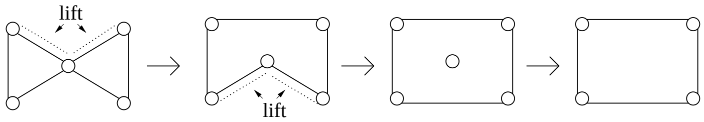


Figure 1.2: $C_4 \leq_i K_1 + 2K_2$

A *quasi-order* is a reflexive, transitive relation. A quasi-ordered set $(X, \leq)$ is *well-quasi-ordered* if (1) any subset of X has finitely many minimal elements and (2) X contains no infinite descending chain $x_1 \geq x_2 \geq x_3 \geq ...$ of distinct elements.

Theorem 1.1 *([RS2]) Graphs are well-quasi-ordered under immersion.*

Theorem 1.1 tells us that, given an immersion-closed family of graphs F, a membership algorithm always exists. The following theorem gives us even more: that a polynomial-time algorithm always exists.

Theorem 1.2 *([FL4]) For every fixed graph H, the problem that takes as input a graph G and determines whether $H \leq_i G$ is solvable in time $O(n^{h+3})$, where h is the order of the largest graph in the obstruction set for F.*

Theorems 1.1 and 1.2 together are powerful tools with wide applicability. See [FL1] and [FL2] for many examples. In this work, we focus on problems from the realm of FPGA partitioning, many of which are closed under the immersion order.

Other WQOs are known; one of the most useful in terms of VLSI applications is the *minor* order, under which a graph H is less than or equal to a graph G ($H \leq_m G$) if and only if a graph isomorphic to H can be obtained from G by a series of these two operations: subgraph and edge contraction.

As in the case of the immersion order, there exists a polynomial-time decision algorithm for any minor-closed family of graphs ([RS4]). However, in the case of the minor order, the running time of the algorithms is much faster. Letting n denote the number of vertices in G, the time to recognize G is $O(n^3)$ ([RS3]).

Under either the immersion or the minor order, if a family of graphs has treewidth bounded by some constant h, then a linear-time recognition algorithm exists. Given h, and a graph G, it is possible in linear time either to determine whether the treewidth of G exceeds h (in which case G is a "no" instance), or to find a tree-decomposition of G with treewidth at most h ([Bod]). Given such a tree-decomposition, testing for obstruction containment can be done in linear time ([RS3]).

The results just mentioned are *nonconstructive*. They can be used to show the *existence* of polynomial-time *decision* algorithms. They do not address the issue of actual algorithm construction, which depends upon specific knowledge of an obstruction set. They do not give us any information on how to find the obstruction set. They do not give us any information on how to solve the *search* version of the problem; i.e. how to construct positive evidence of a "yes" instance.

While these remarkable theoretical findings give us exciting new tools to cope with previously elusive problems, they also introduce a host of issues that must be resolved for any practical application. Of primary importance are the issues of nonconstructivity and high polynomial degree in the case of the immersion order.

1.2 Hardware Technology

The technology of very-large-scale integrated (VLSI) circuit design continues to progress rapidly. A relatively recent addition to the component library is the field-programmable gate array (FPGA), a collection of functional blocks with programmable connections ([OD]). Figure 1.3 gives a simplified picture of a conceptual FPGA.

The specific function of each block and the connections between blocks are dynamically programmable. This feature enhances affordability and flexibility, and has

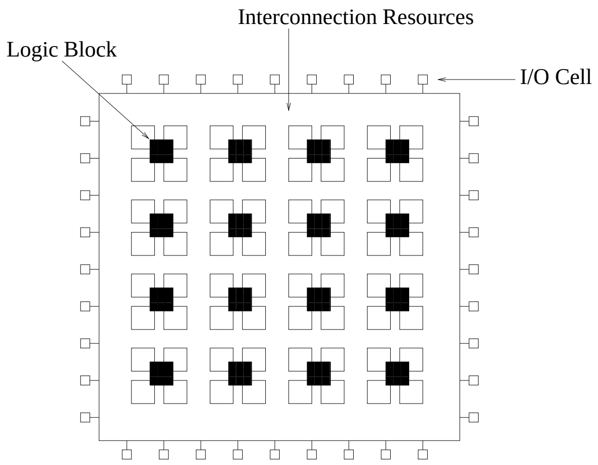


Figure 1.3: The FPGA

significant advantages for the development of prototype systems. A given circuit is implemented by partitioning its logic into blocks and connecting the blocks as required. Since circuits are frequently too large to fit on a single chip, they must be partitioned over several FPGAs.

FPGA chips come in a variety of sizes and styles ([X]). Typically, the functional blocks on a chip consist of an array of identical Configurable Logic Blocks (CLBs). Each CLB is a look-up table with a number of inputs (usually two to five), and one or two outputs.

As mentioned earlier, there are many steps involved in taking a circuit from design to physical reality. One of these steps, which can take place either before or after partitioning, is that of technology mapping. Technology mapping refers to the process of transforming a large circuit at the gate level into a system of smaller units that can be realized as a set of communicating CLBs. Although technology mapping can be performed either before or after partitioning, most developers agree that it is more efficient to perform technology mapping first, and then do partitioning on a circuit system of CLBs. The subject of technology mapping will be discussed in more detail in section 3.2.3.

The usual sequence of events, then, is to perform technology mapping first, and then partitioning. The system of CLBs sent to the partitioner can be modelled by a graph, in which a node represents a CLB, and edges represent the connections. Physical connections between CLBs are usually established during a later phase of the design implementation, and are programmable in either direction, so the partitioner may work with an undirected graphical representation of the CLB system. The I/O cells around the periphery of the chip are also programmable in either direction.

In building systems with multiple FPGAs, fabrication technology imposes severe restrictions: limits on pin counts (I/O cells) affect inter-chip connectivity; limits on chip area and density bound FPGA sizes. These physical dimensions give rise to many difficult combinatorial problems, one of which we explore in great detail in the next chapter.

Chapter 2

A Fundamental Partitioning Problem

We begin with a very fundamental problem. Although it is actually the simplest of all that we will consider in this work, we find that it is indeed very difficult, and of considerable independent interest.

2.1 Problem Definition and Prior Results

A circuit design is usually conceived at a high level, and expressed independently of the hardware in which it will eventually be implemented. Circuit partitioning is the process of dividing a circuit into smaller parts, so that it can be realized by hardware devices. Partitioning a design-level circuit in such a way that it satisfies the physical constraints of a hardware system is a complex problem that has been the subject of extensive study. See [AK], [BKK], [CLCDL], [HK] and [WK] for many examples. In this work, we focus specifically on hardware systems consisting of FPGAs. Within this context, an important question is that of whether a given circuit can be partitioned to fit onto a set of FPGAs such that the size and pin count constraints of each are met. We call this the Min Degree Graph Partition problem (MDGP) ([La1]).

Instance: a graph $G = (V, E)$, and two integers k and d.

Question: Is there is a partition of V into disjoint sets $V_1, \ldots, V_m$ such that $\forall i : |V_i| \leq k$, and such that if E_i is the set of edges with exactly one endpoint in V_i, $max_{1 \leq i \leq m}|E_i| \leq d$?

Figure 2.1 shows a "yes" instance of MDGP(k=2,d=2) that has only one satisfying partition: $V_1 = \{a, b\}, V_2 = \{c, d\}$.

Within this formulation, the parameter k represents the size (in CLBs) of an individual type of FPGA chip, and the parameter d represents the pin-count of a chip. Given a graphical representation of a circuit, with each node representing a CLB and each edge representing a connection between two CLBs, MDGP asks whether the circuit can be realized on a set of FPGAs of a given type .

Note the similarity between this problem and the Graph Partitioning problem of [GJ]. In the latter problem, the goal is to minimize the sum of all edges that have their endpoints in different subsets, and there is no explicit constraint on the number of edges that may emanate from an individual subset. Therefore, Graph Partitioning does not model the situation in which there is a degree constraint on each subset.

There are some important issues in circuit design, such as cost and performance, that are not addressed by this fundamental problem. Nevertheless, MDGP provides a useful starting point for the study of FPGA partitioning from a theoretical perspective. Much of the knowledge gleaned from this basic problem is of benefit in solving broader problems, some of which we will examine more closely in later sections.

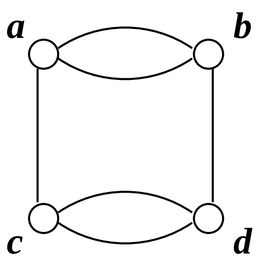


Figure 2.1: A partitioning problem

The MDGP problem is very difficult without parameter bounds, via a reduction from Multiway Cut:

Theorem 2.1 *([Go]) Min Degree Graph Partition is $\mathcal{NP}$-complete.*

Fortunately, however, the aforementioned fabrication limits can be used to advantage. As long as k and d are bounded, the family of "yes" instances is closed in the immersion order, which leads to the following result.

Theorem 2.2 *([La1,LP]) For any fixed k and d, MDGP can be decided in polynomial time.*

Since the parameters k and d represent actual physical constraints, when partitioning for FPGAs we may assume that these parameters are bounded by the technology at hand. Fixed-parameter MDGP is, therefore, a relevant problem from the perspective of circuit partitioning. To distinguish fixed-parameter MDGP from generalized MDGP, we shall use MDGP(k,d) to denote the former.

In section 1.1 we saw that, for any family of graphs closed under the immersion order and of bounded treewidth, a linear-time recognition algorithm exists. Unfortunately, the family of "yes" instances of MDGP(k,d) is not of bounded treewidth. To see this, consider MDGP(1,4). Even this simple family of graphs contains the $w \times w$ grid for any w, a graph with treewidth w.

Prior to this time, little more was known about the complexity of MDGP(k,d), and no efficient algorithms, or even brute force algorithms, were known to exist. Not much more could be said other than that MDGP(k,d) was nonconstructively decidable in polynomial time. Whether it was solvable in low-order polynomial time was an open question, as recently as 1995 ([LP]). Many issues remained, including the following:

- WQO-based solutions are inherently nonconstructive. They depend on the existence of finite obstruction sets and, in general, we do not know what these obstruction sets are, or how to find them.

- Although the algorithms are polynomially bounded, the degree of the polynomial is high: $O(n^{h+3})$, where h is the order of the largest obstruction. This polynomial presents yet another dimension of nonconstructivity: since we do not know the obstruction set, or even the order of the largest obstruction, we do not know the exact degree of the polynomial. Sometimes efficient algorithms can be devised to test for specific obstructions, but this is a difficult task (BGLR).

- Obstruction sets are very difficult to identify. In some cases, obstruction set isolation has been performed exhaustively as part of a major research effort ([KiL]). Other researchers have developed machinery to generate minor-minimal "no" instances of some graph families of bounded treewidth ([CD]). However, in general, there exists no easy, widely-applicable method of finding obstruction sets.

- WQO-based solutions are *decision* algorithms: they simply tell us whether or not a given graph is a member of a particular graph family. They do not address the corresponding *search* problem by constructing evidence. In most practical problems, knowing that a graph is a "yes" instance is not enough. In the case of graph partitioning, for example, a solution in the form of a satisfying partition is essential.

- WQO-based solutions apply only to ordinary graphs. Practical problems, especially those that model VLSI problems, are often represented more accurately by hypergraphs. Although WQOs are known to exist on hypergraphs ([GGL], [Se]), these orders have not yet been shown to be of practical importance for these types of problems.

In subsequent sections, we will address each of these issues.

2.2 New Results

We know by Theorem 2.2 that MDGP(k,d) is solvable in polynomial time. In this section, we present some tools that will ultimately be used to show that MDGP(k,d) is actually solvable in linear time. These tools will also assist in formulating self-reduction strategies, finding MDGP(k,d) obstruction sets, and understanding the complexity of MDGP when it can be assumed that the instance graph has a pre-defined structure.

2.2.1 Algorithmic Tools

We now present some definitions, observations and lemmas that will be of general use throughout most of this work. (Recall that we refer to fixed-parameter MDGP as MDGP(k,d).)

Observation 2.1 *A star graph (see Figure 2.2) with $k + d$ rays is an obstruction to MDGP(k,d); therefore, no obstruction to MDGP(k,d) contains a vertex with more than $k + d$ neighbors.*

Similarly, no "yes" instance of MDGP(k,d) contains a vertex with more than $d + k$ neighbors; hence the "yes" family has bounded degree.

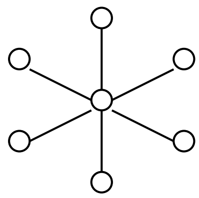


Figure 2.2: A star graph with six rays

Definition 2.1 *Let $N_p(v)$ denote $\{v\} \cup \{w : \exists$ a path from v to w of length $\leq p\}$.*

Definition 2.2 *A connected subset of a graph G is a subset $S \subseteq V_G$ such that the subgraph of G induced by S is connected.*

Lemma 2.1 *G is a "yes" instance of MDGP(k,d) iff there exists a solution in which every subset is connected; thus, in this solution, every vertex v is partitioned only with other vertices in $N_{k-1}(v)$.*

Proof If there exists such a solution for G, then G is a "yes" instance.

For the converse, assume that G is a "yes" instance, and that we have a satisfying partitioning. Consider any subset S such that the subgraph G' of G induced by S is not connected. We can then separate S into distinct connected subsets, one for each connected component of G'. Each of these is of size less than k. Additionally, each is of degree no more than d, because there exist no edges between the new subsets. Because every subset is now connected and of size no more than k, each vertex $v \in S$ is partitioned only with other vertices in $N_{k-1}(v)$. $\Box$

Definition 2.3 *Given k and d, let c_p denote the value $1 + \sum_{i=1}^{p}(k+d)(k+d-1)^{i-1}$.*

Lemma 2.2 *If G is an obstruction to MDGP(k,d), then $\forall v \in V, \forall p > 0, |N_p(v)| \leq c_p$.*

Proof By Observation 2.1, v has at most $k+d$ immediate neighbors, so $|N_1(v)| \leq 1 + (k+d)$. Each neighbor at distance $q > 0$ from v has at most $k+d-1$ neighbors not contained in $N_{q-1}(v)$, so $|N_p(v)| \leq 1 + \sum_{i=1}^{p}(k+d)(k+d-1)^{i-1}$. $\Box$

It is useful to observe that, when $k+d > 2$, $c_p = 1 + (k+d) \times \frac{(k+d-1)^p - 1}{k+d-2}$.

Definition 2.4 *A "kd-satisfying subset" is a subset of size no more than k and degree no more than d.*

Definition 2.5 *A "kd-candidate subset" is a connected kd-satisfying subset. Given k, d and a vertex v, let C_v denote the set of all kd-candidate subsets containing v.*

We note that, because a kd-candidate subset is connected and of size no more than k, its diameter is bounded by $k-1$. Furthermore, for every v in some kd-candidate subset C, every other vertex in C is in $N_{k-1}(v)$.

For example, consider the graph G in Figure 2.3 as an instance of MDGP(3,2). Then $N_{k-1}(a) = N_2(a) = \{a, b, c, d\}$, and $C_a = \{\{a, c\}, \{a, b, c\}\}$.

Lemma 2.3 *Given kd-satisfying subsets $C1$ and $C2$, either $C1 - C2$ or $C2 - C1$ is a kd-satisfying subset.* [1]

Proof Since neither $C1 - C2$ nor $C2 - C1$ can have size exceeding k, we need only consider their respective degrees.

If $C1 \cap C2 = \emptyset$, then we are done. Otherwise, let $I = C1 \cap C2, A = C1 - C2, B = C2 - C1, D = V - C1 - C2$ (see figure 2.4).

Denote by N_{AB} the number of edges having an endpoint in A and an endpoint in B. $N_{AD}, N_{AI}, N_{BD}, N_{BI}$ and N_{DI} have analogous meanings. The degree of $C1$ is $N_{AD} + N_{AB} + N_{DI} + N_{BI}$, and the degree of $C2$ is $N_{AB} + N_{BD} + N_{AI} + N_{DI}$.

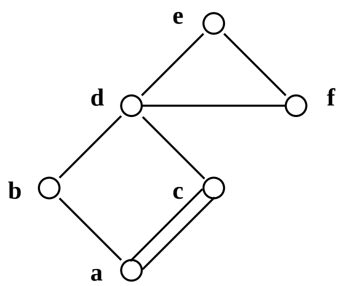


Figure 2.3: An instance of MDGP(3,2)

[1] Independently proved in [CLCDL].

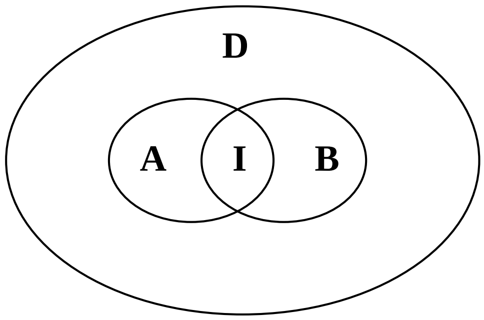


Figure 2.4: $C1 = A \cup I, C2 = B \cup I$

By the definitions above, we have

$$N_{AD} + N_{AB} + N_{DI} + N_{BI} \leq d$$

and

$$N_{AB} + N_{BD} + N_{AI} + N_{DI} \leq d.$$

Summing yields

$$N_{AD} + 2N_{AB} + 2N_{DI} + N_{BI} + N_{BD} + N_{AI} \leq 2d,$$

so

$$N_{AD} + 2N_{AB} + N_{BI} + N_{BD} + N_{AI} \leq 2d.$$

Thus either

$$N_{AB} + N_{AI} + N_{AD} \leq d$$

or

$$N_{AB} + N_{BI} + N_{BD} \leq d.$$

The former bounds the degree of $C1 - C2$, the latter the degree of $C2 - C1$. $\Box$

Lemma 2.4 *Given kd-satisfying subsets $C_1, C_2, ..., C_p$, a disjoint set of kd-satisfying subsets $D_1, D_2, ..., D_q$ exists such that $C_1 \cup C_2 \cup ... \cup C_p = D_1 \cup D_2 \cup ... \cup D_q$.* [2]

Proof The proof is by induction on p. For the basis case, $p = 1$, the set of subsets is already disjoint and satisfying. As inductive hypothesis, assume that, given $C_1, C_2, ..., C_p$, $p \geq 1$, an appropriate set of subsets $D_1 \cup D_2 \cup ... \cup D_q$, $q \geq 1$, exists. Given C_{p+1}, we construct a set of kd-satisfying subsets $D_1 \cup D_2 \cup ... \cup D_{q'}$ such that $D_1 \cup D_2 \cup ... \cup D_{q'} = C_1 \cup C_2 \cup ... \cup C_{p+1}$. Initially $D_1 \cup D_2 \cup ... \cup D_{q'} = D_1 \cup D_2 \cup ... \cup D_q$, so $D_1 \cup D_2 \cup ... \cup D_{q'} = C_1 \cup C_2 \cup ... \cup C_p$, and the D_i's are disjoint and satisfying. Let $T = C_{p+1}$. For each $D_i, 1 \leq i \leq q'$, we do the following. If $D_i \cap T = \emptyset$, do nothing. Otherwise, $D_i \cap T = I \neq \emptyset$, with $I \cap D_j = \emptyset, \forall\, 1 \leq j \leq q', j \neq i$. By Lemma 2.3, either $D_i - T$ or $T - D_i$ is satisfying. If the former, we change D_i to $D_i - T$; if the latter, we change T to $T - D_i$. At the end of consideration of each D_i, if T is nonempty, q' is incremented by one, and $D_{q'}$ is set to T. Finally, any empty D_i may be removed, and q' decremented accordingly. □

The proof of Lemma 2.4 suggests a subset disjointing algorithm. Such an algorithm is used by [CLCDL] in an FPGA partitioning heuristic. The heuristic first forms subsets to satisfy constraints, and then makes the subsets disjoint in a later step. Our work proceeds further, however, as we shall now describe.

The algorithm suggested by the proof of Lemma 2.4 is of quadratic-time complexity. It can, however, be implemented to run in time linear with respect to $|V|$, assuming we are given one kd-candidate (rather than mere kd-satisfying) subset for each $v \in G$. In the proof described above, T is compared against each D_i. If we begin with kd-candidate subsets, however, there is only a constant number of D_i's with which any T can have a nonempty intersection. Each of T, D_i is initially formed from a kd-candidate subset, and never made larger. If there exist $u \in T, v \in D_i$ such that $u \notin N_{2*(k-1)}(v)$, the intersection of T, D_i must be empty, by the following reasoning. Suppose otherwise: $\exists x \in T \cap D_i$. Because $x \in T$ and $u \in T$, it must be the case that

[2] Independently claimed in [CLCDL].

the distance from u to x is at most $k-1$. Because $x \in D_i$ and $v \in D_i$, it must be the case that the distance from x to v is also at most $k-1$. Therefore, the distance from u to v is at most $2*(k-1)$, which contradicts $u \notin N_{2*(k-1)}(v)$.

Therefore, each subset can be indexed by the vertex for which it serves as a *kd*-candidate subset, and no checking need be done of subsets indexed by vertices "too far apart." Specifically, the subset disjointing algorithm proceeds as follows. We are given a set of $p = |V|$ *kd*-candidate subsets, one for each $v \in V$. As the algorithm progresses, and subsets are modified, they may lose their connectivity. They, will, however, always be *kd*-satisfying. Furthermore, because vertices are never added to subsets, it will always be the case that for any subset C, $u \in C, v \in C \rightarrow u \in N_{k-1}(v)$. We will denote the subset indexed by vertex v as S_v.

Step 1: For each vertex v, compute $\{x | x \in N_{2*(k-1)}(v)\}$.

Step 2: This step consists of an outer loop, which executes for each vertex $v_i, 2 \leq i \leq |V|$. For each v_i, an inner loop executes for each $\{v_j | v_j \in N_{2*(k-1)}(v_i)$, and $j < i\}$. The inner loop is as follows:

```
if S_{v_i} ∩ S_{v_j} ≠ ∅
   then
      if S_{v_i} − S_{v_j} is of degree no more than d
         then
            S_{v_i} = S_{v_i} − S_{v_j}
         else
            S_{v_j} = S_{v_j} − S_{v_i}
```

Step 1 executes in linear time, as does the outer loop of Step 2. The running time of the inner loop of Step 2 is bounded by a constant. Therefore, the complexity of the entire subset disjointing algorithm is linear.

This algorithm is of linear time complexity, does not possess large constants of proportionality, and runs quickly in our experimentation. It does, however, require the availability of $|V|$ *kd*-candidate subsets, the computation of which is much more costly, as we shall see in Section 2.2.3.

The proposition that follows is an important tool in our subsequent work, and finds many applications within, including the following:

- finding linear-time decision and search algorithms for MDGP(k,d), as well as a host of related problems
- showing that the obstruction set for MDGP(k,d) is computable
- characterizing the simple tree obstructions to MDGP(k,d)
- proving complexity results for MDGP and other problems

Additionally, it has also been used in ([Go]) in which it was dubbed the *Locality Condition*, a term that we will retain here.

Proposition 2.1 The Locality Condition: *G is a "yes" instance of MDGP(k,d) iff* $\forall v \in V, C_v \neq \emptyset$.

Proof Suppose $\forall v \in V, C_v \neq \emptyset$. Then, by Lemma 2.4, a satisfying partition can be found, ensuring that G is a "yes" instance. For the converse, suppose G is a "yes" instance. Then, by Lemma 2.1 and by the definition of kd-candidate subset, $\forall v \in V, C_v \neq \emptyset$. $\square$

2.2.2 Self-reduction

It is sometimes possible to solve a search problem by reducing it to a related decision problem. For example, one might seek to find a satisfying subset assignment for Min Degree Graph Partition with the aid of a routine that merely tells whether such an assignment exists.

This approach to algorithm design is called *self-reducibility*, and has been formulated in many ways in the literature. In its most limited form, an assortment of restrictions is placed on the decision algorithm, its input and the lexicographic position of the output produced (see, for example, [Sc]). In more general forms, input/output limitations are eliminated and decision algorithms quite distant from the original problem are permitted (see, for example, [FL3]). Additional variations exist, some even incorporating randomness or parallelism (see, for example, [FF], [KUW]).

It is not difficult to see that, for any fixed k and d, MDGP is self-reducible in polynomial time. That is, one can construct a satisfying subset assignment, if any exist, with at most a polynomial number of calls to a decision algorithm, known from the last section also to run in polynomial time. It can in fact be self-reduced with only a linear number of calls.

Theorem 2.3 *The search version of MDGP(k,d) can be solved in $O(np(n))$ time, where $p(n)$ denotes the time required to solve the decision version of the problem.*

Proof First, use the decision algorithm to ensure that the graph is a "yes" instance. If the graph is a "yes" instance, we know, by Lemma 2.1 that there exists a solution in which every subset is connected. We now describe an algorithm that constructs such a solution. The algorithm does this by modifying the input graph G. As subsets are constructed, if a vertex v is assigned to a nonempty subset S, this assignment is forced by the placement of at least $d+1$ copies of an edge between v and some vertex $w \in S$. At the end of the algorithm, the subsets in the modified graph are identified as follows. Vertices u and v are in the same subset S iff there exists a path from u to v such that there are at least $d+1$ copies of each edge in the path.

In what follows, we will refer to a vertex assigned to a subset as a *committed* vertex. Those not yet assigned to a subset are *uncommitted.* Initially, no vertices are committed. An outer loop executes at most once for each vertex.

An arbitrary uncommitted vertex v is selected for this inner loop, and v is now committed to a new subset S. We next show how, in an inner loop, S is constructed in $O(p(n))$ time.

Every time the inner loop begins, the current version of the graph is known to be a "yes" instance. By Lemma 2.1, we know, therefore, that there exists a solution in which every subset is connected (as the algorithm progresses, it zeroes in on one of perhaps many potential initial solutions). The number of uncommitted immediate neighbors of v is bounded by a constant; these vertices form a "neighbor pool." At all times, the neighbor pool consists of all uncommitted immediate neighbors to vertices in S. Initially all vertices in the neighbor pool are unmarked. A vertex in the neighbor pool will be marked if it can be determined that its addition to S produces a graph that is a "no" instance.

If at any time in the inner loop there are no unmarked vertices in the neighbor pool, then there is no way to expand S while maintaining its connectivity. In that case, by Lemma 2.1, and the fact that the modified graph remains a "yes" instance, it must be that S is a kd-candidate subset, and we exit the inner loop.

We select any unmarked vertex w from the neighbor pool, and any vertex $y \in S$ for which an edge wy exists. We augment the graph with d additional copies of wy. If the augmented graph is a "no" instance, the added edges are taken back out. Additionally, vertex w is marked, because its commitment to S produces a graph that is a "no" instance. If the augmented graph is still a "yes" instance, then the extra edges are retained. Additionally, w is now committed to S. All uncommitted neighbors of w are added to the neighbor pool. If the size of S is now k, then S cannot be expanded; thus the graph is a "yes" instance, and we discard the neighbor pool and exit the inner loop. If there are no unmarked vertices in the neighbor pool, then S cannot be made larger while preserving connectivity; thus the graph is still a "yes" instance, and we discard the neighbor pool and exit the inner loop. If neither of these conditions occurs, then the inner loop is not exited, and a new unmarked w is selected.

The neighbor pool is always of size bounded by a constant. This is because the number of neighbors of every vertex is bounded, and no more than k vertices are ever in S. The process continues until a subset size of k is reached, or until no neighbor in the neighbor pool can be pulled into the subset. One of the inner loop terminating conditions occurs in $O(p(n))$ time. □

2.2.3 More on Decision and Search

Theorem 2.4 *The decision version of MDGP(k,d) can be solved in linear time.*

Proof In linear time, any graph containing a vertex with at least $k+d$ neighbors can be eliminated as a "no" instance. Otherwise, $|N_{k-1}(v)|, \forall v \in V$ is bounded by a constant; $|C_v|$ for each v is of constant size; the set of all kd-candidate subsets can be computed in linear time; and by the Locality Condition, a solution exists iff each set is nonempty. □

Theorems 2.3 and 2.4 yield a quadratic time search algorithm for MDGP(k,d). However, we can do even better than that.

Theorem 2.5 *The search version of MDGP(k,d) can be solved in linear time.*

Proof If a solution is known to exist, one can be constructed as follows. Find an arbitrary kd-candidate subset for each vertex. This can be done in linear time, since the complete set of kd-candidate subsets for each vertex can be computed in constant time. Eliminate overlapping as described in the proof of Lemma 2.4. As per the discussion following that proof, this can be done in linear time. □

It must be pointed out that, although solving MDGP(k,d) is asymptotically efficient, in practice this is not really the case. This is due to the large constants of proportionality introduced by our methods. The search algorithm for MDGP(k,d) consists of two parts: 1) finding kd-candidate subsets for each vertex, and 2) eliminating overlapping. The second part is quite efficient, as discussed following the proof of Lemma 2.4. This is not the case, however, for the first part.

Finding a kd-candidate subset containing some vertex v can be done by examining every possible combination of at most $k-1$ vertices from $N_{k-1}(v)$. Since $|N_{k-1}(v)| \leq c_{k-1}$, the constant of proportionality for this method is bounded by $\sum_{i=0}^{k-1} \binom{c_{k-1}-1}{i}$

$> \sum_{i=0}^{k-1} \binom{2^k}{i} > \binom{2^k}{k-1} > (\frac{2^k-k+2}{k-1})^k$. It may not be necessary to consider all of these combinations, since a kd-candidate subset must be connected. However, the multiplicative constant c_{k-1} introduced by the size of $N_{k-1}(v)$ remains, and this is exponential in k.

Although the constants of proportionality of these methods are large and prohibitive, they pale in comparison to those introduced by WQO methods. WQO constants arise from testing for minor containment, which consists of "towers of 2's" functions. See [FL1], [BL], and [RS1] for more on this subject.

2.2.4 Obstruction Sets

If the obstruction set for an immersion-closed family of graphs is known, then a constructive decision algorithm automatically exists. Unfortunately, there exist very few examples of immersion- or minor-closed families of graphs for which complete obstruction sets have been isolated. As an example of the difficulty of identifying complete obstruction sets, the reader is referred to [KiL], the major result of which is the identification of the complete 110-element obstruction set for a single instance of a minor-closed family of graphs.

In this section, we show that, given any fixed k and d, the obstruction set for MDGP(k,d) is computable. This enables the generation, in principle, of the obstruction set for any fixed-parameter instance of MDGP. Such a task is formidable in its magnitude, however, as we shall see later.

Observation 2.2 *An obstruction to MDGP(k,d) contains at most $d+1$ copies of any edge.*

Lemma 2.5 *An obstruction to MDGP(k,d) contains at most c_k vertices.*

Proof Suppose G is an obstruction to MDGP(k,d), with $|V| > c_k$. Because G is a "no" instance, by the Locality Condition there exists some v such that $C_v = \emptyset$. By

Lemma 2.2 there exists some $w \in V$ such that $w \notin N_k(v)$. Consider $G' = G - \{w\}$. Because every element in C_v must be drawn from $N_{k-1}(v)$, C_v for G' is also empty. Thus, by the Locality Condition, G' is a "no" instance, so G was not minimal. $\square$

Theorem 2.6 *The obstruction set to MDGP(k,d) is computable.*

Proof By Lemma 2.5 there is a bound on the number of vertices in an obstruction, and by Observation 2.2 the number of copies of any edge in an obstruction is bounded. The obstruction set can be computed by generating and checking the finite number of graphs that satisfy these bounds. $\square$

Although this is a finite number, it is very large. The upper bound on the number of vertices is c_k, with at most $\frac{{c_k}^2 - c_k}{2}$ edges, each of multiplicity up to $d+1$. As a rough upper bound on the number, we consider the number of labelled simple (p, q) graphs (graphs with p vertices and q edges). This number is given by $\binom{\binom{p}{2}}{q}$ ([HP]). A better, but still inexact, bound would be g_p, the number of unlabelled graphs of p vertices, although even this would not take into account edge multiplicity.

Counting unlabelled graphs is difficult (see [HP] for a thorough discussion of this topic). The precise answer for g_p is known, but cannot be stated simply. An approximate answer of $g_p \sim \frac{2^{\binom{p}{2}}}{p!}$ is also known ([Wi]). This number is greater than 2^p for $p \geq 10$, which can be seen as follows.

We note that $\frac{2^{\binom{p}{2}}}{p!} = \frac{2^{(\frac{p^2-p}{2})}}{p!} = \frac{((\sqrt{2})^{p-1})^p}{p!} > (\frac{(\sqrt{2})^{p-1}}{p})^p$. This last quantity is larger than 2^p when $(\sqrt{2})^{p-1} > 2p$, which is always the case when $p \geq 10$.

Figure 2.5 shows some sample MDGP(k,d) obstructions for small values of k and d (note that MDGP(k,d) obstructions must be connected). From these examples, several structural observations are evident. For example, C_{k+1} is an obstruction for MDGP(k,1); a graph with one vertex v of degree $d+1$ and no other vertices except for v's immediate neighbors is an obstruction for MDGP(1,d) (which, in the case of simple graphs, is a star graph).

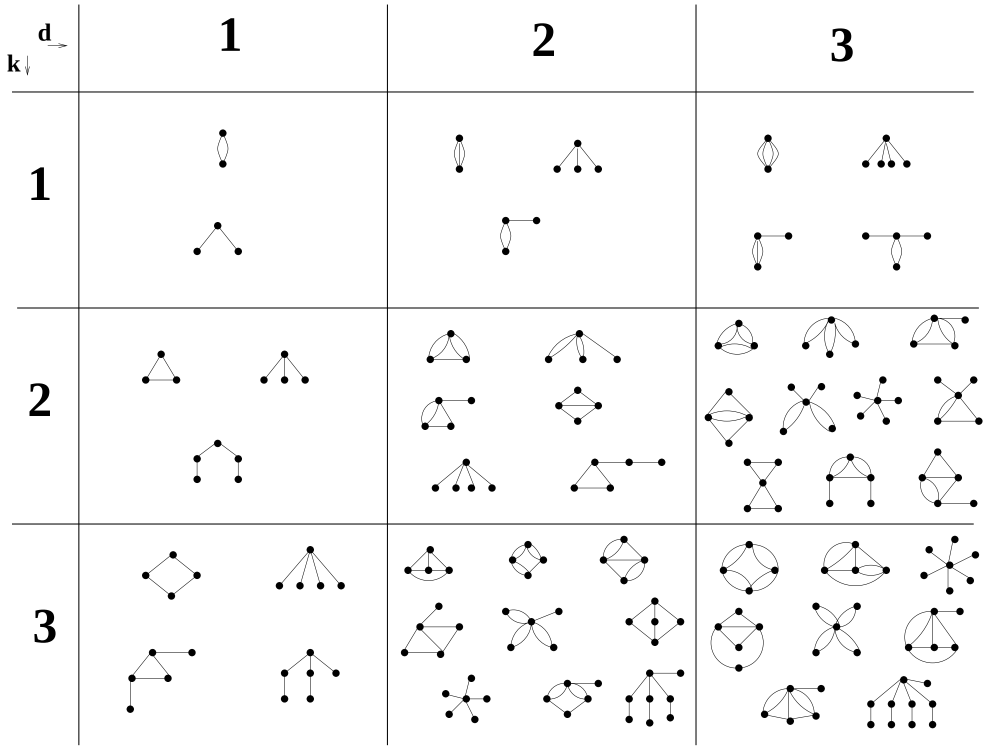


Figure 2.5: Some MDGP(k,d) obstructions

We next show that there is an exponential lower bound on the size of this obstruction set. We do this by completely characterizing the simple tree obstructions to MDGP(k,d), and then showing an exponential lower bound on trees satisfying this configuration. Thus, all trees defined and discussed in the remainder of this section are simple.

We define a *kdq-tree* T_{kdq}, for $1 \leq q \leq k$ as follows:

1. T_{kdq} contains a vertex c with $|N(c)| = \delta(c) = d + q$.
2. Some set of q neighbors of c form the roots of subtrees. These subtrees are of sizes $s_1, s_2, ..., s_q$, where $s_1 + s_2 + ... + s_q = k$.
3. Each of the remaining d neighbors of c forms the root of a subtree of size $\max(s_i)$, $1 \leq i \leq q$.

Figure 2.6 shows a sample *kdq*-tree for $k = 8, d = 2, q = 2$. Figure 2.7 shows all of the *kdq*-trees for k and d ranging from $1 - 3$.

For any tree T and vertices $u, v | \exists uv \in E_T$, we will denote by T_{uv} the connected subtree of $T - uv$ with root v. Any such connected subtree, relative to some vertex u, will be referred to as a subtree of u.

Lemma 2.6 *Any kdq-tree T_{kdq} is a "no" instance to MDGP(k,d).*

Proof Assume some T_{kdq} is a "yes" instance. Consider any subset S containing c in a satisfying partition of T_{kdq}. For each subtree T_{cy} of c, one of the following must hold:

1. $y \notin S$. The degree of S is at least one for each such y.
2. $y \in S$, but T_{cy} not entirely included in S. The degree of S is at least one for each such y.
3. $y \in S$, and T_{cy} entirely included in S.

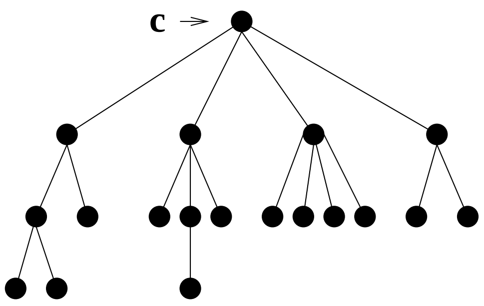


Figure 2.6: A kdq-tree ($k = 8, d = 2, q = 2$)

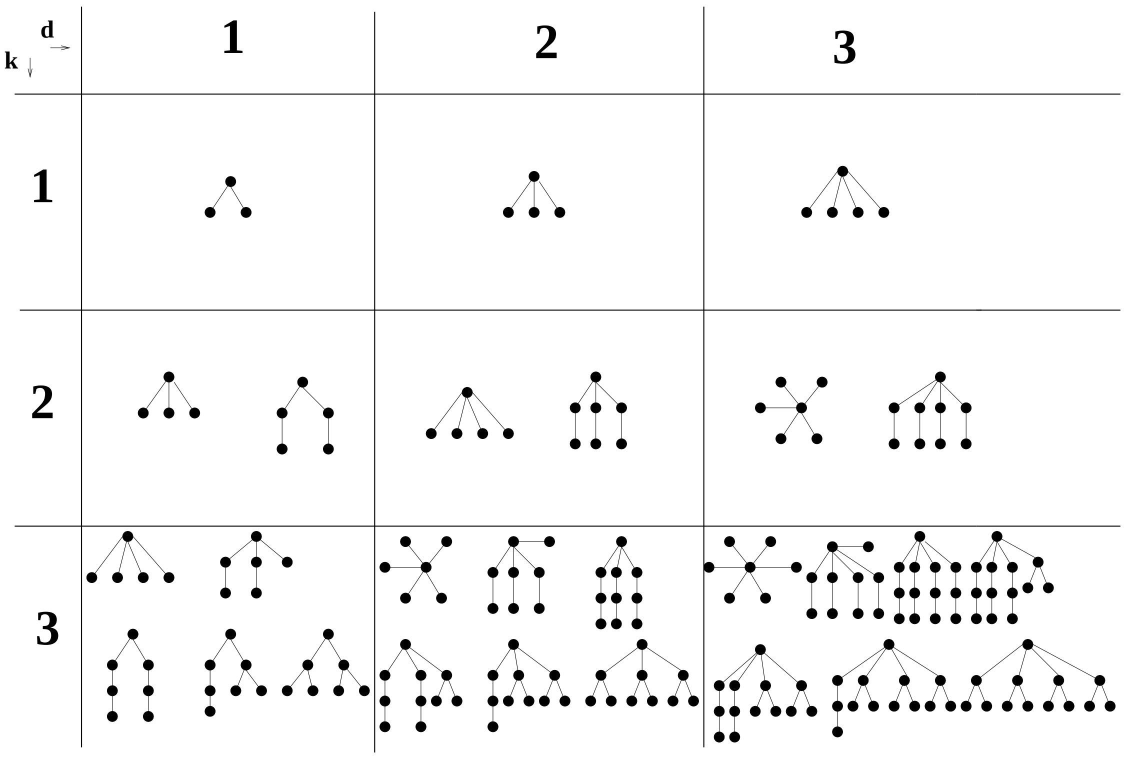


Figure 2.7: kdq-trees

Since the size of S is bounded by k, and by the definition of T_{kdq}, there can be at most $q-1$ y's of the third type. The total number of y's is $d+q$, therefore there are at least $d+1$ y's of the first and second types, each contributing at least one to $\delta(S)$. This contradicts the assumption that T_{kdq} was properly partitioned. □

Lemma 2.7 *Any kdq-tree T_{kdq} is a minimal "no" instance to MDGP(k,d).*

Proof To show that T_{kdq} is minimal, we must consider the graph obtained by any immersion operation, and show that some partition P exists. An immersion operation can be one of 1) edge removal; 2) vertex removal; or 3) lifting a pair of adjacent edges.

Before considering each of these operations in turn, we examine four scenarios:

Scenario 1: Suppose the degree of c is reduced by one of these operations: 1) removal of an edge cx or 2) removal of a vertex x adjacent to c. In either of these cases, at least one edge cx is removed. By the definition of *kdq*-tree, there exist $q-1$ subtrees of c, not including T_{cx}, of total size no more than $k-1$. A subset S of size no more than k can be formed consisting of these $q-1$ subtrees along with c. The degree of c was reduced to at most $d+q-1$ by the immersion operation, and at least $q-1$ subtrees of c have been included in S, which is therefore of degree no more than d. What remains of T_x can be partitioned into a subset by itself, as can the other subtrees of c. Each of these subsets is of size no more than k, and degree 1 or 0.

Scenario 2: Suppose two edges uc and cw, both incident on c, are lifted. By the definition of *kdq*-tree, there exist $q-1$ subtrees of c, not including T_{cu}, of total size no more than $k-1$. A subset S of size no more than k can be formed by taking the union of c with these $q-1$ subtrees. The degree of c was reduced to at most $d+q-2$ by the immersion operation, and at least $q-1$ subtrees of c have been included in S, which is therefore of degree no more than d. All other subtrees of c can be partitioned by themselves into subsets of degree 1.

Scenario 3: Suppose some immersion operation does not reduce the degree of c, but disconnects the graph. There would then exist some set of at least q subtrees of c

of size no more than $k-1$, along with some disconnected component. The q subtrees could be partitioned along with c into a subset of degree d and size no more than k. The remaining subtrees of c could be partitioned into subsets of size no more than k and degree 1. The disconnected component could be partitioned by itself into a subset of size no more than k and degree 0.

Scenario 4: Suppose some immersion operation does not reduce the degree of c or disconnect the graph, but results in reduction in the size of some subtree of c. This situation is the same as Scenario 2, except that there is no disconnected component.

We now examine each of the three immersion operations in turn.

1. Edge removal. If an edge adjacent to c is removed, Scenario 1 results. Otherwise, Scenario 3 results.

2. Vertex removal. If the vertex removed is c, each subtree of c fits into a subset of size no more than k and degree 0. If the vertex removed is one adjacent to c, we have Scenario 1. If the vertex is of degree 1, we have Scenario 4. Otherwise, we have Scenario 3.

3. Lifting a pair of adjacent edges. If both of the lifted edges were incident on c, we have Scenario 2. Otherwise, the result is Scenario 3.

□

Lemma 2.8 *For any tree T, and any $v \in V_T$ with $\delta(v) > d$, any kd-candidate subset C including v includes at least $\delta(v) - d$ entire subtrees of v. Additionally, if any set of at least $\delta(v) - d$ entire subtrees of v is of total size at most $k-1$, these subtrees, along with v, form a kd-candidate subset.*

Proof Suppose C is a kd-candidate subset for v, but includes fewer than $\delta(v) - d$ entire subtrees of v. Then C excludes part of more than $\delta(v) - (\delta(v) - d) = d$ subtrees of v, each of which contributes at least 1 to the degree of C, and thus C is not a kd-candidate subset including v, a contradiction.

For the second statement of the lemma, we need only determine the degree of the subset, because its size is at most k, and it is connected. The degree of the subset is exactly 1 for every excluded subtree of v, and this number is no more than $\delta(v) - (\delta(v) - d) = d$. Therefore, the subset is of degree no more than d, and satisfies the definition of kd-candidate subset including v. □

We observe that Lemma 2.8 also holds for forests.

Lemma 2.9 *Any tree obstruction to MDGP(k,d) is a kdq-tree.*

Proof Suppose we have a tree obstruction T. Because T is a "no" instance, by the Locality Condition there exists some vertex $v \in V(T)$ that has no kd-candidate subset. Because v has no kd-candidate subset, $\delta(v) > d$, and because T is an obstruction, by Observation 2.1 $\delta(v) \leq d + k$. For $1 \leq q = \delta(v) - d \leq k$, we have:

1. T contains a vertex $c = v$ with $|N(c)| = \delta(c) = d + q$.

Suppose that, associated with this vertex c, there exists a set of q subtrees of c containing a total of fewer than k vertices. Then, by Lemma 2.8, c would have a kd-candidate subset. Therefore, every such set of q subtrees of c contains at least k vertices.

To see than no such set contains more than k vertices, note that removal of any vertex of degree 1 from any subtree of c would still yield a set of q subtrees of c containing at least k vertices, and by Lemma 2.8, c would still have no kd-candidate subset. Thus such a T would not be minimal.

We conclude that:

2. Some set of q neighbors of c form the roots of subtrees. These subtrees are of sizes $s_1, s_2, ..., s_q$, where $s_1 + s_2 + ... + s_q = k$.

Now, consider the remaining d neighbors of c, and the subtrees of which they are roots. By reasoning analogous to that above, we note the following: if one of these subtrees is of size less than $\max(s_i)$, $1 \leq i \leq q$, then c has a kd-candidate subset; if one of these subtrees is of size greater than $\max(s_i)$, $1 \leq i \leq q$, then T is not minimal. Thus it must be the case that:

3. Each of the remaining d neighbors of c forms the root of a subtree of size $\max(s_i)$, $1 \leq i \leq q$.

Therefore, by $1, 2$, and 3, T satisfies the definition of kdq-tree. □

Theorem 2.7 *A tree is an obstruction to MDGP(k,d) iff it is a kdq-tree.*

Proof Follows from Lemmas 2.7 and 2.9. □

We now address the issue of a lower bound on the size of the MDGP(k,d) obstruction set.

Theorem 2.8 *When $k > d + 4$, the size of the obstruction set of MDGP(k,d) is at least $max\{2^{d+1}, 2^{k-3}\}$.*

Proof Consider any tree consisting of a root vertex with $d+2$ children: one degree-1 vertex, and $d+1$ roots of arbitrary T_{k-1} trees (trees containing $k-1$ vertices). (See Figure 2.8.) Any such tree is a *kdq-tree*; specifically it is a $kd2$-tree. By Theorem 2.7, such a tree is an obstruction to MDGP(k,d).

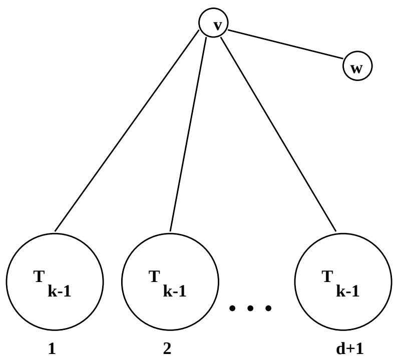


Figure 2.8: A general tree obstruction to MDGP(k,d)

We begin by examining the number of nonisomorphic rooted trees containing p vertices. (The topic of exactly counting rooted trees is thoroughly discussed in [HP]; here we seek only to show an exponential lower bound.) We show by construction that the number of nonisomorphic rooted trees with p vertices ($p \geq 2$) is at least 2^{p-2}. For $p = 2$, we observe that there exists only one rooted tree containing 2 vertices. For $p > 2$, we construct 2 trees with p vertices for each nonisomorphic rooted tree T_{p-1} containing $p-1$ vertices. One of these trees consists of a new root vertex with the root of T_{p-1} as its single child. The other tree is identical to T_{p-1}, except for the addition of a new vertex of degree 1 incident on the root. Figure 2.9 shows the nonisomorphic rooted trees containing $2-5$ vertices constructed by this method. The roots are double-circled in the figure. Arrows indicate the most recently added vertices.

No two of these newly constructed trees are isomorphic. If they were, their roots would have to be of the same degree, hence they were constructed from nonisomorphic subtrees. Removing the most recently added vertex would yield isomorphic subtrees.

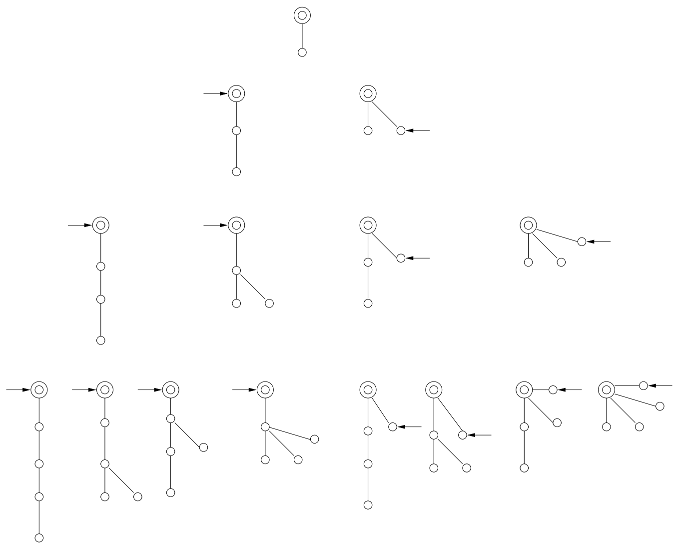


Figure 2.9: Some nonisomorphic rooted trees

We observe that, in any graph matching the configuration of Figure 2.8, there is only one vertex that can be designated as c. Any other vertex that has one subtree of size 1 must have at least one other subtree of size exceeding k.

The general obstruction shown in Figure 2.8 contains rooted subtrees with $k-1$ vertices each; hence even if all of these subtrees had the same configuration, the number of such obstructions is at least 2^{k-3}.

Additionally, this general obstruction contains $d+1$ of these rooted subtrees. Even if no repetition were allowed in the configuration of these subtrees, the number of obstructions matching this configuration would still be bounded below by $\binom{2^{k-3}}{d+1}$ $> (\frac{2^{k-3}-d}{d+1})^{d+1} > 2^{d+1}$ when $2^{k-3} > 3d+2$. This is always the case when $k > d+4$. $\square$

It should be mentioned that the lower bound on MDGP(k,d) obstruction set size established here is quite loose. In addition to the omissions mentioned in the proof above, no attempt has been made to count kdq-trees in the case of $q \neq 2$. Furthermore, the MDGP(k,d) obstruction set includes many graphs that are not trees. Even without considering all of these possibilities, however, the MDPG(k,d) obstruction set is seen to be exponential in both k and d.

2.2.5 Tractability on Restricted Classes of Graphs

Thus far, we have considered the fundamental problem from a very general perspective. In reality, the graphs that serve as input to real instances of FPGA partitioning might not be so generalized. Circuits may, in fact, have a certain measure of underlying structure. Many circuit graphs are of treewidth at most two, a class of graphs known as *series-parallel*. In some situations it may even be possible to assume that the input graph is a tree or a forest, or even a simple tree or simple forest. Although the tractability of MDGP on most of these graph families is still an open question, we can show that MDGP, restricted to simple trees (and hence, forests) is efficiently solvable.

Theorem 2.9 *MDGP, restricted to simple trees, is decidable in $O(n^2 \log n)$ time.*

Proof Given a simple tree T, first check whether any vertex has degree $d+k$ or more. If so, T is a "no" instance, because it contains an obstruction (the star graph with $d+k$ vertices).

Otherwise, for each $v \in T$, do the following. If the degree of v is no more than d, then $\{v\}$ is a kd-candidate subset for v. If the degree of v is more than d, we perform the following steps:

1. Compute the size of each subtree of v. This takes $O(n)$ time.

2. Sort the sizes of the subtrees of v. This takes $O(n\ log\ n)$ time.

3. Check the total size t of the smallest $\delta(v) - d$ subtrees of v. This takes $O(n)$ time.

4. If t is less than k, then v along with the set of smallest $\delta(v) - d$ subtrees of v form a kd-candidate subset, by Lemma 2.8. Otherwise, by Lemma 2.8, v has no kd-candidate subset.

By the Locality Condition, if any vertex has no kd-candidate subset, then T is a "no" instance, otherwise it is a "yes" instance.

The outer loop executes at most once for every vertex, and the inner loop is in $O(n\ log\ n)$. Therefore, the complexity of this procedure is $O(n^2\ log\ n)$. $\square$

Because each simple tree can be handled independently, Theorem 2.9 generalizes to simple forests.

We conclude this section by noting that the algorithm outlined in the proof of Theorem 2.9 is not designed for efficiency; our primary purpose here is to establish that the problem is in $\mathcal{P}$. We conjecture that, by careful use of tree traversals and data structures, the complexity may be $O(n\ log\ n)$ or even better.

Chapter 3

Extending the Fundamental Problem: FPGA Minimization

The fundamental problem has given us a basis for theoretical study of FPGA partitioning. In this chapter, we proceed further to incorporate one of the primary issues in VLSI design, that of cost minimization.

3.1 Problem Definition and Prior Results

Although some results have been obtained for MDGP and MDGP(k,d), the problem as stated is not entirely representative of the issues inherent in FPGA partitioning. MDGP is useful as a starting point, however, and can be generalized in ways that address other issues.

A primary consideration in FPGA partitioning is cost. While we have shown algorithms that can decide whether an input graph is a "yes" instance of MDGP(k,d), and while we can even find a feasible partition in linear time, thus far we have ignored the question of *minimizing* the number of subsets in a partition. Since the number of subsets in a partition represents the number of FPGAs used in the realization of a circuit (hence the cost), it is important that this issue be considered.

It is easy to modify the definition of the problem to accommodate this additional constraint, in a problem that we call FPGA Minimization:

Instance: a graph $G = (V, E)$, and three integers k, d and p.

Question: Is there is a partition of V into disjoint sets $V_1, \ldots, V_m$ such that $m \leq p$, $\forall i : |V_i| \leq k$, and such that if E_i is the set of edges with exactly one endpoint in V_i, $max_{1 \leq i \leq m} |E_i| \leq d$?

FPGA Minimization is $\mathcal{NP}$-complete, since it contains MDGP as a special case in which $p = |V|$.

However, once again the physical limitations inherent in partitioning for FPGAs allow us to assume constant bounds on some of the parameters. Since the size and pincounts of the devices are constrained by the technology, we consider the variant of FPGA Minimization in which these two parameters are fixed. In this situation, which is more representative of the real problem of partitioning a circuit over the minimum number of FPGAs, we wish to minimize p, the number of subsets in a partition. We will refer to the decision version of this problem as *p-way* MDGP(k,d). Unfortunately, even this restricted version is very difficult.

Theorem 3.1 *([Go]) p-way MDGP(k,d) is $\mathcal{NP}$-complete.*

3.2 New Results

In this section, we present a theoretical study of FPGA Minimization and *p-way* MDGP(k,d). We will also look at the version of the problem in which all three parameters are fixed, which will be denoted by MDGP(k,d,p). We find that many of the results for MDGP(k,d) apply in this setting as well, although FPGA Minimization and its variants provide some curiosities of their own. Additionally, we learn that FPGA Minimization remains $\mathcal{NP}$-complete even on very restricted graph families.

Since *p-way* MDGP(k,d) is of potential relevance in real circuit partitioning, in this section we also turn our attention to the task of developing a practical algorithm to solve this problem. Because it is $\mathcal{NP}$-complete, we know there exist no efficient exact algorithms (unless $\mathcal{P} = \mathcal{NP}$). For this reason, most researchers depend upon heuristics to provide quick and workable solutions. We will present a new approach that is motivated by the theoretical study of MDGP(k,d).

3.2.1 Refining the Tractability of FPGA Minimization

We begin with a further exploration into the tractability of FPGA Minimization. Specifically, we look at what happens when the input instance graph must conform to a certain structure. We find that even with severe restrictions, including some for which it is known that MDGP is in $\mathcal{P}$, FPGA Minimization remains $\mathcal{NP}$-complete.

By Theorem 2.9, we know that MDGP, when restricted to simple forests, is in $\mathcal{P}$. We also know that connectivity is not an issue for MDGP (each connected subgraph can be solved independently). In the case of FPGA Minimization, the problem remains $\mathcal{NP}$-complete for disconnected graphs, even for forests in which each component is a simple chain. This can be shown via an easy reduction from Partition [GJ]:

Instance: a finite set A and a "size" $s(a_i) \in Z^+$ for each $a_i \in A, 1 \leq i \leq |A|$.

Question: Is there a subset $A' \subseteq A$ such that

$$\sum_{a_i \in A'} s(a_i) = \sum_{a_i \in A - A'} s(a_i)?$$

Given an arbitrary instance P of Partition, we construct in polynomial time an instance of FPGA Minimization, consisting of a graph G and integers k, d and p. G is composed of a disconnected collection of $|A|$ simple chains, each corresponding to some $a_i \in A, 1 \leq i \leq |A|$ and containing $s(a_i)$ nodes. We set $k = \lfloor \frac{\sum_{i=1}^{|A|} s(a_i)}{2} \rfloor, d = 0$, and $p = 2$.

If P is a "yes" instance of Partition, then G may be partitioned into 2 subsets, each of degree 0 and size k. The first subset contains the chains corresponding to the a_i's $\in A$; the other contains the chains corresponding to the a_i's $\in A'$. Conversely, suppose G is a "yes" instance of FPGA Minimization for these values of k, d and p. Then, because $d = 0$, each chain corresponding to some a_i is completely contained in one subset. There are 2 subsets, each of size k. Therefore one subset contains the chains representing the a_i's $\in A$ in some solution to P; the other subset contains the chains representing the a_i's $\in A'$. Figure 3.1 illustrates the FPGA Minimization instance produced from an instance of Partition in which $|A| = 7$, and $s(a_1) = 8, s(a_2) = 7, s(a_3) = 5, s(a_4) = 3, s(a_5) = 3, s(a_6) = 2, s(a_7) = 2$, with a satisfying partitioning indicated in dotted lines.

The complexity of FPGA Minimization restricted to simple trees is still an open question, but we have the following result for FPGA Minimization on trees.

Theorem 3.2 *FPGA Minimization, restricted to trees, is $\mathcal{NP}$-complete.*

Proof Given an instance P of Partition, we construct a tree instance of FPGA Minimization as follows. G consists of $|A|$ nonsimple chains, $C_1, C_2, \ldots, C_{|A|}$, with C_i containing $s(a_i)$ nodes and every edge having multiplicity $|A| + 1$.

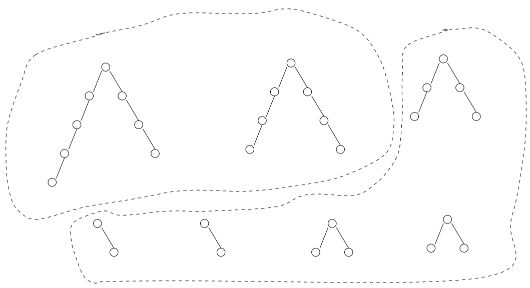


Figure 3.1: A disconnected instance of FPGA Minimization

Additionally, G contains a root vertex v connected by one edge to each C_i, and also connected by one edge to $\lfloor \frac{\sum_{i=1}^{|A|} s(a_i)}{2} \rfloor - 1$ other vertices, $v_1, v_2, \ldots, v_{\lfloor \frac{\sum_{i=1}^{|A|} s(a_i)}{2} \rfloor - 1}$. Finally, $k = \lfloor \frac{\sum_{i=1}^{|A|} s(a_i)}{2} \rfloor$, $d = |A|$, and $p = 3$. Figure 3.2 illustrates the tree FPGA Minimization instance produced from a Partition instance in which $|A| = 4$, and $s(a_1) = 4, s(a_2) = 3, s(a_3) = 2, s(a_4) = 1$.

If P is a "yes" instance of Partition, then G may be partitioned as follows. One subset, of degree $|A| = d$, contains v and $v_1, v_2, \ldots, v_{\lfloor \frac{\sum_{i=1}^{|A|} s(a_i)}{2} \rfloor - 1}$, and is of size $\lfloor \frac{\sum_{i=1}^{|A|} s(a_i)}{2} \rfloor = k$. There are 2 other subsets, one containing all the chains corresponding to the s_i's in A, the other containing all the chains corresponding to the s_i's in A'. Each is of size $\lfloor \frac{\sum_{i=1}^{|A|} s(a_i)}{2} \rfloor = k$, and of degree no more than $|A| = d$.

Figure 3.3 shows the partitioning of the tree instance of Figure 3.2. Each subset is outlined in dotted lines.

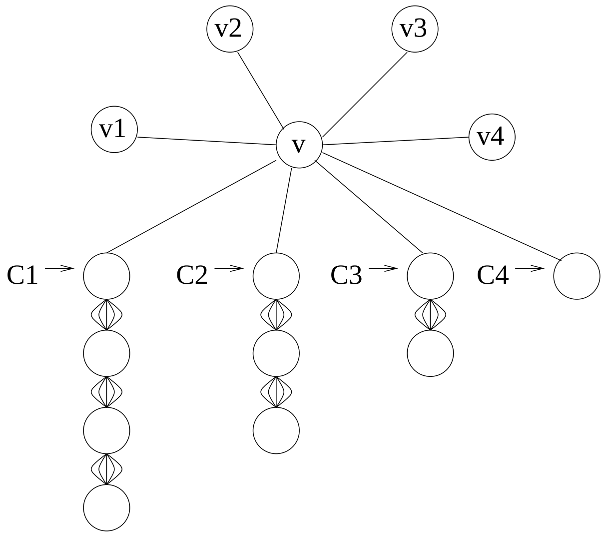


Figure 3.2: A tree instance of FPGA Minimization

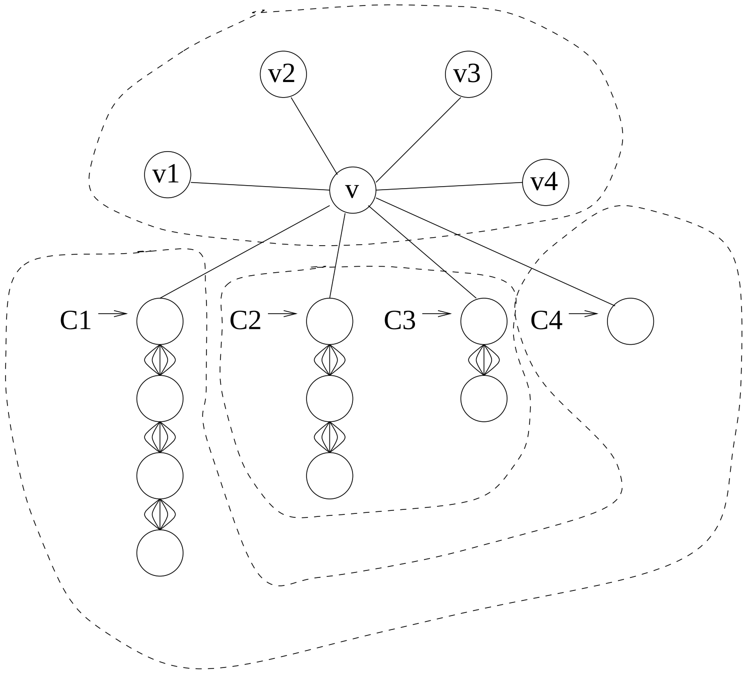


Figure 3.3: Partitioning a tree instance of FPGA Minimization

Conversely, suppose G is a "yes" instance of FPGA Minimization. We observe that the subset S containing v must also contain $\lfloor\frac{\sum_{i=1}^{|A|} s(a_i)}{2}\rfloor - 1$ subtrees rooted by v's neighbors, otherwise the degree of S would exceed $|A| = d$. All of these subtrees must be of size 1, otherwise the size of S would exceed $k = \lfloor\frac{\sum_{i=1}^{|A|} s(a_i)}{2}\rfloor$. Therefore, S contains v and $v_1, v_2, \ldots, v_{\lfloor\frac{\sum_{i=1}^{|A|} s(a_i)}{2}\rfloor - 1}$. Each chain representing some a_i must be completely contained in one subset, and there can be no more than 2 other subsets, each of size $\lfloor\frac{\sum_{i=1}^{|A|} s(a_i)}{2}\rfloor = k$; therefore each of these subsets represents either A or A' in a solution to P. $\Box$

This result generalizes to show that, for non-simple graphs, FPGA Minimization is $\mathcal{NP}$-complete for many classes of graphs, including series-parallel graphs, and all graphs of bounded treewidth.

Table 3.1 summarizes, for comparison, the complexity results for MDGP and FPGA Minimization.

Table 3.1: Complexity of MDGP and FPGA Minimization

Graph Class	MDGP	FPGA Minimization
General Graphs	$\mathcal{NP}$-complete	$\mathcal{NP}$-complete
Simple trees	in $\mathcal{P}$	unknown
Trees	unknown	$\mathcal{NP}$-complete
Simple forests	in $\mathcal{P}$	$\mathcal{NP}$-complete
Forests	unknown	$\mathcal{NP}$-complete
Simple Series-Parallel Graphs	unknown	unknown
Series-Parallel Graphs	unknown	$\mathcal{NP}$-complete
Simple Graphs of Bounded TW	unknown	unknown
Graphs of Bounded TW	unknown	$\mathcal{NP}$-complete

3.2.2 MDGP(k,d,p) Results

In this subsection, we present some findings pertinent to MDGP(k,d,p), the version of FPGA Minimization in which all three parameters are fixed. MDGP(k,d,p) is somewhat of a curiosity. When all three parameters are fixed, we find that WQO theory applies (the "yes" family is closed under immersion). At the same time, however, fixing all three parameters trivializes the problem from a complexity perspective. Any graph with more than $k \times p$ vertices is a "no" instance, thus the problem can (in principle) be solved in constant time by table lookup.

In practice, however, the time required to construct such a table is prohibitive; MDGP(k,d,p) cannot be practically solved in this manner. It may still be beneficial to examine this problem from a WQO-theoretic point of view, in hopes of finding a fast obstruction-based heuristic. For example, in [GLR], it was shown that an obstruction-based heuristic for Layout Optimization (a problem closed under the minor order) was extremely effective, if not exact. This heuristic was based upon the observation that the vast majority of "no" instances contained one of a very small set of obstructions, all of which had fast containment tests.

Self-reduction

At this time it is unknown whether there exists a fast obstruction-based heuristic for MDGP(k,d,p). If an efficient decision heuristic were found, however, it could be used together with a fast self-reduction algorithm. We now show that such a self-reduction exists. It assumes $G = (V, E)$ is a "yes" instance (this can be checked with the decision algorithm) and then constructs a solution as outlined subsequently.

A set of p (or fewer) "core" vertices (representatives of distinct subsets) is identified as the algorithm progresses (initially this set is empty). Each vertex v is tested to see if its commitment to the same subset as some core vertex c still results in a "yes" instance (the commitment is done by adding $d+1$ edges between v and c). If the resulting graph is still a "yes" instance, the added edges are retained, forcing those vertices to occupy the same subset during the remainder of the algorithm. If the resulting graph is a "no" instance for every candidate c, then v is a new core vertex, and will never be assigned the same subset as any other core vertex. Since the graph was a "yes" initially, at most p vertices will be designated as core vertices. Each vertex is tested with at most p other vertices. Since p and d are constants, the algorithm runs in linear time.

Obstruction Sets

A decision algorithm based on table lookup for this problem is certainly infeasible, because of the large number of "yes" instances. One might entertain the concept of an obstruction-based approach instead, if one could be devised that used only a small subset of obstructions in an efficient manner. Although we have no positive results in this direction, we present here some findings on the size of the MDGP(k,d,p) obstruction set. At this time, these results are of no known practical value, and are included solely as a combinatorial exercise of unknown potential for future use.

Lemma 3.1 *Any graph consisting of p $(d+1)$-regular $(d+1)$-edge-connected $(k-1)$-components, along with a 2-vertex component connected by $d+1$ edges, is an obstruction to MDGP(k,d,p).*

Proof Let G be any such graph. We first show that G is a "no" instance to MDGP(k,d,p). (See Figure 3.4 for sample $(d+1)$-regular $(d+1)$-edge-connected $(k-1)$-components for $k = 10, d = 7$.) Each $(k-1)$-component must be self-contained in a subset, because of its $(d+1)$-edge-connectivity. The 2-vertex component cannot be separated into two subsets, since the vertices are connected by $d+1$ edges. Additionally, the 2-vertex component does not fit into any subset containing a $(k-1)$-component. Therefore, a satisfying partitioning of G requires $p+1$ subsets.

Next we show that G is minimal. Removing any vertex from the 2-vertex component allows the remaining vertex to fit into a subset with any $(k-1)$-component.

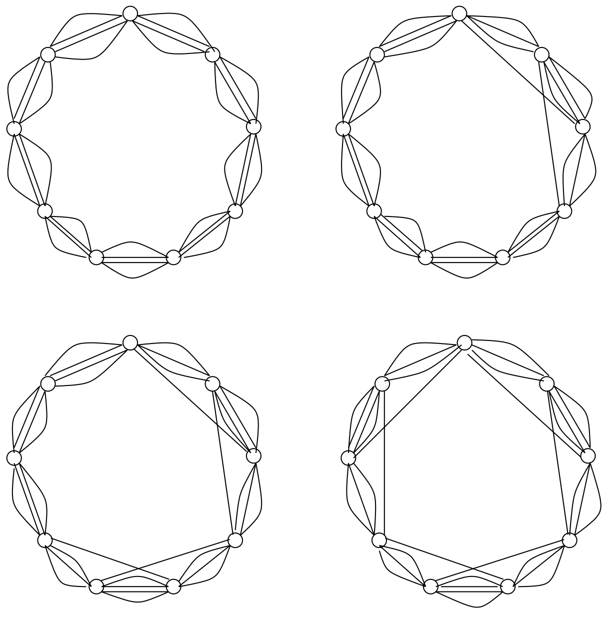


Figure 3.4: Some 8-regular, 8-edge-connected 9-components

Removing any vertex from a $(k-1)$-component allows the remainder of that component to fit into a subset with the 2-vertex component. Removing an edge from the 2-vertex component allows its two vertices to fit into subsets with any two $(k-1)$-components. Any other immersion operation causes some vertex v from a $(k-1)$-component to have its degree reduced to no more than d. A satisfying partitioning can then be done by placing v into a subset with some other $(k-1)$-component, and placing the 2-vertex component with the remainder of v's component. $\Box$

Lemma 3.2 *The number of $(d+1)$-regular $(d+1)$-edge-connected $(k-1)$-graphs is proportional to kd.*

Proof We consider here the case where d is odd (a similar construction applies when d is even). A $(k-1)$-cycle, in which each edge appears $\frac{d+1}{2}$ times satisfies the definition of $(d+1)$-regularity and $(d+1)$-edge-connectivity. Consider any u, v, w, x in this graph, such that $u \neq v \neq w \neq x$, and $|uv| = |vw| = |wx| = \frac{d+1}{2}$. The graph obtained by removing one copy of uv and one copy of wx, and adding uw and vx is still $(d+1)$-regular and $(d+1)$-edge-connected. This can be repeated $\frac{k}{3} * (\frac{d+1}{2} - 1)$ times to produce $\frac{k}{3} * (\frac{d+1}{2} - 1)$ nonisomorphic $(d+1)$-regular $(d+1)$-edge-connected $(k-1)$-graphs. $\Box$

Theorem 3.3 *The size of the set of immersion-minimal elements of fixed-parameter MDGP(k,d,p) is at least exponential in p.*

Proof By Lemma 3.2, we know that there are $O(kd)$ $(d+1)$-regular $(d+1)$-edge-connected $(k-1)$-components. By Lemma 3.1, any graph consisting of p such components (along with the 2-component) is an obstruction to MDGP(k,d,p), and there are at least $O(\binom{p+kd}{p}) = O(\frac{(p+kd)!}{p!kd!})$ such graphs. When $kd > 2p$ (a likely situation in FPGA partitioning), $\frac{(p+kd)!}{p!kd!} > \frac{kd^p}{p^p} > 2^p$. $\Box$

There are some cases when the MDGP(k,d) obstruction set is completely contained in the MDGP(k,d,p) set. In the most general setting, then, the obstruction set for MDGP(k,d,p) is larger than that for MDGP(k,d). It can contain the entire MDGP(k,d) obstruction set, along with exponentially (in p) many more obstructions.

3.2.3 *p-way* MDGP(k,d): A Practical Heuristic

As observed earlier, *p-way* MDGP(k,d) is of practical significance, in that it describes the problem of partitioning a logic circuit to fit onto a minimum number of FPGAs. The decision version of the problem is $\mathcal{NP}$-complete, so no practical exact algorithms are known. It is, however, a well-studied problem, and many efficient heuristics have been proposed for it. This will be discussed in more detail later on.

In this section, we explore some of the issues involved, including some of the difficulties of applying theory to practice in this setting. We then present a new heuristic for FPGA Minimization that employs some of the traditional approaches, but is also driven in part by theoretical results.

Circuit Characteristics

We begin with an overview of the process of converting a logic design into a format suitable for partitioning.

A combinational logic circuit may be represented as a directed acyclic graph (DAG), in which nodes represent I/O and boolean functions. At this level of representation, the functions consist of primitive gates. Directed edges represent the flow of output from one node to input of another.

As mentioned in Section 1.2, the FPGA contains a set of CLBs, each of which is a look-up table. A look-up table is a programmable logic block with x inputs and y outputs. It is capable of simultaneously implementing any set of y functions over x or fewer inputs. Typically, the number of inputs is approximately four, and the number of outputs two, but this varies somewhat for different FPGA types ([X]).

An important step in the process of converting a circuit from a design consisting of gates to an implementation with CLBs is that of *technology mapping*. This is the process of splitting the design into communicating components, each of which can be realized with a single CLB. Technology mapping is itself a complex process, and a topic of independent interest ([MR]).

Partitioning the circuit over a set of FPGAs can be performed either before or after technology mapping, and there are pros and cons to both choices. Partitioning before technology mapping allows the partitioner more latitude, in that technology mapping forces an early commitment of gates to the same CLB. On the other hand, technology mapping greatly reduces the complexity of the circuit. An FPGA typically has $20-30$ times more gates than CLBs ([X]). Thus the partitioning instance is simpler at the CLB level than the gate level. Experiments were performed in [We] to compare the two approaches, and the results of these experiments indicate that it is preferable to perform technology mapping first.

In what follows, therefore, we assume that the logic circuit has already been technology mapped. Our instance then consists of a set of CLBs with interconnections. In addition, there are inputs to the system called *primary inputs* (PIs) and outputs called *primary outputs* (POs). The graphical instance remains directed and acyclic after technology mapping. Vertices representing PIs have no incoming edges, and vertices representing POs have no outgoing edges. The precise function performed by each CLB is of no consequence to the partitioner.

Figure 3.5 illustrates a simplified example, consisting of three CLBs, each with two inputs and one output. There are two PIs and one PO.

A *net* in a circuit is a set of pins (I/O's of the chip or of the CLBs) connected by the same wire. A *netlist* is a list of nets. In the circuit of Figure 3.5 the netlist consists of five nets: $\{PI1, A\}, \{PI2, A, C\}, \{A, B, C\}, \{B, PO1\}$, and $\{B, C\}$. Two of these nets ($\{A, B, C\}$ and $\{B, C\}$) are *internal*, in that they are not connected to any PI or PO. Nets that are connected to I/O are said to be *external*.

When partitioning a circuit over FPGAs, if two (or more) CLBs connected by an

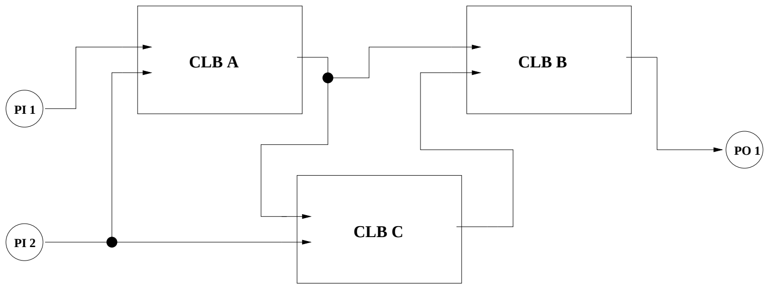


Figure 3.5: An example circuit

internal net are placed onto different FPGAs, each of the FPGAs containing one of these CLBs will require one I/O pin to accommodate that net. In graphical terms, such a net contributes 1 to the degree of each subset. Any external net will require one I/O pin on an FPGA if *any* CLB in that net is contained on the FPGA, even if all CLBs involved in the net are partitioned onto the same FPGA.

At this point, we become aware of some discrepancies between our theoretical model and applications. There are three primary issues:

1. Our model consists of an undirected graph, while circuits have direction (from PIs to POs). For the purposes of FPGA partitioning, it turns out that this is not a problem, because the FPGA is programmable. I/O pins may be programmed in either direction. Thus, we may safely ignore direction in our theoretical model for partitioning.

2. In our graphical representation of a circuit, each node represents a CLB, and edges represent connections between CLBs. However, from the above discussion, it is clear that edges between CLBs and PIs/POs play an important part in the partitioning process. For example, if the circuit in Figure 3.5 were partitioned to fit onto a single FPGA, the subset representing that FPGA would still need to have a degree of 3, representing the nets required for the PIs and PO. Fortunately, for theoretical purposes we can also accommodate this shortcoming

with graph gadgets. Every PI and PO can be represented as a $(d+1)$-edge-connected k-component, and the parameter p in FPGA Minimization can be incremented by the number of PIs and POs.

3. The third issue is not so easily dismissed. When converting a circuit into its graphical counterpart, we note that many of its nets require *hyperedges*: edges with more than two endpoints. We could change a hypergraph into an ordinary graph by converting each hyperedge into a clique: a set of vertices all of which are connected to each other. This "fix" would preserve connectivity information. However, Figure 3.6 illustrates what happens when a hyperedge consisting of vertices $\{a, b, c\}$ is converted to a clique. In the example, nodes a and b have been partitioned into one subset and node c into another. In the hypergraph representation, each subset has a degree of 1; in the simple graph representation, each subset has a degree of 2. Such a "fix" can never cause a partitioner to find a partitioning for a "no" instance, however, it could fail to find a partitioning for a "yes" instance. Converting a hypergraph to a simple graph, for the purposes of partitioning, is a well-known issue, and it has been speculated in [Le] that this cannot be done in a way to preserve complete correspondence. The heuristic that we will present handles hypergraphs. In later sections, we find ways to deal with hypergraphs from a theoretical point of view.

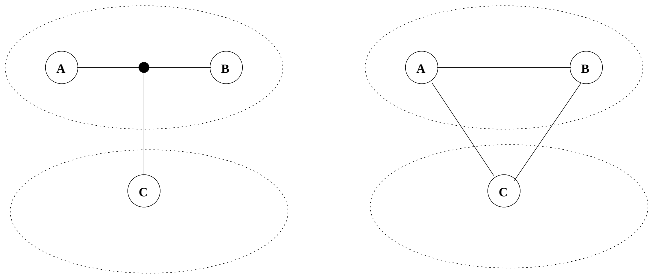


Figure 3.6: Partitioning a hypergraph and a simple graph

Prior Work

Circuit partitioning is a widely studied problem. It has been formulated in many ways, and many different types of heuristics exist. See [AK] for a quite comprehensive survey. Partitioning specifically for FPGAs has also been well researched. See [CLCDL], [HK], [KBK] and [WK] for several examples.

These heuristics vary greatly, but they sometimes contain some common elements. There may be a clustering phase, in which CLBs are committed to sharing FPGAs early in the partitioning process. This is often done in a greedy fashion. There is often an element of randomization, perhaps in the choice of initial CLBs for clustering. Randomization allows the potential for different partitioning runs to produce different results; usually the partitioner is run many times and a best solution chosen. Another element that is almost always present is some kind of iterative improvement phase, usually based on swapping the placement of individual nodes in order to improve the quality of an existing partition. See [KL] and [FM] for a thorough discussion of these techniques.

A New Approach

The heuristic we present here has many of the same characteristics as other known heuristics. It differs in that it was initially motivated by the theoretical study of MDGP and FPGA Minimization, and is driven by some of those ideas.

Many heuristics rely very heavily upon the iterative improvement phase; indeed some heuristics even begin with an arbitrary partitioning and depend solely upon iterative improvement. This appears to work reasonably well when the objective function is to minimize the number of connections between subsets, without regard for minimizing the number of subsets. Because we seek to minimize the number of subsets, this strategy alone is not of much use, because it does not incorporate any way to eliminate subsets.

In our heuristic, we attempt to concentrate more on the early clustering phase, and less on later improvement. We build our subsets one at a time, with a strong focus on the efficient packing of each subset. After each new subset has been created, an iterative improvement pass is made over that subset and all other existing subsets. This iterative improvement phase attempts to swap CLBs from subset to subset in such a way to improve their packing, in order to make room for more CLBs in each subset. This will be described in more detail later.

By the lemmas and theorems presented in Chapter 2, we know that we can (in principle) efficiently find a solution to MDGP(k,d). This can be done by exploiting the Locality Condition: if a solution exists, we can find it by confining our search for kd-candidate subsets to a bounded-size neighborhood for each vertex. No vertex v need ever share a subset with another vertex not in $N_{k-1}(v)$. This property no longer holds generally for FPGA Minimization. Indeed, Figure 3.7 shows a graph that can only be partitioned properly by violating this "near neighborhood" property, assuming $k = 2, d = 2$, and $p = 2$.

Nevertheless, it still seems reasonable to begin with clusters from a bounded-size neighborhood. Even though graphs can be contrived to thwart almost any heuristic approach (see [Go] for a discussion of this topic), our experiments (results to follow) indicate good results from confining the search for cluster expansion to near neighborhoods of the initially chosen vertex.

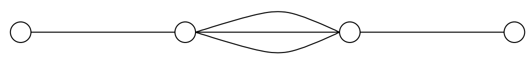


Figure 3.7: An instance of FPGA Minimization

Although many partitioning algorithms are not designed for hypergraphs, for purposes of FPGA partitioning we must cope with graphs containing hyperedges. The primary data structures for the program are the CLB lists and the netlists, both of which are maintained as linked lists of pointers crossreferencing each other. For the circuit illustrated in Figure 3.5, the CLB list and netlist data structures are as in Figure 3.8.

The general strategy we present here for FPGA Minimization is quite simple. The main partitioning algorithm, **FPGA_Min** is as follows:

```
                Algorithm FPGA_Min

num_subsets ← 0
 do
    num_subsets ← num_subsets + 1
    randomly select a seed CLB for the new subset
    expand_subset
    if num_subsets > 1
        improve_partition
 until all CLBs assigned
```

The only routines that need further explanation are *expand_subset* and *improve_partition.*

In procedure *expand_subset*, the current subset is expanded from some randomly chosen initial CLB v. The expansion is done by selecting the best candidate CLB from $N_{k-1}(v)$. The best CLB is the one which, when added to the current subset, yields the highest value. Value is calculated using the following formula:

$$value = \frac{subset\_size}{subset\_degree} + (subset\_size * SIZE\_BONUS)$$

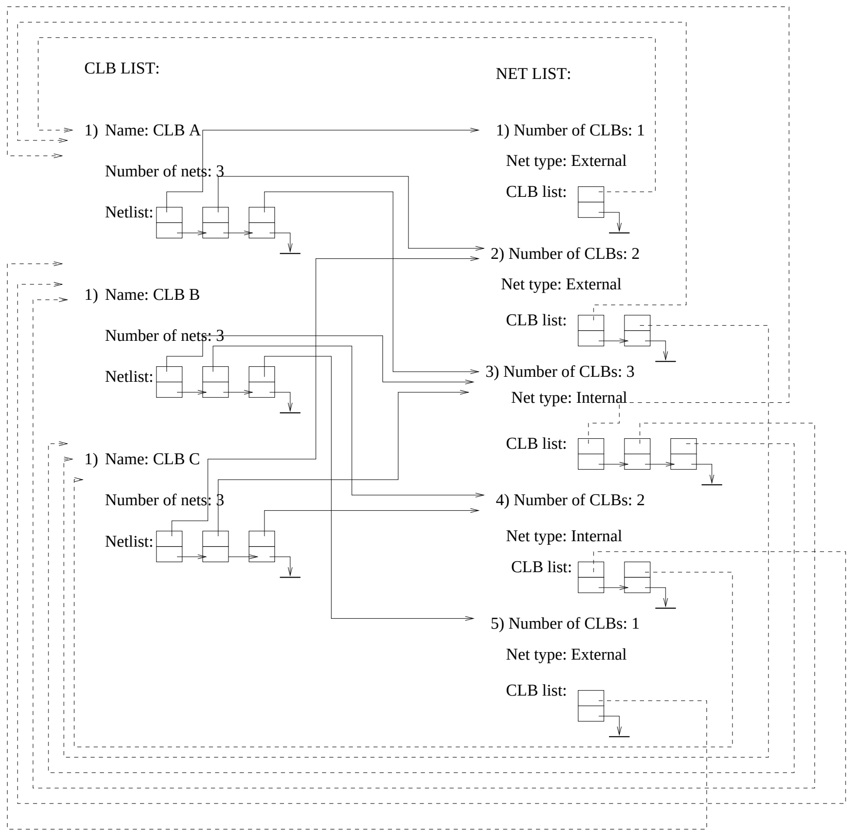


Figure 3.8: Data structure for FPGA Minimization

This formula reflects the fact that, given two subsets of the same size, we favor the one with the smaller degree, and given two subsets of the same degree, we favor the one with the larger size. Additionally, in the case of more than one subset of identical subset_size:subset_degree ratio, the larger subset receives a higher value, via a positive value for *SIZE_BONUS*. In experimental runs, a value of 0.01 for *SIZE_BONUS* produced the best results, so this value was used throughout. Ties were broken arbitrarily. The process continues until no candidate CLB can be added without violating size or pincount constraints.

The value formula given here is defined for an individual subset. It is used to guide the addition of new CLBs to an existing subset. The idea is somewhat similar to the ratio cut of [WC], a more sophisticated concept used as a metric of overall partition quality. The ratio cut between each two subsets is the number of edges between those two subsets divided by the product of the two subset sizes.

The *improve_partition* procedure is a simplified iterative improvement algorithm, inspired by the method of [KL]. It iterates through pairs of CLBs that have already been mapped to subsets. If swapping CLB x from subset X with CLB y from subset Y produces an improvement in the sum of the values of the two subsets, without violating constraints, the swap is performed. At the end of each *improve_partition* execution, a check is made to see if any subset has been modified in such a way to admit additional CLBs to be added and, if so, the CLBs are added. If any more CLBs are added, *improve_partition* executes again.

The outer loop executes at most once per CLB. The *expand_subset* routine is confined to $N_{k-1}(v)$ for seed CLB v, so its complexity is constant.

The *improve_partition* routine is the most time-consuming step. It executes at most once per CLB pair, and then repeats if at least one new CLB can be added to any subset. (Repetition until no better solution is found is a characteristic of most iterative improvement procedures.) Therefore, *improve_partition* is of cubic complexity, and the complexity of the overall algorithm is $O(n^4)$. The algorithm performed sufficiently well for our purposes. For this reason, and because this heuristic is not of significant independent interest, no attempt was made to improve this efficiency.

Many attempts were made to extend this basic approach. For example, the *improve_partition* procedure was modified to move one, two, or three CLBs in a single "swap." However, none of these attempts significantly improved the performance of the algorithm, and all increased its running time.

It is of note that the iterative improvement phase of this algorithm differs from that of traditional [KL] and [FM] type algorithms. Usually these strategies allow hill-climbing out of local minima as follows. At the beginning of each pass, every node is unlocked. As the pass proceeds, the algorithm iteratively selects, swaps and locks the module pair with the highest gain. Thus, in each iterative improvement pass, every module moves exactly once. If, at the end of the pass, any intermediate solution is an improvement over the solution at the beginning of the pass, the better solution is kept and the process repeats. During the iterative improvement phase of our algorithm, however, a swap is done only on pairs that allow immediate gain. This simplifies the iterative improvement phase, at the expense of eliminating the possibility of movement out of local minima. However, our experimental results have demonstrated that the technique is effective in this case, in terms of both solution quality and runtime, probably because the initial clustering itself is quite good.

Experimental Results

In spite of its simplicity, the heuristic we have discussed here for FPGA Minimization produces results that compare favorably with other known methods. For comparison purposes, tests were run over the standard partitioning benchmarks [Be], and results compared to those found by [CLCDL], [HK] and [KBK]. In each case, the circuits were technology mapped to an FPGA with a capacity of 64 CLBs and 58 I/Os.

The comparisons are tabularized in Table 3.2, in which the total number of FPGAs calculated for each circuit is given. For all but circuit c3540, we ran each test ten times, and selected the best partitioning result. Circuit c3540 was easy to partition into seven subsets, but we were only able to find a partitioning into six subsets on two of perhaps a thousand runs. It is difficult to compare runtimes, since these are unreported in most cases [CLCDL, HK, KBK]. However, our runtimes are very close to those that have been reported, but are not an improvement.

It should be noted that, although our simple heuristic produces results that compare favorably with other known heuristics, it does not outperform them, either in partition quality or runtime. In fact, two of the benchmark tests (c3540, c6288) cannot possibly be partitioned onto fewer FPGAs of capacity 64 CLBs. Therefore, no partitioner will ever demonstrate improvement in solution quality for these circuits in this configuration. An effort is presently underway to develop new benchmarks, that better represent current circuits [Al]. The heuristic we present here differs from many others in that it was motivated by the theoretical study of MDGP, but it may not necessarily represent advancement in circuit partitioning methods. One of its main purposes in this work is to serve as the first step in a two-step method for delay minimization, a topic we discuss subsequently.

Table 3.2: Partitioning results

circuit	(CLBs, IOBs, nets)	CLCDL	HK	KBK	ours
c3540	(373, 72, 569)	6	6	7	6
c5315	(535, 301, 936)	12	12	11	11
c7552	(611, 313, 1057)	11	11	11	11
c6288	(833, 64, 1472)	14	14	14	14

Chapter 4

Extending the Fundamental Problem: Delay Minimization

Although FPGA Minimization is a significant problem in the FPGA arena, there are other issues in addition to minimizing the number of chips utilized. One important concern is minimization of delay through the system.

4.1 Problem Definition

Recall that for the FPGA Minimization problem, the objective is to realize the system on as few chips as possible while satisfying constraints, in order to minimize cost. A Configurable Computing Machine (CCM) system is often composed of a fixed set of FPGAs, and may also incorporate memory, a CPU, and other components ([VM]). Since such a system has a predefined number of FPGAs already available, the "cost" of an implementation is the same regardless of the number of chips actually utilized.

Within a static system of FPGAs, an important issue in a partitioning solution is the delay through the system. In this section, we turn our attention to this new problem variation, which we call Delay Minimization.

Before presenting the formal definition of the problem, we introduce some new concepts.

A *topology graph* T_G is an undirected, simple graph that describes the connectivity of the FPGAs in a specific CCM system. The nodes of T_G are in one-to-one correspondence with the FPGAs in the system. For every pair of FPGAs that are directly connected in the CCM, there exists an edge between the nodes of T_G representing these FPGAs. There are no other edges.

A circuit instance contains primary inputs (PIs), primary outputs (POs) and CLBs. (Recall that PIs are the external inputs to the circuit, and POs are the external outputs.) Paths in a combinatorial circuit are acyclic, and flow from PIs to POs. Each such path begins with a PI, proceeds through a series of CLBs and ends with a PO. Each step in the path (from PI to CLB, from CLB to CLB, from CLB to PO) incurs a delay. The precise value of the delay depends upon the particular underlying hardware, although we may make some assumptions.

In the case of a step from CLB to CLB, the delay depends upon the partitioning and the topology. During partitioning, "virtual" CLBs of a circuit instance are assigned to "physical" CLBs of FPGAs. Each FPGA is represented by one node of the topology graph T_G. The delay incurred by a step from CLB A to CLB B depends upon where these two CLBs are located relative to each other after partitioning. Communication between two CLBs residing on the same FPGA chip will be less costly than that between two CLBs residing on different chips. Furthermore, communication between two CLBs residing on different chips will be less costly if those two chips are directly connected. We will use the terms *delta_local, delta_neighbor*, and *delta_global* to describe these three different delay values.

If CLBs A and B both lie on the same FPGA chip, the cost of a step from A to B is *delta_local*. If these CLBs are on different FPGAs, but the FPGAs are directly connected (as indicated by T_G), the cost of the step is *delta_neighbor*. The final possibility is that the CLBs are on different, unconnected FPGAs, in which case the cost of the delay is *delta_global*.

The actual values of these delays may vary from system to system. We can reasonably assume, however, that *delta_neighbor* is approximately ten times *delta_local*, and that *delta_global* is at least 50 percent greater than *delta_neighbor*. For our purposes, we will assume values of $delta\_local = 3$, $delta\_neighbor = 30$ and $delta\_global = 50$ [Bou]. We may also assume that the delay from a PI to a CLB, or from a CLB to a PO, is *delta_local* [Bou].

The *critical path* is the longest (in terms of delay) path from any PI to any PO, in the partitioned circuit. Given a partitioning P of some circuit instance represented by a DAG G, relative to some topology graph T_G, denote by $cp(P)$ the delay of the critical path of P. In G, any node with no incoming edges is a PI, and any node with no outgoing edges is a PO.

We now define Delay Minimization as follows.

Instance: a directed acyclic graph G, a simple, undirected graph T_G, and three integers k, d and t.

Question: Is there is a partition P of V_G into disjoint sets $V_1, \ldots, V_m$ such that

1. $m \leq |V_{T_G}|$,
2. $\forall i : |V_i| \leq k$,
3. if E_i is the set of edges with exactly one endpoint in V_i, $max_{1 \leq i \leq m} |E_i| \leq d$, and
4. $cp(P) \leq t$?

The complexity of Delay Minimization follows in a straightforward manner from that of FPGA Minimization.

Theorem 4.1 *Delay Minimization is $\mathcal{NP}$-complete.*

Proof An instance of FPGA Minimization could be solved by Delay Minimization as follows.

The FPGA Minimization instance consists of a graph G, and three integers k, d, and p. We form an instance of Delay Minimization consisting of G', T_G, k', d' and t.

To form G', we begin with G and then direct the edges as follows. For each edge $uv \in G'$, if we have already constructed in G' a directed path from u to v, then direct uv from u to v. If not, then direct uv from v to u. This can be done in polynomial time, and introduces no cycles in G'. Since G' is a DAG, it contains at least one source (PI) and one sink (PO).

T_G consists of a graph containing p isolated vertices. The longest possible delay through G' is *delta_local* (delay from some source node representing a PI to the first node representing a CLB in the critical path) $+(|V_{G'}|-3)\times$ *delta_global* + *delta_local* (delay from the last node representing a CLB in the critical path to some sink node representing a PO). Therefore, we set $t =$ *delta_local* $+(|V_{G'}|-3)\times$ *delta_global* + *delta_local* $+1$. Finally, $k' = k$ and $d' = d$.

The critical path of G' cannot exceed t in any partitioning satisfying the other constraints. Therefore, G, k, d, p is a "yes" instance of FPGA Minimization if and only if G', T_G, k, d, t is a "yes" instance of Delay Minimization. □

Analogously, it can be seen that Fixed-k, d Delay Minimization is $\mathcal{NP}$-complete, and that Delay Minimization, like FPGA Minimization, remains $\mathcal{NP}$-complete on many classes of graphs.

We conclude this subsection by addressing a discrepancy between our theoretical model and the circuit it represents. Earlier we discussed ways to accommodate, in our graphical model, PIs and POs for MDGP and the FPGA Minimization. Recall that these problems were stated in terms of undirected graphs. In the case of Delay Minimization, some vertices already represent PIs and POs. However, these nodes do not represent CLBs, and hence must not be partitioned into the same subsets as nodes representing CLBs.

The model can accommodate this requirement. To describe more accurately a real circuit instance, the graph representing the circuit could be augmented with chains containing $k-1$ nodes, one chain for every PI and PO node. Each edge in each chain is of multiplicity $d+1$, and a chain is connected to every PI and PO node, by an edge of multiplicity $d+1$. For PIs, all of these edges are directed toward the PI. For POs, all of these edges are directed away from the PO. Finally, the topology graph T_G is augmented with isolated vertices, one for each PI and PO. These "gadgets" force each PI and PO to lie in a unique subset. Critical path computation can be modified to accommodate these changes, which are described for theoretical purposes only.

4.2 A Practical Heuristic

Delay Minimization differs from the other problems we have considered so far, in that it is defined over directed graphs. The WQO theory that we have discussed earlier in this work no longer applies. Notions of closure under immersion order and obstruction sets are undefined with regard to this problem. For this reason, and because Delay Minimization is $\mathcal{NP}$-complete, we focus our efforts in this section toward the development of a new heuristic.

As was the case for FPGA Minimization, the heuristic we present here works on hypergraphs, since real circuit instances contain nets with more than two endpoints. Many of the example circuits that follow contain such nets.

4.2.1 Circuit Characteristics

Consider again the circuit depicted in Figure 3.5, and reproduced here in Figure 4.1 for clarity.

This circuit has five paths. Suppose this circuit is to be implemented on a system with a K_3 topology (three completely-connected FPGAs). Recall that we assume delay penalties of 3 nanoseconds for *delta_local* and 30 nanoseconds for *delta_neighbor*. Some possible mappings and resulting delays are presented in Table 4.1. (The notation $A:1$ denotes that CLB A is mapped to FPGA 1.)

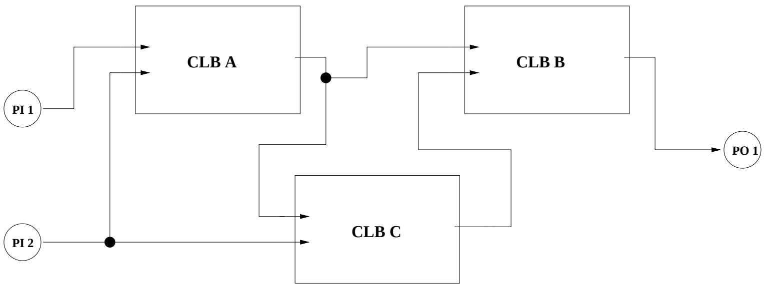


Figure 4.1: An example circuit

Table 4.1: Possible mappings for circuit 3.5

	A:1 B:2 C:3	A:1 B:1 C:2	A:1 B:2 C:1	A:1 B:1 C:1
$PI1 \rightarrow A \rightarrow B \rightarrow PO1$	36	9	36	9
$PI1 \rightarrow A \rightarrow C \rightarrow B \rightarrow PO1$	66	66	39	12
$PI2 \rightarrow A \rightarrow B \rightarrow PO1$	36	9	36	9
$PI2 \rightarrow A \rightarrow C \rightarrow B \rightarrow PO1$	66	66	39	12
$PI2 \rightarrow C \rightarrow B \rightarrow PO1$	36	36	36	9
System Delay	66	66	39	12

In this particular example, it is always better to use fewer FPGAs, but this is not necessarily true, as we now demonstrate.

Assuming $k = 9, d = 4$, Figure 4.2(a) illustrates a circuit whose delay is shorter on three chips than two. With these parameters, there is only one way to partition this circuit onto two chips. This two-chip partitioning is shown (in dotted lines) in Figure 4.2(b). Because every chip crossing has delay either *delta_neighbor* or *delta_global*, the path of longest delay is that which goes through the double-circled node and makes two chip crossings. Figure 4.2(c) shows a partitioning of this circuit onto three chips, in which no path makes more than one chip crossing.

4.2.2 Prior Work

Performance-driven partitioning is a relatively new research area, but it is already a topic of strong interest. See [KS], [NS], [RW], [ST], [TSO] and [We] for a sampling of various approaches to the problem. The problem formulations and objective functions often differ significantly from researcher to researcher, as do the solution approaches. Sometimes device constraints are considered; sometimes not. Sometimes replication is utilized; sometimes not. Sometimes the focus is on clustering for delay minimization. Sometimes partitioning is done before technology mapping, although this tends to be the exception. Because the ways of defining the problem are so various, it is difficult to compare directly the merits of one method to another.

Comparing results is further complicated by the fact that, although partitioning benchmarks exist, there are no standard timing constraints for these benchmarks. Therefore, the objective timing function is defined in various ways, which depend heavily upon other problem parameters specific to a particular technique.

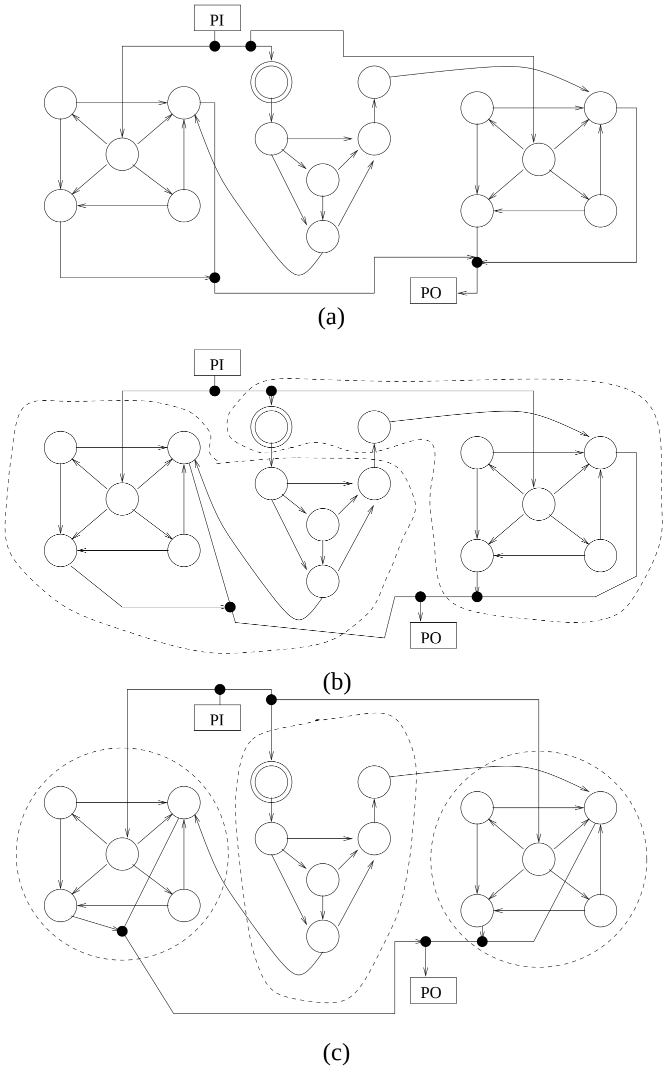


Figure 4.2: A Delay Minimization example

In this work, we focus on the development of an iterative improvement delay optimization strategy that works in conjunction with an independent partitioning step. For this reason, we refer to our approach as the *two-step* method [La2]. The first step is partitioning, and the second step consists of assigning the partitioned subsets to physical FPGAs and performing iterative timing improvement.

4.2.3 A New Approach - The "Two-Step" Method

As mentioned in the previous section, many timing heuristics operate by incorporating delay considerations into a partitioning heuristic. Such approaches have merit; however, our two-step method is different in that it performs timing optimization as an independent step after partitioning.

There are several advantages to this approach:

1. Since the system has already been partitioned, the complexity of the timing step is reduced.

2. If a good partitioning of a system is already known, this can be used as a starting point for timing optimization.

3. Our post-partitioning timing heuristic can be viewed as an independent iterative-improvement step which could be applied as a post-processing step after any other timing heuristic.

The first step of the two-step method, that of partitioning, can be performed by the partitioner of choice. For our tests, we used our own partitioner, but any partitioner may be used.

The second step consists of two parts. In the first part, "virtual" FPGAs (the subsets of the partitioning) are assigned to physical FPGAs. The second part consists of an iterative improvement algorithm to improve the delay through the system, by moving CLBs of the circuit graph from one subset (hence FPGA) to another. When a virtual FPGA is assigned to a physical one, virtual CLBs (the CLBs of the circuit graph) are also assigned to physical CLBs. In the discussion that follows, the term CLB is used to refer either to a virtual or a physical CLB when the context makes the meaning clear.

In the first part of the second step of the two-step method, virtual subsets of the partitioned circuit are assigned to physical FPGAs, represented by the topology graph. Initially, this was done in a greedy fashion, as follows. Each virtual subset is assigned to the available FPGA which results in the fewest number of nets requiring global communication. This approach seemed to have little effect on the critical path, however, at the expense of some computation time.

An experiment was done using one circuit and a partitioning of that circuit into five subsets. Every possible way of mapping those five subsets onto adjacent FPGAs in a linear array topology was examined. For each of these mappings, we compared the initial delay with the final delay after performing the delay optimization heuristic. There seemed to be no discernible pattern at all. Those mappings with the best final delay were not associated with those having the best initial delay. For each of these mappings, we also counted the number of nets containing at least two CLBs mapped to non-adjacent FPGAs. The number of such nets varied very little from mapping to mapping, ranging from a low of 630 to a high of 650. Again, there was no correspondence with final outcome.

Based on the above experiment, the decision was made to perform the virtual-to-physical FPGA mapping arbitrarily, with one exception. Whenever possible, connected physical FPGAs are utilized, in order to eliminate artificial "improvement" induced by an obviously inefficient initial placement. For example, if only 4 non-adjacent FPGAs in a 16-FPGA system are utilized, a large improvement would be seen by simply re-assigning the subsets to adjacent chips. We avoid this "artificial" improvement by initially choosing, as much as possible, FPGAs that can directly communicate.

The focus of the discussion for the remainder of this chapter is on the second part of the second step of the two-step method: the iterative improvement algorithm.

4.2.4 An Iterative Improvement Algorithm for Improving Delay in a Partitioned Circuit

The second step of the two-step method begins with a technology-mapped circuit, and a partitioning of that circuit, as input. The technology-mapped circuit is represented by the netlist file produced by the technology mapping software found in [Be], used without modification. The partitioning is represented by a file that enumerates the subsets of the partitioning, and the specific CLBs contained in each subset.

The topology of the FPGA system is also part of the program input, and includes the number of FPGAs available, and their connectivity. This information dictates the delay values (*delta_local, delta_neighbor* or *delta_global*).

The Connection Graph

A directed acyclic graph (DAG) called a *connection graph* ([WKMKY]) is constructed from the technology-mapped circuit. If each CLB computes one function, the connection graph is formed as follows. A node is created for each PI and each PO, and a node is created for each CLB. If a PI is connected to a CLB, a corresponding directed edge is placed into the connection graph. Similarly, directed edges are added to represent connections from CLBs to POs, and from the output of one CLB to the input of another.

The connection graph corresponding to the circuit of Figure 4.1 is shown in Figure 4.3.

Many FPGAs contain CLBs that can implement two functions ([X]). In this case, the connection graph must contain a node for each CLB function. We illustrate this situation.

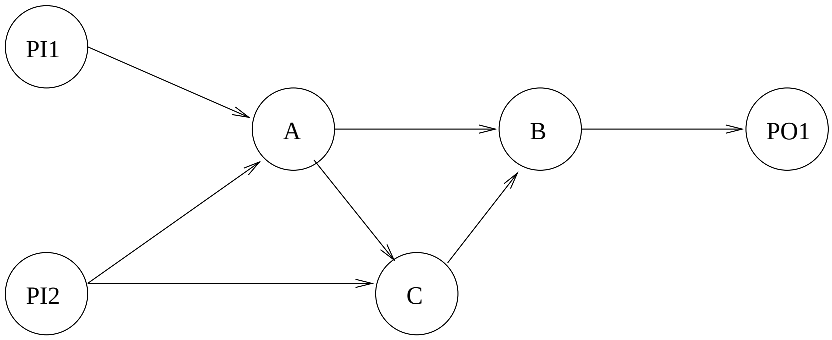


Figure 4.3: A connection graph

Figure 4.4 shows a technology-mapped circuit containing two CLBs, and Figure 4.5 shows the corresponding connection graph. Figure 4.6 illustrates the situation if such CLBs are not separated into two nodes. The resulting graph is no longer a DAG. Additionally, information is lost, for example the fact that PI1 serves as input to CLB A function1 but not to CLB A function2. In this situation, it is not possible to determine the critical path.

After the mapping from virtual to physical FPGAs has been performed, the edges of the connection graph are weighted with delays corresponding to communication costs. Edges to POs or from PIs are given a delay of *delta_local.* Each edge delay, then, is either *delta_local, delta_neighbor*, or *delta_global.*

For example, suppose the circuit of Figure 4.4 were implemented on a system of two connected FPGAs. Furthermore, assume *delta_local* = 3 nanoseconds and *delta_neighbor* = 30 nanoseconds. If CLB A were mapped to FPGA1, and CLB B to FPGA2, the edge delays on the connection graph would be as depicted in Figure 4.7.

Finding the Critical Path

From the connection graph with edge delays, we compute the maximum delay through the system. Since we wish to find the longest (in terms of delay) path from any input to any output, we augment the connection graph with two special nodes v_s and v_t.

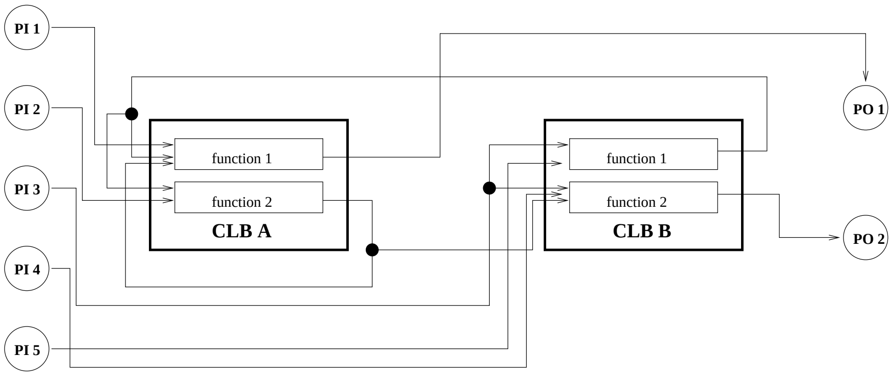


Figure 4.4: Two-function CLBs

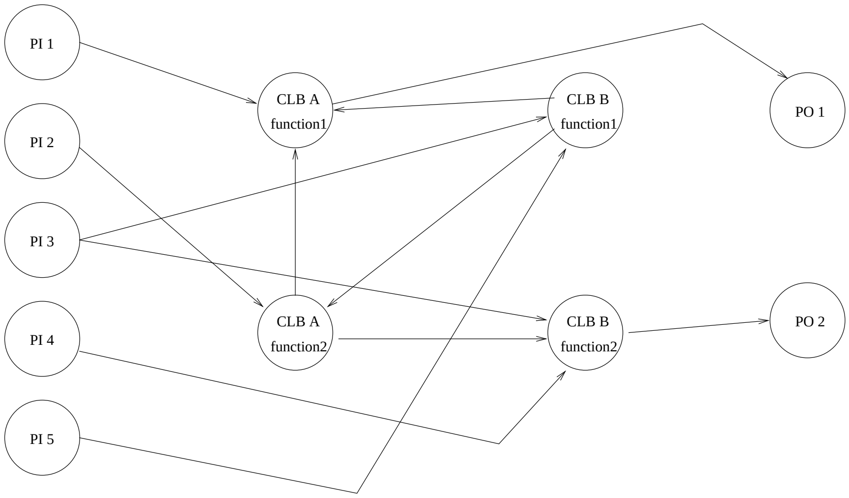


Figure 4.5: A connection graph for two-function CLBs

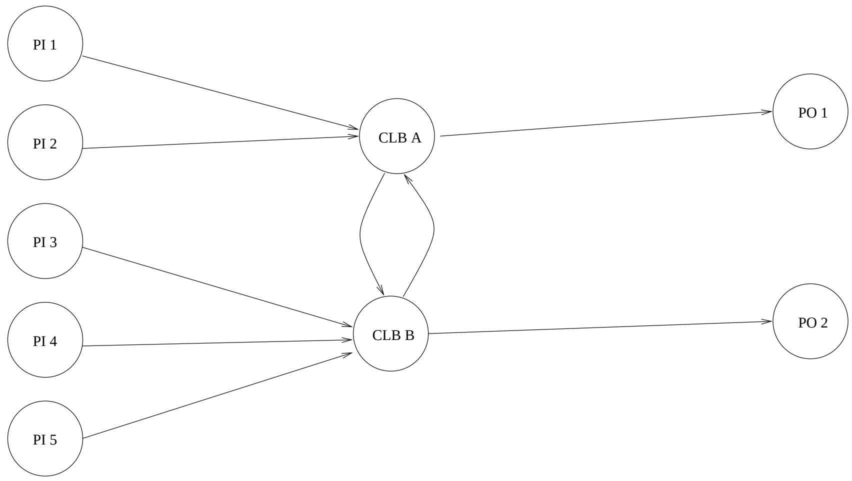


Figure 4.6: A cyclic connection graph

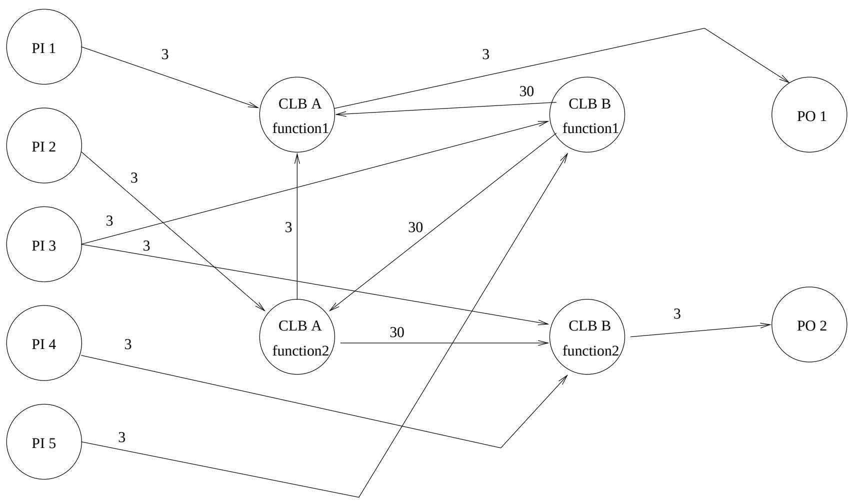


Figure 4.7: A connection graph with edge delays

The node v_s is of in-degree 0 and is connected by a directed out-edge of delay 0 to every PI node. The node v_t is of out-degree 0, and every PO node has a directed out-edge of delay 0 to v_t. Now, to find the delay through the system, we simply compute the longest (in terms of delay) path from v_s to v_t in the directed, acyclic connection graph.

The Longest Path problem is known to be $\mathcal{NP}$-complete for general graphs, but in $\mathcal{P}$ for DAGs [GJ]. In [WKMKY], the longest weighted path from v_s to v_t in the connection graph is computed with a breadth-first search. Such an approach suffices for DAGs, and was initially utilized in our heuristic. However, breadth-first search does not necessarily observe topological sort[1] order, which can cause nodes to be processed multiple times.

Consider the DAG of Figure 4.8, in which each edge is of delay 1. When processing v_s, suppose node x (longest path delay from v_s of 1) is pushed onto the stack first, followed by node y (longest path delay from v_s of 1).

Because y is at the top of the stack, it is processed next, and node z is assigned a current longest path delay of 2. When x is removed from the stack, its neighbor y has its longest path delay increased to 2, so must be pushed onto the stack again. Similarly, when processing y, the longest path delay of z increases again.

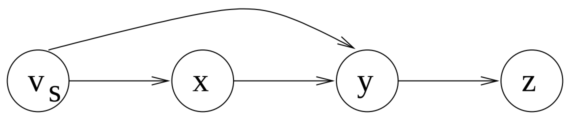


Figure 4.8: A DAG with edge delays

[1] A *topological sort* of the nodes of a DAG is the operation of arranging the nodes in order in such a way that if there exists an edge (i, j), then i precedes j in the list. [BB]

An efficient alternative exists that does not use a stack, and guarantees that each edge will be processed exactly once. First, the nodes of the DAG are sorted into topological order. (This can be done in $O(n)$ time using a depth-first search [BB], and only needs to be done once.) The longest path to each node is initialized to 0. Then, for each node u (proceeding in topological order), the longest path to each neighbor v of u is updated if the longest path to u plus the delay of edge uv exceeds the current longest path to v. When processing each u, it is evident that the current longest path to u is at its maximum, since all vertices with edges to u have already been processed.

Critical path is re-computed extremely often in our heuristic. By using the more efficient topological sort technique rather than breadth-first search to compute critical path, we were able to greatly improve the efficiency of our heuristic. Experimental results pertaining to this will be presented in Section 4.2.4.

Iterative Improvement Overview

Iterative improvement strategies for partitioning usually operate by swapping pairs of modules (i.e., CLBs), or moving a single module from subset to subset. Such approaches were attempted for delay optimization in this research, and poor results were achieved.

The difficulty with these approaches is that simple pairwise swaps, or single module movement, seldom produce improvement in the delay of the critical path. Therefore, we have developed an iterative improvement scheme that we call *critical path compression*. Critical path compression works by reassigning *groups* of CLBs, specifically chosen for potential delay improvement, from one FPGA to another. This scheme proceeds as follows.

At any given moment, there exists a critical path in the connection graph (ties may be broken arbitrarily). We will refer to this specific path as Π. If the number of chip crossings in Π can be reduced, by reassigning CLBs from Π to different FPGAs, the result will be a decrease in the delay of Π. It could be the case that the reduced-delay Π is still the critical path. However, it also may be the case that some different path, Π', is now the critical path. In fact, it may be the case that the delay of Π' is bigger than the original delay of Π.

Figure 4.9 shows an example in which compressing Π produces a new, worse critical path. In (a), module a is in one FPGA, modules b, c, e and f are in a second FPGA, and module d is in a third FPGA. If all FPGAs are connected, $delta\_local = 3$ and $delta\_neighbor = 30$, the critical path is $PI \to a \to b \to c \to d \to PO$ and is of delay 69. Moving module b from the second FPGA to the first, and moving module c from the second FPGA to the third, reduces the delay of path $PI \to a \to b \to c \to d \to PO$ to 42, but produces a new critical path, $PI \to e \to b \to c \to f \to PO$ of delay 96.

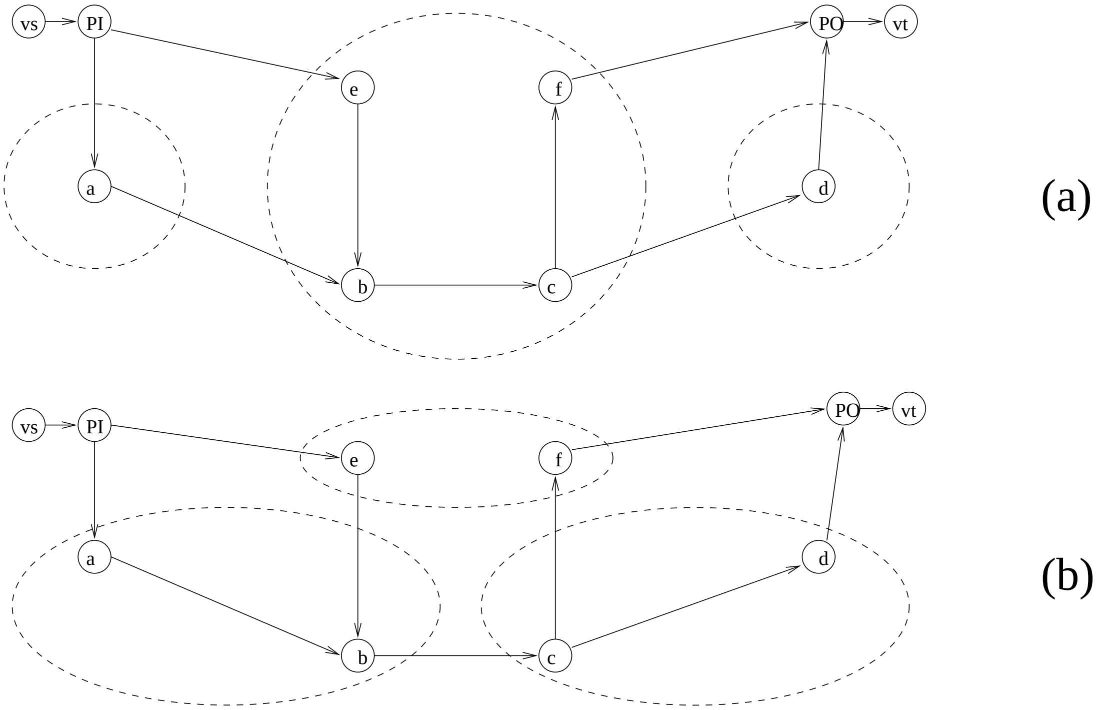


Figure 4.9: Compressing a critical path

It is infeasible, of course, to try all possible ways of re-assigning the CLBs from Π to FPGAs. Therefore, we have devised specific patterns in the critical path for which to search. Specifically, we look for patterns in the FPGAs to which consecutive CLBs in the critical path are assigned.

For example, in the example of Figure 4.9, we saw the following pattern in the critical path. A CLB (a) was assigned to some FPGA x, some set S of CLBs following a (b and c) were assigned to some FPGA $y \neq x$, and the CLB following S (d) was assigned to some FPGA $z \neq y$. In any such pattern, the number of chip crossings is reduced if some initial segment of S is reassigned to FPGA x, and the rest of S assigned to FPGA z.

There are five distinct types of patterns for which we search. We refer to these five strategies as critical path compression techniques. They are called 1) Elimination I, 2) Elimination II, 3) Substitution I, 4) Substitution II, and 5) Resequencing. Each of these critical path compression techniques will be described later.

Each of these five techniques can be activated or de-activated. This enabled us to test each for effectiveness individually, and in combination with others. The user then has the flexibility of choosing which of the techniques to utilize, and may choose to deactivate those that have a higher running time and/or smaller potential for gain. More will be said about this in the section describing experimental results.

The iterative improvement algorithm operates by examining Π, and sequentially attempts each of the activated critical path compression techniques. With each technique, Π is examined for the existence of some specific pattern of assignments of CLBs to FPGAs. For each occurrence of this pattern, the following is done.

The CLBs are reassigned to FPGAs, in the manner dictated by the critical path compression technique in effect at the moment. This reassignment requires that edge delays in the connection graph be modified to reflect the new placement of the CLBs. Then Π is recomputed, and its delay recorded. Finally, the changes are undone, and the next occurrence of the pattern is processed.

It is possible that no matching patterns are found, in which case the iterative improvement algorithm terminates.

It is also possible that matching patterns are found, but for each attempt, either constraints are violated, or the new critical path is of delay no better than that of the original. In this situation, a local minimum has been reached, and the algorithm terminates. (The topic of escaping local minima will be discussed in the next subsection.)

The final possibility is that at least one of the reassignments produced a critical path of delay shorter than the original. In that case, this reassignment is applied again and maintained. The new critical path is again designated as Π (this path may or may not be the same as the original Π), and the iterative improvement algorithm repeats.

Because the algorithm only repeats if the delay of the critical path is reduced, the algorithm eventually terminates.

Because each critical path compression technique is independently coded, and so can be run independently, new critical path compression strategies can easily be incorporated into the existing code. The overall idea is somewhat similar to that of peephole optimization, a technique for optimization of compiler output. In this strategy, a "peephole" is passed along the code stream, and the code within the peephole is examined for the existence of certain patterns. If the pattern is found, an appropriate substitution is attempted, and retained if it can be successfully implemented.

Strategies for Escaping Local Minima

A standard consideration in iterative improvement algorithms is escaping local minima. If the algorithm only implements changes that result in improvement, it can never effect a potentially greater improvement that requires an "uphill move:" going through an intermediate, worse solution.

Many traditional swapping algorithms (e.g. [FM, KL]) enable uphill moves in the following way. A module is "locked" after it has been moved from one subset to another. At the beginning of each "run," all modules are unlocked. Until all modules are locked, all pairs of unlocked modules are examined, and the pair with the greatest gain is selected, swapped and locked. Note that this gain may indeed be negative in terms of overall solution, effecting an uphill move. After all modules are locked, the best intermediate solution is chosen. If this solution is better than what existed at the beginning of the run, it is kept and another run is performed. Note that this strategy requires that all modules be moved in each pass.

Our critical path compression techniques move sets of modules, not always pairs. Additionally, they work by examining the critical path only, and any module movement may result in the formation of a completely different critical path, with different candidate CLBs. If each module were locked after being moved, any group of modules containing a locked module could not be considered for movement. For this reason, the locking mechanism does not provide enough flexibility for our purposes. Therefore, to enable uphill moves, we employ the following strategy.

A user-defined value, *look_ahead*, is obtained. This value must be an integer at least 1, and is typically a small value. (The effects of various precise values will be examined in Section 4.2.4.)

The following is then done, for each activated critical path compression technique. A counter is initialized to *look_ahead*. The smallest critical path seen is recorded. This is initialized to the value of the current critical path, and updated any time a smaller critical path is found. Recall that for each critical path compression technique, some specific pattern of assignments of CLBs to FPGAs is sought. This pattern may occur many times in Π, and, depending upon the technique at hand, may involve several different ways of reassigning CLBs to FPGAs. Each possibility is attempted, and the one which produces the largest decrease in critical path is noted. (Note that this decrease may in fact be negative, if the reassignment increases the delay of the critical path.) If the counter is 1, processing terminates for this critical path technique, and the configuration that produced the best critical path is applied and maintained. Otherwise, the counter is decremented, and the set of module movements that produces the largest decrease in critical path is implemented, even if the result is an increase in the critical path delay. The current critical path compression technique is applied again. In this way, it is sometimes possible to find a better overall solution, that could not have been found without going through the intermediate, worse solution.

It is noteworthy that, without locking modules, it is possible for a looping situation to develop. However, if this does happen, it will terminate when the number of iterations is done. Experimental results with differing values of *look_ahead* will be presented later.

Critical Path Compression Techniques

We now describe the five different strategies for critical path compression: the process of compressing the delay of the critical path, Π, through a partitioned circuit.

The algorithm considers only the sequence of CLBs in Π. This path is searched for every occurrence of a particular pattern of assigned FPGAs in consecutive CLBs. The precise definition of the pattern depends upon which of the five strategies is under consideration. In this subsection, we establish a common framework for all of these strategies.

Each strategy considers only the current critical path, Π. In each strategy, Π is searched for some segment of consecutive CLBs matching some pattern of assignment to FPGAs. Returning again to the example of Figure 4.9, we saw the following pattern in the critical path. A single CLB (a) was assigned to some FPGA x; some set S of n CLBs (in this case $n = 2$) following a (b and c) were assigned to some FPGA $y \neq x$; and the single CLB following S (d) was assigned to some FPGA $z \neq y$. In this setting, then, the pattern sought is a set of n consecutive CLBs assigned to the same FPGA, such that the CLBs immediately preceding and immediately following the set are assigned to different FPGAs. The set of n consecutive CLBs is referred to as the "target sequence" because these are the CLBs that we will attempt to reassign to different FPGAs. In this particular example, we require the target sequence to consist of consecutive CLBs assigned to the same FPGA, but this will vary from strategy to strategy. In each of the strategies, the target sequence is also defined in terms of the CLBs immediately preceding and immediately following. We will refer to the "extended target sequence" as the target sequence, along with the single CLB immediately preceding the target sequence, and the single CLB immediately following the target sequence.

Definition 4.1 *Denote by $t_1, t_2, \ldots, t_n$ the set of n CLBs of a target sequence. Denote by x the CLB immediately preceding a target sequence. Denote by y the CLB immediately following a target sequence.*

We note that either of x, y may be a "dummy CLB" if the target sequence is at the beginning or the end of Π.

A representative snapshot of an extended target sequence is illustrated in Figure 4.10.

Definition 4.2 *Given CLB c, denote by $f(c)$ the index of the FPGA to which c is currently assigned. If CLB c is a "dummy CLB," then $f(c) = 0$.*

Definition 4.3 *Given FPGA indices $i \geq 0$ and $j \geq 0$, denote by $d(i, j)$ the communication delay between the FPGAs with these indices. If either $i = 0$ or $j = 0$, $d(i, j) = 0$.*

We note that $\forall i, j \; d(i, j) \in \{0,$ *delta_local, delta_neighbor, delta_global*$\}$.

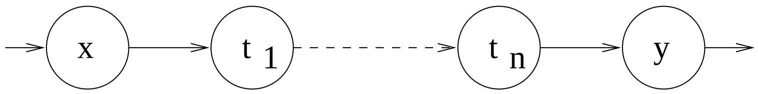


Figure 4.10: An extended target sequence

Elimination I

The first critical path compression strategy we describe is called *Elimination I.* We define its extended target sequence as follows.

Definition 4.4 *An Elimination I extended target sequence is one such that:*

- $f(t_1) = f(t_i) \forall i \in [1, n]$,
- $f(x) \neq f(t_1)$,
- $f(y) \neq f(t_1)$ *and*
- *either* $f(x) \neq 0$ *or* $f(y) \neq 0$.

For example, suppose that Π, in its entirety, consists of six CLBs, assigned to FPGAs 1, 2, and 3, as shown in Figure 4.11.

There first two extended target sequences are:

1. x is the "dummy CLB," $f(x) = 0$, $n = 1$, $t_1 =$ CLB 1, $f(t_1) = 2$, $y =$ CLB 2, $f(y) = 3$.
2. $x =$ CLB 1, $f(x) = 2$, $n = 2$, $t_1 =$ CLB 2, $t_2 =$ CLB 3, $f(t_1) = 3$, $y =$ CLB 4, $f(y) = 1$.

We can now define Elimination I:

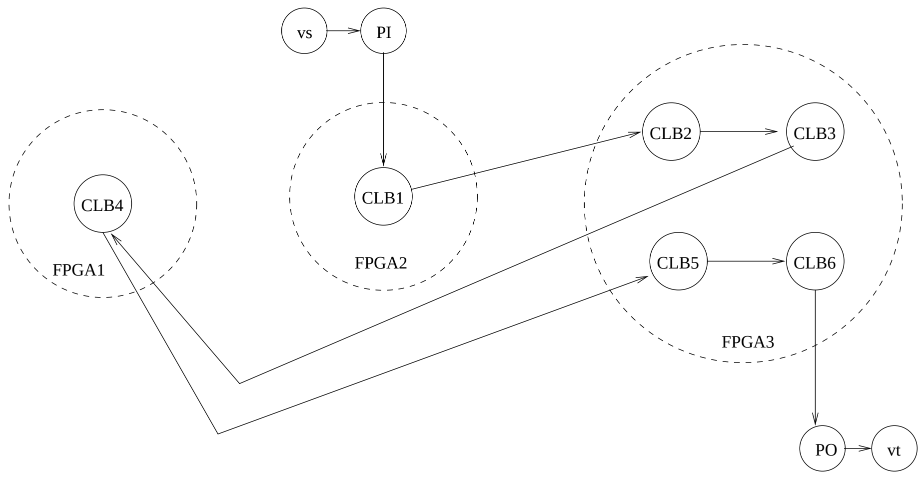


Figure 4.11: A critical path

Definition 4.5 *Elimination I is the assignment, for some $p+q=n$, of the first p CLBs of an Elimination I target sequence to the FPGA indexed by $f(x)$, and the last q CLBs of the sequence to the FPGA indexed by $f(y)$, if appropriate FPGAs exist, and changes can be made without violating size or pincount constraints.*

Appropriate FPGAs do not exist if, for example, x is the "dummy CLB" and $p > 0$. We reiterate that it is possible for one, but only one, of x, y to be the "dummy CLB."

Prior to Elimination I, the $n+2$ CLBs in the extended target sequence are assigned to the following FPGAs, in the following order:

$$1 : f(x), 2 : f(t_1), \ldots, (n+1) : f(t_1), (n+2) : f(y).$$

The delay of this sequence is

$$d(f(x), f(t_1)) + ((n-1) \times delta\_local) + d(f(t_1), f(y)).$$

After Elimination I, the sequence of assigned FPGAs becomes:

$$1 : f(x), 2 : f(x), \ldots, (p+1) : f(x), (p+2) : f(y), \ldots, (p+q+2) : f(y).$$

The delay of this sequence is

$$(p \times delta\_local) + d(f(x), f(y)) + (q \times delta\_local) = (n \times delta\_local) + d(f(x), f(y)).$$

We may then compute the change in delay of critical path Π (d_c) as follows:

$$d_c = d(f(x), f(y)) + delta\_local - d(f(x), f(t_1)) - d(f(t_1), f(y))$$

Figures 4.12, 4.13, and 4.14 illustrate the effects of Elimination I in a specific example. Assume $k = 2$ and $d = 5$. Assume also that there is direct communication between FPGAs 1 and 2, and between FPGAs 2 and 3, but that FPGAs 1 and 3 must communication by means of a global bus.

In Figure 4.12, $\Pi = PI \rightarrow A \rightarrow C \rightarrow D \rightarrow PO1$, and is of delay 86. The first extended target sequence for Elimination I consists of a dummy x, $t_1 = A$, $y = C$. Moving CLB A into FPGA $f(y) = 3$ is not possible, because this would violate k. The next extended target sequence is $x = A$, $t_1 = C$, $y = D$. Again, it is not possible to move CLB C from FPGA 3 into FPGA $f(x) = 2$ because of constraint violation. It is, however, possible to move CLB C into FPGA $f(y) = 1$, as shown in Figure 4.13. The delay of the original Π is reduced to 39. The current Π is now $PI \rightarrow A \rightarrow B \rightarrow E \rightarrow PO2$, and is of delay 66.

Elimination I is attempted again on the new Π. The first extended target sequence consists of a dummy x, $t_1 = A$, $y = B$. CLB A can be moved into FPGA $f(y) = 3$, as shown in Figure 4.14. The delay of Π is now 39, which cannot be improved.

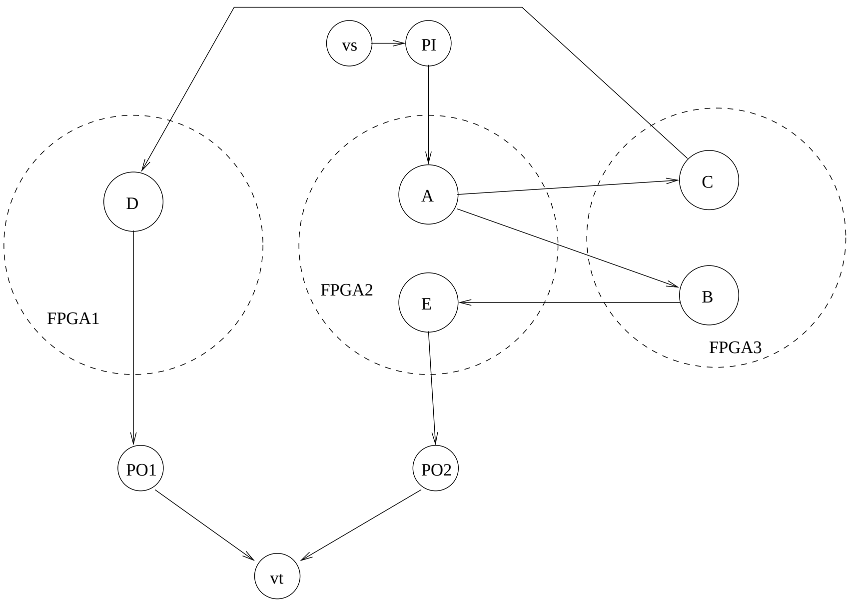


Figure 4.12: Elimination I example: part 1

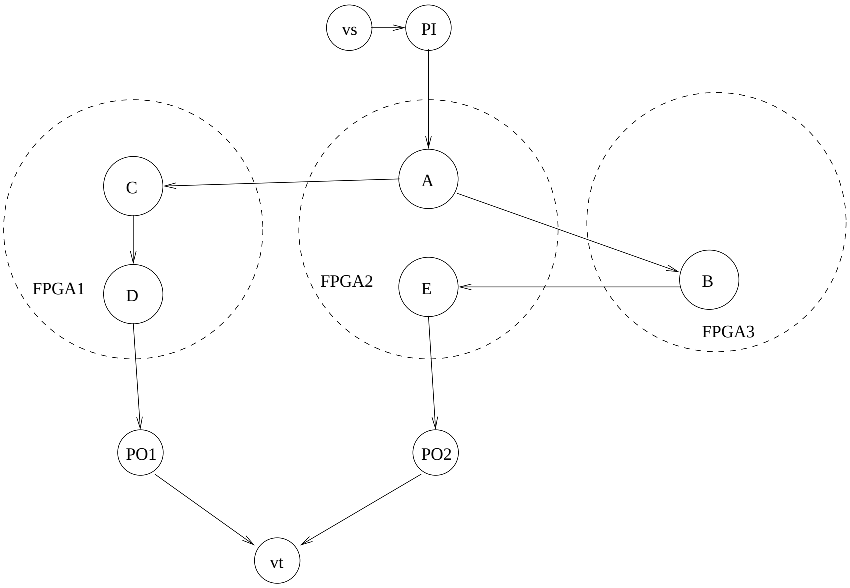


Figure 4.13: Elimination I example: part 2

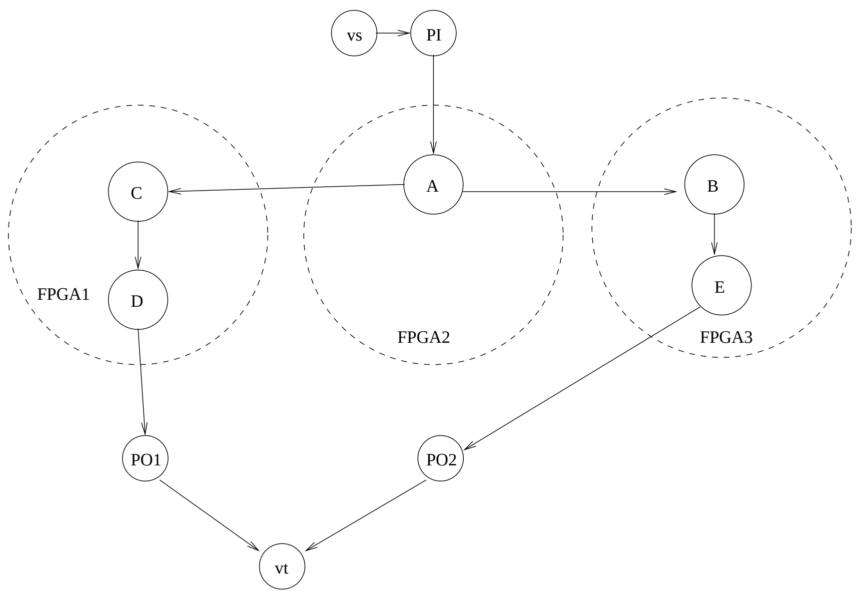


Figure 4.14: Elimination I example: part 3

Can Elimination I increase the delay in Π? If this is the case, then we have

$$d(f(x), f(y)) + delta\_local > d(f(x), f(t_1)) + d(f(t_1), f(y))$$

Table 4.2 illustrates the possibilities. In the table (and others that follow) d_l indicates *delta_local*, d_n indicates *delta_neighbor*, and d_g indicates *delta_global*. The only situation in which the delay of Π increases is when $d(f(x), f(y)) = delta\_global$, $d(f(x), f(t_1)) = delta\_neighbor$, $d(f(t_1), f(y)) = delta\_neighbor$, and $delta\_global \geq (2\times\ delta\_neighbor) - delta\_local$. Using our assumed values of 50 for *delta_global*, 30 for *delta_neighbor*, and 3 for *delta_local*, Elimination I always results in a decrease in the delay of Π.

Recall that it is possible that a reduction in the delay of Π results in the production of a new critical path of even larger delay. Such a move would actually be implemented only if uphill moves have been activated. This applies to all of the critical path compression strategies.

Table 4.2: Effect of Elimination I

$d(f(x), f(y))$	$d(f(x), f(t_1))$	$d(f(t_1), f(y))$	Path delay effect
0	0	d_n	decrease
0	0	d_g	decrease
0	d_n	0	decrease
0	d_g	0	decrease
d_l	d_n	d_n	decrease
d_l	d_g	d_n	decrease
d_l	d_n	d_g	decrease
d_l	d_g	d_g	decrease
d_g	d_n	d_n	decrease if $d_g < (2 \times d_n) - d_l$
d_g	d_g	d_n	decrease
d_g	d_n	d_g	decrease
d_g	d_g	d_g	decrease
d_n	d_n	d_n	decrease
d_n	d_g	d_n	decrease
d_n	d_n	d_g	decrease
d_n	d_g	d_g	decrease

Elimination II

Elimination II is very similar to Elimination I. The only difference is in the definition of the target sequence, which now requires the existence of at least two CLBs assigned to different FPGAs.

Definition 4.6 *An Elimination II extended target sequence is one such that:*

- $\exists i \in [1, n] | f(t_1) \neq f(t_i)$,
- $f(x) \neq f(t_1)$,
- $f(x) \neq f(t_i)$,
- $f(y) \neq f(t_1)$,
- $f(y) \neq f(t_i)$,
- $\forall j \in [1, n] f(t_j) \in \{f(t_1), f(t_i)\}$, *and*

- *either $f(x) \neq 0$ or $f(y) \neq 0$.*

Referring again to Figure 4.11, the two extended target sequences are:

1. x is the "dummy CLB," $f(x) = 0$, $n = 3$, t_1 = CLB 1, t_2 = CLB 2, t_3 = CLB 3, $i = 2$, $f(t_1) = 2$, $f(t_i) = 3$, y = CLB 4, $f(y) = 1$.

2. x = CLB 1, $f(x) = 2$, $n = 3$, t_1 = CLB 2, t_2 = CLB 3, t_3 = CLB 4, $i = 4$, $f(t_1) = 3$, $f(t_i) = 1$, y = CLB 5, $f(y) = 3$.

We define Elimination II:

Definition 4.7 *Elimination II is the assignment, for some $p + q = n$, of the first p CLBs of an Elimination II target sequence to the FPGA indexed by $f(x)$, and the last q CLBs of the sequence to the FPGA indexed by $f(y)$, if appropriate FPGAs exist, and changes can be made without violating size or pincount constraints.*

Elimination II has even greater potential than Elimination I for reducing delay in the current critical path, because at least one additional chip crossing (that between the FPGAs indexed by $f(t_1)$ and $f(t_i)$) is always eliminated.

Substitution I

Substitution I applies a different method to an Elimination I extended target sequence. Rather than assigning the CLBs of the target sequence to the FPGAs indexed by $f(x)$ or $f(y)$, an attempt is made to assign them to a completely different FPGA. As was the case for Elimination I, it is required that at least one of $f(x), f(y)$ be nonzero. If both $f(x), f(y)$ were zero, that would mean that the target sequence encompasses the entirety of Π, and that all CLBs in Π are assigned to the same FPGA. Reassigning all of Π to a different FPGA would have no effect on the delay of Π.

Definition 4.8 *Substitution I is the assignment of all n CLBs of an Elimination I target sequence to some FPGA indexed by z, such that $z \notin \{f(x), f(t_1), f(y)\}$, if changes can be made without violating size or pincount constraints.*

In contrast to Elimination I, where an attempt is made to move the target sequence to one or two different FPGAs ($f(x)$ and/or $f(y)$), in Substitution I attempts are made to move the target sequence to any FPGA other than $f(x)$, $f(t_1)$ or $f(y)$.

Prior to Substitution I, the $n+2$ CLBs in the extended target sequence are assigned to the following FPGAs, in the following order:

$$1 : f(x), 2 : f(t_1), \ldots, (n+1) : f(t_1), (n+2) : f(y).$$

The delay of this sequence is

$$d(f(x), f(t_1)) + ((n-1) \times delta\_local) + d(f(t_1), f(y)).$$

After Elimination I, the sequence of assigned FPGAs becomes:

$$1 : f(x), 2 : z, \ldots, (n+1) : z, (n+2) : f(y).$$

The delay of this sequence is

$$d(f(x), z) + ((n-1) \times delta\_local) + d(z, f(y)).$$

We may then compute the change in delay of critical path Π (d_c) as follows:

$$d_c = d(f(x), z) + d(z, f(y)) - d(f(x), f(t_1)) - d(f(t_1), f(y))$$

The delay of Π decreases with Substitution I only if

$$d(f(x), z) + d(z, f(y)) < d(f(x), f(t_1)) + d(f(t_1), f(y))$$

Table 4.3 summarizes all of the possibilities.

Table 4.3: Effect of Substitution I

$d(f(x), z)$	$d(z, f(y))$	$d(f(x), f(t_1))$	$d(f(t_1), f(y))$	Path delay effect
0	d_n	0	d_n	no change
0	d_n	0	d_g	decrease
0	d_g	0	d_n	increase
0	d_g	0	d_g	no change
d_n	0	d_n	0	no change
d_n	0	d_g	0	decrease
d_n	d_n	d_n	d_n	no change
d_n	d_n	d_n	d_g	decrease
d_n	d_n	d_g	d_n	decrease
d_n	d_n	d_g	d_g	decrease
d_n	d_g	d_n	d_n	increase
d_n	d_g	d_n	d_g	no change
d_n	d_g	d_g	d_n	no change
d_n	d_g	d_g	d_g	decrease
d_g	0	d_n	0	increase
d_g	0	d_g	0	no change
d_g	d_n	d_n	d_n	increase
d_g	d_n	d_n	d_g	no change
d_g	d_n	d_g	d_n	no change
d_g	d_n	d_g	d_g	decrease
d_g	d_g	d_n	d_n	increase
d_g	d_g	d_n	d_g	increase
d_g	d_g	d_g	d_n	increase
d_g	d_g	d_g	d_g	no change

It is noteworthy that the actual values of *delta_local, delta_neighbor* and *delta_global* do not affect whether or not Substitution I increases or decreases the delay of Π. These values do, of course, affect the amount of increase or decrease.

Substitution II

Substitution II is very similar to Substitution I, in that an attempt is made to reassign all of the FPGAs of the target sequence to a different FPGA. The CLBs in the target sequence must be initially assigned to two FPGAs rather than one, however. A Substitution II extended target sequence is the same as an Elimination II extended target sequence, except that both x and y may be the "dummy CLB." If $f(x) = 0$ and $f(y) = 0$ (the target sequence encompasses all of Π), it would decrease the delay of Π to reassign all of these CLBs to some other FPGA, if they are all reassigned to the same FPGA.

Definition 4.9 *Substitution II is the assignment of all n CLBs of a Substitution II target sequence to any FPGA z, such that $z \notin \{f(x), f(t_1), f(t_i), f(y)\}$, if changes can be made without violating size or pincount constraints.*

Substitution II has greater potential than Substitution I for critical path compression, because it will always eliminate at least one additional chip crossing (that between the FPGAs indexed by $f(t_1)$ and $f(t_i)$). As was the case with Substitution I, it does not guarantee reduction in the delay of Π. Under our assumptions, however, there is only one situation in which Substitution II fails to reduce the delay of Π.

If $f(t_n) = f(t_1)$, then the target sequence contains at least two chip crossings, and Substitution II always reduces the delay of Π. Therefore, for the following discussion, assume $f(t_n) = f(t_i)$. Substitution II replaces a delay of $d(f(t_1), f(t_i))$ in the target sequence with a delay of *delta_local*. In addition, delays of $d(f(x), f(t_1))$ and $d(f(t_i), f(y))$ are replaced, respectively, with $d(f(x), z)$ and $d(z, f(y))$. The delay of Π increases, then, only if

$$delta\_local + d(f(x), z) + d(z, f(y)) > d(f(t_1), f(t_i)) + d(f(x), f(t_1)) + d(f(t_i), d(y)).$$

Using our assumed values for $delta\_local = 3$, $delta\_neighbor = 30$, and $delta\_global$ $= 50$, this cannot happen unless both of $d(f(x), z), d(z, f(y))$ are *delta_global*, and all of $d(f(t_1), f(t_i)), d(f(x), f(t_1)), d(f(t_i), d(y))$ are *delta_neighbor*. (Recall that none of these latter three values is *delta_local*, by the definition of the extended target sequence.)

Resequencing

For Resequencing, we require a two-subset target sequence, identical to that for Substitution II, except that the target sequence must contain at least two chip crossings. This means that there must exist some CLB $t_k, i < k \leq n$, such that $f(t_k) = f(t_1)$.

We now define Resequencing, with the assumption that $p > 0$ CLBs in the target sequence are initially assigned to $f(t_1)$ and $q > 0$ CLBs in the target sequence are initially assigned to $f(t_i)$.

Definition 4.10 *Resequencing is either*

1. *the assignment of the first p CLBs of a Resequencing target sequence to $f(t_1)$ and the last q CLBs of the sequence to $f(t_i)$, or*

2. *the assignment of the first q CLBs of a Resequencing target sequence to $f(t_i)$ and the last p CLBs of the sequence to $f(t_1)$,*

if changes can be made without violating pincount constraints.

We note that the predetermined values of p and q ensure it is not possible for Resequencing to violate size constraints.

Resequencing eliminates all but one chip crossing in the target sequence. Since the number of chip crossings in the target sequence may be more than two, it is impossible to calculate in general the delay of the critical path after a Resequencing. Furthermore, the analysis is dependent upon the assumed delay values. Using our assumptions, however, Resequencing always reduces the delay of Π, as we now demonstrate.

By the definition of Resequencing, there are two different resequencing possibilities: either the first p CLBs are assigned to $f(t_1)$ and the last q CLBs to $f(t_i)$, or the first q CLBs to $f(t_i)$ and the last p CLBs to $f(t_1)$. There are also two possibilities for the value of $f(t_n)$, which may be either $f(t_1)$ or $f(t_i)$. Note that, if $f(t_n) = f(t_i)$, there are at least three chip crossings in the target sequence. This gives rise to four possibilities. In each case, we assume the minimum number of chip crossings in the target sequence. If more chip crossings exist, Resequencing produces even further reduction in the delay of Π.

1. First resequencing order, $f(t_n) = f(t_1)$.

   A delay of $d(f(t_1), f(t_i))$ is replaced with a delay of *delta_local*, and a delay of $d(f(t_1), f(y))$ is replaced with a delay of $d(f(t_i), f(y))$. The delay of Π can increase only if

   $$delta\_local + d(f(t_i), f(y)) > d(f(t_1), f(t_i)) + d(f(t_1), f(y))$$

   The largest possible value of the left hand side is *delta_local* + *delta_global* = 53. The smallest possible value of the right hand side is 2× *delta_neighbor* = 60. Therefore, the delay of Π is decreased.

2. First resequencing order, $f(t_n) = f(t_i)$.

   Two delays of $d(f(t_1), f(t_i))$ are replaced by two delays of *delta_local*. Since $d(f(t_1), f(t_i))$ is at least *delta_neighbor*, the delay of Π is decreased.

3. Second resequencing order, $f(t_n) = f(t_1)$.

A delay of $d(f(t_1), f(t_i))$ is replaced with a delay of *delta_local*, and a delay of $d(f(x), f(t_1))$ is replaced with a delay of $d(f(x), f(t_i))$. The delay of Π can increase only if

$$delta\_local + d(f(x), f(t_i)) > d(f(t_1), f(t_i)) + d(f(x), f(t_1))$$

The largest possible value of the left hand side is $delta\_local + delta\_global = 53$. The smallest possible value of the right hand side is $2\times$ $delta\_neighbor = 60$. Therefore, the delay of Π is decreased.

4. Second resequencing order, $f(t_n) = f(t_i)$.

   Two delays of $d(f(t_1), f(t_i))$ are replaced with two delays of *delta_local*. A delay of $d(f(x), f(t_1))$ is replaced with a delay of $d(f(x), f(t_i))$, and a delay of $d(f(t_i), f(y))$ is replaced with a delay of $d(f(t_1), f(y))$. The delay of Π can increase only if

$$(2 \times delta\_local) + d(f(x), f(t_i)) + d(f(t_1), f(y)) >$$

$$(2 \times d(f(t_1), f(t_i))) + d(f(x), f(t_1)) + d(f(t_i), f(y))$$

   The largest possible value of the left hand side is $(2\times$ *delta_local*$)$ + $(2\times$ *delta_global*$) = 106$. The smallest possible value of the right hand side is $4\times$ $delta\_neighbor = 120$. Therefore, the delay of Π is decreased.

Experimental Results

We tested our heuristic on all combinational circuits from [Be], except for those that were partitioned onto a single chip, for which delay would then be optimum. We also excluded circuit c499xc2, which easily partitions onto two chips, and the optimization heuristic could not improve the delay. The statistics of the remaining circuits are shown in Table 4.4. The circuits are listed in order of size. “LP” refers to the number of CLBs in the longest path between any PI/PO pair.

The “xc2” circuits have been technology mapped for FPGAs of the Xilinx 2000 series, with chip capacity of 64 CLBs and 58 I/O pins. The “xc3” circuits have been technology mapped for the Xilinx 3000 series, with chip capacity of 144 CLBs and 96 I/O pins. All of our experiments were performed on a Sun ULTRA-1 workstation.

Three different hardware topologies were utilized in these tests, each containing 16 FPGAs: linear array, mesh and ring.

In every case, we begin with a satisfying partitioning of the circuit, as the first step in the two-step method. Recall that the partitioner of choice may be used for the first step; in our experiments we used our own partitioner.

There is no known efficient way to determine the optimal delay through a circuit, so in general we cannot compare the current delay to the optimum. Therefore, for purposes of comparing the critical path compression strategies, and the overall effectiveness of the algorithms, we measure percentage improvement in the delay.

Table 4.4: Circuit statistics

Test	CLBs	PIs	POs	Nets	LP
c2670xc3	150	157	64	361	12
c3540xc3	283	50	22	489	23
c3540xc2	373	50	22	567	21
c5315xc3	377	178	123	699	12
c7552xc3	489	206	107	921	11
c5315xc2	535	178	123	936	14
c7552xc2	610	206	107	1056	13
c6288xc2	833	32	32	1456	90
c6288xc3	833	32	32	1472	91

We begin by analyzing the effectiveness of the uphill move strategy. For each of the nine circuits, and for each of one hundred partitionings of each circuit, we tested various values of *look_ahead*. Because the purpose of this experiment was to analyze the uphill move strategy only, we fixed all other program parameters. We chose to activate all five critical path compression strategies and use a linear array topology. The values of *look_ahead* used were 1 (which disables uphill moves completely), 2, 4, 6, 8 and 10. Summaries of these test results appear in Tables 4.5, 4.6 and 4.7.

Each column in each table represents one value of *look_ahead*. In Table 4.5, each table entry is the percentage improvement in the delay, averaged over the one hundred runs. In Table 4.6, each table entry is the best final delay. In Table 4.7, each table entry is the average CPU time (in seconds) of the entire program, including processing all input files.

Table 4.5: Hill-climbing experiment: percentage improvement

test	*look_ahead* = 1	*look_ahead* = 2	*look_ahead* = 4	*look_ahead* = 6	*look_ahead* = 8	*look_ahead* = 10
c2670xc3	9.43	10.95	13.47	12.35	13.00	12.11
c3540xc3	15.90	21.53	24.98	25.64	23.27	23.01
c3540xc2	13.08	14.38	17.43	16.20	16.73	16.62
c5315xc3	21.48	22.06	24.30	24.14	22.99	23.71
c7552xc3	21.23	21.80	24.57	25.76	25.27	28.18
c5315xc2	16.42	20.09	23.19	23.70	22.80	23.35
c7552xc2	13.05	15.40	17.16	18.61	17.10	18.69
c6288xc2	13.95	17.10	18.15	19.07	18.53	18.72
c6288xc3	15.56	18.53	19.94	19.56	20.05	20.41

Table 4.6: Hill-climbing experiment: final delay

test	$look\_ahead = 1$	$look\_ahead = 2$	$look\_ahead = 4$	$look\_ahead = 6$	$look\_ahead = 8$	$look\_ahead = 10$
c2670xc3	77	77	77	77	77	77
c3540xc3	126	102	102	102	102	102
c3540xc2	232	232	211	211	211	211
c5315xc3	92	89	89	89	89	89
c7552xc3	128	101	101	101	101	101
c5315xc2	207	177	177	177	177	177
c7552xc2	221	209	204	198	198	198
c6288xc2	676	663	636	636	636	636
c6288xc3	525	525	505	505	491	491

Table 4.7: Hill-climbing experiment: CPU time

test	$look\_ahead$ = 1	$look\_ahead$ = 2	$look\_ahead$ = 4	$look\_ahead$ = 6	$look\_ahead$ = 8	$look\_ahead$ = 10
c2670xc3	0.14	0.16	0.20	0.22	0.26	0.29
c3540xc3	0.27	0.43	0.61	0.69	0.74	0.85
c3540xc2	0.43	0.59	0.84	1.02	1.20	1.37
c5315xc3	0.37	0.42	0.52	0.60	0.68	0.75
c7552xc3	0.55	0.68	0.71	0.97	1.06	1.16
c5315xc2	0.72	1.02	1.39	1.63	1.91	2.09
c7552xc2	0.80	1.06	1.34	1.69	2.13	2.17
c6288xc2	2.05	3.09	4.61	5.98	7.09	8.57
c6288xc3	1.72	2.66	4.00	5.01	5.86	6.63

In our testing, improvement was seldom seen beyond a look-ahead of six. CPU time increases significantly with larger values of $look\_ahead$. Therefore, a value of 10 for $look\_ahead$ seemed more than adequate, and was utilized throughout the remainder of the testing.

The next set of experiments was performed to compare the results over the three different topologies (linear array, mesh and ring). Again, one hundred partitionings of each of the circuits were utilized. In every case, all five critical path compression strategies were activated. The results are shown in Tables 4.8 and 4.9. Each entry of Table 4.8 is the average percentage improvement of the one hundred runs. Each entry of Table 4.9 is the best final delay. CPU times for all topologies are comparable to those of the last column of Table 4.7, and are not reported specifically.

Table 4.8: Topology comparison: percentage improvement

test	linear array	mesh	ring
c2670xc3	12.11	13.42	12.43
c3540xc3	23.01	25.46	23.38
c3540xc2	16.62	19.17	15.57
c5315xc3	23.71	26.64	23.65
c7552xc3	28.18	28.86	25.87
c5315xc2	23.35	24.74	22.57
c7552xc2	18.69	20.10	17.80
c6288xc2	18.72	20.20	18.68
c6288xc3	20.41	18.91	19.68

Table 4.9: Topology comparison: final delay

test	linear array	mesh	ring
c2670xc3	77	77	77
c3540xc3	102	102	102
c3540xc2	211	214	228
c5315xc3	89	89	89
c7552xc3	101	101	105
c5315xc2	177	166	186
c7552xc2	198	180	198
c6288xc2	636	600	656
c6288xc3	491	485	491

There does not appear to be any predictable difference in the behavior of the algorithm under different hardware topologies.

We then ran tests to compare the effectiveness of the different critical path compression strategies. (Recall that *look_ahead* has been set at 10.) For these tests, we used a linear array topology. Again, each circuit was run on one hundred different partitionings, and the results averaged. Each circuit was tested in six different modes: 1) only Elimination I activated; 2) only Elimination II activated; 3) only Substitution I activated; 4) only Substitution II activated; 5) only Resequencing activated; and 6) all five strategies activated. Tables 4.10, 4.11 and 4.12 show the results of these experiments. Each entry in Table 4.10 is the average percentage improvement; each entry in Table 4.11 is the final delay; and each entry in Table 4.12 is the average CPU time.

Table 4.10: Strategy comparison: percentage improvement

test	Elim I	Elim II	Subst I	Subst II	Reseq	ALL
c2670xc3	8.61	0.00	1.92	8.56	0.91	12.11
c3540xc3	20.90	0.48	6.90	2.25	1.16	23.01
c3540xc2	13.98	0.77	4.26	5.62	1.47	16.62
c5315xc3	18.00	5.57	6.13	10.68	1.05	23.71
c7552xc3	21.37	0.80	9.71	10.57	1.47	28.18
c5315xc2	13.12	3.08	5.08	17.20	3.43	23.35
c7552xc2	8.74	1.24	4.47	11.79	2.42	18.69
c6288xc2	10.93	2.89	5.75	16.68	3.65	18.72
c6288xc3	18.30	4.37	6.97	12.65	2.67	20.41

Table 4.11: Strategy comparison: final delay

test	Elim I	Elim II	Subst I	Subst II	Reseq	ALL
c2670xc3	78	80	80	80	80	77
c3540xc3	102	126	126	126	126	102
c3540xc2	245	289	255	263	275	211
c5315xc3	89	92	92	92	92	89
c7552xc3	101	168	151	148	145	101
c5315xc2	207	227	234	187	227	177
c7552xc2	227	225	225	224	249	198
c6288xc2	683	757	769	673	723	636
c6288xc3	525	599	573	565	573	491

Table 4.12: Strategy comparison: CPU time

test	Elim I	Elim II	Subst I	Subst II	Reseq	ALL
c2670xc3	0.16	0.13	0.19	0.14	0.11	0.29
c3540xc3	0.36	0.20	0.36	0.18	0.17	0.85
c3540xc2	0.31	0.27	0.68	0.42	0.20	1.37
c5315xc3	0.33	0.29	0.52	0.33	0.27	0.75
c7552xc3	0.48	0.41	0.76	0.48	0.39	1.16
c5315xc2	0.50	0.48	1.07	0.78	0.40	2.09
c7552xc2	0.52	0.48	1.22	0.82	0.47	2.17
c6288xc2	1.04	0.82	4.00	1.80	0.90	8.57
c6288xc3	1.68	0.86	3.04	1.40	0.93	6.63

From these experiments, it seems evident that, although all of the strategies produce results, the most successful ones are Elimination I and Substitution II. Substitution I appears to take the most CPU time, relative to percentage improvement.

In Section 4.2.4, we discussed a topological sort technique for finding the longest weighted path in a DAG. Our heuristic was initially coded using breadth-first search to find the critical path, and then modified to use the topological sort technique when it became evident that this was much more efficient. Our final set of experimental results compares CPU times of using these two methods for computing critical path. Table 4.13 reports CPU times only. In each case, a value of 10 was used for *look_ahead*, a linear array topology was used, and all five critical path compression strategies were activated. The average CPU time over one hundred runs is reported.

In summary, critical path compression seems to be an effective tool for improving the delay in a partitioned circuit. Additionally, the algorithmic platform is expandable, and can be augmented in the future with new strategy techniques. The running times seem quite dependent upon the length of the longest PI/PO path, which is

Table 4.13: Breadth-first search (BFS) vs. topological sort (TS): CPU time

test	BFS	TS
c2670xc3	0.40	0.29
c3540xc3	1.18	0.85
c3540xc2	2.22	1.37
c5315xc3	1.00	0.75
c7552xc3	1.30	1.16
c5315xc2	3.57	2.09
c7552xc2	3.49	2.17
c6288xc2	206.43	8.57
c6288xc3	176.97	6.63

what one would expect. The topological sort technique is superior to breadth-first search for computing critical path. This is especially evident in a computation in which critical path computation is done frequently on paths of significant length.

Chapter 5

Variations of the Fundamental Problem

In this chapter, we examine some other problems that are related to the fundamental problem. Most of these problems are of independent interest. Some have already been studied by other researchers. We discuss them here to present some new findings.

5.1 Hypergraphs

As discussed in Section 3.2.3, circuit designs are often represented by hypergraphs rather than ordinary graphs. A hypergraph differs from an ordinary graph in that more than two vertices are allowed in an edge. In such a representation, vertices represent circuit nodes (for example, CLBs), and edges represent nets that may connect more than two nodes. As such, hypergraphs are important as a representation tool in VLSI applications.

We generalize MDGP to hypergraphs as Hypergraph MDGP:

Instance: a hypergraph G, and two integers k and d.

Question: Is there is a partition of V into disjoint sets $V_1, \ldots, V_m$ such that $\forall i : |V_i| \leq k$, and such that if E_i is the set of hyperedges with at least one endpoint in V_i and at least one endpoint not in V_i, $max_{1 \leq i \leq m}|E_i| \leq d$?

If more than one vertex of some hyperedge E is partitioned into set S_1, with at least one vertex of E partitioned into set S_2, E contributes only 1 to the degrees of both S_1 and S_2. This is a consequence of the fact that we assume the routing is done internally on the CLB.

Hypergraph MDGP is, of course, $\mathcal{NP}$-complete because it is a generalization of MDGP.

Because Hypergraph MDGP is no longer defined in terms of ordinary graphs, none of the WQO-theoretic results discussed here apply directly. There is at least one known WQO over hypergraphs, however ([GGL]). A hypergraph H is a minor of another hypergraph G if H arises from G as the result of successive elementary operations, performed in any order. Elementary operations consist of the deletion of a node or an edge (subgraph operation), the replacement of an edge by any subset of itself (generalization of subgraph operation), and the identification of two nodes in an edge (generalization of contraction). Whether practical use can be made of this hypergraph WQO is an open question. None of the problems discussed in this work is closed in the ordinary minor order. Therefore, if practical use can be made of the hypergraph minor order, it will probably not be with partitioning problems of this type.

Even though none of the WQO results from Chapter 2 seem to apply to Hypergraph MDGP(k,d) (the fixed-parameter version of the problem), some of the non-WQO results from that chapter do hold.

We now define a path in a hypergraph: a path from vertex v to vertex w consists of a sequence of hyperedges $E_1, E_2, \ldots, E_n$, such that $v \in E_1, w \in E_n$, and $E_i \cap E_{i+1} \neq \emptyset, \forall 1 \leq i \leq n-1$.

Lemma 5.1 *A hypergraph H is a "yes" instance of Hypergraph MDGP iff there exists a solution in which every subset is connected; hence every v is partitioned only with other vertices in $N_{k-1}(v)$.*

Proof See proof of Lemma 2.1. □

We observe that, if G is a "yes" instance of Hypergraph MDGP(k,d), the number of vertices that are neighbors of v can be unbounded. For example, consider a graph G with n vertices, and exactly one hyperedge that contains all n vertices. G is a "yes" instance of Hypergraph MDGP(k,d) for any $k, d \geq 1$, although the number of neighbors of every vertex is unbounded. See Figure 5.1 for a partitioning of such a graph, with $k = d = 1$ (subsets are indicated by dotted lines). Thus, Lemma 2.2 does not hold for hypergraphs, for any constant.

Using the same definitions of kd-satisfying subset and kd-candidate subset, Lemma 2.3 holds for hypergraphs, with only slight modification.

Lemma 5.2 *Given kd-satisfying subsets $C1$ and $C2$, either $C1 - C2$ or $C2 - C1$ is kd-satisfying.* [1]

Proof Since neither $C1 - C2$ nor $C2 - C1$ can have size exceeding k, we need only consider their respective degrees.

If $C1 \cap C2 = \emptyset$, then we are done. Otherwise, let $I = C1 \cap C2, A = C1 - C2, B = C2 - C1, D = V - C1 - C2$ (see figure 2.4).

Denote by N_{AB} the number of edges with an endpoint in A and an endpoint in B. $N_{AD}, N_{AI}, N_{BD}, N_{BI}$ and N_{DI} have analogous meanings. When dealing with hypergraphs, we must also consider N_{ABC}, etc., which denotes the number of edges with endpoints in A, B, and C.

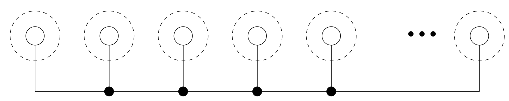


Figure 5.1: A "yes" instance of Hypergraph MDGP(k,d)

[1] Independently proved in [CLCDL].

The degree of $C1$ is $N_{AB}+N_{AD}+N_{BI}+N_{DI}+N_{ABD}+N_{ABI}+N_{ADI}+N_{BDI}+N_{ABDI}$, and the degree of $C2$ is $N_{AB}+N_{AI}+N_{BD}+N_{DI}+N_{ABD}+N_{ABI}+N_{ADI}+N_{BDI}+N_{ABDI}$.

By the definitions above, we have

$$N_{AB}+N_{AD}+N_{BI}+N_{DI}+N_{ABD}+N_{ABI}+N_{ADI}+N_{BDI}+N_{ABDI} \leq d$$

and

$$N_{AB}+N_{AI}+N_{BD}+N_{DI}+N_{ABD}+N_{ABI}+N_{ADI}+N_{BDI}+N_{ABDI} \leq d.$$

Summing and simplifying yields

$$2N_{AB}+N_{AD}+N_{AI}+N_{BD}+N_{BI}+2N_{ABD}+2N_{ABI}+N_{ADI}+N_{BDI}+2N_{ABDI} \leq 2d.$$

Thus either

$$N_{AB}+N_{AD}+N_{AI}+N_{ABD}+N_{ABI}+N_{ADI}+N_{ABDI} \leq d$$

or

$$N_{AB}+N_{BD}+N_{BI}+N_{ABD}+N_{ABI}+N_{BDI}+N_{ABDI} \leq d.$$

The former bounds the degree of $C1-C2$, the latter the degree of $C2-C1$. □

Lemma 5.3 *Given kd-satisfying subsets $C_1, C_2, ..., C_p$, a disjoint set of kd-satisfying subsets $D_1, D_2, ..., D_q$ exists such that $C_1 \cup C_2 \cup ... \cup C_p = D_1 \cup D_2 \cup ... \cup D_q$.*[2]

Proof See proof of Lemma 2.4. □

Proposition 5.1 The Hypergraph Locality Condition: *$G=(V,E)$ is a "yes" instance of Hypergraph MDGP(k,d) iff $\forall v \in V, (C_v, G) \neq \emptyset$.*

Proof See proof of Proposition 2.1. □

Theorem 5.1 *The search and decision versions of Hypergraph MDGP(k,d) can be solved in polynomial time.*

[2]Independently claimed in [CLCDL].

Proof Because Lemma 2.2 does not hold for Hypergraph MDGP(k,d), we cannot limit the search for kd-candidate subsets to a bounded neighborhood. However, we can still determine whether a kd-candidate subset exists for each vertex (hence, by the Hypergraph Locality Condition whether the graph is a "yes" instance) by examining all $\binom{|V|}{i}$, $1 \leq i \leq k$, possible subsets. Since k is a constant, this can be done in $O(n^k)$ time. If there exists a kd-candidate subset for every vertex, a disjoint set can be found in polynomial time, by the proof of Lemma 5.3. □

Although this problem is in $\mathcal{P}$, the degree of the polynomial is high. It is not known whether Hypergraph MDGP(k,d) can be solved in low-order polynomial time.

Recall that the heuristic presented in Chapter 3 accommodates hypergraphs for the FPGA Minimization problem.

5.2 Partitioning for Heterogeneous Systems

In the MDGP problem, we were given two parameters, k and d, which represent, respectively, the size and pin-count of a type of FPGA chip. In some circuit partitioning situations, there exists a variety of chip types from which to choose. In this section, we generalize MDGP to allow for such a system of *heterogeneous* FPGAs. Rather than considering a single style of FPGA with k logic blocks and d pins, we consider a set of x FPGA types, with logic block and pin count constraints $k_1, d_1; k_2, d_2; ...; k_x, d_x$ ([BKK]). We call this problem Heterogeneous MDGP, and formalize it as follows.

Instance: a graph G, and a pair list L containing $2 \times x$ integers: $k_1, d_1; k_2, d_2; ...; k_x, d_x$.

Question: Is there is a partition of V into disjoint sets $V_1, \ldots, V_m$ such that $\forall V_i, 1 \leq i \leq m, \exists j, 1 \leq j \leq x$, such that $|V_i| \leq k_j$, and $\delta(V_i) \leq d_j$?

The fixed-parameter version of Heterogeneous MDGP will be referred to as MDGP(L).

Since Heterogeneous MDGP is a generalization of MDGP, its $\mathcal{NP}$-completeness follows from that of MDGP. However, when all parameters are fixed, we have the following.

Theorem 5.2 *MDGP(L) can be decided in polynomial time.*

Proof We observe that MDGP(L) is immersion closed. Given a satisfying partition, neither the subgraph operation nor edge lifting invalidates that partition. $\square$

Although Heterogeneous MDGP is very similar to MDGP, it is not possible in general to convert an instance of MDGP(L) to an instance of MDGP(k,d).

Consider MDGP(L), with L consisting of two pairs: $k_1 = 2, d_1 = 1, k_2 = 1$ and $d_2 = 2$. We will refer to this specific instance of MDGP(L) as MDGP(2,1; 1,2). The graph G in Figure 5.2 is an obstruction. If MDGP(2,1; 1,2) = MDGP(k,d) for some k and d, then G is also an obstruction for MDGP(k,d). G is a "yes" instance for any $k \geq 3$, so if it's an obstruction, it must be the case that $k = 1$ or $k = 2$. Suppose $k = 1$. Then G is a "yes" for any $d \geq 3$, so it must be the case that $d = 1$ or $d = 2$. On the other hand, if $k = 2$, then G is a "yes" for any $d \geq 2$, so it must be the case that $d = 1$. So we have three possibilities:

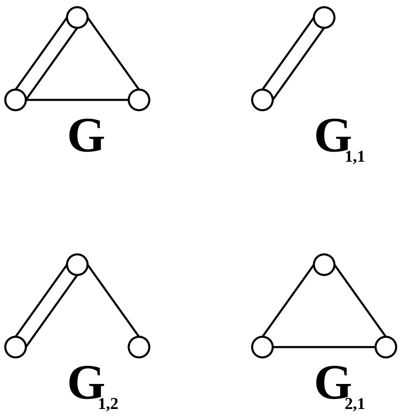


Figure 5.2: MDGP(2,1; 1,2) $\neq$ MDGP(k,d)

1. MDGP(2,1; 1,2) = MDGP(1,1). The subgraph $G_{1,1}$ of Figure 5.2 is a "no" instance; therefore G is not an obstruction.

2. MDGP(2,1; 1,2) = MDGP(1,2). The subgraph $G_{1,2}$ of Figure 5.2 is a "no" instance; therefore G is not an obstruction.

3. MDGP(2,1; 1,2) = MDGP(2,1). The subgraph $G_{2,1}$ of Figure 5.2 is a "no" instance; therefore G is not an obstruction.

Therefore, MDGP(2,1; 1,2) is not the same as MDGP(k,d) for any values of k and d.

It is also possible to find "yes" instances of MDGP(L) that are "no" instances of MDGP(k_i, d_i), $\forall 1 \leq i \leq x$. For example, consider the graph of Figure 5.3 which is a "yes" instance of MDGP(2,1; 1,2) but is a "no" for both MDGP(2,1) and MDGP(1,2).

We find that almost all of the known results for MDGP(k,d) hold for MDGP(L), with only slight modification.

Definition 5.1 *Let* $t_{max} = max(k_i + d_i), 1 \leq i \leq x$.

Observation 5.1 *A star graph with* t_{max} *rays is an obstruction to MDGP(L); therefore, no obstruction to MDGP(L) contains a vertex with more than* t_{max} *neighbors.*

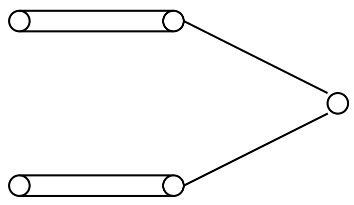


Figure 5.3: A "yes" instance of MDGP(2,1; 1,2)

Similarly, no "yes" instance of MDGP(L) contains a vertex with more than t_{max} neighbors; hence the "yes" family has bounded degree.

Similar to the lemmas and definitions we had for MDGP(k,d), we have the following:

Lemma 5.4 *G is a "yes" instance of MDGP(L) iff there exists a solution in which every subset is connected; hence every v is partitioned only with other vertices in $N_{k_i-1}(v)$, for some $1 \leq i \leq x$.*

Definition 5.2 *Given t_{max}, let c'_p denote the value $1 + \sum_{i=1}^{p}(t_{max})(t_{max} - 1)^{i-1}$.*

Lemma 5.5 *If G is an obstruction to MDGP(L), then $\forall v \in V, \forall p > 0, |N_p(v)| \leq c'_p$.*

Definition 5.3 *A "$k_i d_i$-satisfying subset" is a subset of size no more than k_i and degree no more than d_i, for some $1 \leq i \leq x$.*

Definition 5.4 *A "$k_i d_i$-candidate subset" is a connected $k_i d_i$-satisfying subset. Given $x, k_1, d_1; \ldots ; k_x, d_x$ and a vertex v, let C'_v denote the set of all $k_i d_i$-candidate subsets containing v.*

Lemma 5.6 *Given $k_m d_m$-satisfying subset $C1$, and $k_n d_n$-satisfying subset $C2$, either $C1 - C2$ is a $k_m d_m$-satisfying subset or $C2 - C1$ is a $k_n d_n$-satisfying subset.*

Proof Since $C1 - C2$ cannot have size exceeding k_m, and $C2 - C1$ cannot have size exceeding k_n, we need only consider their respective degrees.

If $C1 \cap C2 = \emptyset$, then we are done. Otherwise, let $I = C1 \cap C2, A = C1 - C2, B = C2 - C1, D = V - C1 - C2$ (see Figure 2.4).

Denote by N_{AB} the number of edges with an endpoint in A and an endpoint in B. $N_{AD}, N_{AI}, N_{BD}, N_{BI}$ and N_{DI} have analogous meanings. The degree of $C1$ is $N_{AD} + N_{AB} + N_{DI} + N_{BI}$, and the degree of $C2$ is $N_{AB} + N_{BD} + N_{AI} + N_{DI}$.

By the definitions above, we have

$$N_{AD} + N_{AB} + N_{DI} + N_{BI} \leq d_m$$

and

$$N_{AB} + N_{BD} + N_{AI} + N_{DI} \leq d_n$$

Summing yields

$$N_{AD} + 2N_{AB} + 2N_{DI} + N_{BI} + N_{BD} + N_{AI} \leq d_m + d_n,$$

so

$$N_{AD} + 2N_{AB} + N_{BI} + N_{BD} + N_{AI} \leq d_m + d_n.$$

Thus either

$$N_{AB} + N_{AI} + N_{AD} \leq d_m$$

or

$$N_{AB} + N_{BI} + N_{BD} \leq d_n.$$

The former bounds the degree of $C1 - C2$, the latter the degree of $C2 - C1$. □

Lemma 5.7 *Given $k_i d_i$-satisfying subsets $C_1, C_2, ..., C_p$, a disjoint set of $k_i d_i$-satisfying subsets $D_1, D_2, ..., D_q$ exists such that $C_1 \cup C_2 \cup ... \cup C_p = D_1 \cup D_2 \cup ... \cup D_q$.*

Proof See the proof of Lemma 2.4. □

Proposition 5.2 Heterogeneous Locality Condition *G is a "yes" instance of MDGP(L) iff $\forall v \in V, C'_v \neq \emptyset$.*

Proof See the proof of Proposition 2.1. □

In a straightforward manner, other results from MDGP(k,d) follow:

Theorem 5.3 *The search version of MDGP(L) can be solved in $O(np(n))$ time, where p(n) denotes the time required to solve the decision version of the problem.*

Theorem 5.4 *The decision and search versions of MDGP(L) can be solved in linear time.*

Theorem 5.5 *The obstruction set to MDGP(L) is computable.*

The proofs are all analogous to those for MDGP(k,d).

The complexity of Heterogeneous MDGP restricted to simple trees (hence, simple forests) can also be addressed in a manner similar to that for MDGP.

Lemma 5.8 *For any simple tree T, and any $i, 1 \leq i \leq x$, and $v \in V(T)$ with $\delta(v) > d_i$, any $k_i d_i$-candidate subset C including v includes at least $\delta(v) - d_i$ entire subtrees of v. Additionally, if any set of at least $\delta(v) - d_i$ entire subtrees of v is of size less than k_i, these subtrees, along with v, form a $k_i d_i$-candidate subset.*

Proof See the proof of Lemma 2.8. $\square$

Theorem 5.6 *Heterogeneous MDGP, restricted to simple trees, is in $\mathcal{P}$.*

Proof Given a simple tree T, first check whether any vertex has degree t_{max} or more. If so, T is a "no" instance, because it contains an obstruction.

Otherwise, for each $v \in T$, do the following. If the degree of v is no more than d_i, for some $1 \leq i \leq x$, then $\{v\}$ is a $k_i d_i$-candidate subset for v. If the degree of v is more than $d_i, \forall 1 \leq i \leq x$, then we perform the following steps:

1. Compute the size of each subtree of v. This takes $O(n)$ time.
2. Sort the sizes of the subtrees of v. This takes $O(n \; log \; n)$ time.
3. Mark v.
4. For each $i, 1 \leq i \leq x$, if the total size of the smallest $\delta(v) - d_i$ subtrees of v is less than k_i, then unmark v. For each i, this takes $O(n)$ time, hence the overall step takes $O(n^2)$ time.
5. If v is unmarked, then v has a $k_i d_i$-candidate subset, by Lemma 5.8. Otherwise, by Lemma 5.8, v has no $k_i d_i$-candidate subset.

By the Heterogeneous Locality Condition, if any vertex has no $k_i d_i$-candidate subset, then T is a “no” instance, otherwise it is a “yes” instance. □

The complexity of the algorithm as a whole, then, is $O(n^3)$.

Because each tree in a simple forest can be handled independently, Theorem 5.6 generalizes to simple forests.

5.3 Labelled Graphs

It may be possible that there exists a set of special components in a circuit that must be mapped onto FPGAs in such a way that no more than one of the special nodes is present in a single FPGA. We model this situation as a graph that contains special nodes designated as *terminals*. The immersion order on such graphs is known to be well-quasi-ordered ([RS3]). We define a *terminal partition* of G as a partition in which each subset contains at most one terminal from G. Labelled MDGP can then be formulated as follows.

Instance: a graph G, in which some of the vertices are terminals; two integers k and d.

Question: Is there is a terminal partition of V into disjoint sets $V_1, \ldots, V_m$ such that $\forall i : |V_i| \leq k$, and such that if E_i is the set of edges with exactly one endpoint in V_i, $max_{1 \leq i \leq m} |E_i| \leq d$?

This problem is a generalization of MDGP, hence is $\mathcal{NP}$-complete. Fixed-parameter Labelled MDGP (Labelled MDGP(k,d)) is immersion closed, and is amenable to other MDGP techniques in a fairly straightforward manner. The problems are not interchangeable, however. An instance of Labelled MDGP(k,d) cannot always be cast as an instance of MDGP(k,d). A simple example is a graph consisting of only two vertices, both of which are labelled, connected by $d+1$ edges. Such a graph is an obstruction to Labelled MDGP(k,d) for any $k > 0$. Specifically, it is an obstruction to Labelled MDGP(2,d). The only MDGP family for which this graph is an obstruction is MDGP(1,d). Therefore, it would have to be the case that Labelled MDGP(2,d) is the same as MDGP(1,d), which is untrue. A graph consisting of two unlabelled vertices connected by $d+1$ edges is a "yes" instance of Labelled MDGP(2,d), but a "no" instance of MDGP(1,d).

A generalization of the labelled version of MDGP is the colored version, in which a subset of the vertices is colored from a finite set of t colors, and a satisfying partitioning requires all vertices of each color to be in the same subset. On closer inspection, we observe that the colored version and the labelled version are equivalent. The labelled version is a special case of the colored version, in which there is exactly one vertex of each color. The colored version can be solved using any algorithm for the labelled version by connecting all vertices of the same color with $d+1$ edges, labelling exactly one vertex of each color, and then removing colors.

Sometimes in a graphical representation of a circuit, the set of nodes is separated into two disjoint sets, V_x and V_y, where V_x denotes the set of *interior* nodes and V_y denotes the set of *boundary* nodes ([BKK]). This reflects the function performed by a specific node, and the fact that a given FPGA has distinct interior logic blocks and exterior I/O pins. We can easily state this problem in terms of a graph, however, it does not seem possible to model it in a way to obtain immersion closure. The difficulty is that the lifting operation can either increase or decrease the number of boundary vertices in a subset.

5.4 Balanced Partitioning

Another occasional goal in circuit partitioning is to obtain a solution in which the subset sizes are balanced; i.e. no two subsets differ in size by more than some constant c. The problem we define here, Balanced MDGP, is identical to MDGP, except that we insist that the sizes of any two subsets in the partition be within c of each other.

Instance: a graph $G = (V, E)$, and three integers k, d and c.

Question: Is there is a partition of V into disjoint sets $V_1, \ldots, V_m$ such that $\forall i : |V_i| \leq k$, such that $\forall i,j : ||V_i| - |V_j|| \leq c$, and such that if E_i is the set of edges with exactly one endpoint in V_i, $max_{1 \leq i \leq m}|E_i| \leq d$?

This problem is a generalization of MDGP, hence it is $\mathcal{NP}$-complete. What is interesting is what happens when all three integer parameters are constants. We refer to this version of the problem as MDGP(k,d,c). The complexity of the problem then depends on the value of the constant c.

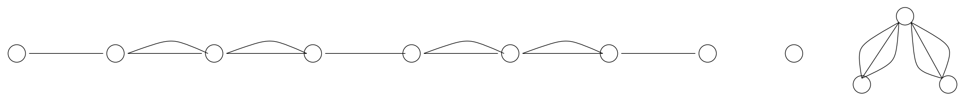
If $c \geq k-1$, then MDGP(k,d,c) is identical to MDGP(k,d) and is in P. However, if $c = 0$, we have the following result:

Theorem 5.7 *For any fixed k and d, MDGP(k,d,0) is $\mathcal{NP}$-complete.*

Proof Let M be an arbitrary instance of *p-way* MDGP(k,d), consisting of a graph G and an integer p. The constants k and d are the same for both problems. We could then decide whether M is a "yes" instance by solving the following instance of MDGP(k,d,0). We form the graph instance of MDGP(k,d,0), G', by augmenting G as follows. We add $(pk) - |V(G)|$ isolated vertices. We also add one component consisting of a k-path, with each edge having multiplicity $d+1$. Figure 5.4(a) shows an instance of *p-way* MDGP(k,d) ($k = 3, d = 2, p = 3$), and Figure 5.4(b) shows the corresponding instance of MDGP(3,2,0).

If G is a "yes" instance of *p-way* MDGP(k,d), then G' is a "yes" instance of MDGP(k,d,0). G can be partitioned into p subsets, each of size no more than k and degree no more than d. The vertices of G' that correspond to those of G can be partitioned in the same way. The $(pk) - |V(G)|$ isolated vertices can be distributed among the p subsets so that each subset is of size k. The $d+1$-edge-connected k-component of G' is self-contained in a single subset. Thus, G' can be partitioned into $p+1$ subsets of identical size.

(a)



(b)

Figure 5.4: Instances of *p-way* MDGP(k,d) and MDGP(k,d,0)

Figures 5.5(a) and (b) show the partitionings of the instances depicted in Figures 5.4(a) and (b).

Conversely, suppose G is a "no" instance of *p-way* MDGP(k,d). It is either the case that every partitioning violates either k or d, or that G can be partitioned to satisfy k and d, but the number of subsets always exceeds p. In the first case, G' would also not be partitionable. In the second case, any partitioning of G' would consist of one subset of size k containing the $d+1$-edge-connected k-component, along with more than p other subsets. Since the total number of vertices in G' is $pk+k$, at least one of the subsets is of size less than k, and a perfect balance is not achieved. $\square$

The cases of $c=0$ and $c=k-1$ are, of course, the easiest and least interesting. We have not addressed the complexity of the problem when confined to connected instances, nor have we considered Balanced FPGA Minimization. The complexity of MDGP(k,d,c) for $0<c<k-1$ remains an open question.

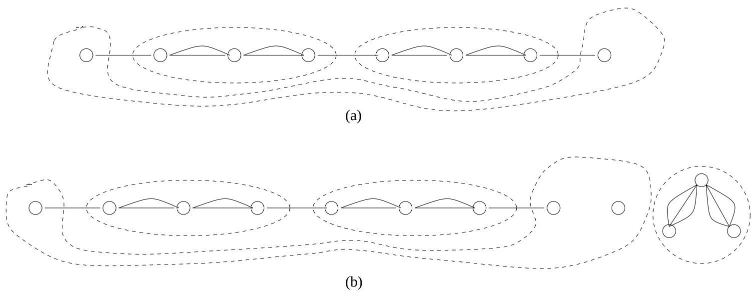


Figure 5.5: Partitioning the graphs of Figures 5.4(a)

Chapter 6

Future Directions and Conclusion

In this chapter, we present some research results that may have future application potential, as well as some open problems.

6.1 Theoretical Directions

6.1.1 Closure-Preserving Operators

The topic of this section has been previously studied by [BFL]. Many of these results were independently discovered, although to the best of our knowledge have not been published, with one exception which will be pointed out later.

We define sixteen families of graphs, each of which in turn is defined in terms of an arbitrary minor-closed (or immersion-closed) family of graphs. We examine the properties of the resulting families, and consider the question of whether minor closure (or immersion closure) is preserved.

As an example application, consider the question of planarity. The family of planar graphs is closed under the minor order (but not the immersion order). A graph may be said to be "almost planar" if there exists a way to remove a small fixed number of vertices to produce a planar graph. Knowing that the family of planar graphs is minor closed, can we assume that the family of "almost planar" graphs is minor closed?

A general way to state the question we consider is as follows. Given a graph G that is not necessarily a member of some family F, where F is minor closed (or immersion closed), do there exist some k vertices (or edges) that can be added to (or taken away from) G to form $G' \in F$? These alternatives (minor vs. immersion order, vertices vs. edges, added vs. taken away) produce eight families.

To obtain eight more families, we rephrase the question as follows. Given a graph G that is not necessarily a member of some family F, where F is minor closed (or immersion closed), is it the case that for *every* set of k vertices (or edges) added to (or taken away from) G forms $G' \in F$?

We use a shorthand notation of **abcde** to denote each family, in which

- **a** is either "M" (F is minor closed) or "I" (F is immersion closed),
- **b** is either "∃" (some set of vertices or edges) or "∀" (all sets of vertices or edges),
- **c** is either "+" (adding vertices or edges) or "-" (removing vertices or edges),
- **d** is a constant denoting the number of vertices (or edges) to be added (or removed), and
- **e** is either "v" (vertices) or "e" (edges).

In [BFL], **M∃-kv** was examined, and shown to be minor closed. Given a minor-closed family F of graphs, G is in **M∃-kv** if there exists a way to remove k vertices from G, forming G' such that G' is in F. Returning to our example application above, we conclude that the family of "almost planar" graphs is minor closed.

We now examine all sixteen graph families in detail.

1. F_k = **M∃+kv**: $G = (V_G, E_G) \in$ **M∃+kv** if there exists a set S of vertices, $S \cap V_G = \emptyset$, $|S| = k$, such that $G' = (V_G \cup S, E_G) \in F$.

Theorem 6.1 **M∃+kv** *is minor closed, and* **M∃+kv** $\subseteq F$*.*

Observation 6.1 *We note that for any* $G \in F_k$*, it is already the case that* G *is in* F*, because* G *is a subgraph of* G'*.*

Proof Noting that S is a set of disjoint vertices, it is easy to see that for any minor $H = (V_H, E_H)$ of $G = (V_G, E_G)$, $H' = (V_H \cup S, E_H)$ is a minor of $G' = (V_G \cup S, E_G)$. Therefore, $H' \in F$, $H \in$ **Mv∃+kv**, and **M∃+kv** is minor closed. □

Because adding isolated vertices does not seem to destroy any inherent structure in a family of graphs, it is tempting to conjecture that **M∃+kv**$=F$. However, consider F defined by graphs G such that either 1) G has 5 or fewer vertices, or 2) G has a vertex cover (a set of vertices that includes at least one endpoint of each edge) of size 1 or less. (The second property has been added to make F infinite, which is not necessary to disprove equivalence, but shows that inequivalence applies to both finite and infinite families.) F is minor closed. If G is any graph with 5 vertices that does not have a vertex cover of 1 or less, $G \in F$, but $G \notin$ **M∃+kv** for any positive value of k.

2. $F_k =$ **M∀M+kv**: $G = (V_G, E_G) \in$ **M∀+kv** if for every set S of vertices, $|S| = k$, $G' = (V_G \cup S, E_G) \in F$.

Theorem 6.2 **M∀+kv** *is minor closed, and* **M∀+kv** $\subseteq F$*.*

Proof This family is observed to be identical to **M∃+kv**. If $G' = (V_G \cup S, E_G) \in F$ for some set S of size k, then certainly $G' \in F$ for all sets S of size k, since S is a set of disjoint vertices.□

3. $F_k =$ **M∃-kv**: $G = (V_G, E_G) \in$ **M∃-kv** if there exists a set S of vertices, $S \subseteq V_G, |S| = k$, such that $G' = (V_G - S, E_G) \in F$.

Theorem 6.3 (BFL) **M∃-kv** *is minor closed, and* $F \subseteq$ **M∃-kv**.

Observation 6.2 *Every graph in* F *is also in* F_k*, because* F *is closed under subgraphs.*

4. $F_k =$ **M∀-kv**: $G = (V_G, E_G) \in$ **M∀-kv** if for all sets S of vertices, $S \subseteq V_G, |S| = k$, $G' = (V_G - S, E_G) \in F$.

Theorem 6.4 **M∀-kv** *is minor closed, and* $F \subseteq$ **M∀-kv**.

We restrict our attention only to graphs for which $|V_G| \geq k$.

Proof Consider any $H = (V_H, E_H) \leq_M G = (V_G, E_G)$, such that H has at least k vertices. If H was obtained by removing a vertex or an edge from G, then for S any set of k vertices in H, $H' = (V_H - S, E_H)$ is a subgraph of $G' = (V_G - S, E_G)$. Therefore, by the minor closure of F, $H' \in F$, and $H \in$ **M∀-kv**.

Now suppose H was obtained by contracting edge (u, v) in G (removing u), and let S be any set of k vertices in H. If $v \notin S$, then $H' = (V_H - S, E_H) \leq_M G' = (V_G - S, E_G)$. If $v \in S$, then $H' = (V_H - S, E_H) = (V_G - S - u, E_G)$, which is a subgraph of $(V_G - S, E_G) \in F$.

Observation 6.2 applies to this family. □

Observation 6.3 *There is only a finite number of graphs in* F_k *that do not belong to* F.

Consider some $G \in F_k$, but not in F. Then G contains some obstruction O. But we must be able to remove any set of k vertices (edges) to get G' in the closed family. Therefore, no matter how we remove the vertices (edges), we need to capture O. Thus all graphs in F_k that are not in F are of size bounded by the members of F's obstruction set.

For any F_k to which this observation applies, there exists a low-order polynomial time recognition algorithm, even if F_k is not minor (immersion) closed.

5. F_k = **M∃+ke**: $G = (V_G, E_G) \in$ **M∃+ke** if there exists a set $E_K, E_K \cap E_G = \emptyset, |E_K| = k$, such that $G' = (V_G, E_G \cup E_K) \in F$.

Theorem 6.5 **M∃+ke** *is not minor closed, and* **M∃+ke** $\subseteq F$.

Proof Let F be the (minor-closed) family of graphs that have no cycles, and let $k = 1$. The graph G_1 illustrated in Figure 6.1(a) is in F, and is also in **M∃+ke** because there exists a way to add an edge to this graph, with the resulting graph still in F. However, consider the graph H_1 of Figure 6.1(b) that is a minor of G_1. There is no way to add an edge to this graph and still remain in F.

Observation 6.1 applies to this family. □

(a) G_1 **(b) H_1**

Figure 6.1: Graphs G_1 and H_1

6. F_k = **M∀+ke**: $G = (V_G, E_G) \in$ **M∀+ke** if for every set $E_K, E_K \cap E_G = \emptyset, |E_K| = k$, $G' = (V_G, E_G \cup E_K) \in F$.

Theorem 6.6 **M∀+ke** *is minor closed, and* **M∀+ke** $\subseteq F$.

Proof Consider $H = (V_G - v, E_G)$, and let E_K be any set of k edges that can be added to H. Then $H' = (V_G - v, E_G \cup E_K)$ is a subgraph of $G' = (V_G, E_G \cup E_K)$, $G' \in F$, so by the minor closure of F, $H' \in F$, and $H \in$ **M∃+ke**.

Consider $H = (V_G, E_G - (x, y))$, and let E_K be any set of k edges that can be added to H. If $(x, y) \notin E_K$, then $H' = (V_G, E_G \cup E_K - (x, y))$ is a subgraph of $G' = (V_G, E_G \cup E_K)$. If $(x, y) \in E_K$, then $H' = (V_G, E_G \cup E_{K-1})$, where E_{K-1} is of size $k-1$. Since $(V_G, E_G \cup E_K) \in F$ for any E_K of size k, $(V_G, E_G \cup E_{K-1}) \in F$.

If $H = (V_H, E_H)$ was formed by contracting edge (u, v) in G (removing u), then, for any set E_K of k edges, $H' = (V_H, E_H \cup E_K)$ is a minor of $G' = (V_G, E_G \cup E_K)$, $H' \in F$, and $H \in$ **M∀+ke**.

Observation 6.1 applies to this family. □

7. F_k = **M∃-ke**: $G = (V_G, E_G) \in$ **M∃-ke** if there exists a set E_K of edges, $E_K \subseteq E_G, |E_K| = k$, such that $G' = (V_G, E_G - E_K) \in F$.

Theorem 6.7 **M∃-ke** *is not minor closed, and* $F \subseteq$ **M∃-ke**.

Proof Let F be the family of graphs all of whose vertices are of degree $0, 1$, or 2. F is minor closed. Consider $k = 1$. The graph G_2 in Figure 6.2(a) is in **M∃-ke**, because removal of the middle edge yields a graph in F. However, graph H_2 shown in Figure 6.2(b), which is a minor of G_2, is not in **M∃-ke**.

Observation 6.2 applies to this family. □

8. F_k = **M∀-ke**: $G = (V_G, E_G) \in$ **M∀-ke** if for all sets E_K of edges, $E_K \subseteq E_G$, $G' = (V_G, E_G - E_K) \in F$.

(a) G $_2$ **(b) H** $_2$

Figure 6.2: Graphs G_2 and H_2

Theorem 6.8 **M∀-ke** *is minor closed, and* $F \subseteq$ **M∀-ke** *.*

Proof For $H = (V_G - v, E_G)$, $H' = (V_G - v, E_G - E_K)$ for any set E_K of k edges, is a subgraph of $G' = (V_G, E_G - E_K) \in F$. Thus $H' \in F$. The same reasoning shows that $H = (V_G, E_G - (x, y)) \in$ **M∀-ke**.

If $H = (V_H, E_H)$ was formed by contracting edge (u, v) (eliminating u) in G, then consider $H' = (V_H, E_H - E_K)$ for any set E_K of k edges. H' is a minor of $G' = (V_G, E_G - E'_K)$, where E'_K contains all the edges of E_K, except edges of the form (v, x) where (v, x) is not an edge of G are replaced with (u, x).

Observation 6.2 applies to this family. □

Observation 6.3 applies to this family.

9. F_k = **I∃+kv**: $G = (V_G, E_G) \in$ **I∃+kv** if there exists a set S of vertices, $S \cap V_G = \emptyset$, $|S| = k$, such that $G' = (V_G \cup S, E_G) \in F$.

Theorem 6.9 **I∃+kv** *is immersion closed, and* **I∃+kv** $\subseteq F$*.*

Proof Noting that S is a set of disjoint vertices, observe that for any immersed $H = (V_H, E_H)$ of $G = (V_G, E_G)$, $H' = (V_H \cup S, E_H)$ is immersed in $G' = (V_G \cup S, E_G)$. Therefore, $H' \in F$, $H \in$ **I∃+kv**, and **I∃+kv** is immersion closed.

Observation 6.1 applies to this family. As in **M∃+kv**, we might conjecture that **I∃+kv**=F. However, consider F defined by graphs G such that either 1) G has 5 or fewer vertices, or 2) G is cycle free. F is immersion closed. In any graph with 5 vertices that contains a cycle, $G \in F$, but $G \notin$ **M∃+kv** for any positive value of k. □

10. F_k = **I∀+kv**: $G = (V_G, E_G) \in$ **I∀+kv** if for every set S of vertices, $|S| = k$, $G' = (V_G \cup S, E_G) \in F$.

Theorem 6.10 **I∀+kv** *is immersion closed, and* **I∀+kv** $\subseteq F$.

Proof This family is seen to be identical to **I∃+kv**. If $G' = (V_G \cup S, E_G) \in F$ for some set S of size k, then certainly $G' \in F$ for all sets S of size k, since S is a set of disjoint vertices. □

11. F_k = **I∃-kv**: $G = (V_G, E_G) \in$ **I∃-kv** if there exists a set S of vertices, $S \subseteq V_G$, $|S| = k$, such that $G' = (V_G - S, E_G) \in F$.

Theorem 6.11 **I∃-kv** *is not immersion closed, and* $F \subseteq$ **I∃-kv**.

Proof Consider the family F of graphs that have no edges, which is observed to be closed under immersion, and let $k = 1$. The graph G_3 shown in Figure 6.3(a) is in **I∃-kv**, because removal of the middle vertex yields a graph in F. However, for the immersed H_3 of G_3 shown in Figure 6.3(b), there is no way to remove a single vertex to obtain a graph in F.

Observation 6.2 applies to this family. □

12. F_k =**I∀-kv**: $G = (V_G, E_G)$ ∈**I∀-kv** if for all sets S of vertices, $S \subseteq V_G$, $|S| = k$, $G' = (V_G - S, E_G) \in F$.

Theorem 6.12 **I∀-kv** *is not immersion closed, and* $F \subseteq$ **I∀-kv**.

(a) G_3 (b) H_3

Figure 6.3: Graphs G_3 and H_3

Proof Consider the family F of graphs that have no cycles, which is observed to be closed under immersion, and let $k = 1$.

The graph G_4 shown in Figure 6.4(a) is in **I∀-kv**, because removal of any vertex yields a graph in F. However, for the immersed H_4 of G_4 shown in Figure 6.4(b), removal of the isolated vertex does not yield a graph in F.

Observation 6.2 applies to this family. □

Observation 6.3 applies to this family.

13. F_k =**I∃+ke**: $G = (V_G, E_G) \in$ **I∃+ke** if there exists a set $E_K, E_K \cap E_G = \emptyset, |E_K| = k$, such that $G' = (V_G, E_G \cup E_K) \in F$.

Theorem 6.13 **I∃+ke** *is not immersion closed, and* **I∃+ke** $\subseteq F$.

Proof Let F be the family of cycle-free graphs, which is immersion closed, and let $k = 1$. There exists a way to add an edge to the graph G of Figure 6.1(a), with the resulting graph still in F. However, for the immersed H of Figure 6.1(b), there is no way to add an edge without introducing a cycle.

Observation 6.1 applies to this family. □

14. F_k = **I∀+ke**: $G = (V_G, E_G) \in$ **I∀+ke** if for every set $E_K, E_K \cap E_G = \emptyset, |E_K| = k$, $G' = (V_G, E_G \cup E_K) \in F$.

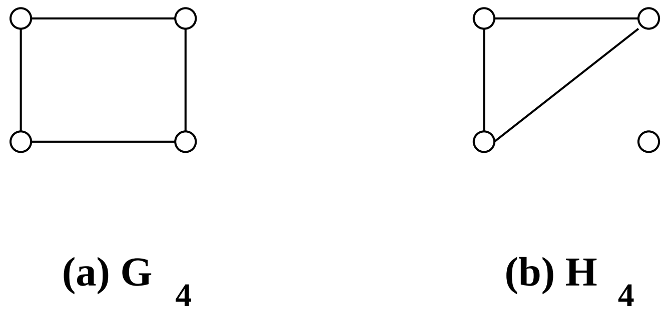


Figure 6.4: Graphs G_4 and H_4

Theorem 6.14 **I∀+ke** *is immersion closed, and* **I∀+ke** $\subseteq F$.

Proof Consider $H = (V_G - v, E_G)$, and let E_K be any set of k edges that can be added to H. Then $H' = (V_G - v, E_G \cup E_K)$ is a subgraph of $G' = (V_G, E_G \cup E_K)$, $G' \in F$, so by the immersion closure of F, $H' \in F$, and $H \in$ **I∃+ke**.

Consider $H = (V_G, E_G - (x, y))$, and let E_K be any set of k edges that can be added to H. If $(x, y) \notin E_K$, then $H' = (V_G, E_G \cup E_K - (x, y))$ is a subgraph of $G' = (V_G, E_G \cup E_K)$. If $(x, y) \in E_K$, then $H' = (V_G, E_G \cup E_{K-1})$, where E_{K-1} is of size $k-1$. Since $(V_G, E_G \cup E_K) \in F$ for any E_K of size k, $(V_G, E_G \cup E_{K-1}) \in F$.

If H was formed by lifting $(u, v), (v, w)$ (adding (u, w)), consider E_K any set of k edges that can be added to H to form H'. If $E_K \cap \{(u, v), (v, w)\} = \emptyset$ then H' is immersed in $G' = (V_G, E_G \cup E_K)$. If either or both of $(u, v), (v, w) \in E_K$, then H' is a subgraph of $G' = (V_G, E_G \cup E_K \cup (u, w))$, which is in F because the set of added (unredundant) edges is of size $\leq k$.

Observation 6.1 applies to this family. □

15. $F_k =$ **I∃-ke**: $G = (V_G, E_G) \in$ **I∃-ke** if there exists a set E_K of edges, $E_K \subseteq E_G, |E_K| = k$, such that $G' = (V_G, E_G - E_K) \in F$.

Theorem 6.15 **I∃-ke** *is immersion closed, and* $F \subseteq$ **I∃-ke**.

Proof Let E_K denote the set of k edges whose removal from G yields a $G' = (V_G, E_G - E_K) \in F$. If H is a subgraph of G, then it is easy to see that $H' = (V_H, E_H - (E_K \cap E_H))$ is a subgraph of G'.

Suppose H was formed by lifting $(u,v),(v,w)$ (adding (u,w)). If $E_K \cap \{(u,v),(v,w)\} = \emptyset$ then $H' = (V_H, E_H - E_K)$ is immersed in G'. Otherwise, a set of edges of size k (or less) can be removed from H by removing E_K, which contains at least one non-existent edge from H, as well as (u,w). The resulting graph is a subgraph of G'.

Observation 6.2 applies to this family. □

16. F_k = **I∀-ke**: $G = (V_G, E_G) \in$ **I∀-ke** if for all sets E_K of edges, $E_K \subseteq E_G$, $G' = (V_G, E_G - E_K) \in F$.

Theorem 6.16 **I∀-ke** *is immersion closed, and* $F \subseteq$ **I∀-ke**.

Proof If $H = (V_G - v, E_G)$, then $H' = (V_G - v, E_G - E_K)$, where E_K is any set of k edges from H, is a subgraph of $G' = (V_G, E_G - E_K)$. If $H = (V_G, E_G - (x,y))$ then $H' = (V_G, E_G - (x,y) - E_K)$ is a subgraph of $G' = (V_G, E_G - E_K)$.

Suppose H was formed by lifting $(u,v),(v,w)$ (adding (u,w)), and consider any set E_K of k edges in H. If $(u,w) \in E_K$, then H' is a subgraph of $G' = (V_G, E_G - (E_K - \{(u,w)\} \cup \{(u,v)\}))$, which is in F. If $(u,w) \notin E_K$, then H' is immersed in $G' = (V_G, E_G - E_K)$.

Observation 6.2 applies to this family. □

Observation 6.3 applies to this family.

The theorems presented in this section guarantee only the existence of polynomial-time decision algorithms for the closed families of graphs. In practice, of course, what is usually required is not only a yes answer, but proof in the form of a specific solution.

For nine of the families that are closed under their corresponding orders, a solution can be easily and quickly constructed. They are the families for which a solution may be constructed by choosing *any* k vertices or edges to be added or taken away. These families are: **M∃+kv**, **M∀+kv**, **M∀-kv**, **M∀+ke**, **M∀-ke**, **I∃+kv**, **I∀+kv**, **I∀+ke**, and **I∀-ke**.

The other two closed families, **M∃-kv** and **I∃-ke**, appear to be the only two of the sixteen that are of theoretical and potential practical interest. For both of these families, a solution can be constructed via self-reduction.

In [BFL] a self-reduction algorithm is presented to show that a solution to **M∃-kv**, in the form of the construction of S, can be obtained in $O(|V|^4)$ time. We note that a somewhat simpler self-reduction can be performed, yielding the same time bound, by employing "related" oracles for **M∃-nv**, where n takes on the successive values $k, k-1, ..., 0$. The same approach yields a search algorithm for **I∃-ke**. In [FL4], a general $O(n\ log\ n)$ self-reduction technique called *scaffolding* is introduced. Scaffolding also uses related oracles, but is primarily applicable to layout permutation problems.

Table 6.1 summarizes closure-preserving operator results.

6.1.2 Other Circuit Partitioning Problems

In addition to practical generalizations of the MDGP problem, there exist other combinatorial problems of relevance to FPGA partitioning. See [Go] for a sampling of such problems, along with many open questions. In this subsection, we describe some new results for one of these, Minimum Degree Cut, which is defined as follows.

Instance: a graph $G = (V, E)$, some of whose vertices are terminals, and an integer d.

Question: Does G have a terminal partition in which each subset has degree d or less?

Recall that a *terminal partition* of G is a partition in which each subset contains

Table 6.1: Summary of closure-preserving operators

F_k	Closure	Notes
1. **M∃+kv**	yes	$F_k \subseteq F$
2. **M∀+kv**	yes	$F_k \subseteq F$
3. **M∃-kv**	yes	$F_k \supseteq F$
4. **M∀-kv**	yes	$F_k \supseteq F^*$
5. **M∃+ke**	no	$F_k \subseteq F$
6. **M∀+ke**	yes	$F_k \subseteq F$
7. **M∃-ke**	no	$F_k \supseteq F$
8. **M∀-ke**	yes	$F_k \supseteq F^*$
9. **I∃+kv**	yes	$F_k \subseteq F$
10. **I∀+kv**	yes	$F_k \subseteq F$
11. **I∃-kv**	no	$F_k \supseteq F$
12. **I∀-kv**	no	$F_k \supseteq F^*$
13. **I∃+ke**	no	$F_k \subseteq F$
14. **I∀+ke**	yes	$F_k \subseteq F$
15. **I∃-ke**	yes	$F_k \supseteq F$
16. **I∀-ke**	yes	$F_k \supseteq F^*$

$*$ Only finite number of graphs $\in F_k, \notin F$

at most one terminal from G.

This problem has polynomial-time complexity ([Go]), which makes the fact of its immersion closure less interesting. No practical algorithm is known, however. We present some results about the obstruction set of the fixed-parameter version of Minimum Degree Cut (MDC(d)). It is unlikely that an obstruction-based algorithm will be practical for this problem. Nevertheless, knowledge gleaned from the study of these sets may still be useful. This was the case with MDGP(k,d), in which study of the obstruction set paved the way to linear-time search and decision algorithms.

Observation 6.4 *Every obstruction to MDC(d) contains at least* 2 *terminal vertices.*

Observation 6.5 *No obstruction to MDC(d) contains an edge with multiplicity exceeding* $d+1$.

Observation 6.6 *A graph consisting of a single non-terminal vertex, with three terminal neighbors, is an obstruction for MDC(1).*

Lemma 6.1 *No obstruction to MDC(d) contains a non-terminal vertex with fewer than three neighbors.*

Proof First observe that there can be no obstruction with an isolated nonterminal.

Denote by H some obstruction to MDC(d). Suppose some non-terminal $v \in V_H$ has exactly one neighbor, w. $H' = H - \{v\}$ has a terminal partition P in which w belongs to some subset S. But adding v to S in P yields a terminal partition of H.

Suppose some non-terminal $v \in V_H$ has only two neighbors, u and w. H' obtained by replacing $\{u,v\},\{v,w\}$ with $\{u,w\}$ has a terminal partition P.

If u and w are in the same subset S of P, we can obtain a terminal partition of H by adding v to S. So we must have $u \in S_1, w \in S_2$ for S_1, S_2 of P. But then $P - S_1 \cup (S_1 \cup v)$ is a terminal partition of H. $\square$

Observation 6.7 *Any connected graph consisting only of terminal vertices is a "no" instance of MDC(d) if any of the terminals is of degree greater than d.*

Definition 6.1 *A d-star terminal graph S_d is defined as follows:*

1. *There is one terminal vertex v of degree $d + 1$.*
2. *Every neighbor of v is a terminal, with no other neighbors.*
3. *S_d is connected.*

Lemma 6.2 *Any d-star terminal graph S_d is an obstruction to MDC(d).*

Proof By Observation 6.7, S_d is a "no" instance of MDC(d). We only need to show that S_d is minimal. We note that the only situation in which a vertex other than v could have degree more than d is that in which the graph consists of only two terminals connected by $d + 1$ edges. Any immersion operation, then, results in all terminals having degree less than d, and the resulting graph is a "yes" instance of MDC(d). □

Figure 6.5 shows the set of S_d obstructions for MDC(4). Note that all vertices in Figure 6.5 are assumed to be terminals.

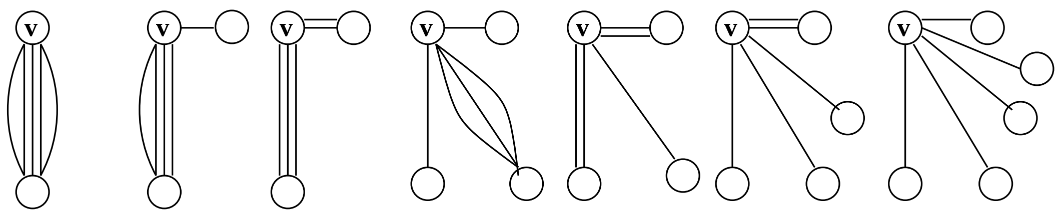


Figure 6.5: Some obstructions to MDC(4)

Theorem 6.17 *The size of the obstruction set to MDC(d) is at least exponential in* $\sqrt{d}$.

Proof The proof hinges on counting the number of S_d obstructions. Such graphs can be put into correspondence with sets of positive integers totalling $d+1$, and the number of these is exponential in $\sqrt{d}$ ([Ro]). □

6.1.3 Faster Immersion Testing

We have seen that WQO theory provides a powerful tool for proving polynomial-time decidability. In the case of the minor order, every immersion-closed family F automatically has an asymptotically fast algorithm ($O(n^3)$ at worst). In the case of the immersion order, the best we can guarantee is $O(n^{h+3})$, where h is the order of the largest member of the obstruction set for F.

These complexity orders are a consequence of the time required to decide whether a fixed graph H is a minor of (immersed in) a given graph G. Should an algorithm be found that could perform immersion testing faster, this would translate into a faster known algorithm for all immersion-closed families.

At this time, it is unknown whether or not there exists a $O(n^k)$ (k any fixed constant) algorithm for deciding immersion containment, in which k does not depend upon the obstruction set for F.

6.1.4 Other Issues

In Chapters 2, 3, and 4, we investigated the complexity of MDGP, FPGA Minimization, and Delay Minimization, when restricted to certain graph families. These results for MDGP and FPGA Minimization were summarized in Table 3.1. The complexity of Delay Minimization, under these restrictions, is the same as that for FPGA Minimization.

The entries marked "unknown" in the table are open questions. Additionally, the complexity of many of the problems in Chapter 5 when restricted to particular graph families, is unknown. Since many of these generalizations have potential applications to circuit partitioning, and because it is sometimes the case that assumptions may be made about the structure of real circuits, these issues are worthy of further study.

There are also many open questions with respect to partitioning problems over hypergraphs, and for heterogeneous partitioning. We have shown some results for the fundamental problem in these settings. We have no positive results for FPGA Minimization, however, except for the heuristic which does works on hypergraphs.

6.2 Practical Directions

One area of potential promise for future research is that of more practical heuristics, especially in the area of timing. Hardware technology continues to advance rapidly, and the software for realizing rapid prototype systems on this hardware must keep pace.

There is an open question related to critical path compression, that encompasses both theory and practice. The question is, under what circumstances can compression of the current critical path result in the creation of a new, worse critical path? If this information could be known in advance, even some of the time, critical path compression could be made more efficient.

Code replication is a significant topic in circuit design. In fact, many researchers consider it an essential tool ([TSO]), without which near-optimal delays are almost impossible. A possible project would be to incorporate replication into the critical path compression technique.

The CPU time for our critical path compression algorithm increases significantly when the current critical path is very long. One way to deal with this shortcoming would be to ignore target sequences that exceed some predefined length. The reasoning behind this is that very long sequences might be unlikely candidates for reassignment. The amount of time spent investigating reassignment of very long target sequences is perhaps not justified.

Other areas of potential research include, but are not limited to, the following:

- More effective clustering methods for partitioning, that incorporate timing concerns.
- Expanded iterative improvement techniques for delay optimization.
- Improved implementation strategies to make the code itself more efficient.
- Scalable strategies that can handle extremely large circuits, or circuits with extremely long critical paths.

6.3 Conclusion

In summary, we have examined a set of partitioning problems that have relevance to VLSI design, particularly FPGA partitioning. We have explored theoretical properties of these problems, and have found some results concerning their tractability. We have seen that many of these problems are in $\mathcal{P}$ when all parameters are fixed, and many have been shown for the first time to be solvable in linear time. We have learned a great deal about the immersion order obstruction sets for some of these families, and have discovered that many of these sets are computable.

From this theoretical perspective, we have also explored more practical algorithmic possibilities. A promising area of new research is that of partitioning for delay minimization, and we have developed a new iterative improvement technique toward this end. Many unresolved problems and open issues have been discovered along the way.

Bibliography

Bibliography

[AK] C. J. Alpert and A. B. Kahng, "Recent Directions in Netlist Partitioning: a Survey," *INTEGRATION, the VLSI Journal* 19 (1995), 1–81.

[Al] C. J. Alpert, "The ISPD98 Circuit Benchmark Suite," preprint available from http://vlsicad.cs.ucla.edu/˜cheese/ispd98.html.

[BB] G. Brassard and P. Bratley, *Algorithmics Theory and Practice*, Prentice Hall, Englewood Cliffs, New Jersey, 1988.

[Be] Benchmark directory pub/Benchmark_dirs/Partitioning93. Send email to benchmarks@cbl.ncsu.edu for details on ftp access.

[BFL] D. J. Brown, M. R. Fellows and M. A. Langston, "Polynomial-Time Self-Reducibility: Theoretical Motivations and Practical Results," *International Journal of Computer Mathematics* 31 (1989), 1–9.

[BGLR] H. D. Booth, R. Govindan, M. A. Langston and S. Ramachandramurthi, "Fast Algorithms for K_4 Immersion Testing," Accepted for publication in *Journal of Algorithms*. (A preliminary version of this paper was released as a technical report.)

[BKK] F. Brglez, R. Kuznar and K. Kozminski, "Cost Minimization of Partitions into Multiple Devices," *Proceedings of the 30th ACM/IEEE DAC* (1993), 315–320.

[BL] D. Bienstock and M. A. Langston, "Algorithmic Implications of the Graph Minor Theorem," *Handbooks in Operations Research and Management Science*, Elsevier Science B.V., 1995, 481–502.

[Bod] H. L. Bodlaender, "A Linear Time Algorithm for Finding Tree-decompositions of Small Treewidth," *Proceedings, 25^{th} Annual ACM Symposium on Theory of Computing* (1993), 226–234.

[Bou] D. W. Bouldin, private communication.

[CD] K. Cattell and M. Dinneen, "A Characterization of Graphs with Vertex Cover up to Five," *Proceedings ORDAL '94* (1994), 86–99.

[CLCDL] N. Chou, L. Liu, C. Cheng, W. Dai and R. Lindelof, "Local Ratio Cut and Set Covering Partitioning for Huge Logic Emulation Systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* Vol. 14, No. 9 (1995), 1085–1091.

[FF] J. Feigenbaum and L. Fortnow, "Random Self-reducibility of Complete Sets," *SIAM Journal on Computing* 22 (1993), 994–1005.

[FL1] M. R. Fellows and M. A. Langston, "Nonconstructive Advances in Polynomial-Time Complexity," *Info. Proc. Letters* 26 (1987), 157–162.

[FL2] M. R. Fellows and M. A. Langston, "Nonconstructive Tools for Proving Polynomial-Time Decidability," *J. of the ACM* 35 (1988), 727–739.

[FL3] M. R. Fellows and M. A. Langston, "Fast Search Algorithms for Layout Permutation Problems," *International Journal of Computer Aided VLSI Design* 3 (1991), 325–342.

[FL4] M. R. Fellows and M. A. Langston, "On Well-Partial-Order Theory and its Application to Combinatorial Problems of VLSI Design," *SIAM J. Disc. Math.* 5 (1992), 117–126.

[FM] C. M. Fiduccia and R. M. Mattheyses, "A Linear Time Heuristic for Improving Network Partitions," *Proceedings ACM/IEEE Design Automation Conference,* (1982), 175–181.

[GGL] R. L. Graham, M. Grötschel, L. Lovász, *Handbook of Combinatorics*, The MIT Press, Cambridge, Massachusetts, 1995, 390–391.

[GJ] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman and Company, New York, 1979.

[GLR] R. Govindan, M. A. Langston, and S. Ramachandramurthi, "A Practical Approach to Layout Optimization," *6th International Conference on VLSI Design* (1993), 222–225.

[Go] R. Govindan, "Algorithmic Methods for Circuit Layout and Partitioning," Ph.D. Dissertation, University of Tennessee, 1998.

[H] F. Harary, *Graph Theory*, Addison-Wesley Publishing Company, 1972.

[HK] D. J.-H. Huang and A. B. Kahng, "Multi-Way System Partitioning into a Single Type or Multiple Types of FPGAs," *FPGA '95* (1995), 140–145.

[HP] F. Harary, E. M. Palmer, *Graphical Enumeration*, Academic Press New York and London, 1973.

[KBK] R. Kuznar, F. Brglez and K. Kozminski, "Cost Minimization of Partitions into Multiple Devices," 30^{th} *ACM/IEEE Design Automation Conference* (1993), 315–320.

[KiL] N. G. Kinnersley and M. A. Langston, "Obstruction Set Isolation for the Gate Matrix Layout Problem," *Discrete Applied Mathematics,* 54 (1994), 169–213.

[KL] B. W. Kernighan and S. Lin, "An Efficient Heuristic Procedure for Partitioning Graphs," *Bell System Technical Journal,* 49(2) (1970), 291–307.

[KS] C. Kim and H. Shin, "A Performance-Driven Logic Emulation System: FPGA Network Design and Performance-Driven Partitioning," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 15(5) (1996), 560–568.

[KUW] R. M. Karp, E. Upfal and A. Wigderson, "The Complexity of Parallel Search," *Journal of Computer and Systems Sciences* 36 (1988), 225–253.

[La1] M. A. Langston, "WQO-Based Methods," *International Workshop on Combinatorial Methods for Circuit Design,* Schloss Dagstuhl, Germany, October, 1993.

[La2] M. A. Langston, private communication.

[Le] T. Lengauer, *Combinatorial Algorithms for Integrated Circuit Layout*, John Wiley and Sons, 1990.

[LP] M. A. Langston and B. C. Plaut, "On Algorithmic Applications of the Immersion Order," *Discrete Mathematics,* 182 (1998) 191–196.

[MR] H.-G. Martin and W. Rosenstiel, "A Comparing Study of Technology Mapping for FPGA," *Proceedings. Design, Automation and Test in Europe,* 98EX123

(1998) 939–940.

[NS] K. Roy-Neogi and C. Sechen, "Multiple FPGA Partitioning with Performance Optimization," *Proceedings, FPGA '95* (1995) 146–152.

[OD] J. V. Oldfield and R. C. Dorf, *Field Programmable Gate Arrays*, John Wiley & Sons, Inc., 1995.

[Ro] H. E. Rose, *A Course in Number Theory*, second edition, Clarendon Press, Oxford, 1994.

[RS1] N. Robertson and P.D. Seymour, "Graph Minors V. Excluding a planar graph," *J. Combin. Theory Ser. B* 41 (1986), 92-114.

[RS2] N. Robertson and P.D. Seymour, "Graph Minors IV. Tree-width and Well-quasi-ordering," *J. Combin. Theory Ser. B* 48 (1990), 227–254.

[RS3] N. Robertson and P. D. Seymour, "Graph Minors XIII. The Disjoint Paths Problem," *J. Combin. Theory Ser. B* 63 (1995), 65–110.

[RS4] N. Robertson and P. D. Seymour, "Graph Minors XVI. Wagner's Conjecture," to appear.

[RW] R. Rajaraman and D. F. Wong, "Optimal Clustering for Delay Optimization," *Proceedings, 30th ACM/IEEE Design Automation Conference* (1993) 309–314.

[Sc] C. P. Schnorr, "Optimal Algorithms for Self-reducible Problems," *Proceedings, 1976 International Conference on Automata, Programming and Languages* (1976), 322–337.

[Se] P. D. Seymour, private communication.

[ST] P. Sawkar and D. Thomas, "Multi-Way Partitioning for Minimum Delay for Look-up Table Based FPGAs," *Proceedings, 32nd Design Automation Conference* (1995), 201–205.

[SY] S. M. Sait and H. Youssef, *VLSI Physical Design Automation*, IEEE Press, 1995.

[TSO] N. Togawa, M. Sato, and T. Ohtsuki, "A Circuit Partitioning Algorithm with Path Delay Constraints for Multi-FPGA Systems," *IEICE Trans. Fundamen-*

tals E80-A, No. 3 (1997), 494–505.

[VM] J. Villasenor and W. H. Mangione-Smith, "Configurable Computing," *Scientific American*, July 1998.

[WC] Y.-C. Wei and C.K. Cheng, "Ratio Cut Partitioning for Hierarchical Designs," *IEEE Transactions Computer-Aided Design* Vol. 10, No. 7 (1995), 911–921.

[We] U. Weinmann, *More FPGAs*, Abingdon EE&CS Books, 1994.

[Wi] H. S. Wilf, *Algorithms and Complexity*, Prentice-Hall, Inc., 1986.

[WK] N. Woo and J. Kim, "An Efficient Method of Partitioning Circuits for Multiple-FPGA Implementation," *30th ACM/IEEE Design Automation Conference* (1993), 202–207.

[WKMKY] S. Wakabayashi, H. Kusumoto, H. Mishima, T. Koide, and N. Yoshida, "Gate Array Placement Based on Mincut Partitioning with Path Delay Constraints," *Proceedings, ISCAS'93* (1993), 2059–2062.

[X] Xilinx, Inc., *The Programmable Logic Data Book*, San Jose: Xilinx, 1994.

Vita

Barbara Catherine (Johnson) Plaut was born in La Crosse, Wisconsin, on November 24, 1952. In 1986, she obtained a Master's Degree in Computer Science from the University of Kentucky in Lexington. The next three years were spent working for SofTech, Inc., in Alexandria, Virginia. From 1989 until 1992 she stayed at home with her two young daughters, where she produced and sold a nutritional software package. In 1992 she began her pursuit of the PhD degree in computer science at the University of Tennessee in Knoxville.