PLACEMENT PROBLEMS ARISING FROM AUTOMATIC LOGIC COMPILATION

William Wei-Lian Lin

(Thesis)

CS-TR-201-89

June 1989

PLACEMENT PROBLEMS ARISING FROM AUTOMATIC LOGIC COMPILATION

William Wei-Lian Lin

A DISSERTATION
PRESENTED TO THE FACULTY
OF PRINCETON UNIVERSITY
IN CANDIDACY FOR THE DEGREE
OF DOCTOR OF PHILOSOPHY

RECOMMENDED FOR ACCEPTANCE
BY THE DEPARTMENT OF
COMPUTER SCIENCE

June 1989

Acknowledgments[†]

I'd like to thank my advisor, Andrea LaPaugh, for her guidance and suggestions, for her role in making this dissertation a reality.

I'd like to thank my readers, Bernard Chazelle and Ken Steiglitz, for taking the time to look over my thesis and make comments.

Thanks also to Mihalis Yannakakis for suggesting the algorithm described in Section 3.3. For the work in Chapter 6, I'd like to thank my collaborators, Luen Heng, Andrea LaPaugh, and Ron Pinter.

---

† This work was supported in part by DARPA Contract N00014-82-K-0549 and NSF Grants MCS-8202594 and MIP-8619335.

To Nora

TABLE OF CONTENTS

Placement Problems Arising from Automatic Logic Compilation

William W. Lin

Princeton University

ABSTRACT

Automatic logic compilation is the process of taking an input description consisting of Boolean logic equations and outputting an IC layout mask description implementing the equations. There are a number of problems that must be solved by the compiler. In this thesis, we shall discuss some placement problems that arose during the building of a logic compiler, the *Weinberger Array Generator*.

The first problem is *Linear Arrangement with Critical Paths*. The input is a graph G and a set of critical paths (paths are subsets of G) with corresponding limits. The objective is to place the nodes of G in a straight line so that the length of any critical path is less than or equal to the limit for that path. In some versions of this problem, the main goal is to minimize the cutwidth of the linear arrangement while satisfying the path limits. This problem is called *Mincut Linear Arrangement with Critical Paths*. In the general case, both of the above problems are NP-complete. We analyze restrictions on the problems to determine the complexity of these restricted problems. Some algorithms are given for solving special cases of the two general problems.

The second problem is *Cluster Placement in an Array of Gates* (*CPAG*). The input

consists of an ordered set of gates. Each gate consists of a set of clusters, where each cluster is a group of signals that must remain together. The objective is to place the signals so that clusters do not mix and the total routing space needed is minimized. Since the general problem is NP-complete, we present some heuristics to solve *Cluster Placement in an Array of Gates*, along with experimental results.

The next problem we consider is the restriction of *CPAG* to a single pair of gates and the channel between those gates. This problem is still NP-complete, so we consider a heuristic for this problem that provides a lower bound for the channel density and splits the problem into smaller subproblems. Each subproblem is constrained such that the top terminals are in fixed positions and each net may have at most one top terminal and one bottom terminal. We present an algorithm that finds a placement that achieves the optimal possible density for each subproblem.

Finally, we consider a situation where the terminals' vertical alignments are fixed, but we may add empty columns to a channel. We describe how the width of the channel is lower bounded by a metric called *smooth-flux*. The goal is to reduce the *smooth-flux* to a given value. We present an algorithm that calculates how many columns to add and where to place them.

Chapter 1
Introduction

The task of designing a digital integrated circuit (IC) is a difficult one. Hand designs are fine for small systems; but large systems, especially very large scale integration (VLSI) systems, are usually complex enough to confound human-only efforts to construct them efficiently. So, chip builders must turn to computers for help.

In recent years, a vast amount of work has been done on computer-aided design for VLSI. Analysis tools exist for such tasks as simulation of switching circuits (e.g., *MOSSIM* [Bryant]), timing estimations (e.g., *CRYSTAL* [Ousterhout1]), and design rule checking (e.g., *LYRA* [Mayo]). Design tools exist for placement and routing problems such as channel routing [Rivest] [Yoshimura] [Burstein]. For the custom chip designer, there are a number of programs for specifying layout masks. These range from procedural languages like *ALLENDE* [Mata] to interactive graphics packages like *MAGIC* [Ousterhout2]. For a survey of recent work on computer-aided design for layout, peruse [Ohtsuki].

1.1. Logic Compilation and Weinberger Arrays

One particular area of interest is logic compilation, which is the process of taking an input description consisting of Boolean logic equations and outputting an IC layout mask description implementing the equations. This difficult compilation process is usually decomposed into four subtasks: function compilation, logic optimization, topological optimization, and layout generation. Function compilation takes the input Boolean

equations and converts them into an appropriate internal representation. An example of one such representation is a truth table. The logic optimization phase implements the functions under two main criteria. First, we may use only the types of gates allowed by the target technology (this may be considered part of function compilation [Rowen]); second, we want to use as few gates as possible. The third phase, topological optimization, places the gates produced by the logic optimization phase. The goal of this phase is to minimize the cost associated with the topology of the layout. There may be restrictions on the allowed topologies, depending on the paradigm one uses. Finally, the layout generation produces an IC mask from the description provided by the first three phases.

The target technology (or technologies) can greatly affect the phases of compilation. We assume that we are designing for nMOS or CMOS technology, with two metal layers for routing. We also assume that all wire segments must either be vertical or horizontal.

When building an automatic logic compiler, one of the first questions that must be addressed is what template to use for the layout. Two well-known paradigms are the programmable logic array (PLA) and the gate array.

A PLA implements the Boolean equations using two levels of logic. The functions are rendered into disjunctive normal form (DNF). Operations are performed to decrease the number of conjunctive terms used. These minimized DNF equations are then translated directly into two levels of gates, commonly known as the ''AND'' and ''OR'' planes, in a particular array layout. For a treatment of these ideas, see [Brayton].

For gate array implementations, the functions must be built from a small set of allowable gates. Typically, these gates have restricted fan-in (about 6) and may not be complex. These gates are placed within an array grid, and the common signals are wired together. For a general source on gate arrays, one might peruse [Hollis]

A less widely-used paradigm, but one that is quite interesting, is the Weinberger array [Weinberger]. The basic general structure of a one-dimensional Weinberger array is shown in figure 1-1. Logic ''1'' (= Vdd) and ''0'' (= GND) signals traverse the top

and bottom of the array, respectively. The active elements (*gates*), represented by dashed boxes in the figure, are placed one after another, in a straight horizontal line. Input and output signals may enter or leave the array from any of the four rectilinear sides. The signals are routed *through* the gates, where they may form transistors (e.g., *In1* in *gate 1*), just pass through (e.g., *In2* in *gate 2*), continue (e.g., *In2* forms a transistor in *gate 1*, then continues through the gate), or terminate (e.g., *In1* stops in *gate 1*).

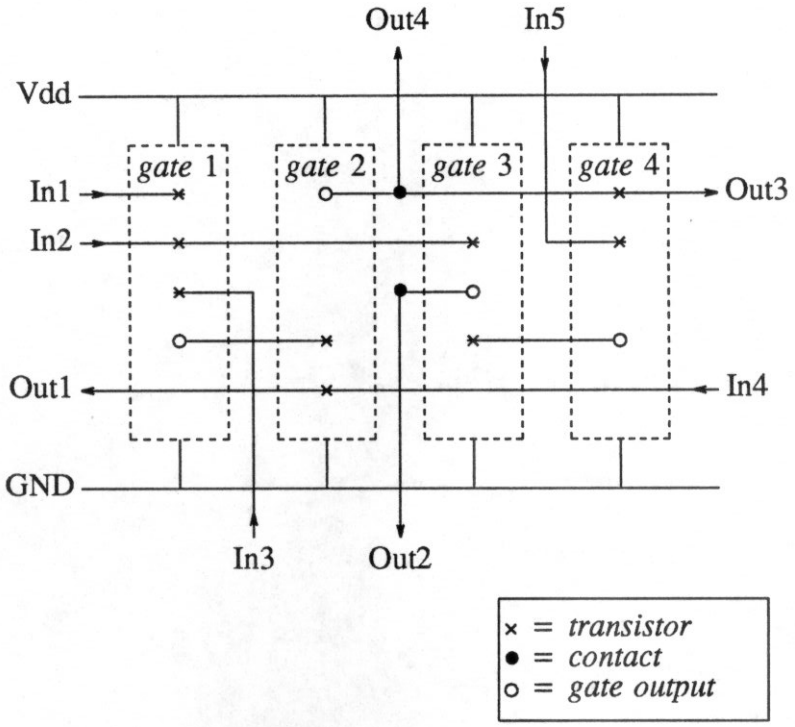


figure 1-1 Weinberger array: general structure

The Weinberger array paradigm offers certain advantages over other structures. Unlike PLAs, which usually only allow simple gates such as NORs and NANDs, the Weinberger array allows complex gates such as a NOR-of-ANDS (see fig. 1-2). Although some gate array systems may allow complex gates, the allowed gates are of restricted size; contrariwise, there is no set limit on the size of Weinberger array gates (in practice, however, gate sizes will be limited in order to reduce propagation delays). Also, unlike PLAs, the Weinberger array structure allows the use of multi-level logic.

Multi-level logic allows signals to propagate through more than two levels of gates. For example, in figure 1-1, we have the propagation path *In* 1 → *gate* 1 → *gate* 2 → *gate* 4 → *gate* 3 (*Out* 2). The use of complex gates and multi-level logic allows one to lay out the circuit using fewer gates; thus, the area needed should decrease.

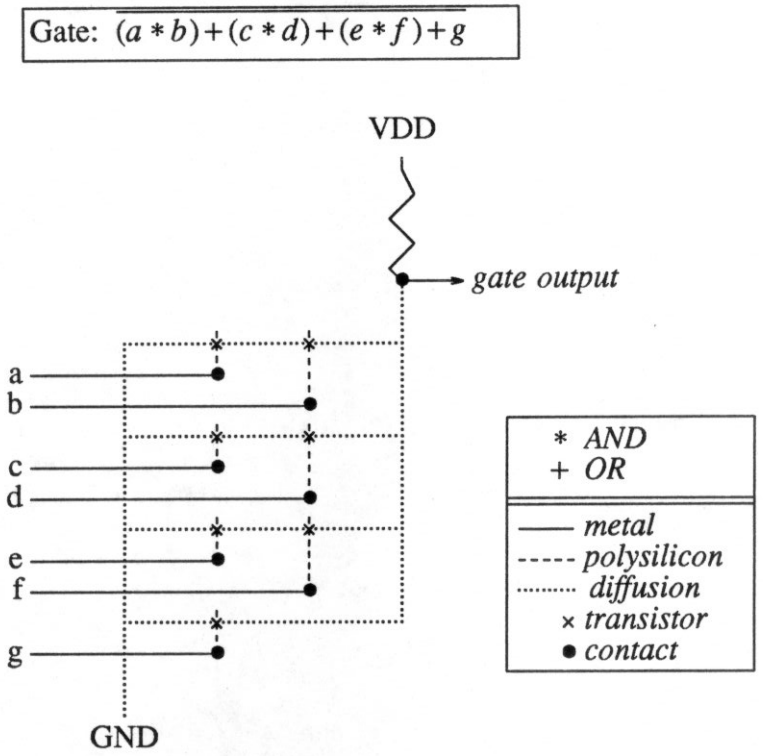


figure 1-2 Example of complex gate: a NOR-of-ANDs (nMOS technology)

Of course, there are possible disadvantages for Weinberger array layouts. One disadvantage arises from the flexibility of having complex gates and multi-level logic. Since the complex gates may have larger signal transition delays and the signal paths may be longer than for two-level logic, the signal delays may be longer. This is not always the case, though. If we are able to use many fewer gates, and the gates have smaller fan-in and fan-out, the propagation delay may actually decrease. [Rowen] One other disadvantage of the one-dimensional Weinberger array is that the aspect ratio may

become very bad. There are at least two fixes to the aspect ratio problem. One is to "fold" the array after a certain number of gates have been placed (see figure 1-3). Another fix is to allow more than one gate per column, as mentioned by Rowen [Rowen]. Of course, these variations create other issues that must be handled.

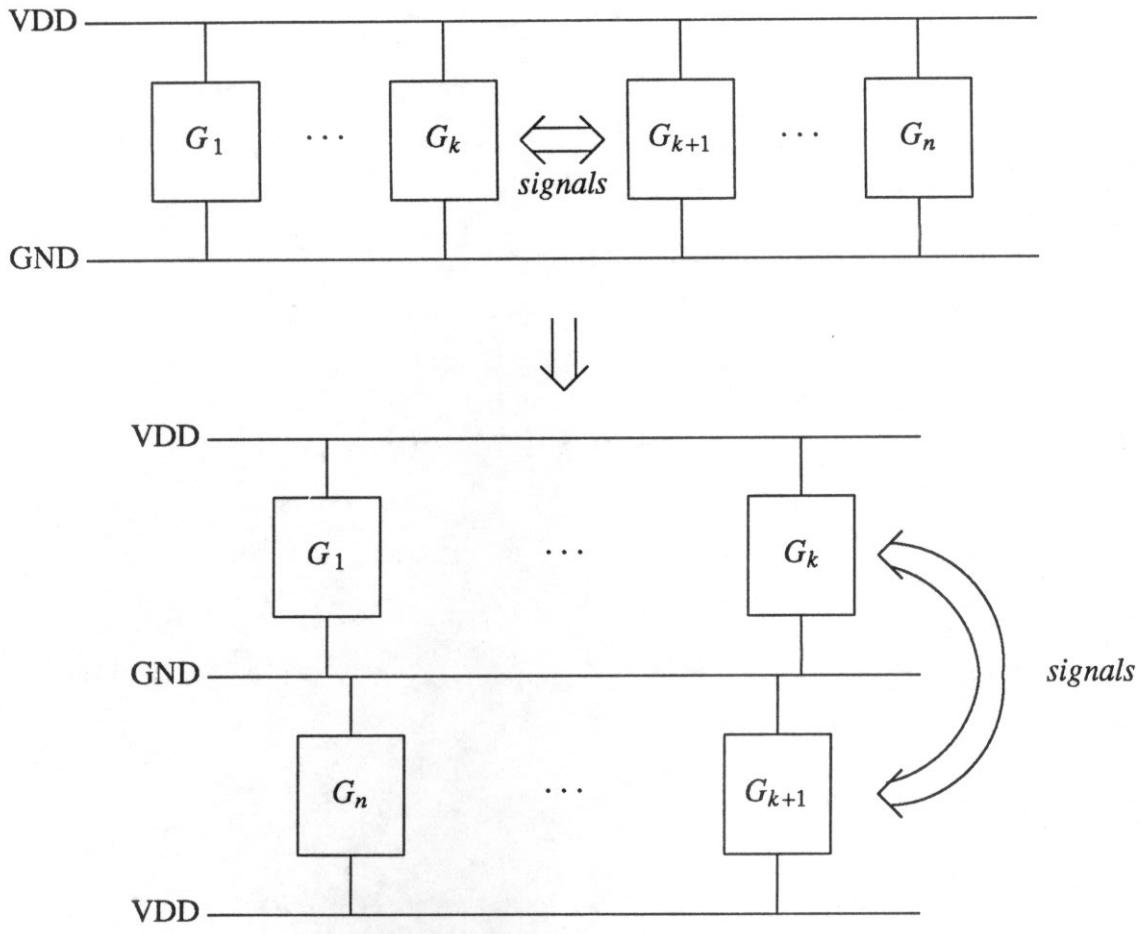


figure 1-3 Example of array folding

The types of gates allowed and the structure of those gates may have a large impact on the performance of the compiler. On the one hand, using more kinds of gates theoretically allows the logic optimization to perform better. On the other hand, the more kinds of gates there are, the harder it is to find efficient layouts for all of them. In most of the previous work with Weinberger arrays, implementers chose to restrict tightly the set of allowed gates. The choices were limited to some subset of NANDS, NORS, ANDS, and

ORS [Siskind] [Southard] [Sabety] [JohnsonS].

Lin, Yeh, and LaPaugh chose to use NOR-of-ANDS, NAND-of-ORS, NANDS, NORS, and NOTS in their implementation of the Weinberger Array Generator (*WAG*) [Lin]. They added a further restriction that no gate may have a long AND chain, since a long series of transistors increases the delay propagation. Their structure for a NOR-of-ANDS gate is shown in figure 1-2. *WAG*'s choice of allowing a more varied mixture of gates led to certain hurdles in the placement and routing for an array.

1.2. Problems arising from work on *WAG*

The problems we concentrate on occur in the topological optimization phase of the logic compilation. After the logic minimization phase decides what gates to use to implement the Boolean equations, the normal next step is to order the gates in a line. We may shift the problem into a graph theoretical framework by representing each gate and each signal as a vertex. Then, if a signal or gate output forms a transistor in another gate, there is an edge between the corresponding vertices in the graph. (See figure 2-1.) Placing the vertices in a line is called *linear arrangement*. Formally, a linear arrangement is a function $f : V \rightarrow \{1, \ldots, |V|\}$.

When deciding which linear arrangement to use, we may have several goals in mind. One possible goal is to minimize the total length of wire used along certain "critical" signal propagation paths. That is, if there is a certain path that is critical to the propagation delay for the whole circuit, we would like that path to be as short as possible. This task may be abstracted to a graph theoretic problem that we call *Linear Arrangement with Critical Paths* (the same problem has been studied by Simonson under the name *Routing with Critical Paths* [Simonson].) Another possible goal is to minimize the number of signals that must run between any two consecutive gates, since this should minimize the number of horizontal tracks needed for routing signals. This goal translates into *cutwidth minimization* for linear arrangements of graphs. In Chapter 2 of this thesis,

we look at the problem of *Linear Arrangement with Critical Paths*. We give the complexity for the general problem and determine the complexity for special cases of the problem. We examine in Chapter 3 the effects of adding cutwidth minimization as an additional goal.

Once the gates are ordered, we must decide the relative ordering of signals into each gate. For the case where only simple gates are used, the problem of ordering is not hard. An optimal solution is given by a greedy track assignment, such as the one described in [Ullman]. But if complex gates are allowed, it becomes harder. Refer back to figure 1-2. Note that the seven transistors are all created on only *two* vertical tracks. This is possible because the signals that are ''ANDed'' together (e.g., ''a'' and ''b'') are adjacent to each other when they enter the gate. If they weren't, then it would require more than two vertical tracks to wire the gate.

Note that not all NOR-of-ANDS gate structures require certain signals to be adjacent (see figure 1-4). Here it is not important in what order the signals arrive; and since the order of the signals does not matter, a ''greedy'' algorithm will pack the signals into as few tracks as possible. However, there is a disadvantage to the structure shown in figure 1-4. This NOR-of-ANDS gets very wide if there are a number of terms to be NORed together; also, it would need many long vertical diffusion runs. Also, while the alternate NOR-of-ANDS uses fewer horizontal tracks than the one in figure 1-2, the excess of vertical tracks is a greater worry.

WAG's gate structures are not unique in requiring adjacency constraints for signals. For example, Schlag, et. al., require variables to be in the proper track for alignment in cascode-switch macros [Schlag]. In order to make our work a bit more general, we abstract the idea of adjacency. We assume that a signal must use the same horizontal track to enter and leave a gate; and we assume that a gate requires certain signals to be grouped together when they enter the gate. For simplicity, we assume that the gate output must use a separate track from the incoming signals. (This is not necessarily the

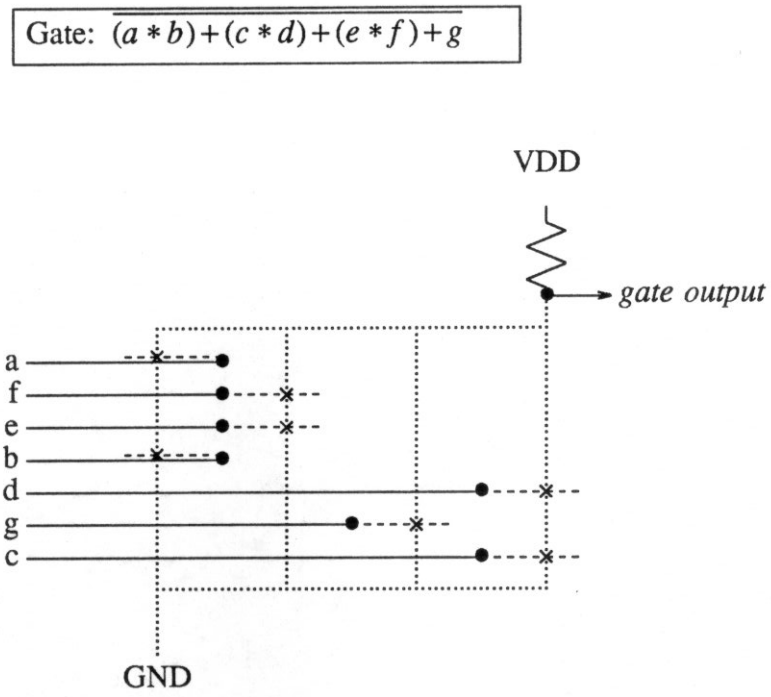


figure 1-4 Alternative NOR-of-ANDs gate (nMOS technology)

case. If a signal terminates within a gate, it might possibly be able to share a track with the gate output.) Then, we define a *cluster* to be a set of signals, none of which may be intermixed with signals from another cluster.

In our problem conversion from the Weinberger array realm, gates become sets of clusters. The task is, for each gate, to order (place) the gate's signals so that no clusters intermix and the amount of space required for routing between gates is minimized. Since identical signals must be wired together, we may need some extra vertical tracks between the gates to perform the routing. (See figure 1-5.)

The converted problem is *Placement of Clusters in an Array of Gates*, and it will be examined in Chapter 4 of this thesis. We discuss some of the issues to consider in

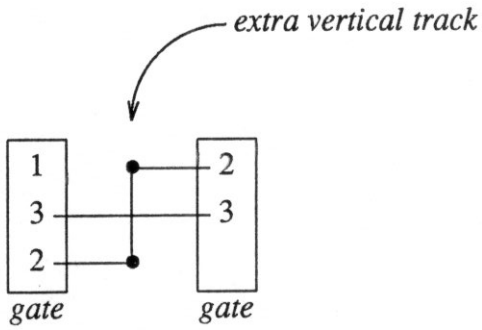


figure 1-5 Example where routing requires "extra" vertical track

designing a heuristic to solve this problem. Then, we present some experimental results with heuristics that were implemented. The methods used are "greedy", "randomized" iterative improvement, simulated annealing, and successive partitioning.

In Chapter 5, we consider a restriction on the cluster placement problem. In particular, we allow an array to consist of only two gates. We call our problem *Single Channel Placement*. The problem is to position the terminals in order to minimize the number of extra vertical columns needed to connect the signals. Note that placing the terminals is only half the task. Given a positioning of the terminals into two rows, we want to be able to find an optimal routing of the signals in the space between the two rows.

The problem of finding an optimal wiring of signals between two rows of terminals is called *channel routing*. Following convention, the input consists of the *top* and *bottom* rows, whose terminals must be connected across an intervening horizontal channel. Optionally, there may be specified nets that must enter the channel from the *left* or leave the channel from the *right*. The object is to connect all terminals that belong to a common net, for each net, using as few horizontal *tracks* as possible. The smallest number of tracks that can possible be used is called the channel's *width*.

How well one can perform a channel route depends on the routing model one uses. A model specifies the number of layers one may use to route signals, and it specifies in what ways (if any) wire segments may cross or overlap. There are various models, but

one that matches our intended target technology is the commonly used 2-layer Manhattan model. This model allows only 2 layers for routing, with one layer reserved for vertical wire segments and the other layer reserved for horizontal wire segments. Wire segments on different layers that intersect may *cross* (not affecting one another), or they may be connected electrically via contact cuts.

Channel routing under the 2-layer Manhattan model is known to be NP-complete [LaPaugh] [Szymanski]. Most heuristics try to route an instance using a number of tracks very close to the *density* — the maximum, over all horizontal positions along the channel, of the number of nets crossing that position — of the instance. In the Manhattan model, any net η whose leftmost or rightmost terminal lies in column i is considered to cross position i, unless the whole extent of net η consists solely of column i. We mention this because in a model known as ''knock-knee,'' if column j contains two terminals, one being the leftmost terminal for net η_1 and the other being the rightmost terminal for a different net η_2, then the density at j due only to nets η_1 and η_2 is one.

Our problem of terminal placement is different from the classical channel routing problem in that the terminals in our problem are not in fixed positions. In Chapter 5, we analyze the complexity of certain restrictions on the problem of *Single Channel Placement*. Then, we consider the problem of *Bottom Terminal Placement*, which adds the further restriction that the top terminals are in fixed positions. We show that *Bottom Terminal Placement* is NP-complete, even if we fix the positions of the top terminals in a fixed-length channel. Finally, we present a heuristic method to solve the problem. This method splits the problem into smaller subproblems that are interesting in themselves. We give an algorithm that finds placements that achieve the optimal densities for these smaller subproblems.

In Chapter 6, we consider a problem somewhat related to Single Channel Placement. Assume that we have a positioning of terminals that achieves a good density, so that we do not want to shift the vertical alignment of terminals. Assume further that we

have some extra space that may be used to stretch the channel, thus increasing its length. That is, we may add empty columns to the channel. These extra columns cannot affect the density of the channel, but it can affect a value called *flux*, which is a lower bound on the width of the channel.

The bound of flux is not well-behaved with respect to the adding of empty channels. We show how to modify the definition of flux to create a bound, *smooth-flux*, that is well-behaved. Then, we present a ''greedy'' algorithm that calculates how to decrease the value of smooth-flux to a given target value.

Chapter 2
Linear Arrangement With Critical Paths

A linear arrangement of a graph $G = (V, E)$ is formally defined as a function $f : V \rightarrow \{1, 2, \ldots, |V|\}$. The graph my be either undirected or directed. (See figure 2-1. Note the corresponding nMOS implementation, where horizontal wires correspond to the graph edges. A correspondence holds for CMOS, but in some flavors of CMOS, the number of horizontal wire segments is double the number of edges.) Conceptually, a linear arrangement of a graph is simply the embedding of that graph onto a line (graph). This in itself is not an interesting problem. The complexity arises if we want to restrict the allowable arrangements or optimize some feature of the layout.

Consider an input graph $G = (V, E)$ and an associated path $P = v_1 \rightarrow v_2 \rightarrow v_3 \rightarrow \cdots \rightarrow v_k$. It is assumed that $v_i \in V$, for all i, and $(v_i, v_{i+1}) \in E$, for all $1 \leq i < k$. Note that P is viewed as having direction, although the underlying graph is undirected. We call v_1 the *source* of the path and v_k the *sink*. For a linear arrangement f of G, the *dilation* of P is $dil(P) = |f(v_2) - f(v_1)| + |f(v_3) - f(v_2)| + \cdots + |f(v_k) - f(v_{k-1})|$. (For example, see figure 2-2.) We shall have occasion to refer to the starting and ending nodes in a path, and we call them, respectively, the *source* and the *sink*. When used in this thesis, these terms always refer to a path and not to the (possibly directed) input graph.

In this chapter, we consider a problem called *Linear Arrangement with Critical Paths*. The question is, given a graph, whether or not there is a linear arrangement of the graph such that certain paths (subsets of the graph) are not "stretched" too much.

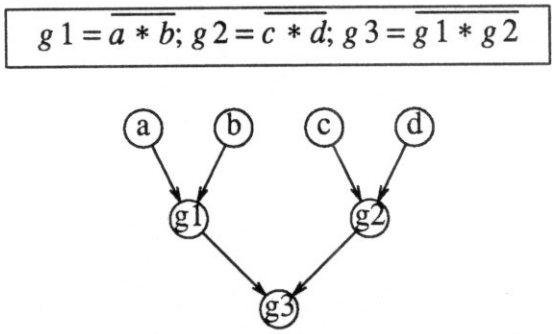


figure 2-1(a) Equations and corresponding directed graph

figure 2-1(b) Possible linear arrangement

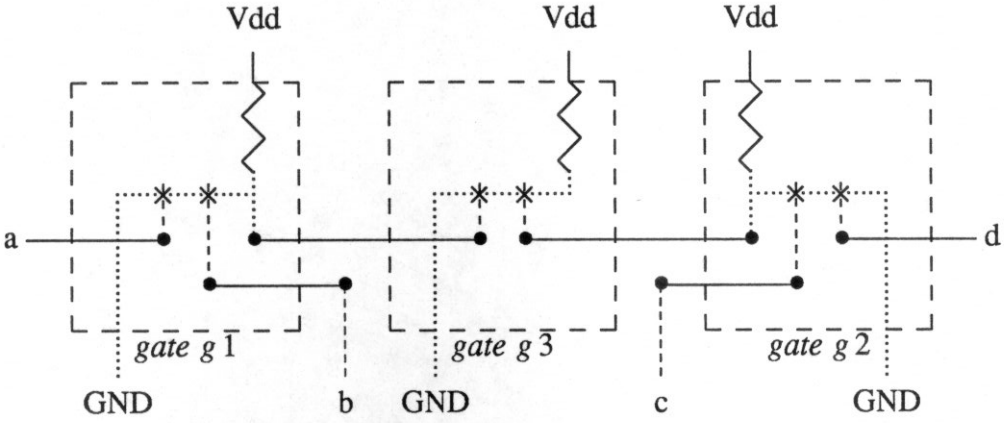
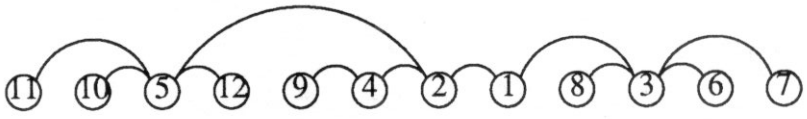


figure 2-1(c) Possible nMOS implementation

Formally, the problem may be defined as follows:

Path $P = (1, 2, 5, 12)$



$$\begin{aligned} dilation\,(P) &= |f\,(2)-f\,(1)|+|f\,(5)-f\,(2)|+|f\,(12)-f\,(5)| \\ &= |7-8|+|3-7|+|4-3| \\ &= 1+4+1 \\ &= 6 \end{aligned}$$

figure 2-2 Dilation example

Linear Arrangement with Critical Paths (LACP):

Input: Graph $G = (V, E)$.

Set of ''critical'' paths $P = \{p_1, \ldots, p_m\}$.

Set of path limits $L = \{lim\,(p_1), \ldots, lim\,(p_m)\}$.

Question: Is there a linear arrangement of G such that for all $p_i \in P$, $dil\,(p_i) \leq lim\,(p_i)$?

We also consider a problem that we term *Directed LACP (DLACP)*. *DLACP* is defined identically to *LACP* with the addition of two constraints. First, the input must be a directed acyclic graph $G = (V, A)$; second, the nodes of G must be topologically ordered — that is, if $(x, y) \in A$, then $f\,(x) < f\,(y)$. (Note that *LACP* does not become ''directed'' simply by having the graph be directed.)

There has not been very much previous work done on linear arrangement problems with non-simple (i.e., not just consisting of a single edge) critical paths. In fact, the only work we know of was done by Simonson, who studied the problem under the name ''Routing with Critical Paths'' [Simonson]. He showed that *LACP* is NP-complete, even

for bipartite graphs. Further, he showed that under the restriction that all limits in L are bounded by a constant (and all paths have bounded lengths), the problem is solvable in polynomial time.

A closely related problem that has been more heavily studied is *Bandwidth Minimization*, in which the object is to minimize the maximum distance between any two vertices that are endpoints of a common edge. The general bandwidth problem was shown to be NP-complete by Papadimitriou [Papadimitriou]. Subsequently, Garey, Graham, Johnson, and Knuth proved that bandwidth minimization remains NP-complete for trees whose nodes have maximum degree of 3 [Garey1]. Garey, et. al., also give a polynomial algorithm for the case when the maximum allowed distance between the vertices of an edge is bounded by a constant. A survey of recent results on bandwidth is given in [Chinn].

In Section 2.1, we present an independently derived proof of the NP-completeness of *LACP* and *DLACP*. In Section 2.2, we present an algorithm to solve the problem of *LACP* with the restriction that there is only one source for all critical paths, the paths are ''disjoint'' and ''non-crossing''. In Section 2.3, we look at the directed problem under the constraint that there are a bounded number of sources. We show that we can decide in polynomial time where to place the nodes by performing a reduction to a solved problem.

2.1. NP-Completeness Proofs for *LACP* and *DLACP*

Independently of Simonson, we have found a proof of the NP-completeness of the *LACP* problem. While Simonson's proof works for bipartite graphs, it does not classify the problem for trees. Our proof shows that *LACP* is NP-complete for trees, even if all the paths are ''proper'' — that is, they start and end at vertices of degree one. It is easy to show that *LACP* is in NP. Following is a reduction from *Bandwidth Optimization on*

Trees (*BOT*), which is known to be NP-complete [Garey1]. Note that if the critical paths are not constrained to be proper, we may perform a very straightforward reduction that makes each edge in the graph a critical path.

Input: β, a *BOT* instance with tree $G = (V, E)$, integer $K \leq |V|$, and with question "Is there a linear arrangement f of G such that $|f(v) - f(u)| \leq K$, for all $(u, v) \in E$?".

Reduction: Create ρ, an *LACP* instance.

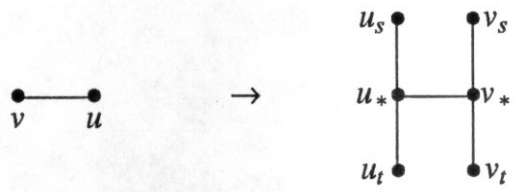
(A) Create new graph $G' = (V', E')$: (see figure 2-3)

(1) For each $v \in V$, create v_s, v_*, v_t .

$V' = \{v_s, v_*, v_t \mid v \in V\}$

(2) Create E' consisting of

- edges (v_s, v_*), (v_*, v_t) , for each $v \in V$
- (u_*, v_*) , for each edge $(u, v) \in E$



(a) graph G **(b) graph G'**

figure 2-3 Sample edge transformation

(B) Create set of paths P consisting of

(B1) $p_v = v_s \rightarrow v_* \rightarrow v_t$, with $lim(p_v) = 2$, for all $v \in V$.

(B2) $p_{uv} = u_s \rightarrow u_* \rightarrow v_* \rightarrow v_t$, with $lim(p_{uv}) = 3K + 2$, for all $(u, v) \in E$.

(C) Ask the question "Is there a linear arrangement of G' satisfying the limits for the critical paths in P ?"

Correspondence between β and ρ:

For any linear arrangement f of G, consider the corresponding ''block'' placement f' of the nodes in G'. Specifically, let $f'(v_*) = 3*(f(v)-1)+2$, $f'(v_s) = f'(v_*)-1$, and $f'(v_t) = f'(v_*)+1$. If f is a solution to the β, then f' is a solution to the ρ.

Proof:

From inspection, we can see that f' is a legal linear arrangement of G', since it is a one-to-one function from the set V' to the set $\{1,\ldots,|V'|=3|V|\}$. Also, from our definition of f', it is clear that paths of the form (B1) have their limits satisfied. For a path p_{uv} of type (B2), the dilation under f' is

$$\begin{aligned} dil_{f'}(p_{uv}) &= |f'(u_*)-f'(u_s)| + |f'(v_*)-f'(u_*)| + |f'(v_t)-f'(v_*)| \\ &= 1 + |[3*(f(v)-1)+2] - [3*(f(u)-1)+2]| + 1 \\ &= 2 + |3*(f(v)-f(u))| \\ &\leq 2 + 3*K \quad , \quad \textit{since } f \textit{ a solution } \beta \end{aligned}$$

So, paths of the form (B2) also have their limits satisfied; thus, f' is a solution to ρ.

□

Finally, we must show that any solution f' of ρ has a corresponding solution f of β. Assume ρ has a solution f'.

Proof:

The limit of 2 (= path length) on paths of form (B1) forces v_s, v_*, and v_t to be together in legal linear arrangements of G'. There are two possible orderings of these nodes: $f(v_s)+2 = f(v_*)+1 = f(v_t)$ OR $f(v_s) = f(v_*)+1 = f(v_t)+2$. That is, the three nodes related to v must be placed as a block (*block v*), with v_s and v_t flanking v_*. Since v_* must lie in

middle of block v and $f' : V' \rightarrow \{1, \ldots, 3|V|\}$, $f'(v_*) = 2 \pmod 3$. Further, given a path p_{uv} of type (B2), $dil_{f'}(p_{uv}) = 2 + |f'(v_*) - f'(u_*)| \leq lim(p_{uv}) = 3K + 2$. So,

$$|f'(v_*) - f'(u_*)| \leq 3K \tag{2.1}$$

Now, consider the linear arrangement f on G where $f(v) = \lceil f'(v_*) / 3 \rceil$. For edge (u, v), the dilation is

$$\begin{aligned} dil_f(u, v) &= |f(v) - f(u)| \\ &= |\lceil f'(v_*) / 3 \rceil - \lceil f'(u_*) / 3 \rceil| \\ &= |[f'(v_*) - f'(u_*)] / 3| \\ &\leq [(3K + 2) - 2] / 3 = K \quad , \quad \textit{equation } (2.1) \end{aligned}$$

So, the edges (u, v) satisfy their limits, and f is a solution to β.

□

The basic proof structure for *LACP* also can be used to prove that *DLACP* is NP-complete, even if all critical paths are proper. There are two basic differences. First, the reduction is from *Directed Bandwidth on Trees*, which is shown to be NP-complete in [Garey1]. Second, there is only one possible ordering for the block v: $f(v_s) + 1 = f(v_*) = f(v_t) - 1$. Also, note that after the reduction from Directed Bandwidth on Trees, the resulting directed tree is no longer rooted, since none of the nodes v_s have ancestors.

2.2. Undirected, Single Source, Paths Disjoint and Non-crossing

In this section, we consider a restriction on the *LACP* problem. We only consider problem instances where there is a single *source* node for all critical paths and the critical paths are "disjoint" (i.e., the only node that may be shared by two critical paths is the source). We also restrict the allowed linear arrangements to those where the critical

paths do not ''cross'' over the source node; that is, all paths must lie entirely to the left or the right of the source node. We call nodes or paths lying to the left of the source node to be on the *left side*, nodes or paths lying to the right to be on the *right side*. The problem is to find a linear arrangement that satisfies all path limits, if one exists. We shall refer to our problem as *USS* (*Undirected, Single Source*). Note that there is a basically identical problem where, instead of a single source for the paths, we allow only a single *sink* for all paths. (Just reverse the sense of direction of every path arc.)

Without loss of generality, we shall assume that all nodes in the graph are in some critical path. We may do this since nodes not in critical paths have no restrictions on their placement, and may thus be placed arbitrarily far from the source node. Also, let $s(i)$ = length of path p_i, $n = |V| - 1$, and $M = |P|$.

To solve *USS*, we shall create a dynamic programming algorithm that is a variation on the pseudo-polynomial algorithm to solve the *PARTITION* problem [Garey2]. Let $N = \min\left\{ n , \max_{p_i} [lim(p_i)] \right\}$, the farthest away from the root any node may possibly be placed.

Algorithm 2.1:

(1) Order the paths by non-decreasing order of $lim(i)$.

Thus, we assume that $lim(j) \le lim(k)$, if $j < k$.

(2) Build table $t[1{:}M,\ 0{:}N,\ 0{:}N]$, where $t[i,\ l,\ r]$ indicates whether or not there is a placement of the first i paths such that there are l nodes on the left side and r nodes on the right side:

(a) $t[1,0,s(1)] = TRUE$ if $s(1) \le lim(p_1)$, *else FALSE*

(b) $t[1,s(1),0] = TRUE$ if $s(1) \le lim(p_1)$, *else FALSE*

(c) $t[1,l,r] = FALSE$, *except* for *cases covered by* (*a*) and (*b*)

(d) for $1 < i \le M$, $t[i,l,r] = TRUE$ *iff*

$$OR \begin{cases} t[i-1,\ l-s(i),\ r] = TRUE\ , & s(i) \le l \le lim(p_i) \\ t[i-1,\ l,\ r-s(i)] = TRUE\ , & s(i) \le r \le lim(p_i) \end{cases}$$

(3) There is a solution iff $t[M, X, Y] = TRUE$, for some $X, Y \leq N$.

We may backtrack from $t[M, X, Y]$ to find a solution. See the proof of theorem 2.1 below for the procedure.

The main idea of the algorithm is contained in part 2(d). We place the nodes (except the source) of each critical path as an inseparable block. The paths are placed one at a time, each path being placed to the left or to the right of all preceding paths. Paths with smaller limits are placed first, so that they are closer to the source node. Meanwhile, we keep track of all possible pairs of totals for the number of nodes on the left side (l) and the number of nodes on the right side (r). An (l, r) pair is legal for path i under two different conditions. One is if path p_i is placed on the left side and p_i's limit is less than or equal to l. There is a parallel condition for legality if p_i is placed on the right side.

Now, we shall prove that algorithm 2.1 correctly solves the *USS* problem. First, we show that we may place nodes of any given path so that the nodes farther from the source in the path are also farther from the source in the linear arrangement. We say that the nodes of the path are *internally ordered.* For a given linear arrangement, if all paths have their nodes internally ordered, we say that the arrangement is internally ordered. Thus, for any path, we only have to worry about the position of the sink.

Lemma 2.1:

Assume a critical path $p_i = v_{source} \rightarrow v_1 \rightarrow \cdots \rightarrow v_{s(i)}$ in an instance of *USS*. If there is a solution f to the instance, there must be one f' such that the set of positions occupied by the nodes of p_i are the same as in f, i.e., $\{f(1), \ldots, f(s(i))\} = \{f'(1), \ldots, f'(s(i))\}$, and such that the nodes of p_i are internally ordered, i.e.,

$$\forall_{1 \leq j \leq s(i)-1} f'(j) < f'(j+1) \quad \textbf{or} \quad \forall_{1 \leq j \leq s(i)-1} f'(j) > f'(j+1).$$

Proof:

Consider the case when p_i is placed on the right side. There is a parallel case for p_i on the left side. Assume a solution f such that $f(v_{source}) = x_0$. Also, assume that the vertices in p_i are placed at positions $x_1, x_2, \ldots, x_{s(i)}$, where $x_1 < x_2 < \cdots < x_{s(i)}$. There is a path from v_{source} to the node at $x_{s(i)}$, so we have a lower bound on the dilation:

$$dil_{\min}(p_i) \geq x_{s(i)} - x_0.$$

We may form a linear arrangement f' where the nodes of the path are placed in order (i.e., $f'(v_j) < f'(v_{j+1})$) and on the same positions. We then have that $f'(v_j) = x_j$, for all $1 \leq j \leq s(i)$. Then, the new dilation is

$$\begin{aligned} dil_{ord}(p_i) &= |x_1 - x_0| + |x_2 - x_1| + \cdots + |x_{s(i)} - x_{s(i)-1}| \\ &= x_{s(i)} - x_0 \leq dil_{\min}(p_i), \end{aligned}$$

which is optimal for any fixed set of vertex positions.

□

Now, we show that the critical paths (disregarding the source node) may be disjoint from one another; thus a solution may always be *disjoint*.

Lemma 2.2:

Assume a critical path $p_i = v_{source} \rightarrow v_1 \rightarrow \cdots \rightarrow v_{s(i)}$ in an instance of *USS*. If a solution f exists, there must be an internally ordered solution f^* such that $\max_{1 \leq j \leq s(i)} [f^*(v_j)] - \min_{1 \leq j \leq s(i)} [f^*(v_j)] = s(i) - 1$.

Proof:

We shall only consider the right side of the linear arrangement. There is a parallel argument for the left side. Assume that there are m critical paths on the right side for a solution f to the instance of *USS*; by Lemma 2.1, we may assume that the critical paths are internally ordered. Denote by R_i the position of the rightmost node (the sink) of path p_i, and let R_0 be the position of the source. By

construction and Lemma 2.1, $dil(p_i) = R_i - R_0$. We may re-number the paths so that $R_i < R_j$, if $i < j$.

First, look at path p_1. Consider a (new) linear arrangement f' that places the nodes of p_1 in the positions $R_0+1, R_0+2, \ldots, R_0+s(i) = R'_1$, and the nodes v not in p_1 for which $R_0 < f(v) < R_1$ are shifted over to occupy the positions $R'_1+1, \ldots, R_1$. This shift is done so that the relative ordering of the nodes in any given path is not changed. Clearly, $R'_1 \leq R_1$, since all the nodes in p_1 were originally at or to the left of R_1. The new dilation of p_1 is $R'_1 - R_0 \leq R_1 - R_0 \leq lim(p_1)$, so p_1's limit is satisfied by f'. And the nodes of p_1 are "together," with $\max_j [f'(j)] - \min_k [f'(k)] = s(1) - 1$. No other path's dilation is affected, since no other rightmost points were moved. Since f is a solution, then so is f'. Also, f' has the nodes internally ordered, since the order of nodes within a particular path has not changed.

In the same manner in which we shifted p_1, we may continue shifting the remaining paths (starting with p_2) to create new linear arrangements; and these linear arrangements are all legal solutions. After we shift p_{m-1}, all paths are placed as blocks, with no interleaving between the blocks, and the paths are all internally ordered. Thus, the final arrangement is a solution that is internally ordered and disjoint.

□

There is one final lemma we wish to prove: We may always place the paths by non-decreasing order of limits.

Lemma 2.3:

For a linear arrangement of G, assume a solution f that satisfies Lemmas 2.1 and 2.2. Let the paths placed on the left side be (in increasing order of distance from the source) $(p_{L_1}, p_{L_2}, \ldots, p_{L_u})$ and the paths placed on the right side be

$(p_{R_1}, p_{R_2}, \ldots, p_{R_t})$. Then, there must be a solution such that the critical paths remain internally ordered and disjoint, and, additionally, they are ordered by their limits, i.e., $\forall_{1 \le j \le u_1 - 1}\ lim(p_{L_j}) \le lim(p_{L_{j+1}})$ **and** $\forall_{1 \le j \le t_1 - 1}\ lim(p_{R_j}) \le lim(p_{R_{j+1}})$.

Proof:

Assume we have a solution f that is internally ordered and disjoint. We only consider the nodes on the right side, since there is an identical argument for the left side. Assume that f places t critical paths on the right side. Let the paths be (in increasing distance from the source) $p_{R_1}, p_{R_2}, \ldots, p_{R_t}$, and let there be R_{tot} nodes on the right side.

Let p_{R_x} be a path with maximal limit among the paths placed on the right side, i.e., $lim(p_{R_x}) \ge lim(p_{R_j})$, for all $1 \le j \le t$. Consider a new linear arrangement f' identical to f, except that p_{R_x} has been moved to the far right. The new path ordering is

$$p_{R'_j} = \begin{cases} (1)\ p_{R_j} & , \quad \text{for } 1 \le j \le x-1 \\ (2)\ p_{R_{j+1}} & , \quad \text{for } x \le j < t \\ (3)\ p_{R_x} & , \quad \text{for } j = t \end{cases}$$

Paths of type (1) are in the same positions for both f and f', so their limits must be satisfied in both. For the type (3) path, $lim(p_{R'_t}) \ge lim(p_{R_t}) \ge R_{tot}$, since all path limits are satisfied by f; so the path limit is satisfied in f'. The paths of type (2) have been shifted left, so $dil_{f'}(p_{R_j}) < dil_f(p_{R_j}) \le lim(p_{R_j})$; thus, their limits are also satisfied by f'.

We may recursively shift a path with largest limit to the far right, then consider only paths to its left. After each shift, the new arrangement satisfies all path limits (by the argument above). After the last path is shifted, the paths are arranged in order, by non-decreasing path limits; and the resulting linear arrangement is a solution that is internally ordered and disjoint.

□

Finally, we prove that algorithm 2.1 actually solves the problem of *USS*.

Theorem 2.1:

Algorithm 2.1 correctly solves the problem of *USS*.

Proof:

We want to show that there is a solution if and only if at the end of algorithm 2.1, there is a table element $t[M, X, Y] = TRUE$, for some X, Y. Recall that a solution must place all paths such that the each path is totally placed on one side of the root and such that all path limits are satisfied. We call a placement of a subset of the paths *legal* if the paths are ''non-crossing'' and all placed paths have their limits satisfied.

Our proof will be by induction on the number of paths placed. In particular, we shall show that $t[k, l, r] = TRUE$ *iff* there is a legal placement of the first k paths by order of limits such that l nodes are placed on the left side and r nodes are placed on the right side.

By Lemmas 2.1, 2.2, and 2.3, we may assume that the critical paths are placed as disjoint, internally ordered blocks.

Part 1: ONLY IF

Basis: $k = 1$

Assume that $t[1, l, r] = TRUE$. By the algorithm 2.1, steps 2(a) - 2(c), one of l or r must be 0 and the other $s(1)$. Without loss of generality, assume that $r = 0$ and $l = s(1)$. Consider an internally ordered placement f of path p_1 to the left of the root. By step 2(b) of the algorithm, $s(1) \leq lim(p_1)$, so all path limits are satisfied by f.

Induction step: $(k > 1)$

Assume that our hypothesis holds for $k = 1, \ldots, i-1$. Also, assume that $t[i, l, r] = TRUE$, for some given i, l, r. We show that we can construct an appropriate legal placement of the first i paths.

By the algorithm (step 2(d)), $t[i, l, r] = TRUE$ only if

$$t[i-1, l-s(i), r] = TRUE \text{ and } s(i) \le l \le lim(p_i) \tag{2.2}$$

or

$$t[i-1, l, r-s(i)] = TRUE \text{ and } s(i) \le r \le lim(p_i).$$

Without loss of generality, we assume that the first set of conditions holds. Then by the induction hypothesis, there is a legal placement f of the first $i-1$ paths such that $l-s(i)$ nodes are placed on the left side and r nodes are placed on the right side. Now, consider the placement f' which is identical to f except that path p_i is placed to the left of all other placed nodes.

In placement f', there are $(l-s(i))+s(i) = l$ left nodes and r right nodes. Since their relative placements are the same as in f, all paths $p_1, \ldots, p_{i-1}$ have their limits satisfied by f'. Also, from equation (2.2) above, $s(i) \le l \le lim(p_i)$; thus, path p_i also has its limit satisfied. Therefore, f' is a legal placement of the first i paths.

Part 2: IF

Basis: $k = 1$

Assume a legal placement f of p_1 with l left side nodes and r right side nodes. Since f is a legal layout, all the nodes must be on the left side or on the right side. Without loss of generality, assume that all the nodes are on the left side — $l = s(i)$, $r = 0$. The legality of the placement then forces $l = s(i) \le lim(p_i)$. Therefore, by the algorithm 2.1, step 2(b), $t[1, s(1), 0] = TRUE$.

Induction step: $(k > 1)$

Assume a legal placement f of paths $p_1, \ldots, p_i$ with l left side nodes and r right side nodes. Without loss of generality, we may assume that path p_i is placed on the left side. Then, from the legality of f,

$$s(i) \leq l \leq lim(p_i). \tag{2.3}$$

Consider another placement f' that is identical to f, except that the non-root nodes of p_i are not placed. Since their positions relative to each other and the root are unchanged, the path limits for $p_1, \ldots, p_{i-1}$ are all satisfied by f'; thus, f' is a legal layout of the first $i-1$ paths. Also, the number of left side nodes is $l - s(i)$ and the number of right side nodes is r. Therefore, by the induction hypothesis,

$$t[i-1, l-s(i), r] = TRUE. \tag{2.4}$$

By equations (2.3) and (2.4) above, step 2(d) of algorithm 2.1 gives that $t[i, l, r] = TRUE$.

□

We can bound the running time of algorithm 2.1 by counting the number of array elements that may possibly be accessed during the algorithm's operation. A quick upper bound is the size of the array, which is $M * N^2$. We may get a better bound by making a simple observation: after placing a number of paths, the sizes of the paths placed on the left and the sizes of the paths on the right must sum up to the total number of nodes placed so far. That is, after i paths are placed, we must have that $l + r = s(1) + \cdots + s(i) \leq N$; so, for any given i, we only need to access $O(N)$ table entries. This immediately gives us a bound of $M * N$ on the number of entries we need to consider. Finally, we note that $M = O(N)$, since the critical paths are disjoint. N is $O(n)$; thus, the total number of table accesses done by the algorithm is $O(n^2)$.

2.3. Directed, Constant Number of Critical Path Sources

In this section, we consider a restriction on the *Directed Linear Arrangement with Critical Paths* problem where there are a constant number of critical path source nodes. The task is to determine a linear arrangement such that all path limits are satisfied, if one exists. We shall call this problem *Constant Number of Sources* (CNS), and we shall show that there is a polynomial algorithm to solve it. Note that the problem with a constant number of critical path sinks is equivalent to CNS — the reduction reverses the direction of every graph arc and every critical path arc, thus exchanging path sinks with path sources, and vice versa. Further, note that CNS subsumes the version of DLACP restricted to a constant number of critical paths.

Assume there are exactly K distinct critical path source nodes, where K is bounded. The first step in our polynomial algorithm is to place only the source nodes, assigning each node to one of the positions between *1* and $|V| = n$. We call such a placement of only the source nodes a *skeleton* layout. We may split the skeleton layout process into two parts. In the first part, we relatively order the source nodes; and in the second part, we choose the positions in which to place the source nodes.

Relative Ordering:

Let s_i be the source for path p_i. There are $K!$ permutations of the nodes, but not all the permutations are necessarily legal. In particular, if there exists in the graph a directed (not necessarily a critical) path from source s_j to source s_l, then any permutation in which s_l precedes s_j is illegal. The upper bound on the number of permutations is polynomial since K is bounded by a constant. We shall ignore any skeleton layouts that violate precedence constraints between the source nodes.

Placing the Source Nodes:

Now that we have relatively ordered the source nodes, we have to choose the exact positions for the source nodes. There are n available positions for K source nodes; thus, the total number of ways to choose the positions is

$$\binom{n}{K} = O(n^K)$$

The steps above show that the number of legal skeleton layouts is

$$O(n^K K!)$$

The remainder of the algorithm consists of transforming the skeleton layout and the input graph into a scheduling problem that is known to be solvable in polynomial time.

Before we proceed with the rest of the algorithm, though, we must introduce the notions of *ancestor* and *descendant* sets. We say that a node v is a *descendant* of u if there is a non-trivial path from u to v in the graph G — that is, there is a sequence of nodes $(u_1, u_2, \ldots, u_r)$ such that $(u,u_1), \ldots, (u_{r-1},u_r), (u_r,v) \in A$. Conversely, we say that u is an *ancestor* of v. We denote the set of all ancestors of a node v to be $An(v)$. Likewise, $De(v)$ denotes the set of all descendants of v.

Now, suppose that we are considering a particular skeleton layout f_σ. By assumption, all precedence constraints between the source nodes are satisfied. The task remaining is to try to place the non-source nodes. Given a non-source node v, we may calculate where v may be placed. First, determine the set $S(v) = \{s_i \text{ a source} \mid v \in De(s_i)\}$ Then, v must be placed *after* every node in $S(v)$. Let s be the rightmost node in $S(v)$. We call s the *start* node for v, and we write that $start(v) = f_\sigma(s)$. If $S(v) = \varnothing$, then $start(v) = 0$. Clearly, node v must be placed somewhere after its *start*.

A *legal* placement of the nodes is one where the dilation of every critical path is less than or equal to its limit. Since the layout must be topological, the rightmost node of any critical path must be its sink; therefore, the dilation of path p_i in a legal placement f is $f(t_i) - f(s_i)$, and the following equation must be satisfied:

$$f(t_i) - f(s_i) \leq lim(p_i) \tag{2.5}$$

Define the *end* position of sink t as the rightmost position where t may possibly be placed; we denote t's end position by $end(t)$. The value of $end(t)$ may be calculated by

considering every path which has t as a sink, along with equation (2.5):

$$end(t) = \min_{t \text{ is sink of } p_i} [lim(p_i) + f(s_i)]$$

Now, given the input graph $G = (V, A)$ and a skeleton layout f_σ, we may transform the problem to an instance of *Sequencing on a Single Processor*, with release times and deadlines [Garey2]:

(1) For each node $v \in V$, a task τ_v of length one.

(2) For each arc $(u, v) \in A$, a precedence constraint τ_u must end before τ_v starts.

(3) For each source node $s \in V$, $release(\tau_s) = deadline(\tau_s) - 1 = f_\sigma(s)$. Call τ_s a ''source'' task.

(4) For each non-source node $v \in V$, $release(\tau_v) = start(v) + 1$. Additionally, if v is a critical path sink, then let $deadline(\tau_v) = end(v)$; otherwise, let $deadline(\tau_v) = n$. Call τ_v a ''non-source'' task.

The question is ''Is there a one-processor scheduling of the tasks that satisfies all precedence constraints and release times and meets all deadlines?''

The correspondence between the instances of *CNP* and scheduling is fairly clear:

- Task τ_v starting at time i and finishing at time $i+1$ corresponds to node v being placed at position i.
- Consider tasks τ_u, τ_v corresponding to nodes in arc (u, v). Satisfying the precedence constraint that τ_u ends before τ_v begins corresponds with node u being placed before node v.
- Step (3) above forces the source tasks to be computed at certain times, and those time correspond with the positions of the corresponding source nodes in the skeleton layout.
- Non-source task τ_v starting at or after its release time corresponds to node v being placed after its start node.

- Let t be a sink. Task τ_t finishing by its deadline corresponds to having all critical paths with sink t satisfy their path limits.
- A solution to the scheduling problem corresponds to a legal placement in the linear arrangement problem.

According to Lageweg, et. al., the problem of sequencing on one processor with release times, deadlines, and precedence constraints is solvable in polynomial time if the tasks are of unit length [Lageweg]. Their algorithm modifies release times and deadlines in such a way that precedence constraints are satisfied if and only if the release times and deadlines are satisfied. Given this modification, their algorithm at each step schedules a task having the earliest deadline and whose release time has passed. Their algorithm solves the problem in running time quadratic in the number of tasks. Thus, our scheduling instance is polynomial, as is the corresponding instance of placing the non-source nodes into a given skeleton layout. Since the number of possible skeleton layouts is bounded, there is a polynomial time algorithm for the problem of *Constant Number of Sources*.

2.4. Conclusions and Open Problems

In this chapter, we have shown that both the directed and undirected versions of the problem of Linear Arrangement with Critical Paths are NP-complete, even if the input graph is restricted to be a tree and the critical paths must be proper. We then proceeded to show that certain restrictions on the problems made them tractable: (1) for the undirected problem, if there is a single source node for all paths, all paths are disjoint from one another (except for the source), and the critical paths may not cross over the source in the linear arrangement; (2) for the directed problem, if the number of critical path sources is bounded.

There are a number of problems whose complexities have yet to be determined. One such problem is the undirected version of *Constant Number of Sources*. It would be satisfying to know whether or not the increased flexibility of a non-topological placement makes the previously-polynomial problem intractable. Another interesting question is whether the undirected problem with a single path source and disjoint paths is still tractable if there is no restriction on the placement that the paths must be "non-crossing." There are some other restrictions that might be tried. For instance, we might allow an unbounded number of sources (or sinks) but allow each to be used only once. One other possibility is to have vertex-disjoint critical paths (except for sources and sinks).

Chapter 3
Mincut Linear Arrangement with Critical Paths

Dilation is not the only measure of the ''goodness'' of a linear arrangement. One well-known and important measure is *cutwidth*. For a linear arrangement f of graph $G = (V, E)$, the cutwidth of the layout is

$$cutwidth(f) = \max_{1 \le i \le |V|} [cut(i)] ,$$

$$where \;\; cut(i) = |\{(u, v) \in E : f(u) \le i < f(v)\}|$$

We may consider cuts not at a vertex: at point x, $cut(x) = cut(\lfloor x \rfloor)$. (See figure 3-1.) If the graph G represents a circuit, with its edges E representing wires connecting different components (V), then the cutwidth gives an estimate of how many tracks are needed to route the wires. The term $cutwidth(G)$ means the minimum cutwidth obtainable over all possible linear arrangements of G.

The problem of *Mincut Linear Arrangement* (*MLA*) asks, given a graph G and an integer K, if there is a linear arrangement f of G such that $cutwidth(f) \le K$. The general problem of *MLA* is known to be NP-complete [Gavril]. If K is fixed, then the problem has an $O(n^K)$ algorithm [Gurari]. Also, restriction of mincut to graphs that are trees makes the problem tractable. Yannakakis has found an $O(n \log n)$ algorithm to solve the problem [Yannakakis]. On the other hand, if we add weights to the edges of a tree, the mincut problem becomes NP-complete, even if all weights are polynomial in the instance size [Monien]. The complexity of the directed version of *MLA* is unknown, but *Directed MLA* bears a certain similarity to the problem of *Pebbling with no re-computation*

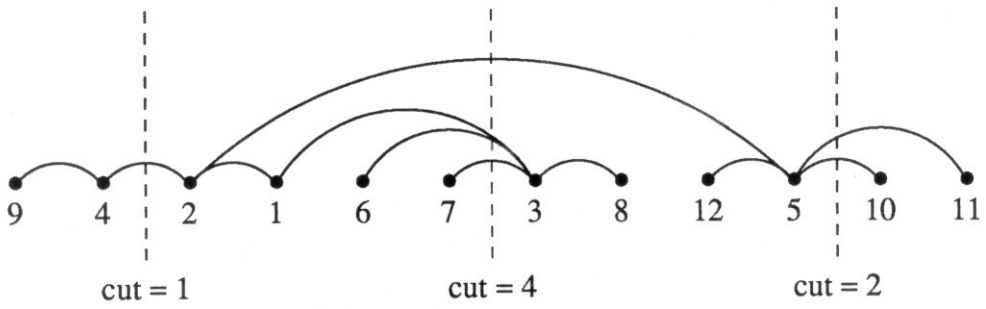


figure 3-1(a) Arrangement with sub-optimal cutwidth of 4

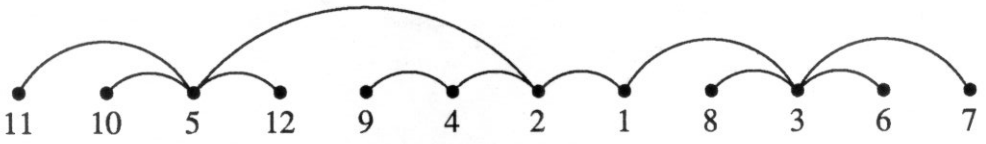


figure 3-1(b) Arrangement with optimal cutwidth of 2

allowed, which is known to be NP-complete [Sethi].

For a Weinberger layout, we would like to minimize both the wiring area needed and the maximum delay for signal propagation; thus, it makes sense to consider the problem of linearly arranging a graph, subject to a cutwidth limit and some critical path limits.

Note that the cutwidth for a linear arrangement in the graph realm does not translate directly to a lower bound on the number of horizontal tracks needed for wiring in the Weinberger array realm. For example, figure 3-2 shows a layout with cutwidth 5 that can be wired using only 4 horizontal tracks. The discrepancy occurs because connectivity represented by multiple edges from a vertex can be implemented in the Weinberger array by chaining vertices together, thus using only one track.

We use cutwidth as an estimate of the number of tracks needed because it is generally close to the actual lower bound and because there has been useful previous work on cutwidth. Future work might look at using a more accurate bound.

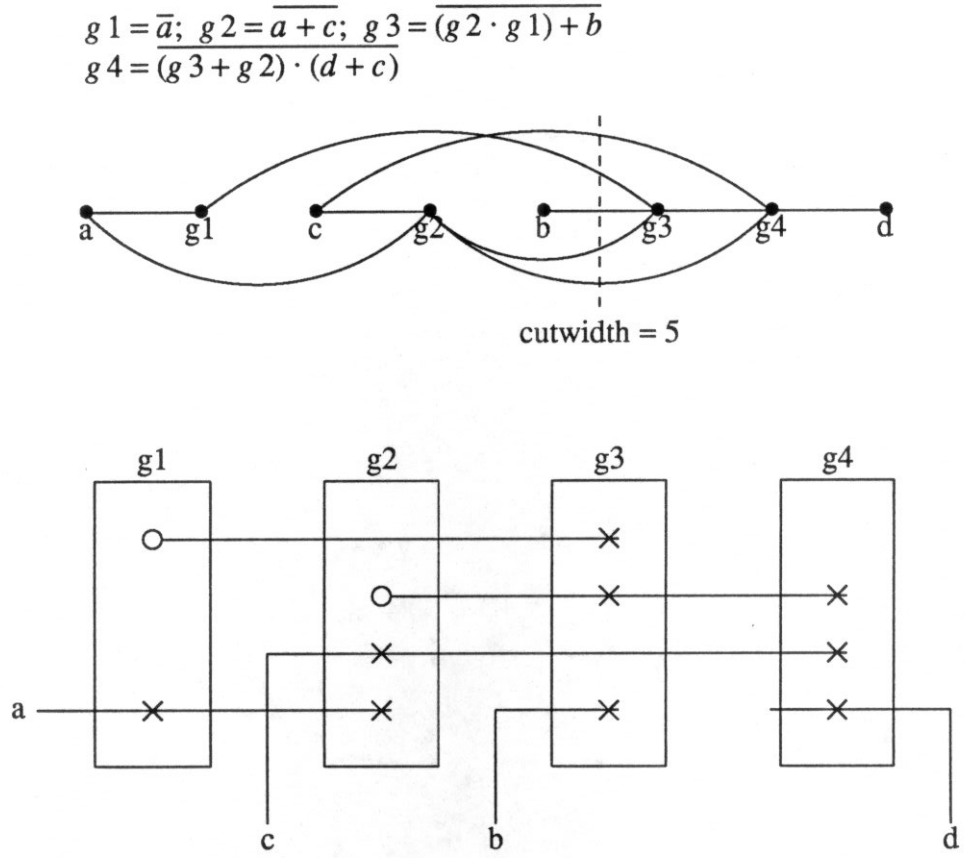


figure 3-2 Example where cutwidth exceeds number of tracks needed

In this chapter, we look at the problem of *Mincut with Critical Paths* (MCP). *MCP* is similar to *LACP* in that certain critical paths must have their limits satisfied. However, MCP entails the optimization of the arrangement's cutwidth while satisfying the path limits. Clearly, *MCP* is NP-complete, since the two sub-problems comprising it are both NP-complete. We shall also consider the corresponding topological version, called *Directed MCP* (*DMCP*), in which we must have $f(u) < f(v)$ if $(u, v) \in A$. In other words, a node must be placed after all of its parents and before all of its children.

We start this chapter by considering the directed case. In Section 3.1, we consider the problem of Mincut Linear Arrangement on directed rooted trees (no critical paths). We prove that the problem has a polynomial algorithm. Then, in Section 3.2, we give some thought to the directed mincut with critical paths problem restricted to rooted trees

and one critical path. In the remaining sections, we consider some aspects of the undirected case. In Section 3.3, we restrict the mincut problem with critical paths to trees and we assume that all critical path limits equal 1. Finally, in Section 3.4, we present a heuristic algorithm for Mincut Linear Arrangement, and we consider the complexity of the three steps of the heuristic.

3.1. Directed Mincut Linear Arrangement on Rooted Trees

We restrict the input graph to be a directed rooted tree — that is, there is a single source node for a graph containing no reconvergence. We want to solve the problem of *Directed Mincut Linear Arrangement.* The main idea is to prove that a disjoint layout of the subtrees is optimal. Then, given that the layout may be disjoint, it is easily shown that the subtree layouts may be placed by non-decreasing order of their cutwidths. First, we prove that the subtrees may be placed disjointly.

Lemma 3.1:

Given a directed linear arrangement f of the rooted directed tree T, there must exist a directed layout f', with $cutwidth(f') \leq cutwidth(f)$, such that for any non-empty subtree T_j, $\max_{v \in T_j} [f'(v)] - \min_{v \in T_j} [f'(v)] = |T_j| - 1$.

Proof:

Our proof is by induction on the height of tree T.

Basis: T consists of a single vertex (height 0)

This case is easy. Since T is a single vertex, there are no non-empty subtrees of T, and the lemma is trivially satisfied.

Induction:

The input is a tree T of height h. Its root node r has arcs to m subtrees. Call the subtrees $T_1, \ldots, T_m$. Assume some directed linear arrangement f of T, where $v_1, \ldots, v_m$ are, respectively, the rightmost nodes in $T_1, \ldots, T_m$. Without loss of

generalization, we may assume that $f(v_1) < f(v_2) < \cdots < f(v_m)$. By definition of a topological layout, r must be the leftmost node; thus, $f(r) = 1$.

Since T is a rooted tree, there is a path from r to every node in the tree. Consider the nodes $v_1, \ldots, v_m$. The nodes belong to node-disjoint and arc-disjoint subtrees; thus, there must be unique, arc-disjoint paths from r to every one of the nodes $v_1, \ldots, v_m$.

Now, consider any node $u \in T_1$. The node v_1 is placed to the left of all the other vertices v_i; therefore, the path from r to any node v_i, $2 \le i \le m$, must pass over u. That is, over any point in the layout of subtree T_1, there must be a contribution of at least $m-1$ due to arcs not belonging to T_1.

So, instead of occupying their positions under the layout f, we can create a new linear arrangement f^1 by shifting the nodes of T_1 over to the left as far as possible (keeping subtree nodes in the same relative order). Thus, subtree T_1 under f^1 will occupy the positions $2, 3, \ldots, |T_1|+1$. Since T_1 is a complete subtree, with no arcs to or from any node except r, the only "outside" arcs that can pass over T_1 are those from r to nodes to the right of T_1, and there are only $m-1$ of those. For any position over T_1, the cut due to only those arcs in T_1 is the same as for f, since the relative ordering of nodes has not changed. The contribution to the cut from outside is exactly equal to $m-1$, which is less than or equal to what it was before.

Finally, we consider the nodes not from T_1 that were shifted to the right, that is, the nodes that in f^1 occupy the positions $|T_1|+2, \ldots, f(v_1)$. The only change in the cut over these nodes is that there is now no contribution from any arc in T_1, so there is no increase in the cut due to the shifting of nodes.

We may continue successively to shift to the left nodes from $T_2, \ldots, T_m$. This creates the succession of layouts $f^2, \ldots, f^m = f'$. All of these f^i have $cutwidth(f^i) \le cutwidth(f)$, and f' has all the subtrees disjoint from one another,

which is what we require.

Thus, we can place the subtrees $T_1, \ldots, T_m$ disjoint from one another. Also, since the subtrees T_i are subtrees of T, they all have height less than or equal to $h-1$. So, by the induction hypothesis, all subtrees within any T_i may also be placed disjointly.

□

Now, we show that the disjoint subtrees may be ordered by non-decreasing cutwidth.

Lemma 3.2:

Assume a directed linear arrangement of the directed tree T. If the subtrees are laid out disjointly, there exists a directed linear arrangement, with cutwidth less than or equal to the original's, such that for any two subtrees T_1 and T_2 sharing the same parent node, if $cutwidth(T_1) < cutwidth(T_2)$, then T_1 is placed totally to the left of T_2.

Proof:

Assume a tree rooted at v, with subtrees of v being $T_1, T_2, \ldots, T_m$, and $cutwidth(T_1) < cutwidth(T_2)$. Consider a disjoint layout f where T_2 lies to the left of T_1. Let r_i be the number of subtrees in the set $\{T_1, T_2, \ldots, T_m\}$ that are placed to the right of any given subtree T_i. Since the layout is assumed to be disjoint, the maximum cut occurring over subtree T_i is $cutwidth(T_i) + r_i$. As T_2 is to the left of T_1, $r_2 > r_1$. If the positions of T_1 and T_2 are switched, the new maximum cut occurring over T_1 is $cutwidth(T_1) + r_2 < cutwidth(T_2) + r_2$, and the new maximum cut over T_2 is $cutwidth(T_2) + r_1 < cutwidth(T_2) + r_2$. But, $cutwidth(T_2) + r_2 \leq cutwidth(f)$; therefore, swapping the two subtrees cannot increase the global cutwidth of the layout. Thus, we may swap any two subtrees that are out of order to get a linear arrangement ordered by non-decreasing order of subtree cutwidths; these swaps maintain the disjointness of the layout.

The above argument works for a tree of any height. Therefore, we may successively apply the subtree ordering procedure on all subtrees of height 0, 1, . . . , *height* (*T*). In the end, the layout remains disjoint, and the subtrees are ordered by non-decreasing cutwidth.

□

From Lemmas 3.1 and 3.2 above, we see that there is a simple polynomial algorithm for *Directed MLA*: (1) Start with the leaf nodes as subtrees. The optimal layout for each subtree is trivial. (2) For each set of subtrees and a parent node, place the parent to the left, then place the subtrees by non-decreasing order of cutwidth. (3) Continue until the root node and all its subtrees are placed.

The ordering of the subtrees takes at most $k \log k$ time for a node with k subtrees. Since the procedure considers each node once as the parent of a set of subtrees, the total time for the algorithm is $O(n \log n)$.

3.2. Directed Tree, Single Critical Path

For the problem of *Directed Mincut with Critical Paths*, a tight restriction is that the input graph must be a rooted tree. We shall further simplify our problem by allowing only one critical path, which must be proper. For a rooted tree, the critical path must start at the root node and end at a leaf node. We call our problem *Directed Tree, Single Path* (DTSP).

The problem of *Directed Linear Arrangement with Critical Paths* on rooted trees is basically just a restricted version of the problem discussed in Section 2.3; thus, it is polynomial. Also, the *Directed Mincut Linear Arrangement* problem is solvable for directed rooted trees, as shown in Section 3.1. With this information in mind, we have hope that *DTSP* is tractable. Unfortunately, we have not been able to find a polynomial algorithm to solve the problem, nor have we found an NP-completeness proof. There are some

observations that may be made, however.

Let the critical path be $p = v_0 \rightarrow v_1 \rightarrow \cdots \rightarrow v_t$. Assume a linear arrangement where we know which nodes are *exterior*, that is, placed to the right of node v_t. From our earlier proof in Section 3.1, we can easily deduce that we may place the exterior nodes as disjoint subtrees, which are then ordered by non-decreasing order of cutwidth. We simply envision all the nodes to the left of v_t and including v_t to be one conglomerate vertex. (See figure 3-3.)

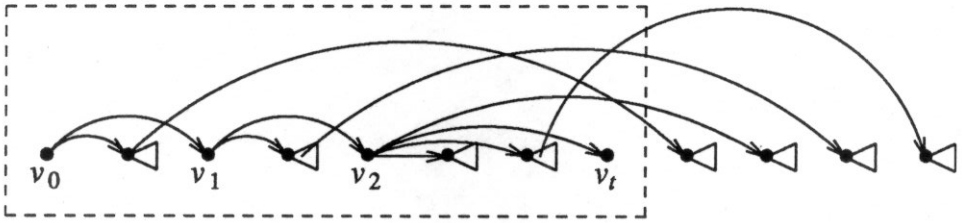


figure 3-3 left side nodes as one giant "vertex"

The tasks remaining are the following: (1) decide which nodes to place to the right of node v_t; (2) position the *interior* nodes, that is, the nodes to the left of v_t. We don't know how to solve question (1); but if we know the answer to question (1), then we may constrain the answer to question (2).

So, we assume that we know which nodes are *interior* and which nodes are *exterior*. For a subtree T_i that does not contain any of the nodes $v_1, v_2, \ldots, v_t$, define T_i to be *split* if it contains both *interior* and *exterior* nodes; otherwise, T_i is *unsplit*. Consider the placement of subtrees that do not contain critical path nodes. By an argument similar to that in Lemma 3.1, we can show that an unsplit subtree not containing a critical path node may be placed disjointly. The point is that we may move whole unsplit subtrees closer to their roots, placing them disjointly from each other. Further, by an argument similar to that in Lemma 3.2, we can show that the unsplit subtrees may be placed disjointly and by order of non-decreasing cutwidth. On the other hand, a split subtree rooted

at a node u must always create a chain of arcs from u 's parent — call it v — to a node placed right of v_t; thus, it must add at least one to the cut over any unsplit subtree that branches directly off node v. (See figure 3-4.) Therefore, we claim that for subtrees of a given node, the unsplit subtrees may be placed completely left of any split subtrees. We shall not give a rigorous proof of our claim, because, in and of itself, the claim does not give an algorithm for placing nodes to achieve minimum density; but we give a short analysis of the situation pictured in figure 3-4.

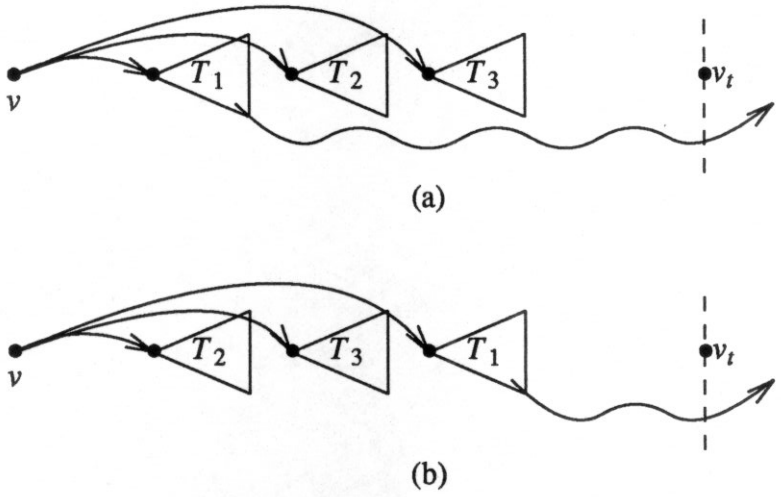


figure 3-4 Ordering of split and unsplit subtrees

Assume that the subtrees T_1, T_2, and T_3 contain no critical path nodes. The squiggly line indicates that there are arcs from T_1 to nodes to the right of v_t. Thus, T_1 is a split subtree, whereas T_2 and T_3 are unsplit. In figure 3-4(a), the maximum global cut over subtree T_2 is at least $cutwidth(T_2)+2$. The max global cut over T_3 is at least $cutwidth(T_3)+1$. We see that if T_1 is shifted to the right of the other two subtrees (fig. 3-4(b)), then the global cut over T_1 is lessened by 2. On the other hand, the global cuts over T_2 and T_3 are $cutwidth(T_2)+2$ and $cutwidth(T_3)+1$, respectively; thus, we have not increased the cutwidth by shifting the split subtree to be to the right of all the unsplit

subtrees.

Our rule for subtree placements leads to a layout with a structure resembling that in figure 3-5. Unfortunately, this does not completely determine where a split subtree is to be placed, since it could be placed farther to the right than the local neighborhood of the parent node.

Critical path = $v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow v_t$

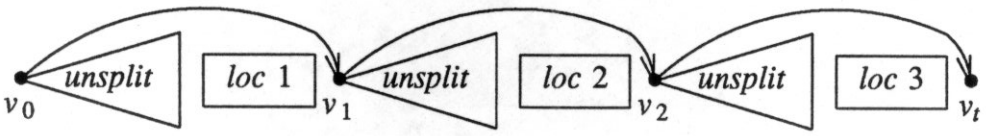


figure 3-5 Allowable locations for split subtrees

The problem of *DTSP* is polynomial if we bound the value of the path limit K to be less than or equal to $len(p) + C$, for some constant C. This result uses the condition shown above that the nodes placed to the right of v_t may be placed in a disjoint subtree layout, with subtrees ordered by non-decreasing order of cutwidth. Specifically, we only have to worry about the placement of the *interior* nodes. Note also that there may be at most C interior nodes, else the critical path limit cannot be satisfied.

So, the procedure for an exhaustive style search, given that the path limit is at most a constant amount larger than the path length, is the following:

(A) Choose how many interior nodes there will be — $\binom{C}{1}$ choices. Assume we have chosen i interior nodes.

(B) Of the $m = |V| - len(p) - 1$ nodes not in the critical path p, choose i nodes to be interior nodes — $\binom{m}{i} = O(m^C)$ choices.

(C) Order the i interior nodes — at most $i! = O(C!)$ choices. Note that some of the choices may be illegal because they violate precedence constraints between vertices.

(D) Place the ordered interior nodes into the $len(p)$ "regions" between the nodes of the critical path (in figure 3-5, these regions are the unsplit areas and the locations between adjacent unsplit areas) — at most $\binom{len(p)+i-1}{i} = O([len(p)+i-1]^i)$ possibilities (some placements may be illegal).

Totally, the maximum number of possibilities is

$$\sum_{i=0}^{C} \binom{m}{i} i! \binom{len(p)+i-1}{i} < \sum_{i=0}^{C} m^C [len(p)+C-1]^i$$

$$= O(m^C [len(p)+C-1]^{C+1}) ,$$

which is polynomial since C is a constant.

3.3. Trees, Disjoint Paths with Unit Limits

In this section, we consider the following restrictions on the problem of *Mincut with Critical Paths*: (a) the graph is a tree, (b) the critical paths are path-disjoint (i.e., they share no nodes), and (c) all path limits equal 1. We call our problem *TUL* (Trees, Unit Limits). The last constraint forces all critical paths simply to be single edges, otherwise there is no solution.

LaPaugh and Yannakakis have suggested an algorithm for this problem [LaPaugh2]:

Algorithm 3.1 (*Expand/Contract Algorithm*)

(1) For each critical path (edge) e, *contract* the edge e and its endpoints into a single node v_e, to create a new graph G'.

(2) Run Yannakakis' algorithm for mincut linear arrangement of trees [Yannakakis] on the resulting graph.

(3) *Expand* each contracted edge to its original form. Order the endpoint nodes to minimize the maximum of all cuts at or adjacent to the expanded edge.

(See figure 3-6.)

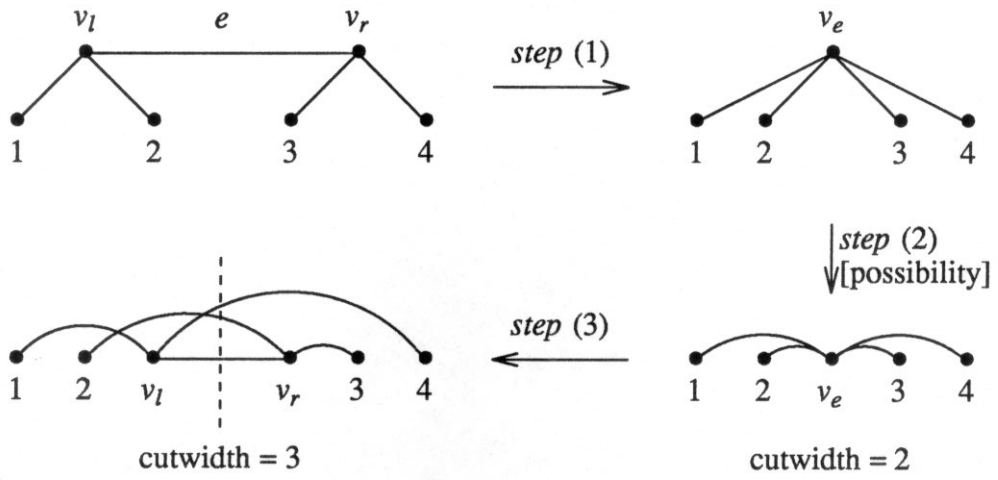


figure 3-6 Example usage of algorithm 3.1

How well does the algorithm perform? First, define the *satwidth* of a graph G to be the minimum cutwidth obtainable over all linear arrangements of G that satisfy the critical path constraints. Also, we define a *local edge cut* for a critical edge (u, v) to be a cut due only to edges that have u or v as an endpoint. Likewise, a *local node cut* for node v is a cut due only to edges that have v as an endpoint. We show that the algorithm finds a linear arrangement having a cutwidth within 1 of the optimal value. First, we prove that contracting critical path edges cannot increase the best cutwidth obtainable.

Lemma 3.3:

Let G' be the graph obtained by contracting the critical path edges of G. Then, $cutwidth(G') \leq satwidth(G)$.

Proof:

For critical edge e, $lim(e) = 1$. This means that the two endpoints of e must be adjacent in a legal layout. Edge e is contracted in step (1) of the algorithm. Let v_l and v_r be the left and right endpoints, respectively, of e for some linear arrangement of G. If we consider the same arrangement, but with v_l and v_r contracted to a single edge v_e, we see that the local node cuts adjacent to v_e must be

less than or equal to the local edge cuts near (v_l, v_r). (For example, see figure 3-7.) The contraction of e can not affect the cut anywhere except adjacent to v_e; thus, if K is the satwidth of graph G, the best obtainable cutwidth for graph G' is less than or equal to K.

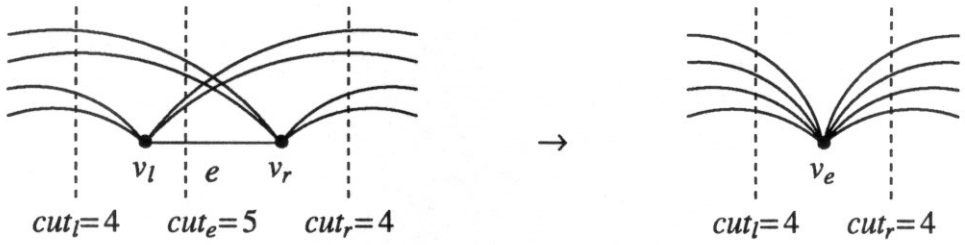


figure 3-7 Local cuts after contraction

□

Next, we show that expanding a linear arrangement for a contracted graph can increase the cutwidth by at most one.

Lemma 3.4:

Given a linear arrangement f' for a contracted graph G', let f'' be the linear arrangement derived by expanding the contracted nodes of G'. Then, $cutwidth(f'') \leq cutwidth(f') + 1$.

Proof:

Given an arrangement f' of the graph with contracted edges, consider the result of expanding a node v_e, the contraction of edge $e = (v_l, v_r)$. We look at the local cuts near v_l, v_r, or v_e.

Let $L(v_l)$ be the *left connectivity* of v_l — the number of connections from v_l to nodes placed to the left of both v_l and v_r — and let $R(v_l)$ be the *right connectivity* of v_l. Similarly, $L(v_r)$ and $R(v_r)$ are defined. Before the expansion, the local node cut to the left of v_e is $left\_cut = L(v_l) + L(v_r)$, and the local node cut to the right of v_e is $right\_cut = R(v_l) + R(v_r)$. After the expansion, the leftmost and

rightmost local edge cuts adjacent to e are still *left_cut* and *right_cut*, respectively; however, the cut above e may be different. In expanding v_e, the relative positioning of v_l with respect to v_r is flexible. If v_l is to the left of v_r, then the cut over e is $R(v_l) + L(v_r) + 1$; otherwise, the cut over e is $R(v_r) + L(v_l) + 1$.

Let $\delta = R(v_l) + L(v_l) + R(v_r) + L(v_r)$. It is clear that the optimal local edge cut over e is

$$loc\_min(e) = \min [R(v_l)+L(v_r), R(v_r)+L(v_l)] + 1 \leq \frac{1}{2} * \delta + 1 \qquad (3.1)$$

On the other hand, the maximum local node cut around v_e in the unexpanded arrangement is

$$loc\_max(v_e) = \max [L(v_l)+L(v_r), R(v_l)+R(v_r)] \geq \frac{1}{2} * \delta \qquad (3.2)$$

Equations (3.1) and (3.2) show that in the expanded arrangement, the maximum of the local edge cuts around v_l and v_r can exceed the maximum of the local node cuts around v_e (in the unexpanded arrangement) by at most 1. In fact, an increase in the local edge cut only occurs if both endpoints have identical amounts of connectivity to both the left and right sides (for example, see figure 3-7).

If the local edge cut is increased, the global maximum may also be affected. But since the critical paths are all path-disjoint, each expansion can only increase the cut at a point between the endpoints of a critical edge; thus, the global maximum cut may be increased by at most 1. So, the arrangement f'' derived by expanding all the contracted nodes in the arrangement f' has a $cutwidth(f'') \leq cutwidth(f') + 1$.

□

Theorem 3.1:

For an instance of *TUL*, algorithm 3.1 gives a linear arrangement that satisfies all path constraints and has a cutwidth within one of optimal.

Proof:

The input graph is G. Let f'' be the final linear arrangement obtained by algorithm 3.1. Also, let f' be the linear arrangement of the contracted graph G', after step (2) of the algorithm. By Lemma 3.4, we get

$$cutwidth(f'') \leq cutwidth(f') + 1 \tag{3.3}$$

Since Yannakakis' algorithm finds an optimal layout for trees [Yannakakis], $cutwidth(f') = cutwidth(G')$. Combining this with equation (3.3) above gives:

$$cutwidth(f'') \leq cutwidth(G') + 1 \tag{3.4}$$

Finally, by Lemma 3.3, we get that $cutwidth(G') \leq satwidth(G)$. This combines with equation (3.4) to give our desired result:

$$cutwidth(f'') \leq satwidth(G) + 1$$

□

We may ask whether or not algorithm 3.1 ever produces a case where step (3) actually increases the cutwidth value obtained in step (2). Figure 3-6 shows an example where this may happen. Here, an optimal solution may be derived from the final layout by simply swapping the positions of nodes "4" and "2". The reason that we found a suboptimal solution is that step (2) of the algorithm did not try to partition the subtrees connected to v_e so that those with high connectivity to v_l were on one side and those with high connectivity to v_r were on the other side.

In fact, Yannakakis' algorithm may be modified to take care of cases like that shown in figure 3-6. Before we show how to do this, we give a very short synopsis of Yannakakis' algorithm. For a fuller treatment, see [Yannakakis].

Yannakakis' algorithm: (highlights)

- For a given layout L of a tree rooted at vertex v, compute the *cost*, a finite sequence of integers $(\gamma_1, \eta_1, \ldots)$. The first integer in the sequence is the

cutwidth of the layout and the remaining integers denote cuts at other points in the layout. For example, a cost of (k,k) indicates that the cutwidth is k and that the max cut occurs immediately to the left and immediately to the right of the root. Cost can be used to compare two layouts. Relevant to our purposes is the relationship between a cost c and a cost (k,k). Simply put, $c < (k,k)$ if c is lexicographically smaller than (k,k); otherwise, $c \geq (k,k)$.

- At a given point in the algorithm, have a root v of a subtree T and layouts, with their costs, for the subtrees rooted at the children of v. Using the costs, determine how to place T.
- Start by picking an arbitrary node as the root of the entire tree, then proceed in a bottom-up fashion, computing at each node the minimal cost of the subtree rooted at the node (and a layout to achieve the cost).

We consider the situation within the running of the algorithm where a contracted node v_e is the root of a tree. We assume that the subtrees have their layouts determined.

Let the arrangement determined by Yannakakis' algorithm be called ϕ and let *cutwidth* $(\phi) = K$. From the proof of theorem 3.1, the expansion phase can only increase the cutwidth if the maximum cut occurs both immediately to the left and to the right of v_e. So, we are only concerned with the cases where the cost $c(\phi) = (K,K)$. Also, we only consider cases where the subtree layouts are disjoint from one another (e.g., sometimes in case *3a* of the main procedure in Yannakakis' algorithm). To see if the bad case for step (3) of algorithm 3.1 occurs, we check to see how the edges are partitioned between v_l and v_r. Specifically, it must be the case that $L(v_l) = R(v_l)$, which would imply that $L(v_r) = R(v_r)$.

Now that we are sure that the bad case occurs, we can try to correct the situation. Let $L_1, \ldots, L_K$ be the order of the subtrees on the left side, and let $R_1, \ldots, R_K$ be the order of the subtrees on the right side, as given in ϕ. (See figure 3-8.) Since the maximum cutwidth occurs adjacent to v_e, the subtree layouts may not have too large a

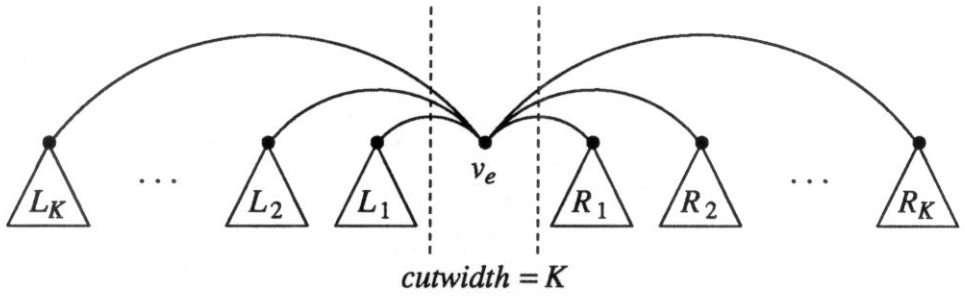


figure 3-8 Example of bad case

cutwidth. In particular, we must have that $c(R_i)$, $c(L_i) < (i,i)$.

Our goal is to re-arrange the placement of the subtree layouts. From the costs of the subtree layouts, we may determine a *limit* on how close to the root node that subtree may be placed. Specifically, for subtree T, if m is the smallest integer such that $c(T) < (m,m)$, then $limit(T) = m$. We may order all the subtrees by non-decreasing order of their limits, renaming the subtrees $T_1, \ldots, T_n$, where the total number of subtrees is $n = 2K$. (Note that $limit(T_1) = 1$, since some subtree must lie right next to the root in ϕ.) We also label each subtree according to whether it has an edge to v_l (label l) or v_r (label r). For subtree T_i, $left?(i) = 1$ if and only if the label for T_i is l, else $left?(i) = 0$.

Our algorithm is a dynamic programming algorithm that keeps track of the number of subtrees with each label that are placed on the left and the right of the root. The algorithm places each subtree, one at a time, to the left or the right of all those placed so far. It builds a table with 3 indices. The first index of the table keeps track of which subtree we are placing. The second index keeps track of the number of subtrees on the left. Then, the number of subtrees on the right is simply the first index minus the second index. The final index tells how many subtrees with label l have been placed on the left side. (We do not have to keep track of how many subtrees labeled r are placed on the left, since that value is simply the second index minus the third index.) In the end, if we are able to partition the subtrees equally to the left and the right *and* there are more l-labeled trees on one of the two sides, then the bad case for step (3) of algorithm 3.1 does

not occur; thus, we may transform a ''bad'' solution given by Yannakakis' algorithm to a ''good'' one.

The formal specification of the algorithm is as follows.

Algorithm 3.2:

(1) Build table $t[1{:}n, 0{:}K, 0{:}K]$:

(a) $t[1, 1, \mathit{left}?(1)] = \mathit{TRUE}$, $\mathit{since}\ \mathit{limit}(T_1) = 1$

(b) $t[1, 0, 0] = \mathit{TRUE}$, $\mathit{since}\ \mathit{limit}(T_1) = 1$

(c) $t[1, x, z] = \mathit{FALSE}$, except for those cases covered by (a) and (b)

(d) for $2 \le i \le n$, $t[i, j, m] = \mathit{TRUE}$ iff

$$\mathit{OR} \begin{cases} t[i-1, j, m] = \mathit{TRUE} \text{ and } \mathit{limit}(i) \le i - j \\ t[i-1, j-1, m - \mathit{left}?(i)] = \mathit{TRUE} \text{ and } \mathit{limit}(i) \le j \end{cases}$$

(2) A solution exists iff $t[n, K, x] = \mathit{TRUE}$, for some $x \ne \frac{1}{2} * \mathit{deg}(v_l)$.

We shall omit the detailed proof that algorithm 3.2 will find an arrangement that avoids the bad case, if one exists. The proof's structure would be very similar to that for algorithm 2.1. Given that the algorithm works, the important point is that a linear arrangement found by the algorithm may be used as a subtree layout to be passed on to the next step of Yannakakis' algorithm.

Algorithm 3.2 is easily shown to be polynomial. We may estimate the time needed by bounding the number of table entries $t[i, j, m]$ we must examine. Clearly, i goes from 1 to n. The value of j is likewise bounded by n. Finally, for any given i, $m \le j$, since there can't be more l-labeled subtrees on the left than there are subtrees on the left; therefore, the number of relevant m is bounded by n. This gives us a bound of $O(n^3)$ table entries that must be examined.

We have not settled all cases produced by Yannakakis' algorithm, which may involve situations where algorithm 3.2 is not applicable and the solution obtained after

expansion of v_e is sub-optimal. For example, see figure 3-9, which shows a layout with cutwidth of 3. Here, the solution obtained by Yannakakis' algorithm forces the subtree rooted at v_1 to be split, with nodes on either side of the tree's root v_e. Assume that $e = (v_l, v_r)$, v_l connects to v_1 and v_3, and v_r connects to v_2 and v_4. After expanding v_e, the new cutwidth is 4. In this case, we can rearrange the nodes (e.g., by swapping the positions of v_2 and v_3) to obtain a new layout with cutwidth equal to 3. However, it is an open question whether there is a way to modify Yannakakis' algorithm so that it finds solutions that for all cases remain optimal after the expansion phase. Further research is necessary.

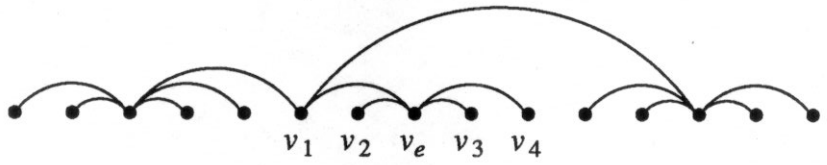


figure 3-9 Example layout with cutwidth of 3

3.4. A Biconnected Component Heuristic for Mincut Linear Arrangement

The success of Yannakakis' algorithm for the mincut linear arrangement of trees suggests a heuristic for the mincut problem on general graphs. Our idea stems from the fact that if each biconnected component of a graph is viewed as a single node, then the resulting graph is a tree. We shall present our heuristic and show that it leads to problems that are known to be or can be proved to be NP-complete. Our algorithm consists of three steps:

(1) Split the input graph into biconnected components.

(2) Individually lay out each component.

(3) Integrate the individual layouts into a single layout, *maintaining* the components derived from step (2). By maintaining components, we mean that for a given component, the relative ordering of the nodes in the final layout is either identical or the complete reverse of their ordering from step (2).

The first step is, of course, easily done. Unfortunately, in the second step, the problem of finding an optimal layout for a given component is not clearly easier than the NP-complete problem of finding an optimal layout for the entire graph. In fact, the whole graph may consist of only a single biconnected component. In general, though, it should be easier to find good layouts for the smaller components than for the entire graph.

Finally, there is the third step of the algorithm, which corresponds approximately to the integration of subtree layouts in Yannakakis' algorithm. If we are given specified layouts for the individual components, can we find an optimal way to integrate those layouts? We shall show that this problem is also NP-complete.

First, we formally define our problem.

Mincut for "Tree" of Ordered Components: [*MTOC*]

Instance: Graph $G = (V, E)$.

Set of ordered components $C = \{C_1, C_2, \ldots, C_m\}$, where each C_i is a linear arrangement for a unique biconnected component of G.

Integer K.

Question: Is there a linear arrangement of G with cutwidth less than or equal to K that maintains each component C_i ?

The problem of *MTOC* is clearly in NP. Simply guess a linear arrangement for the vertices, then check to see that the vertices are placed in the relative orders specified in the components of C and that the cutwidth is less than or equal to K. To complete the NP-completeness proof, we give a reduction from the problem of *Mincut for Edge-Weighted Trees* with polynomial edge weights, which is NP-complete [Monien].

Start:

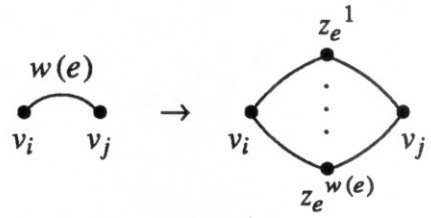
Tree $T = (V, E)$, with polynomial-sized weight $w(e)$ for each edge $e \in E$, with the question "Is there a linear arrangement of T with cutwidth less than or equal to K ?"

Reduction:

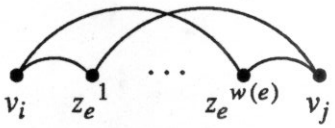
Create graph $G' = (V', E')$:

- For node $v \in V$, let $v \in V'$.
- For edge $e = (u, v) \in E$, create $w(e)$ nodes $z_e^{\,1}, \ldots, z_e^{\,w(e)} \in V'$.
- For edge $e = (u, v)$, create $2\,w(e)$ edges $(u, z_e^{\,1}), \ldots, (u, z_e^{\,w(e)}), (v, z_e^{\,1}), \ldots, (v, z_e^{\,w(e)}) \in E'$.

(See figure 3-10(a).)



(a) Transformation of an edge



(b) A component layout

figure 3-10 Reduction from MEWT to MTOC

The resulting graph G' is a "tree" of biconnected components resembling the one shown in figure 3-10(a). There is a biconnected component corresponding to each edge in E. For the component corresponding to edge (u, v), the ordering of the nodes is $u, z_e^{\,1}, \ldots, z_e^{\,w(e)}, v$. (See figure 3-10(b).)

Correspondence:

First, we note that for a given component, the component layout is an optimal solution. Each node $z_e{}^k$ adds one to the cutwidth, no matter where it is placed; so, totally, the cutwidth is $w(e)$, and it is achieved by the component layout of figure 3-10(b). We further note that the relative ordering of the nodes given by a component is optimal for the final placement, since the only nodes that connect to points outside the biconnected component are the "original" nodes v_i and v_j.

Given a solution to the MTOC problem, we can derive a solution with the same cutwidth to the MEWT problem by simply replacing the set of nodes $z_e{}^k$ (and the attached edges) by a single edge with weight $w(e)$ (opposite direction transformation of that shown in figure 3-10(a)). Likewise, if there is a solution to the MEWT problem, there must be a solution to the MTOC problem. Simply take the solution to MEWT, transform the weighted edges as described above, then place every node $z_e{}^k$ somewhere between the nodes to which it is connected.

□

3.5. Open Problems and Future Research

We still have not fully classified the problem of *DTSP*. We have considered a number of approaches for deciding which nodes to place to the right of v_t. These include contracting the critical path to a single node and swapping subtrees. However, none of the methods is satisfactory. It may be that partitioning the problem into the two particular subproblems we use is not the way to solve it. It may also be useful to characterize the complexity for other restrictions on *DTSP* besides the one we considered. For example, we may bound the limit of the critical path by $limit(p) \leq len(p) + g(len(p))$, for some function g.

The near-optimality of algorithm 3.1 suggests a heuristic for certain other restrictions of the *Mincut with Critical Paths* problem. We could restrict the class of input

graphs to those that are trees after the contraction of certain sets of nodes that are constrained by critical path limits to be placed as groups. Examples of such node sets include the following: (1) a path $v_1 \rightarrow \cdots \rightarrow v_t$ whose limit is equal to its length ($= t-1$); (2) a clique of m nodes that contains a simple critical path with limit equal to $m-1$ (or some other set of critical paths forces the nodes to be together — this includes case (1)).

In case (2) above, the local cut due to just the edges of the clique is always the same, no matter what the ordering of the vertices, so one may order the vertices by non-increasing order of *left_connectivity – right_connectivity*. Case (1) has a tighter constraint on how the vertices may be placed. In fact, the vertices in the path must be in consecutive order, although they may be in backwards order. It is quite clear that using the idea of algorithm 3.1 to contract the entire path into a single node will not ensure that the expansion phase will only increase the cutwidth by at most 1. See figure 3-11, which shows the two possible states after the expansion of a contracted path. It would be satisfying to find an algorithm that guarantees something more concrete about the solution. Further research is necessary.

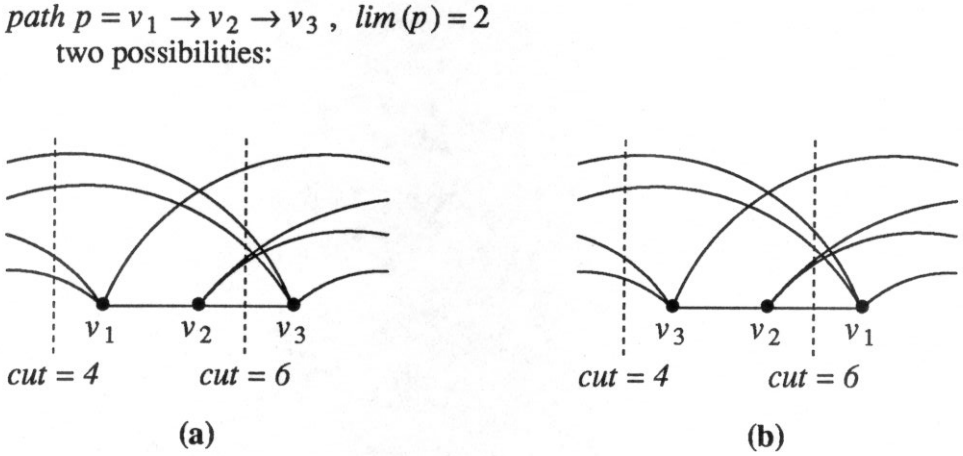


figure 3-11 Example where expansion must increase local cut by more than 1

In addition to those discussed above, there are a host of graphs that become trees when certain subsets are contracted. It would be useful to generalize those structures and

find metrics for determining how well a contract/expand algorithm would work on them.

Chapter 4
Cluster Placement in an Array of Gates

In this chapter, we examine the problem of *Cluster Placement in an Array of Gates* (*CPAG*), which we introduced in Chapter 1. Our goal is to position signals to minimize the area required to implement the circuit, given that the order of the gates is fixed. We differ from previous work on placement within gate arrays and Weinberger arrays in that the signals are grouped together in *clusters* (see Chapter 1).

A *legal* placement into a grid satisfies the following conditions:

- one column per gate
- no clusters interleave
- sufficient routing area, under the 2-layer Manhattan routing model

The formal statement of our problem follows.

Cluster Placement in an Array of Gates:

Input: Ordered set of gates $\{G_1, G_2, \ldots, G_N\}$, where G_i is a set of clusters. Each cluster is a set of signals (integers).

Objective: Find a legal placement of the signals into an array such that the gates are placed in order, left-to-right, and the area required to realize the circuit (in grid terms) is minimized.

Figure 4-1 shows examples of legal and illegal signal placements involving clusters.

We call a placement of the signals a *configuration*. The circuit realization of a configuration requires the connection of all signals having the same value. We call the

Gate = { (1 3), (4 5), (6 7) }

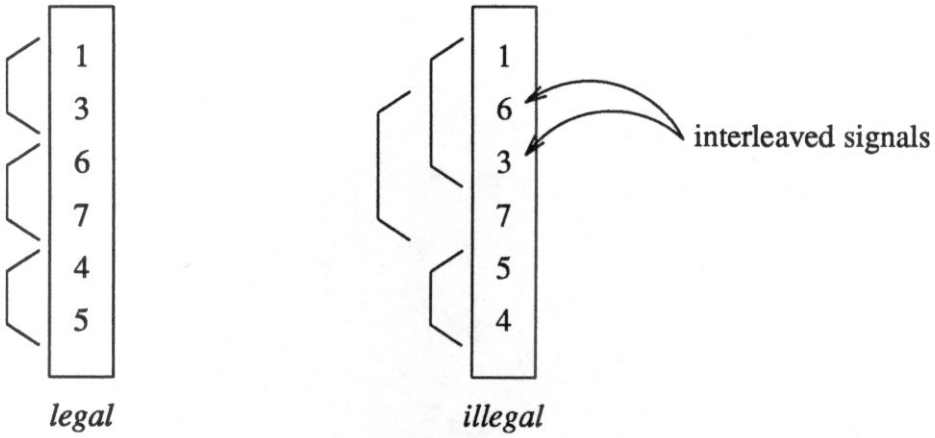


figure 4-1 Examples of legal and illegal cluster placements

aggregate of all instances of a particular signal value to be a *net*. We do not place any constraints on where wires may be placed, as long as they satisfy our routing model.

There are two factors to consider for the space required. First, one would like to place the signals in as few horizontal tracks as possible. Second, one would like to use very few extra vertical columns to *route* (connect identical) signals between gates. All routing must be done in the vertical channels, or free space, between gates. Extra vertical columns are placed within the channels, as needed, to connect nets with signals that are not aligned (i.e., not in the same horizontal tracks). The total number of columns used is *num_extra_columns* + *N*. Note that we have abstracted to a single column a gate from the Weinberger array realm. There, the number of columns required by a given gate depends on the type of the gate and the layout template being used for that type of gate. For example, our NOR-of-ANDS in figure 1-2 requires 4 columns. In the heuristics we have implemented, we fix at the start the number of horizontal tracks used (Section 4.2 further details this choice); thus, we only have to minimize the number of columns. For that purpose, we don't need to know exactly how many columns are used

for routing each gate. Finally, the total area required by a configuration is simply *num_tracks* * *num_columns*.

This chapter presents the work we have done on the problem of *CPAG*. In Section 4.1, we give a short, simple proof that *CPAG* is NP-hard. In Section 4.2, we present some of the issues involved in trying to solve the problem of *CPAG*. We also introduce the notion of *shadow* clusters. In Section 4.3, we describe the heuristics that we studied; and in Section 4.4, we present and analyze the experimental results obtained.

4.1. NP-Completeness Proof

In this section, we show that *CPAG* is NP-complete. First, we re-state the problem as a decision problem:

CPAG — Decision Form:

Input: Ordered set of gates $\{G_1, G_2, \ldots, G_N\}$, where G_i is a set of clusters.
Each cluster is a set of signals.
Integer L.

Question Is there a legal placement of the signals into an array such that the gates are placed in order, from left to right, and the area required to realize the circuit (in grid terms) is less than or equal to L ?

The problem of *CPAG* is in NP. We may guess a given configuration of the signals and a wiring of the configuration. The wiring guesses if there is a wire on each grid segment and if there is a contact at each grid point. The size of the grid is bounded by a polynomial in the number of nets, which is less than or equal to the size of the input. The bound on the grid size occurs because each vertical channel between a pair of gates certainly needs less than or equal to $2 * number\_of\_nets - 1$ columns, and the total number of horizontal tracks need not exceed $2 * number\_of\_nets + size\_of\_largest\_gate$. We may then check to see whether the wiring connects all like signals and whether the total

number of columns used times the total number of tracks used is less than or equal to L. To finish the NP-completeness proof, we prove that CPAG is NP-hard, even if the input only contains a single gate.

Lemma 4.1:

The problem Cluster Placement in an Array of Gates is NP-hard.

The NP-hardness reduction is from the *Mincut Linear Arrangement* (MLA) problem, which is known to be NP-complete [Gavril].

Start: *Min* , an *MLA* instance with graph $G = (V, E)$, integer K, with question "Is there a linear arrangement f of G such that *cutwidth* $(f) \leq K$?"

Reduction:

Create *Arr*, a *CPAG* instance with a single gate G_1:

(1) For each vertex $v_i \in V$, create a cluster C_i.

(2) For each edge $e_j \in E$, create a signal s_j.

(3) For each edge $e_j = (v_i, v_k)$, insert s_j into both C_i and C_k.

(4) Integer $L = (K + 1) * 2|E|$.

Correspondence:

We wish to prove that there is a solution to *Arr* if and only if there is a solution to *Min*. First, we show that if there is a solution f to *Min*, then there is a solution α to *Arr*.

Proof:

(If):

Let f^{-1} be an inverse function for the linear arrangement f, that is, if $f(v_i) = j$, then $f^{-1}(j) = i$. We form a solution α to *Arr* by ordering the clusters, from top-to-bottom, $C_{f^{-1}(1)}$, $C_{f^{-1}(2)}$, . . . , $C_{f^{-1}(|V|)}$, i.e., in the same order as the corresponding vertices in the linear arrangement. Let $e_j = (v_i, v_k)$. For signal

$s_j \in C_i$, place s_j near the top of C_i if C_k is above C_i ($f(k) \leq f(i)$), else place it near the bottom (see figure 4-2).

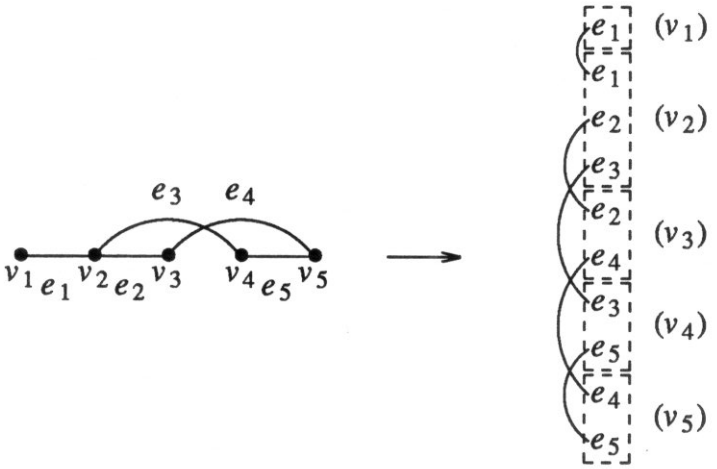


figure 4-2 Example reduction from *MLA* to *CPAG*

Now, consider the routing for the created gate G_1. Recall that we have no constraints regarding where the wires must be placed. Taking G_1 in isolation, there is no topological difference between the "channel" on the left side of G_1 and the "channel" on the right side of G_1. Thus, we get no savings in area by placing wire segments on both sides of the gate. So, we assume for ease of argument that we do all the routing of wires in the left channel. There are "terminals" on only one side of the left channel; therefore, there can be no cycle constraints. The width of the channel is simply its density in the 2-layer Manhattan routing model. From our construction above, it is clear that the local density over a cluster C_i in α cannot exceed the maximum of the local densities immediately adjacent to C_i. To calculate the density between a pair of clusters C_i and C_j, we simply count the number of nets that have signals on both sides. From our construction, this number simply equals the number of edges in f that cross a cut between the nodes v_i and v_j. The maximum cut in f is K, so the maximum

channel density in α is also K, and the total number of columns is $K + 1$.

Since there are $2|E|$ horizontal tracks, the area required by the placement α is exactly $L = (K + 1) * 2|E|$. Thus, α is a solution to *Arr*.

We now finish the proof by showing that if there is a solution α to *Arr*, then there is a solution f to *Min*.

(Only if):

Let α be a solution to *Arr* that requires $area(\alpha) \leq L$. From the reduction, the number of signals placed is $2|E|$; and since there is only one gate for placing the signals, the number of horizontal tracks used by α is at least $2|E|$. This requires that the total number of columns used to by α be less than or equal to $\frac{area(\alpha)}{2|E|}$

$\leq \frac{L}{2|E|} = K + 1.$

One of the columns for α is used by the signals, so at most K columns are used for routing. Since, in our routing model, the density is an upper bound on the width, the density of the channel is less than or equal to K.

Form a solution f to *Min* by contracting each cluster into a single node, maintaining the same ordering as in α. The cuts in f may not exceed the local densities in α, so $cutwidth(f) \leq K$; thus, f is a solution to *Min*.

□

4.2. Issues for Heuristic Algorithms

There are a number of issues to consider when trying to solve the problem of *CPAG*. One is that some signals pass through certain gates without forming transistors. When we analyze such a gate, we must be sure to save a space for the passing signal. The method we choose for dealing with this consideration is to introduce the notion of *shadow* clusters. A *shadow* cluster is a cluster consisting of a solitary signal, and this

cluster may interleave with signals from other clusters in the gate because the signal does not directly interact with the gate. (See figure 4-3.)

G_1: {(1 3) , (2 4)}; G_2: {(1 2) , (4)}; G_3: {(3 4)}
Signal '3' passes through gate G_2 — shadow cluster

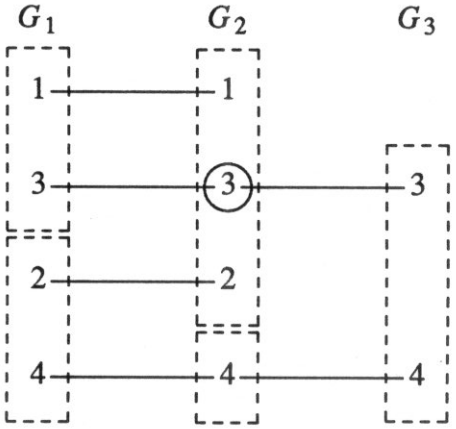


figure 4-3 Example of "shadow" cluster

For each gate, we add exactly one shadow cluster for each net that passes through the gate without interaction. This is a simple method, and it may not allow for better solutions that use extra shadow clusters. In figure 4-4(a), we only allow one shadow cluster for net '5' in the second gate; but in figure 4-4(b), we allow net '5' to have two shadow clusters.

Given that we know exactly what signals occur in each gate, we must decide how to place them. The process of positioning the signals to minimize the area requirement encompasses the two subtasks of determining how many horizontal tracks to use and how many extra vertical columns to use. The number of horizontal tracks used may actually affect the minimum number of vertical columns that must be used to route signals in the channels between the gates. In our implementation, we simply choose the number of tracks to be the smallest possible. That is, for each gate, we count the number of signals (including shadow cluster signals and repeats of signals from the same net) that must

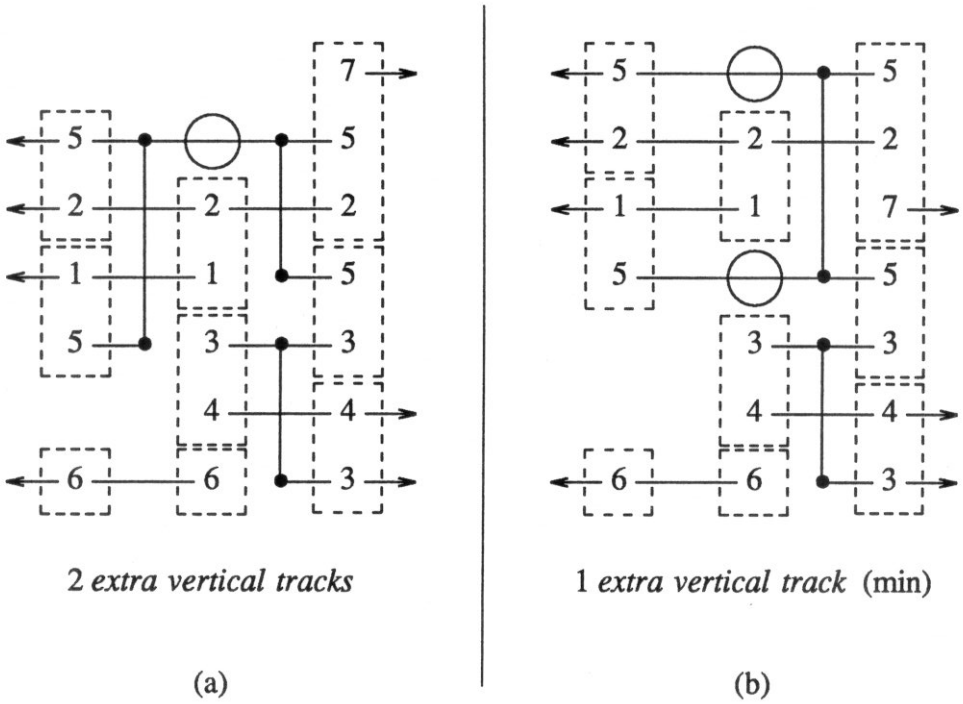


figure 4-4 On the use of multiple shadow clusters

pass through that gate. The maximum of these counts is the least number of tracks that may be used by the largest gate, so it is also a lower bound on how many total horizontal tracks to use.

After determining how many horizontal tracks to use, we may place the signals into the array. Given a fixed positioning for signals (terminals) at each gate, signals from the same net must be connected across each vertical channel between a pair of gates. For a given vertical channel, the goal is to make the connections using as few vertical columns as possible. The smallest number of columns that can possibly be used is called the channel's width. This problem is called *channel routing*, and it is known to be NP-complete under the 2-layer Manhattan routing model [LaPaugh1] [Szymanski]. But this means that we probably cannot know in polynomial time what is the least number of vertical columns required to route the signals across a given channel.

Determining the area required by any given placement of the signals is NP-complete. We therefore use an estimate of the number of extra columns needed (and thus, the area, since we know how many horizontal tracks are used). Our estimate is based on the notion of vertical channel *density*, which is identical to density for a horizontal channel (recall definition from Chapter 1), except for the directions used. We calculate the density of each channel and take the sum of all the individual densities. This sum, plus the number of gates, is our estimate of how many vertical columns are needed.

There are inaccuracies with our method of estimation. These inaccuracies arise because density is a lower bound on the width of only a *single* channel. For multiple channels, the total density may sometimes overestimate the need. For example, the total density in figure 4-4(b) is 2, but the total width is 1; in figure 4-5, the density is 3, but the width is 2. This overestimation occurs because the routing done in one channel may negate the necessity to do some routing in another channel. Despite its shortcomings, the density measure seems to give a good estimate for many applications, including ours. It's not clear how much improvement would be obtained by using a better measure of the width, and the cost in time to calculate and re-calculate such a measure might prove prohibitive.

Density is our primary measure of the "goodness" of a particular configuration. We shall also use a secondary measure, namely the total number of *alignments*. Two signals from the same net and in adjacent gates are said to be *aligned* if they are on the same horizontal track. Placed on the same track, aligned signals may be connected by a straight horizontal wire segment, without the need for an extra vertical routing segment. All our heuristics try to minimize the total density and maximize the number of alignments. Two examples of previous work that consider the maximization of alignment of terminals are [Schlag] and [Widmayer].

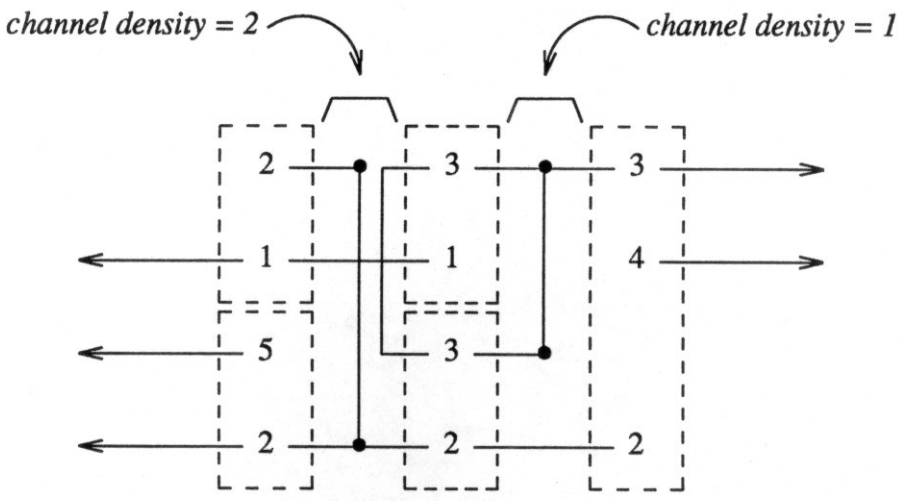


figure 4-5 Example where total density overestimates width

4.3. Heuristics

We have used the following heuristics to solve *CPAG*: "Greedy", Iterative Improvement, Partitioning, and Simulated Annealing. Except for the "greedy" heuristic, all the methods are predicated on the idea of successive improvements from some starting placement of the signals.

4.3.1 "Greedy" Heuristic

The "greedy" method proceeds in a row-by-row manner, starting with the top row. On a given row, we start by placing a signal in a gate that requires the maximum number of horizontal tracks. Within the present row, we try to align signals in neighboring gates with adjacent signals from the same net. For example, if a signal '1' is placed in the gate numbered 'i', then we try to place a signal '1' in gates numbered '$i-1$' and '$i+1$'. If this is not possible and we must place a signal in order to have room to complete the gate, then we place some other signal; otherwise, no signal is placed. In aligning and placing signals, the principal rule is that clusters may not be split (except by shadow clusters).

We did not implement the ''greedy'' method to challenge the other methods. Its primary use is quickly to generate a fairly good starting configuration for the Iterative Improvement or Simulated Annealing heuristic.

4.3.2 Partitioning

In Partitioning, we use a Kernighan-Lin [Kernighan] type of procedure to recursively split the array into two pieces, the *top* and the *bottom*, using a horizontal cut. First, we pad the gates with empty clusters of ''size'' 1 so that all gates contain the same number of signals. Then, we swap clusters between the two pieces, trying to minimize the cost — the number of connections between the two pieces. For any pair of gates, the number of connections is simply the number of nets that have signals on both sides of the cut. This procedure tends to minimize the density at the point of the horizontal cut. When the pieces are small enough, we use a simple greedy strategy to try to align the signals.

Since we have padded the gates to be of the same size, we may regard a partitioning step as a rearrangement of clusters within a fixed-size array. Our partitioning is complicated by the condition that clusters may be of differing sizes; thus, a straight horizontal cut across the array may ''split'' some clusters. For example, consider the following situation: 12 horizontal tracks total, gate 1 has three clusters of 4 signals each, gate 2 has 2 clusters of 6 signals each. Obviously, there is no straight cut that does not split a cluster in one of the two gates. The solution we employ is to use a *fuzzy* cut — that is, one that does not necessarily cross different gates at the same place (see the bold line in figure 4-6(a), which represents a cut within an array). This solution prevents clusters from being on both sides of a cut, but it may lessen the accuracy of the estimate of density provided by the count of signals crossing the cut.

Figure 4-6 shows an example usage of the algorithm. Figure 4-6(a) starts with a random fuzzy partitioning of the eight clusters. The swap that creates the largest

improvement in the number of crossings (cost) is the exchange of clusters C_4 and C_6. After that improvement, no further swap makes an improvement, so we consider a partitioning of the subset of clusters that lie underneath the original cut (figure 4-6(b)). Figure 4-6(c) shows the result of performing the partitioning on the smaller subset of clusters, followed by a greedy alignment of the signals. Note that in figure 4-6(b), the net *1* has signals on both sides of the original cut; but in the final placement of figure 4-6(c), all the signals of net *1* are aligned. This is an example where having a fuzzy partition boundary can cause the estimate of the density to be too large.

4.3.3 Randomized Iterative Improvement

Before giving the algorithm, we shall define what we mean by a *move*, or a change in the configuration. Moves are used by both Iterative Improvement and Simulated Annealing. A move chooses two horizontal tracks h_1 and h_2. For every gate in some chosen set $\{G_s, G_{s+1}, \ldots, G_t\}$, swap the item in track h_1 with the item in h_2. An item may be a terminal or an empty space. Pictorially, an example of a move is given in figure 4-7 below. Note that our move properly contains the set of single swaps, which is commonly used in iterative improvement. We chose the more general move because it increases the size of the *neighborhood* — the configurations reachable in a single move — of a configuration, thus making it more likely that we can reach an optimal solution.

Iterative Improvement (*II*) starts at a legal configuration and considers the change in the cost due to some move. Basically, the cost is equal to the total density, but we stipulate that splitting a cluster adds infinite cost; therefore, no configuration can have a split cluster, and all configurations are legal. If the cost decreases or stays the same, then the move is made and the signal placement is updated. In some forms of iterative improvement, one looks at the effects of all possible moves, then chooses the one that improves the cost the most. This strategy requires much time for our problem because there are many possible moves — in all, there are about $\frac{1}{4} * num\_tracks^2 * num\_gates^2$ possible

moves:

$$\binom{num\_gates}{2} + num\_gates$$

choices for leftmost and rightmost gates to affect

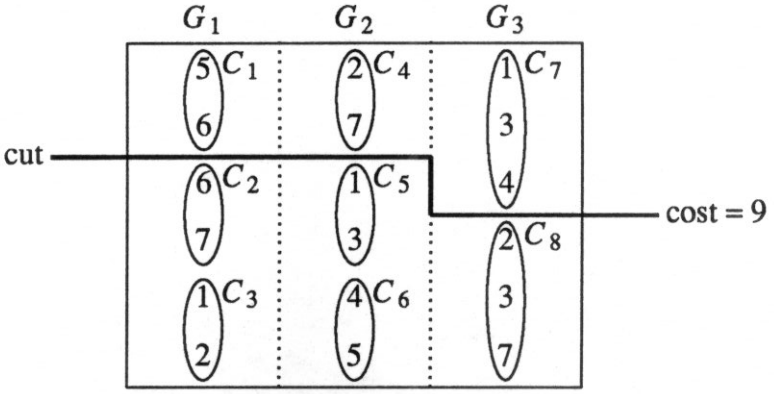
and $\binom{num\_tracks}{2}$ *choices for which 2 tracks to swap.*

After a move is made, we must pick a best move from the new configuration. Since certain configurations may not have moves that actually improve the cost, although a series of moves improves the cost, it may take a huge number of iterations to arrive at an improved solution. To find a ''local'' optimum requires even more time.

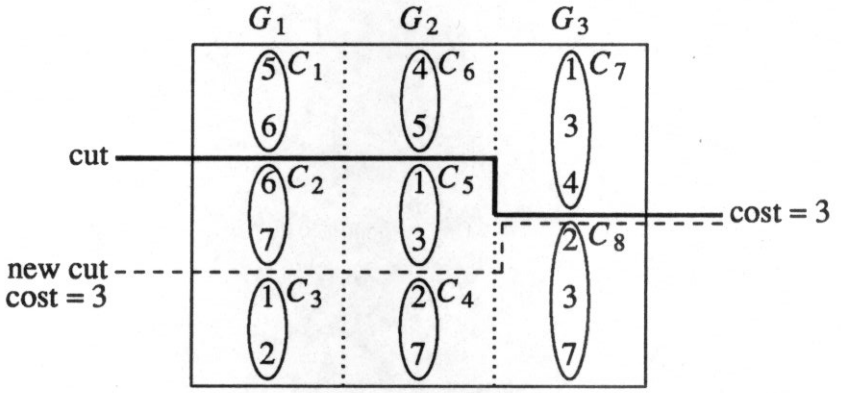
Also a major concern for the *II* heuristic is deciding when to stop. In usual local improvement, changes are made until no improving move can be found. However, if no move improves the cost, but some maintain the same cost, it is not clear whether or not to continue in the hopes that further moves may improve the cost. Our solution is to try only a bounded number of random moves. We choose our bound to be $5 * num\_tracks^2 * num\_gates^2$, which is a moderate number of steps and gives a fairly quick running time. We did try a bound of $10 * num\_tracks^2 * num\_gates^2$ on a number of experiments, but despite the doubling of the moves tried, almost no improvement was found.

One other iterative improvement strategy we considered is based on an idea by Kernighan and Lin [Kernighan]. In this method, we calculate the changes made by a sequence of moves. From this sequence of moves, we choose the subsequence (starting from the original configuration) that makes the largest improvement. The idea is that we may be able to improve the configuration by first increasing its cost, then decreasing it. In this way, we hope to avoid being trapped in local optima in the search space.

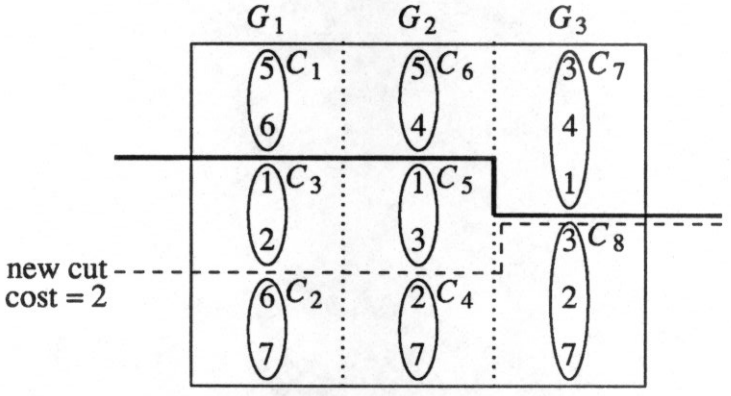
For our implementation, a sequence of $2 * \sqrt{numtracks} + 1$ moves is generated at each iteration. Then we choose the best subsequence, performing all of the moves in the



(a) Starting partition



(b) After one pass: swap C_4 and C_6



(c) After second pass: swap C_2 and C_3
(perform "greedy" placement)

figure 4-6 Sample usage of partitioning heuristic

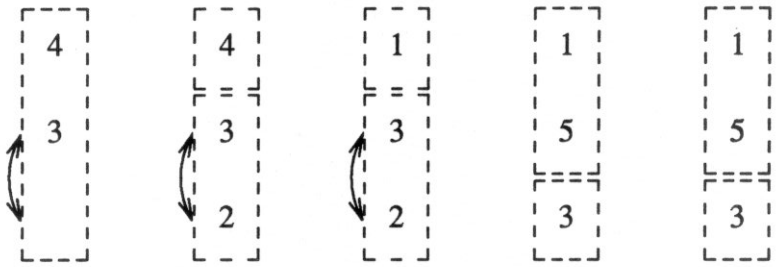


figure 4-7 Example of a *move*

subsequence. We set the number of iterations at $\frac{2*numgates^2*numtracks^2}{\sqrt{numtracks}}$, thus making the run time to be on the same order as that for the simple iterative improvement algorithm.

4.3.4 Simulated Annealing

The final heuristic we tested is Simulated Annealing. The method of simulated annealing was first presented in [Kirkpatrick]. It has since been the subject of much research, for example, [JohnsonD] [Greene] [Sechen]. The idea of simulated annealing is to avoid being trapped in local optima (a shortcoming common to local improvement strategies) by sometimes allowing moves that actually increase the cost of the configuration. Ideally, this method mimics the gradual cooling of a physical system from a high-temperature, high-energy state to a low-temperature, low-energy state.

There are a number of implementation details for any simulated annealing program. We have already described what we mean by a configuration and a move. An important question to answer is whether or not to allow moves that create illegal configurations, something we did not allow in our iterative improvement heuristic. In our problem, an illegal configuration is one where a cluster is split or a terminal is not placed in the correct gate. We allow split clusters during the running of the algorithm, since by allowing split clusters, we allow a broader range of possible solutions. In particular, only

permitting legal configurations means that we may be stuck with the initial relative ordering of the ''real'' clusters. For example, if the clusters are not singular, they cannot be split; thus, the order of the clusters is fixed. On the other hand, we do not allow terminals to shift from gate to gate. This condition is enforced by our definition of a move.

The primary goal of our algorithm is to minimize some *cost function*, which measures the ''goodness'' of a configuration. Clearly, the cost function must contain our estimate of the total area used, namely the total density. We also note that a configuration tends to be better if there are more alignments, since the routing is made easier, so our cost function may have a term containing the number of alignments. The basic form of our cost function is

$$cost = density + C_1 * num\_split\_clusters + \frac{C_2}{total\_alignments + 1}$$

Note the term in the cost that counts the number of split clusters. Also note that a configuration is better if there are more alignments, hence the term that is inversely proportional to the number of alignments. We choose C_1 so that the best solution found so far will never contain a split cluster; thus, $C_1 = initial\_density$, the density of the initial configuration. We have arbitrarily chosen $C_2 = initial\_density$. We also tried $C_2 = 1$ and $C_2 = 0$, which gave comparable or worse results.

Given our value of C_1, it might be possible to perform an entire annealing where the only legal configuration encountered is the starting configuration. The best solution in this case is the starting configuration; however, the final configuration might have a much better density, with perhaps as few as one split cluster. In such a case, we fix up the final configuration so that it is legal, then we anneal some more without allowing any split clusters.

The start configuration is determined by the ''greedy'' method described briefly in Section 4.3.1 and our starting temperature is $5 * initial\_density$. The two other important values, *cooling ratio* and *epoch length* were picked by trial and error. Our cooling

schedule is simply *new_temp* = *cooling_ratio* * *old_temp*, with the cooling ratio being set to .98. We found the value of .98 to be sufficiently large to give an adequate solution, but not so large as to make the computing time too long. From our experiments, we set the *epoch length* (the maximum number of steps at a given temperature) to be $25 * (num\_gates)^2 * num\_tracks$ (see Section 4.4), and we go to a new temperature if we have already accepted *num_gates* * *num_tracks* moves at the present temperature. This last condition is to prevent us from spending too much time at high temperatures, which according to Johnson, et. al., [JohnsonD] does not accomplish much.

One final consideration is the ''freezing'' criterion, or when to stop the annealing because there doesn't seem to be any improvement. We used a basically *ad hoc* approach of counting the number of consecutive temperatures at which no improvement occurred. If this number reached 200, then we assumed the configuration to be frozen. Two other possible stopping strategies are (1) using a different threshold value for freezing than 200 and (2) stopping the annealing at a certain temperature, when there should be few accepted moves, and performing an iterative improvement phase. In Section 4.4, we shall summarize the results of using these other two approaches. One other possibility with which we did not experiment is to stop if a given temperature accepts below a certain fraction r of tested moves. This would have required us to keep track of what moves from what configurations had been tried for the present temperature. Also, we were not sure what value of r to use, since moves that maintain the same cost are always made, and we have no good idea how many zero-cost moves exist for an ''average'' configuration.

4.4. Experimental Results

This section presents the experimental results of running tests on the heuristics described in Sections 4.3.1 - 4.3.4. The computation times are for a Sun 3/260 workstation.

We ran our heuristics on a number of test files. The tests were all generated by a test generation program. The options were:

LIMIT on size of clusters (DEFAULT 5)

LIMIT on number of nets used (DEFAULT 99)

LIMIT on number of clusters per gate (DEFAULT 20)

Explicitly state the number of gates (DEFAULT random ≤ 20)

Following is the generation algorithm we used. All values are generated by a random number generator.

for each gate *G*

Determine the number of clusters for gate *G*.
for each cluster *C* in *G*
Determine the size of *C* — call it *size*.
Generate *size* different signals for cluster *C*.
(The *nets* are determined by signals generated in the above step.)

We shall summarize the results for 6 different input files. Below is a synopsis of the 6 files.

Input file #	# of nets	# of gates	# of tracks	# of clusters	Features
1	37	5	35	22	*very few gates*
2	15	9	26	45	*Many tracks vs. nets*
3	15	13	16	44	*All clusters small*
4	30	8	27	27	*No shadow clusters required, All clusters large*
5	30	10	29	37	*All clusters large*
6	30	15	40	104	*Largest example*

Table 4-1 Characteristics of the input files

For input file 3, the term *small* means that all clusters contain only one or two signals. The term *large* for files 4 and 5 mean that all clusters contain either four or five

signals (for our tests, we constrained clusters to contain at most five signals each).

Before we compare the performance of the different heuristics, we summarize our choices for some of the values used by the Simulated Annealing approach. First, we considered a two different values for the epoch length. Up to a point, the longer the epoch length, the better the solution should be. Johnson, et. al., suggest an epoch length proportional to the size of a configuration's neighborhood [JohnsonD]; thus, one of the epoch lengths we tried is the one we term *long* — $2 * numgates^2 * numtracks^2$. We also tried an epoch length, denoted *short*, that is smaller for an instance with an average number of horizontal tracks: $25 * numgates^2 * numtracks$. In table 4-2 below, we give the results of running Simulated Annealing for the *short* epoch length and the *long* epoch length. For these runs, the value of C_2 was set to *start_dens* and we terminated when we found 200 consecutive non-improving temperatures. The values given for the 10 runs for each input at each epoch length are the following: the best density solution found, the median density solution, and the total CPU time for the 10 runs. A median value of *x-y* indicates that x was the 5^{th} best density found and y was the 6^{th} best density. Below, we see that the results are similar for the short and long annealing, but the long annealing is taking much more CPU time. In fact, the short annealing is performing better. This is probably because the longer epoch value requires a longer time to freeze, since the longer epoch time allows more uphill moves at high temperatures. We might extend the freezing condition to more than 200 consecutive non-improving temperatures, but the run-time seems to be increasing too fast already. Also, it may be that we are decreasing the temperature too fast. Slowing down the cooling process should improve the performance; but once again, it also adds to the computation time. Both inputs 1 and 2 are fairly small, so a large input would probably take an excessively long time on the long annealing, so we shall use the short epoch length annealing as a comparison with the other heuristics.

A second value that must be determined for the simulated annealing heuristic is C_2, the cost weighting factor for the term involving signal alignments. Table 4-3 below

Input	Best Dens		Med Dens		CPU hrs	
	short	long	short	long	short	long
1	5	7	6-7	8	30	79
2	44	45	50	51	61	166

table 4-2 Effects of differing epoch lengths on 2 inputs

shows the results of using three different values of C_2, annealing until we find 200 consecutive non-improving temperatures. For each value of C_2 and each input file, annealing was run ten times. The values are (1) *start_dens*, (2) *1*, and (3) *0*. The effect of the alignment factor on the result of the annealing seems to be small. We shall use the $C_2 = start\_dens$ annealing to compare Simulated Annealing with the other heuristics.

Input	Best Dens			Med Dens			CPU hrs		
	(1)	(2)	(3)	(1)	(2)	(3)	(1)	(2)	(3)
1	5	6	6	6-7	6-7	7	30	30	29
2	44	44	45	50	52-53	51	61	48	39
3	8	8	8	9	8	9	68	91	79

table 4-3 Results with different alignment factors

One final consideration that we make for the Simulated Annealing heuristic is the freezing criterion. The three possibilities we examine are (1) stop after 200 non-improving temperatures, (2) stop after 400 non-improving temperatures, and (3) stop at temperature .025 (then use an iterative improvement fix-up). The value of .025 was chosen because at that value, uphill moves will almost never be accepted. In table 4-4, we give the results of running Simulated Annealing for the three different freezing criteria. The results are for 10 runs on each combination of input file and freezing criterion. Method (3) does not perform quite as well as (1) or (2). Since the value of 200 non-improving temperatures appears to perform as well as the value of 400 does, we shall use method (1) annealing in our comparisons with the other heuristics.

Input	Best Dens			Worst Dens			CPU hrs		
	(1)	(2)	(3)	(1)	(2)	(3)	(1)	(2)	(3)
1	5	5	6	6-7	6-7	8	30	41	8
2	44	44	45	50	50	52-53	61	106	22

table 4-4 Effects of different freezing criteria

Finally, we are ready to compare the heuristics. Table 4-5 below summarizes the density results obtained in the running of four different heuristics on the five small input files. The heuristics tested are *Randomized Iterative Improvement* (II) after a greedy start, Kernighan-Lin style *Iterative Improvement* (K-L II) after a greedy start, *Fuzzy Partitioning* (FP) followed by an iterative improvement fix-up, and *Simulated Annealing* (SA). The number of runs for each heuristic was chosen so that the total running time for the heuristics was approximately equal; thus, Simulated Annealing has the fewest data points. The numbers in parentheses indicate the number of runs for each heuristic.

Input File	Mean / Best Density				Total CPU Hrs.			
	II	K-L II	FP	SA	II (50)	K-L II (50)	FP (50)	SA (10)
1	9 / 6	9 / 6	8 / 6	6 / 5	20	18	22	30
2	54 / 47	55 / 48	54 / 44	50 / 44	33	35	34	61
3	10 / 8	10 / 8	9 / 8	9 / 8	30	29	27	68
4	45 / 40	45 / 40	44 / 36	41 / 37	20	20	24	17
5	62 / 53	64 / 55	60 / 49	61 / 55	60	68	63	50

Table 4-5 Comparison of the different heuristics

Clearly, one of Fuzzy Paritioning or Simulated Annealing always found the best of the densities obtained. In terms of the best density solution found, Fuzzy Partitioning outperformed Simulated Annealing on two of the input files, and Simulated Annealing did better on only one of the input files. On the other hand, Simulated Annealing gives the best mean density on all inputs except input 5.

Figure 4-8 shows the range of solutions obtained by the four heuristic algorithms on input file 4.

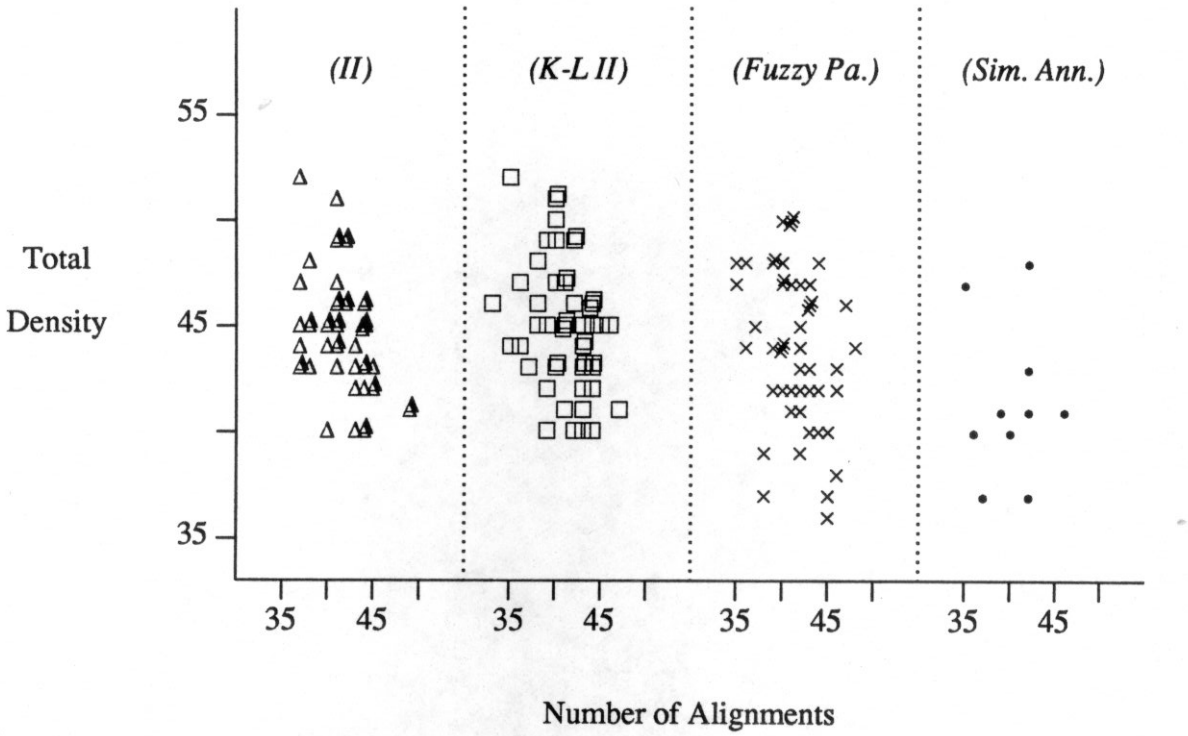


figure 4-8 Ranges of solutions for input file #4

The other input files produce similar graphs. All of the heuristics' solutions exhibit the same kind of distribution. For input file 4, the spreads from best density to worst density are 11 to 14 for the heuristics; the spreads on the number of alignments are 11 to 14. In both cases, Simulated Annealing is the only heuristic with a spread of 11; a smaller spread is expected, since there are only 10 data points rather than 50. Also interesting is that for all the heuristics, the median value for the number of alignments is about 41. This similarity may be in part accounted for by the fact that the three heuristics Iterative Improvement, Kernighan-Lin II, and Fuzzy Partitioning all concluded with an iterative improvement phase. On the other hand, Simulated Annealing is basically an iterative improvement algorithm at low temperatures, when uphill moves are rarely accepted. The

principal difference in the solution distributions is the mean value for the total density. For II and K-L II, the mean density is 45; for Fuzzy Partitioning, the mean is 44; for Simulated Annealing, the mean is 41.

On the average, the mean density value for Fuzzy Partitioning exceeded the mean density value for Simulated Annealing by about ten percent. This suggests that for small instances, if we want to run only one heuristic and we want to run it only once, we should pick Simulated Annealing. On the other hand, if we are given a specified amount of time to find a good solution, Fuzzy Partitioning should perform about as well as Simulated Annealing for small instances.

The results for the largest file, input file #6, finally show what seems to a marked superiority of the Simulated Annealing. In figure 4-9, we present the full range of experimental results obtained for ten runs each of Iterative Improvement and Fuzzy Partitioning (total CPU time about 90 hours for each of the heuristics), seven runs of the Kernighan-Lin II (total CPU time about 490 hours), and two runs of the Simulated Annealing (total CPU time about 400 hours).

The major concern with the annealing heuristic is the CPU time required, which makes comparisons between the heuristics difficult. We did not tune our code for any of the algorithms, so the processing times could easily be three or four time higher than necessary; using parallel processing could also lower the running time. On the other hand, input file #6 isn't really that large, yet annealing is already taking days for each run. Our results so far suggest that our present method of annealing will not be efficient for solving very large input files. More research is necessary to determine if this is in fact the case.

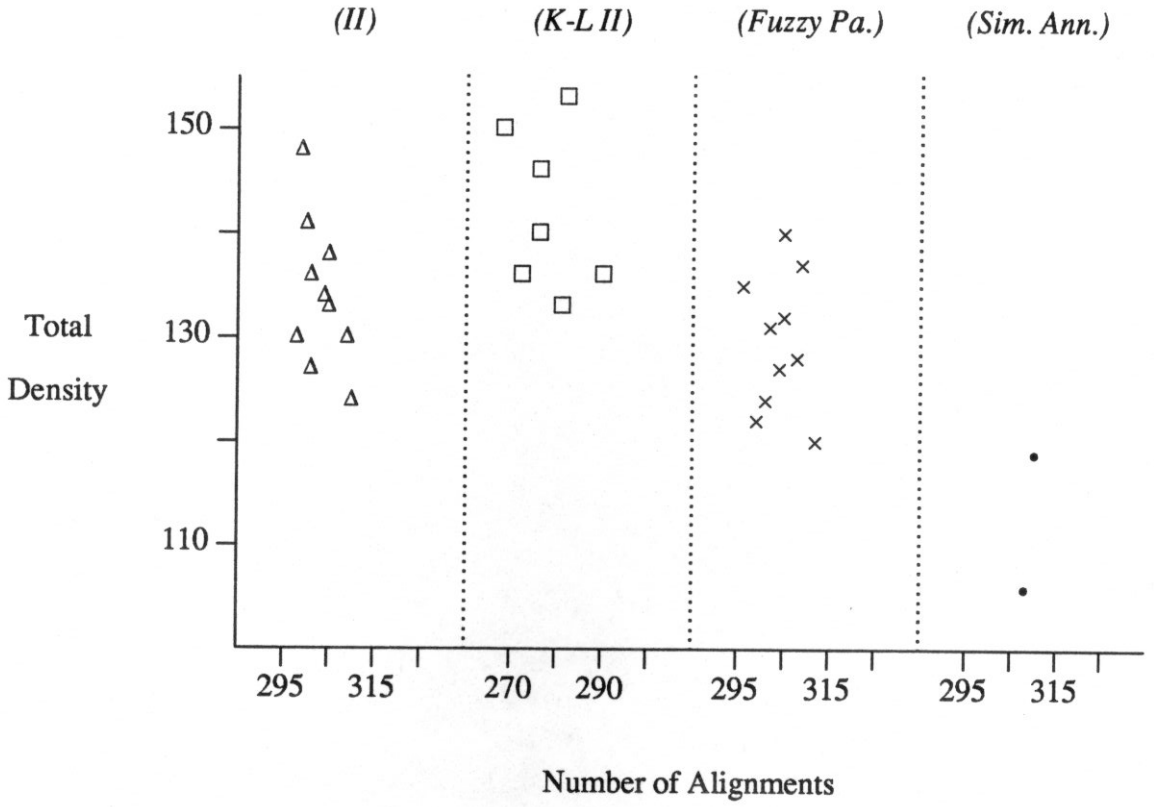


figure 4-9 Ranges of solutions for input file #6

4.5. Open Problems and Future Work

Perhaps the most obvious optimizations can be made to the actual code used to implement the various heuristics. Our run times are much longer than recent figures given by Sechen and Lee for row-based placement by simulated annealing. They report completion in less than 24 hours of computation on a Microvax for circuits of up to 3000 cells [Sechen]. One example of an un-optimized step in our implementation is our fairly straightforward calculation of density at each step of the simulated annealing. Keeping more information about the current extents of each signal at every gate, we might be able to calculate density updates in half the time we presently use. Also, parallelization offers some hope for improvement, since the move updates may be done simultaneously, as may the density updates for all the channels.

There are also variations on the heuristics that may be tried. We might try a different cost function, perhaps utilizing a better estimate of the array's width than density. The problem is that such a metric would probably be harder to calculate than density, so the runtime might increase.

Another possible variation is to use partitioning as a basis, but not use a ''fuzzy'' version. We may, instead, make cuts at the same place at every gate. Then, we might conceptually have to ''split'' a cluster across the boundary cut, which makes keeping track of clusters harder; but there may be some improvement over the estimate used by the ''fuzzy'' partitioning.

On a more basic level, we might allow more than one shadow cluster per gate. This could be done by a pre-processing step that decides if any extra shadow clusters should be used and where they should be placed. We could also let our heuristic determine extra shadow clusters by allowing moves to change an empty space to a shadow cluster, or vice versa.

Finally, we might allow the number of horizontal tracks to vary. One way to do this is to specify more horizontal tracks than the minimum allowed, padding the extra tracks with empty spaces. In calculating the total number of horizontal tracks used, we may disregard any tracks that contain only empty spaces.

Our testing is incomplete in certain regards. First, we would like to test more input instances, perhaps with some of those being ''real'' instances coming from the front end of an automatic logic compiler. The more instances tested, the more accurate the results are likely to be. Along the same lines, we would like to test larger instances, to see the performance of the heuristics on bigger examples. In this regard, we were hampered by the long running time of the algorithms, especially the Simulated Annealing. Finally, we have by no means exhausted the variety of approaches to solving the placement problem. In the next chapter, we shall mention one of the methods that we did not implement.

Chapter 5
Terminal Placement in a Single Channel

In Chapter 4, we considered some local improvement heuristics for *Cluster Placement in an Array of Gates*. The heuristics tried to decrease the total density of the array of gates, taken as a whole. It is possible to fashion heuristics that try to optimize the placement of signals in one channel at a time. One way to do this is first to specify the placement of the signals in the first two gates, then successively use the placement at the i^{th} gate to drive the placement of the signals at the $i+1^{st}$ gate. For example, Terai uses this strategy to improve the placements for terminals in a gate array [Terai]. Bui and Lee mention this method as an approach to solving the problem of cell positioning in a standard cell circuit layout [Bui]. Also, Kobayashi and Drozd consider the problem of terminal placement in a channel with *cells* of fixed extent, each cell consisting of movable (possibly repeating) terminals [Kobayashi].

In this chapter, we consider some problems arising from the above successive optimization heuristic. The main problem we consider is how to place the signals (or terminals) in a single channel to minimize the density. We call this problem *Single Channel Placement* (*SCP*). We know, of course, from the NP-completeness proof for *Cluster Placement in an Array of Gates* that this problem is NP-complete.

Since the main problem is hard, we consider some subproblems that arise in trying to solve the main problem. Some earlier work has been done by Atallah and Hambrusch for the problem of terminal placement into a fixed set of positions, with no clusters. They showed that the problem of achieving the optimal density is solvable in polynomial

time for the following cases: (1) the top terminals have fixed positions and all the bottom nets have the same number of terminals; (2) all top nets have the same number of terminals and all bottom nets have the same number of terminals [Atallah1]. In the same paper, Atallah and Hambrusch showed that the problem of density minimization for multi-terminal nets (no clusters) is NP-complete, even if the positions of the upper terminals are fixed. They also have results for the related problem where the ordering of terminals is known and the only allowed movement is rotation of the terminals [Atallah2].

First, in Section 5.1, we consider some restrictions on the number or order of clusters and signals within the gates; and we give the complexities of some of these cases. We then add the further restriction that the top terminal positions are fixed; we call this new problem *Bottom Terminal Placement* (BTP). In Section 5.2, we shall prove that given a limit on the channel length, the problem of BTP is NP-complete, even if restricted to the case where there are no clusters. In Section 5.3, we present one way of trying to solve BTP that is based on considering clusters as "single points," thus dividing the placement problem into smaller subproblems. Finally, in Section 5.4, we present an algorithm for terminal placement in a channel. We show that the algorithm achieves the optimal density, and we present the motivation for the algorithm, which is to minimize the channel width under a jogless routing model.

Note that in the rest of the thesis, we shall flip the orientation of channels so as to be horizontal, with terminals lying along the top and bottom instead of along the left and right. We do this because it is usual to consider horizontal channels when dealing with channel routing and placement problems.

5.1. Some Restricted Cases for Single Channel Placement

There are special cases of the *Single Channel Placement* problem that are solvable in polynomial time. In this section, we briefly outline the state of knowledge of the problem under four possible restrictions:

(A) Clusters are relatively ordered

(B) Signals within clusters are ordered

(C) At most l clusters, l a constant

(D) At most p signals per cluster, p a constant

Case 0: None of the restrictions hold

This is just a subcase of the general problem for which the proof of NP-completeness still holds.

Case 1: (C) and (D) both hold

This case is polynomial, since there are a bounded number (at most $l * p$) of terminals in each gate. One algorithm is simply to try all possibilities. This type of idea has been used for multi-column pin alignment for cascode-switch trees [Schlag].

Case 2: (B) and (C) both hold

This case is polynomial, since there at most $l!$ possible orderings of the clusters of each gate, and each such ordering forces a strict ordering of the signals. The remaining problem can then be reduced to one of deciding the alignments between the top and bottom rows of a channel, where the terminals of both rows are relatively ordered. This problem was solved in [Gopal], which gives an $O(T^2)$ algorithm, where T is the number of nets.

Case 3: (A) and (B) both hold

This is polynomial, since the terminals in both gates are totally ordered; thus, the algorithm by Gopal, et. al., mentioned above solves the problem.

Case 4: Only (D) holds, with $p \geq 3$

This case is NP-complete. The reduction (practically identical to the one in Section 4.1) is from *Mincut Linear Arrangement* restricted to planar graphs with maximum vertex degree 3, which was shown to be NP-complete by Monien and

Sudborough [Monien].

Remaining cases:

The exact complexities for the remaining combinations of restrictions are unknown.

As a note to case 4 above, there is a simple algorithm that achieves density 2 for the case when $p = 1$ — that is, each cluster consists of a single signal — and we want a minimum-length channel. It is easy to see that for arbitrary-length channels, a density of 1 can be achieved. The algorithm for minimum-length channels follows.

Placement of Singular Clusters:

(1) Initially, all clusters start *unassigned.*

For each signal, have a *top excess set* and a *bottom excess set*, both initially empty.

(2) Until there are no unassigned clusters on the top:

For unassigned top cluster t, if there is an unassigned cluster b from the bottom such that t and b contain identical signals, then form the pair (t, b). Clusters t and b are now considered *assigned.*

If t cannot be paired, then place t in the *top* excess set for the signal contained in t. t is now considered *assigned.*

(3) Until there are no unassigned clusters on the bottom:

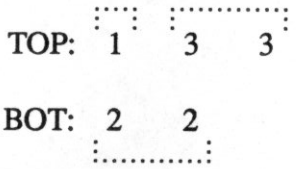
For unassigned bottom cluster b, there cannot be any top cluster available to pair with, so place b in the *bottom* excess set for the signal contained in b. b is considered *assigned.*

(4) First, place the clusters that are in the excess sets. For both the top and bottom rows, start at column 1 and proceed to the right, never skipping a column. Place one excess set at a time, and for each excess set, place its clusters consecutively. Of course, top excess sets are placed on top, and bottom excess sets are placed on the bottom. The result of this step is a tight packing of terminals on both the top and the

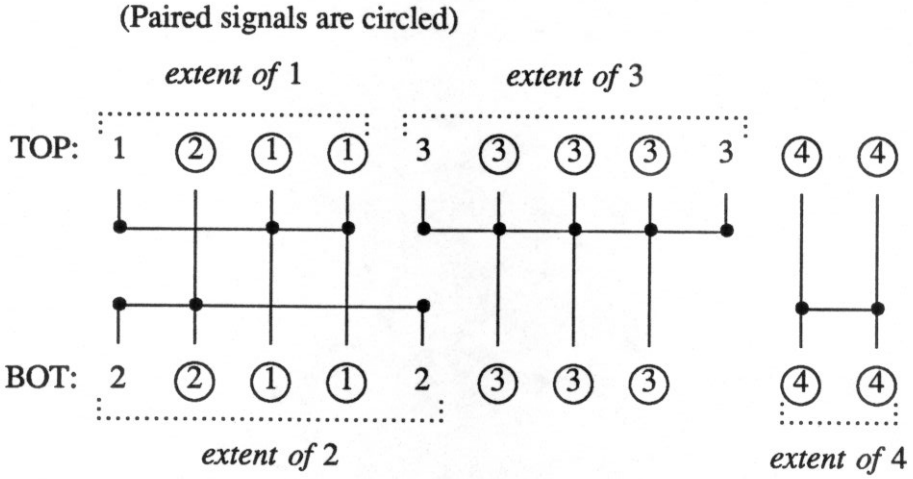
bottom (e.g., see figure 5-1(a)).

(5) Finally, place the paired clusters. For each signal, we define its *extent* to be the range of columns in which it is placed. For each pair, simply insert a column within the extent of the corresponding signal and place the clusters in that column. If the signal's extent is a single column, simply insert a column adjacent to that single column. If the signal's present extent is empty, then simply add a column after the present rightmost column in the channel. Update the extent to include the added column.

Top Clusters: (1), (1), (1), (2), (3), (3), (3), (3), (3), (4), (4)
Bot Clusters: (1), (1), (2), (2), (2), (3), (3), (3), (4), (4)



(a) Placement of excess sets



(b) Possible final placement and routing

figure 5-1 Example usage of algorithm for singular clusters

The final placement can be routed using at most 2 horizontal tracks, since the original extents of excess signals on the same side do not intersect and since the paired signals do not require any extra tracks. Figure 5-1(b) shows a possible final placement and routing.

5.2. Top Terminals Fixed, No Clusters, Length Bounded

Although Atallah and Hambrusch showed that the problem of density minimization for multi-terminal nets (no clusters) is NP-complete, even if the positions of the upper terminals are fixed [Atallah1], they left open the question of whether the problem remains NP-complete if all lower positions within the fixed-length channel are allowed (rather than some specified subset). We consider the following problem, which is a form of the *Bottom Terminal Placement* problem:

Length-Constrained Placement (LCP):

Instance: Integers d, L. A set of nets. A fixed placement of the top terminals (which doesn't necessarily include all the nets); and for each net i, k_i lower terminals. Terminals are placeable at any integer positions.

Question: Can the positions of the terminals be fixed such that the channel density is less than or equal to d and the channel length is less than or equal to L ?

LCP is clearly in NP. One may simply guess a placement of the terminals and check in polynomial time that the channel length and density requirements are met. Similarly to Atallah and Hambrusch's proof in [Atallah1], we give an NP-hardness reduction from a restricted form of *3-Partition*.

Start: A *3-Partition* instance π, with set A consisting of $3q$ elements, a size $s(a)$ for each $a \in A$ such that $B/4 < s(a) < B/2$, a target integer B, and $\sum_{a \in A} s(a) = qB$. Additionally, we require that $|B| = poly(q)$. Since *3-Partition* is strongly NP-complete, the problem is still NP-complete with the restriction that B is

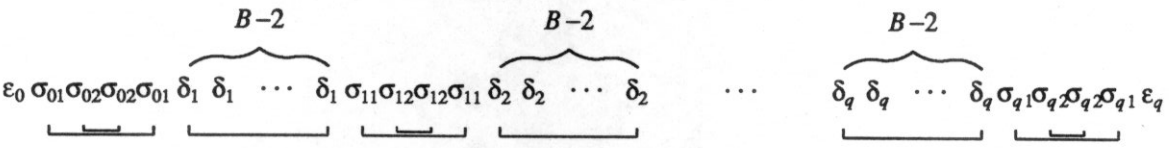
polynomial in the size the input [Garey].

The question is "Is there a partition of A into q (equal-size) disjoint subsets such that sum of each subset's elements' sizes equals B ?"

Reduction: Construct an instance λ of *LCP* with $2qB+2q+10$ terminals in $6q+4$ nets.

- For every i, $1 \le i \le 3q$, create net N_i, which requires exactly $s(a_i)$ lower positions and no upper positions, for $a_i \in A$. [qB terminals]
- For every j, $1 \le j \le q$, create net δ_j, which requires exactly $B-2$ upper positions and no lower positions. [$q(B-2)$ terminals]
- For every k, $0 \le k \le q$, create nets σ_{k1} and σ_{k2}, both of which require exactly 2 upper positions and no lower positions. [$4(q+1)$ terminals]
- Create 2 nets ε_0 and ε_q, each requiring exactly 2 lower positions and one upper position. [6 terminals]
- Let $d = 2$ and $L = qB + 2q + 6$.

The fixed placement of the upper terminals is as follows:



The question for λ is: "Can we place the bottom terminals so that the channel density is d and the channel length is L ?"

Correspondence between π and λ:

From the above construction of λ, we see that the channel density cannot be less than 2, since the top row itself has a density of 2. Also, the top has exactly $L = qB + 2q + 6$ terminals, filling up the entire length of the channel, so every bottom terminal must be placed directly underneath one of the top terminals. Since the density at

any column is at least 1, because of the top terminals, the bottom terminals may add at most 1 to the density of the channel. No bottom net may have terminals on both sides of any top terminal σ_{k2}, since the density at σ_{k2}, for $0 < k < q$, is 2.

The bottom terminals for net ε_0 must be placed in the first two bottom positions, else the net ε_0 would have to cross the positions of net σ_{02}, thus creating a column with density at least 3. Similarly, the bottom terminals for net ε_q must be placed in the last two positions on the bottom.

For the remaining qB nets of the form N_i, there are exactly qB bottom positions legally available, namely those under the terminals δ_j, $1 \leq j \leq q$, and the free ones under the terminals σ_{k1}, for $0 \leq k \leq q$. The allowable free positions are split into q **regions** of B consecutive free positions, each region being separated from its neighboring region by two unavailable positions (those under the terminals σ_{k2}, for $1 \leq k \leq q-1$). Each net N_i may only have terminals in a single region, since otherwise there would be an extra net passing through some position for some σ_{k2}, in which case the density must be at least 3.

If the elements are 3-partitionable in π, then we may place the bottom terminals N_i in λ according to the set partition of π. In the latter problem, a region of B consecutive free positions is equivalent to a set in the former problem. If we have a solution to λ, then we know that each region must contain exactly 3 different N_i (since $B/4 < s(a) < B/2$); thus, since each net is contained in only one region, there is a corresponding solution to π. We thus see that the given instance π of *3-Partition* is equivalent to our constructed instance λ of *LCP*.

□

Note that the proof above works even if the top terminals are relatively ordered rather than fixed in position — the upper terminals exactly fill up all positions on the top row, so there is only one way to position the terminals if they are relatively ordered.

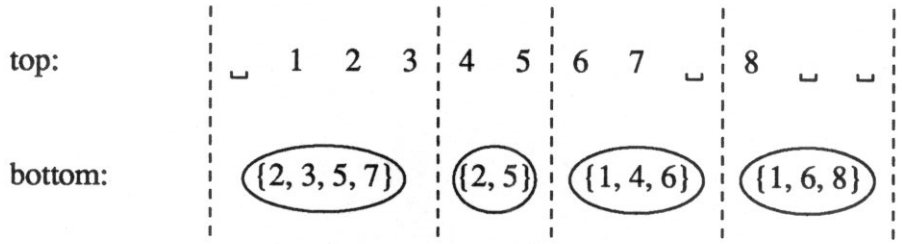
5.3. Transformation to a Smaller Problem

Now, consider using an algorithm that performs the placement of terminals in one gate at a time. Using the fixed placement from some gate i, we place the terminals in gate $i+1$ by trying to minimize the width of the channel between gates i and $i+1$. Let gate i correspond to the top row of the channel and gate $i+1$ correspond to the bottom row. We shall consider the conceptually simpler case where the top terminals are non-repeating. From the Weinberger array realm, this situation occurs if we only allow one line for each net to propagate from one gate to the next.

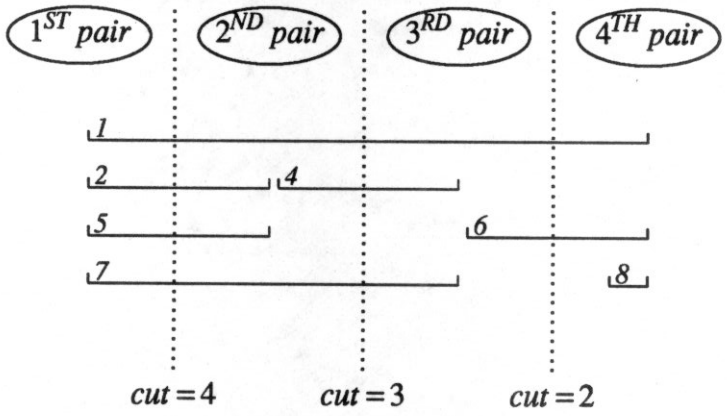
A natural algorithm structure for solving the above problem is first to decide the ordering and *extents* (list of occupied columns) of the clusters on the bottom, then to order and place the signals within each individual cluster. The problem of ordering the clusters on the bottom is akin to the problem of *Mincut Linear Arrangement* of the clusters, but there is the added complication of the fixed terminals on the top and determining the extent of each cluster. Still, the same sort of methods used for *MLA* (e.g., Rowen tries such methods as simulated annealing and iterative improvement [Rowen]) should perform reasonably well.

Suppose we have decided by some method the ordering and extents of the clusters. Then, for each cluster on the bottom, the set of terminals opposite the cluster's extent is fixed. We may independently solve the smaller subproblems of placement within each extent. By considering the set of terminals opposite a cluster as a unit, or *pseudo-cluster*, we can then garner some information on the range of the densities achievable for the channel. For example, figure 5-2(a) shows a possible ordering of clusters, given a fixed terminal placement of the top. Figure 5-2(b) collects each pseudo-cluster and its matching cluster, combining them to form a cluster/pseudo-cluster pair. The intervals represent the ranges of the nets within the channel. A "cut" at a position between two pairs is the number of intervals that cross that position. We call the maximum cut between any cluster/pseudo-cluster pairs the *exterior density*, and it is a lower bound on the achievable

channel density.



(a) Sample ordering of clusters



(b) Ranges of nets, cuts between cluster pairs

figure 5-2 Calculation of density lower bound

By a simple example, we can show that the exterior density may greatly underestimate the global density. Assume that the exterior density is $\frac{3}{2}l$. Figure 5-3 shows a cluster/pseudo-cluster pair of length l. On the top row, $\frac{l}{2}$ terminals with connections only to the right are placed to the left of $\frac{l}{2}$ terminals with connections only to the left. The bottom row consists of l terminals with connections to both the left and the right. In this bad case, the density is $2l$ at a cut in the center of the pair. This figure is significantly larger than the exterior density. This indicates the importance of choosing

well the orderings and extents of the clusters.

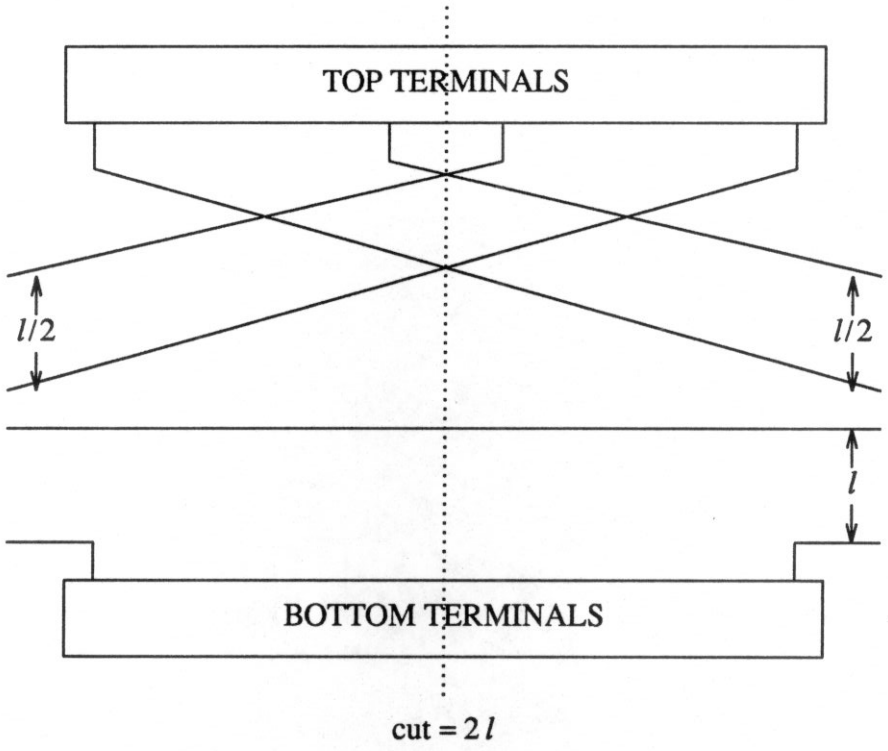


figure 5-3 Example interior placement

An interesting question is, given a cluster/pseudo-cluster pair, how close to the best achievable density can we get? In this problem, every net can have at most one terminal on the top (by assumption) and at most one terminal on the bottom (since clusters are *sets* of signals). In Section 5.4, we shall present an algorithm for placement within a cluster/pseudo-cluster pair. We show that the algorithm achieves the optimal density for the pair.

5.4. An Algorithm for Terminal Placement

In this section, we present an algorithm for the problem of terminal placement within a fixed-length channel under the following conditions:

- Exact positioning of the top terminals of the channel.
- Bottom terminals are totally flexible within the channel.
- A net may have at most a single top terminal and a single bottom terminal.
- Nets may enter from the left and exit to the right of the channel.

The fixed-length channel represents a cluster/pseudo-cluster pair. The top terminals are fixed, and the bottom terminals are the (non-repeating) signals from a single cluster. A net enters from the left if the net connects to a terminal to the left of the cluster/pseudo-cluster pair; a net exits to the right if it connects to a terminal to the right. We shall show that a placement obtained by our algorithm will achieve the optimum channel density, although we have not been able to prove whether or not the placement allows us to achieve the optimal width.

First, we define some terms that we use. A net is said to be *advancing* if it exits to the right of the given channel but does not enter from the left. Likewise, a net is *trailing* if it enters from the left of the given channel but does not exit to the right. If a net both enters from the left and exits to the right, we call it a *passing* net; if a net neither enters from the left nor exits to the right, it is a *stationary* net. A *single* net is one that only has a single terminal within the extent of the channel. A *multiple* net has two terminals, a top one and a bottom one. A multiple net whose terminals are both placed in the same column is called a *matched* net. Conversely, a multiple net whose terminals are placed in different columns is called *split*. Finally, we define a net's *extent* to be the interval from the leftmost column occupied by the net to the rightmost column occupied by the net. For this purpose, trailing nets and passing nets are assumed to occupy a position in an extra leftmost column; advancing nets and passing nets are assumed to occupy a position in an extra rightmost column.

As a final note, we shall sometimes use the net's name in referring to a terminal. We shall try to make clear which usage is meant.

5.4.1. The Algorithm

We now present our algorithm.

Algorithm 5.1:

(1) For each multiple net, match up the terminals, placing them in the same column.

(2) For each single advancing bottom net in turn, place its terminal in the rightmost available bottom position.

(3) For each single trailing bottom net in turn, place its terminal in the leftmost available bottom position.

(4) For each single passing bottom net in turn, place its terminal in any available position.

(5) Fix up the placement to achieve the optimal density by performing swaps of bottom terminals. The only allowed swaps are (a) a single advancing net for a matched trailing or passing net's bottom terminal (see figure 5-4) or (b) a single trailing net for a matched advancing or passing net's bottom terminal.

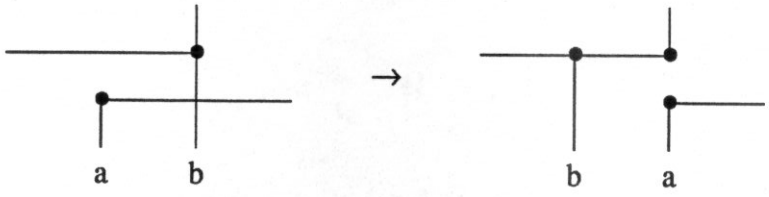


figure 5-4 Example of a swap

Before we give the procedure for step (5), we consider the placement produced by the first four steps of algorithm 5.1. The important features, which are clear from the construction of the placement, are that all multiple nets are matched and that all single advancing bottom nets are placed to the right of all single trailing bottom nets. We may divide the placement into 5 different areas, according the positions of 4 defining nets, as shown in figure 5-5. Here, $L_{\sin}$ is the leftmost single advancing bottom net, $R_{\sin}$ is the

rightmost single trailing bottom net, L_{mat} is the leftmost matched advancing or passing net, and R_{mat} is the rightmost matched trailing or passing net. The areas are defined as follows:

Area 3: The region lying between the two nets $R_{\sin}$ and $L_{\sin}$. Note that by steps (2) and (3) of algorithm 5.1, $R_{\sin}$ must lie to the left of $L_{\sin}$. If $R_{\sin}$ does not exist, then Area 3 extends all the way to the left end of the channel. Likewise, Area 3 extends all the way to the right end, if $L_{\sin}$ does not exist.

Area 1: The region lying to the left of all four of the defining nets and to the left of Area 3.

Area 5: The region lying to the right of all four of the defining nets and to the right of Area 3.

Area 2: The region lying to the right of Area 1 and to the left of Area 3. Note that this region may include up to two of the defining nets, if L_{mat} lies to the left of $R_{\sin}$. If L_{mat} lies to the right of $R_{\sin}$, then Area 2 consists of exactly the column containing $R_{\sin}$.

Area 4: The region lying to the right of Area 3 and to the left of Area 5. It's structure parallels that of Area 2.

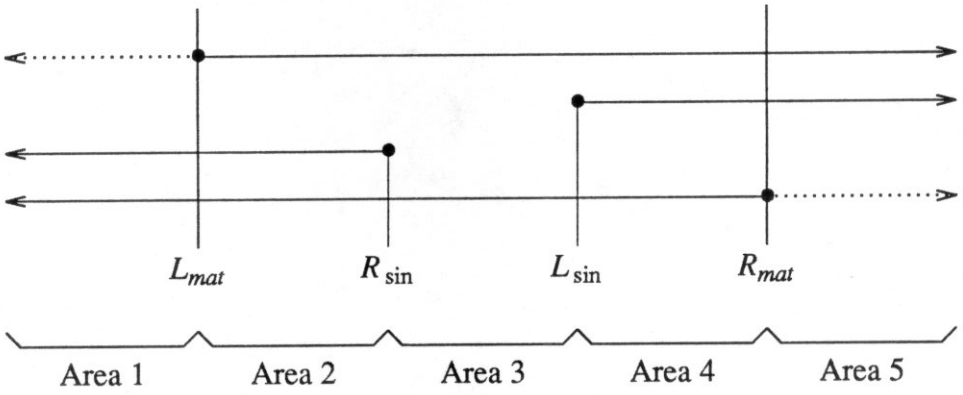


figure 5-5 The 5 areas after the first 4 steps of alg. 5.1

Our procedure for decreasing the density is as follows:

Step (5) of Algorithm 5.1:

- Calculate the densities $dens_1, \ldots, dens_5$ in all five areas.

  Let $d = \max [dens_1, dens_3, dens_5]$.

- While ($dens_2 > d$ and $R_{\sin}$ lies to right of L_{mat})

  Swap the positions of the bottom terminals for the nets $R_{\sin}$ and L_{mat}.

  Update $R_{\sin}$ and L_{mat}, thus expanding Areas 1 and 3, contracting Area 2. Update the values for $dens_1$, $dens_2$, $dens_3$, and d.

- While ($dens_4 > d$ and $L_{\sin}$ lies to left of R_{mat})

  Swap the positions of the bottom terminals for the nets $L_{\sin}$ and R_{mat}.

  Update $L_{\sin}$ and R_{mat}, thus expanding Areas 3 and 5, contracting Area 4. Update the values for $dens_3$, $dens_4$, $dens_5$, and d.

Now, we wish to show that the fix-up step number (5) in the algorithm finds an optimal placement, in terms of density. First, we make some observations about what types of terminals may be placed in the various regions during and at the end of the running of algorithm 5.1. We only consider bottom terminals, since the top terminals are fixed.

(A) All single passing bottom nets must be placed within area 3.

By step (2), (3), and (4), the single passing nets must start between the set of single trailing nets and the set of single advancing nets. Step (5) allows single trailing nets to move only to the left and allows single advancing nets to move only to the right, so the single passing net terminals always remain to the right of $R_{\sin}$ and to the left of $L_{\sin}$.

(B) No multiple trailing net has its bottom terminal placed to the right of its top terminal.

By step (1), all multiple nets start matched. The only possible movement of a

terminal occurs in step (5), where a multiple trailing net's bottom terminal is swapped with a single advancing net's terminal. But this movement always moves the muliple net's bottom terminal to the left.

(C) No multiple advancing net has its bottom terminal placed to the left of its top terminal.

Reasoning parallel to that for item (B).

The following two lemmas show that the density may not possibly be decreased in areas 1, 3, or 5. Analyzing the density of area 5 is sufficient to characterize the density of area 1, since the two regions are equivalent under a flip of the placement about a vertical axis.

We define the following terminology:

P = the set of passing nets, includes single and multiple nets

SAB = the set of single advancing bottom nets

MA = the set of multiple advancing nets

MT = the set of multiple trailing nets

Lemma 5.1:

At any step in the running of algorithm 5.1, the density within area 5, if the area exists, can not be decreased.

Proof:

If area 5 does not exist at the start of the algorithm, it will never exist, since the extent of area 3 never contracts. On the other hand, if area 5 exists at the start of the algorithm, an area 5 will always exist, since the extent of area 5 never contracts. We shall now assume that area 5 exists.

We consider what types of terminals and nets may occur within area 5 at any time during the running of algorithm 5.1. Since the top terminals are fixed, the density due solely to the single top terminals is also fixed; therefore, we only

calculate the density due to nets that have bottom terminals. Also, by definition, $R_{\sin}$ must lie to the left of area 5; therefore, all single bottom nets within area 5 are advancing. Thus, the density of area 5, at some point x, due only to nets having bottom terminals is (using "left" to mean at or to the left, "right" to mean at or to the right)

$$density_{bot}(x) = |P| + |\textit{subset of MA left of } x| + |\textit{subset of SAB left of } x| +$$

$$|\textit{subset of MT with top terminal right of } x|$$

Note that the passing nets in P always add to bottom density at x. Also, the indicated multiple advancing nets in MA must add to the bottom density, since the top terminal is fixed at a position at or to the left of x. A multiple trailing net may either be matched or split. Since by definition, the rightmost matched trailing net R_{mat} must be to the left of area 5, any multiple trailing net of MT within area 5 must be split. By (B) above, a split multiple trailing net has its top terminal placed to the right of its bottom terminal; therefore, if the net adds to the density at x, it always must, since the top terminal is fixed.

Finally, there are the single advancing bottom (SAB) nets within area 5. Consider a specific SAB net named t that is placed to the left of x. Since t is to the left of x, it adds to the density at x. In an arbitrary configuration, t may be placed to the right of x, thus not adding to the density at x. The question is whether or not there exists a configuration with t placed right of x such that the density at x is reduced. Consider the bottom nets placed to the right of x, since if t is moved to the right of x, then something originally to the right of x must be moved to the left of x. The bottom terminals placed to the right of x may be one of the following: a matched stationary net, a matched advancing net, or a single advancing net. The terminals may not be matched passing because R_{mat} is left of x. Note that there are no empty bottom positions to the right of t, since step (2) of

the algorithm places single advancing bottom nets as far right as possible. Also, none of the nets may be split, since if the bottom terminal of a split net lies to the right of x, then it could only have been split by step (5) of the algorithm. This means that $L_{\sin}$ lies to the right of x, which contradicts the original assumption that x lies in area 5. Replacing t by any of the three possible kinds of nets will not decrease the density at x; thus,the density within area 5 can not be decreased.

□

Lemma 5.2:

At any point in the running of algorithm 5.1, the density within area 3 is locally optimal.

Proof:

Using arguments similar to that for lemma 5.2 above, we can show that for some point x within area 3,

$$density_{bot}(x) = |P| + |\textit{subset of MA with either terminal left of } x| + |\textit{subset of MT with either terminal right of } x|$$

From (C), if the bottom terminal from some net in *MA* is at or left of x, then so is the top terminal; thus, that net must affect the density at x. Likewise, from (B), we may deduce that any net in *MT* that affects the density at x will affect the density at x for any placement of the bottom terminals. On the other hand, the passing nets in P will affect the density at all positions. Also, by definition of the areas, all single advancing bottom nets lie to the right of area 3 and all single trailing bottom nets lie to the left of area 3; therefore, they cannot add to the density at x. Thus, the density within area 3 is locally optimal.

□

Theorem 5.1:

The placement produced by algorithm 5.1 achieves the optimal density.

Proof:

Each swap in step (5) may possibly decrease the local density within area 2 or 4. The algorithm stops when the densities within areas 2 and 4 are both less than or equal to the maximum density within the areas 1, 3, and 5. Since, by lemmas 5.1 and 5.2, the local densities within areas 1, 3, and 5 are locally optimal, the termination of the algorithm under such a condition means that the global density of the channel has been optimized. Note that in the extreme case, one or both of the areas 2 and 4 disappear completely.

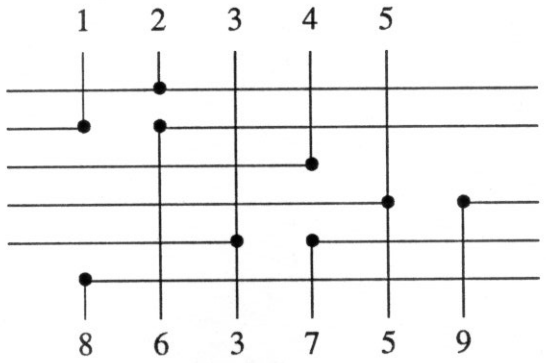
□

An example solution placement (along with an optimal wiring) is shown in figure 5-6. After steps (1) - (4), the order of the bottom terminals is 8, 6, 3, 7, 5, 9 (see figure 5-6(a)). The nets 8 and 5 define area 4. In step (5), terminals 8 and 5 are swapped, since doing so reduces the density. The new area 4 is defined by the nets 6 and 3. Since the density in area 4 is now equal to the density in areas 3 and 5, there is no need to swap terminals 6 and 3, and we are finished (see figure 5-6(b)).

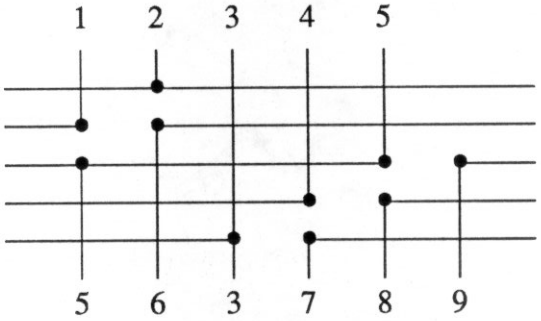
5.4.2. Considerations of Channel Width

The motivation for algorithm 5.1 was not just the minimization of the channel density, but also the minimization of the channel width. We realized that finding the width is a difficult task under most routing models, so we decided to consider the conceptually simple 2-layer, Manhattan, jogless model. In a model that is jogless, we may not have *jogs* — that is, each net's wires may occupy a portion of at most one horizontal track. So, wiring such as that in figure 5-7(a) may not be used. Also, the situation shown in figure 5-7(b) cannot occur, since it is indicative of a jog.

In a jogless routing model, the channel density is a useful estimate of the channel width, although density is more accurate for a routing model which allows jogs. The



(a) After steps (1)-(4)



(b) After step (5)

figure 5-6 Example result of running algorithm 5.1

factors that may throw off the estimate are the *vertical constraints*. A vertical constraint forces a certain net's horizontal segment to be above or below other net's segment. For example, see figure 5-7(b). Here, nets *1* and *2* have terminals in the same column, with *1* being on the top row and *2* being on the bottom row. Net *1*'s segment must be above net *2*'s segment; otherwise there would be a collision of the vertical segments for the two nets (or we would be forced to use jogs). Algorithm 5.1 was designed to keep to a minimum the number of vertical constraints while at the same time minimizing the density. Step (1) of the algorithm, which matches up all multiple net terminals, is the step responsible for keeping the number of vertical constraints small, since matched nets cannot form vertical constraints.

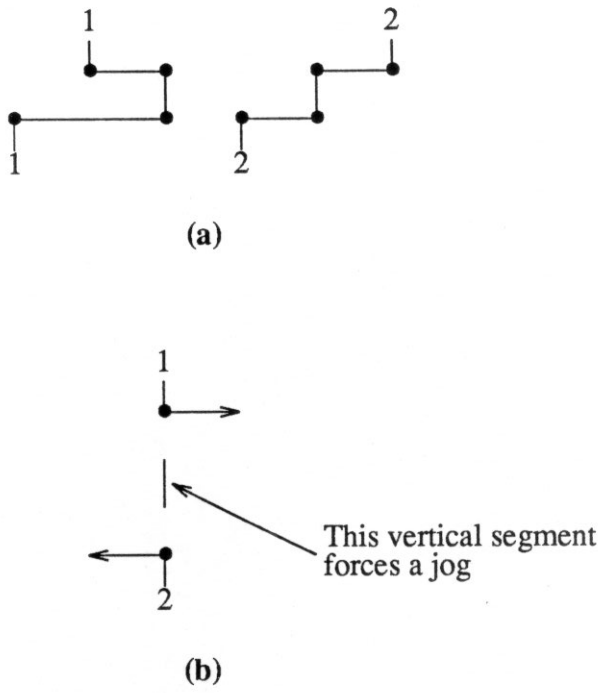


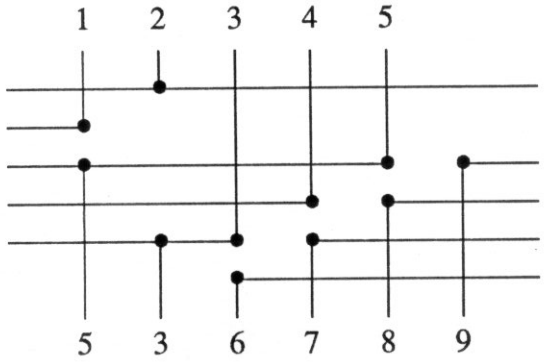
figure 5-7 Examples of wiring using jogs

To keep track of all the constraints, we may construct a *vertical constraint graph*, or VCG [Hashimoto]. A VCG is a directed graph containing a node v_i corresponding to each net i. An arc goes from node v_i to v_j in the VCG if there is a vertical constraint that the segment for net i must be above the segment for net j. For example, the VCG in figure 5-8(b) corresponds to the placement of terminals shown in figure 5-8(a).

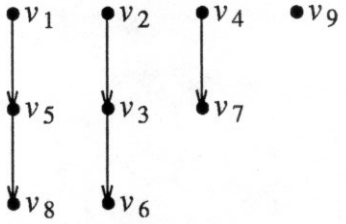
One point that requires mention is that if the VCG contains a cycle, then there is no way to route the channel without using jogs. A cycle requires that some net i occupy a track above the track occupied by itself; but this requires that i occupy a portion of at least two different tracks, which violates our condition that no jogging is allowed. We can show that the placement produced by algorithm 5.1 never contains a cycle in the VCG, hence there will be a legal wiring that does not use jogs.

Lemma 5.3:

The VCG corresponding to the placement produced by algorithm 5.1 does not



(a) Placement with density 5, width 6



(b) Corresponding vertical constraint graph

figure 5-8 Example Vertical Constraint Graph

contain a cycle.

Proof:

After step (1), all multiple nets are matched. Since a cycle requires nets to be split, there is no cycle in the VCG graph. The only step where multiple nets may be split is step (5). However, in each swap, the bottom terminal of the multiple net is replaced by a single bottom net's terminal. This single bottom net is never forced to be above any other net, so no cycle can be formed be formed in the VCG.

□

In fact, the form of the VCG is very highly constrained. A net may have at most two terminals, one on top and one on bottom, so each vertex in the VCG may have at most one incoming edge and one outgoing edge. Also, the maximal length for a chain in

the VCG is three: A chain can only be formed if there exists a split net. By the algorithm, the top terminal of a split net must match with either a single advancing net or a single trailing net. This stops the progress of this chain in the forward ("above") direction. The bottom terminal of a split net had to have swapped with a single bottom net, which must have been originally aligned with a single net; thus the progress of this chain is halted in the backward direction.

In order to achieve the best possible width, horizontal wire segments must be able to share the same track. Since we are using a jogless routing model, we may modify the VCG to account for nets sharing a track. Figure 5-9(a) shows a modified VCG that corresponds to the placement and wiring of terminals shown in figure 5-6(b). The set of squiggly lines is a maximal matching of the VCG nodes, and they correspond to the sharing of horizontal tracks used by segments in the wiring of the channel. Note that given a solution by algorithm 5.1, at most two nets may share any given track, since all nets that require a horizontal segment are advancing or trailing.

Sometimes, we are not able to achieve a width equal to the density, because certain nets cannot share. This type of situation can be observed in a modified VCG in figure 5-9(b), which corresponds to the terminal placement given in figure 5-8(a). For this placement, net *5* may only share with net *9*. This forces net *4* to share with net *8*, in turn forcing net *3* to share with net *7*. Nets sharing the same track also share their vertical constraints, though; so, net *6* cannot share with net *1*, otherwise a cycle $(v_1 \rightarrow v_5 \rightarrow v_8 \sim v_4 \rightarrow v_7 \sim v_3 \rightarrow v_6 \sim v_1)$ is produced. Therefore, the density of five cannot be achieved, and the width of the placement is six.

Note that the terminal placements of figure 5-6(b) and figure 5-8(a) are identical except that the terminals for nets *6* and *3* have swapped positions. The differing widths for the two placements illustrates the danger of making a seemingly innocuous switch. The problem arises because the matching given by the VCG in figure 5-9(a) is the unique maximal matching that does not create a cycle. In figure 5-9(a), there is a ''path'' from

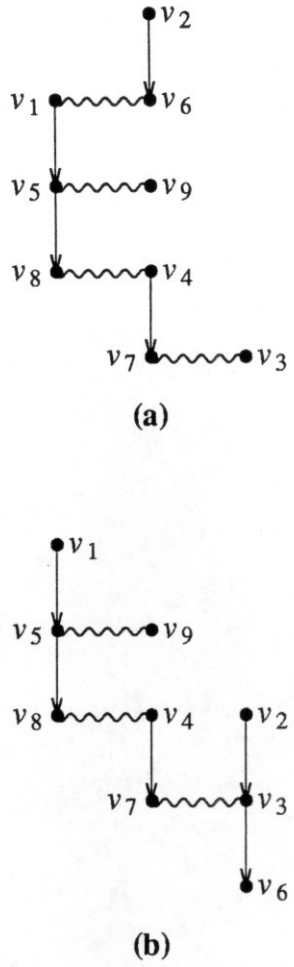


figure 5-9 Track sharing in a VCG

v_6 to v_3; therefore, net *6*'s horizontal segment must lie above net *3*'s horizontal segment. When the terminals are swapped, the constraint is added that net *3* must lie above net *6*, thus disallowing the original maximal matching. If there were no path between the two nodes, then swapping is safe if all matches in the matching are still allowed. Specifically, this requires that no node is shifted to the wrong side of the net with which it shares a track.

Unfortunately, we have not been able to show whether or not a solution found by algorithm 5.1 always allows a routing that achieves the optimal width. There are indications that certain configurations of terminals that are never allowed by the algorithm may

safely be omitted by an optimal placement of the terminals, but we have not been able to prove that the indications are true.

5.5. Open Problems and Future Research

For the problem of Single Channel Placement, there are still a number of problems of unknown complexity that involve restrictions on the order of clusters or signals and on the number or sizes of clusters within a gate. It would be satisfying to classify those problems.

For the problem of Bottom Terminal Placement, we might determine experimentally how well our suggested heuristic performs. Of course, much of that depends on good heuristics for the problem of ordering the clusters, which is quite similar to the problem of ordering gates.

Also, we would certainly like to know if algorithm 5.1 finds an optimal-width placement for the problem where the top terminals are placed in fixed positions and each net may have at most a single top terminal and a single bottom terminal. Section 5.4 shows some progress toward this goal for a jogless routing mode, but it does not answer the question fully. If algorithm 5.1 is not optimal, we would then try to develop another algorithm. There is also the question of how well algorithm 5.1 performs under a model that allows jogs. The complication under a model where jogs are allowed is that nets are no longer bound to have their horizontal segments all lie in a single track, thus enlarging the set of allowable solutions.

Finally, we might in the future implement the algorithm of placing terminals in one channel of the array at a time. It would be interesting to compare its performance against that of the heuristics studied in Chapter 4, although we expect the more global heuristics of Chapter 4 to perform better than the one-channel-at-a-time heuristic, which may find a good placement for early channels, only to force bad placements for later channels.

Chapter 6
Decreasing Lower Bounds by Channel Widening

So far, we have considered only density as a basis for estimating the amount of space needed by a particular channel assignment. Although channel routing under the 2-layer Manhattan model is NP-complete, some heuristics route most instances using a number of tracks very close to the density [Rivest1] [Yoshimura]. It has been noted, however, that some instances require many more tracks to route than would be indicated by the density [Brown]. This has led to a new lower bound called *flux*, which was introduced by Baker, Bhatt, and Leighton [Baker]. They showed that the channel width is upper bounded by a function that is linear in both density and flux.

In Chapter 5, we discussed some issues regarding the minimization of the channel density. In this chapter, we consider a later stage in the placement and routing of a single channel. Suppose that we have performed the density optimization, and we have vertically aligned our terminals the way we want them. To maintain our vertical alignment, we may not switch the order of terminals on either the top or the bottom, and we may not place an empty space in the top without placing an empty space in the corresponding position on the bottom (or vice versa). This means that we cannot change the density, which can only be changed by altering the order or alignment of terminals. It may still be possible to lower the width of the channel, though. If we have the flexibility to do so, we may add an empty column (place empty spaces in corresponding positions on both

† The results in this chapter are due to joint work with Fook-Luen Heng, Andrea S. LaPaugh, and Ron Y. Pinter.

the top and the bottom), thereby reducing some channel width metric other than density. Flux is an interesting metric to use because it captures the idea that routing space may be reduced by maneuvering signals via empty columns; adding empty columns can decrease flux.

In Section 6.1, we distill the essential properties of flux to describe a general class of channel width metrics; then we give an example of a metric, *smooth-flux*, which belongs to this class. *Smooth-flux* is derived from flux, but it has desirable properties not held by flux. In Section 6.2, we present an algorithm that solves the problem of decreasing a channel width metric to some target value, using as few extra columns as possible. This is followed by a proof that the algorithm finds an optimal solution. In Section 6.3, we consider the computational complexity of the lower bound metric *smooth-flux*. In Section 6.4, we analyze the timing requirements for our algorithm. We conclude with some extensions of our problem and some suggestions for future research.

6.1. Smooth-Flux

To define flux, Baker, Bhatt, and Leighton [Baker] consider *horizontal cuts* within the channel, where a horizontal cut isolates from all other terminals those terminals placed on a given row between two given points. Horizontal cuts may either be on the top or bottom row. Flux bounds the number of tracks needed to connect (to each other and to terminals outside the cut) the terminals in a horizontal cut. This is in contrast to a vertical cut, used in calculating density, which isolates the all terminals to the left of some point from those to the right of that point.

We formally define flux as follows. A *trivial net* is one comprised of exactly two terminals, both of which lie in the same column — a trivial column. The flux f is the largest integer for which some horizontal cut spanning $2f^2$ nontrivial columns splits at least $2f^2 - f$ nontrivial nets. A net is *unsplit* by a cut if all of this net's terminals lie

either inside the cut or outside the cut; otherwise, the net is *split*. (See figure 6-1.)

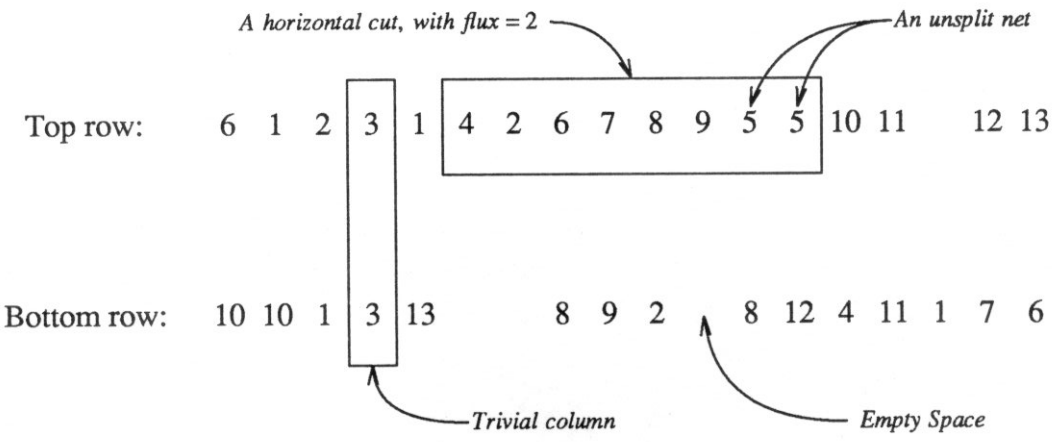


Figure 6-1 Flux terminology

Unfortunately, flux is somewhat anomalous, making it inappropriate as a measure to be minimized. Specifically, flux is not *monotonic*; that is, we may add an empty column and actually *increase* the value of the flux (see figure 6-2). This anomalous behavior occurs due to the coarse granularity between allowed cut sizes in calculating flux — all cuts are of size $2i^2$, for some integer i.

For our purposes, we want a metric that is well-defined and fairly easily computed for any given *window*, where a window is a cut that isolates a set of contiguous horizontal positions from all other positions. Notice that a horizontal cut is a window that contains only positions on the top of the channel or on the bottom of the channel. In general, a window may contain positions from both the top and bottom of the channel. For a window w, define $l(w)$ to be the leftmost terminal or empty space (or column) in w and $r(w)$ to be the rightmost. When we speak of decreasing the metric for a window w by adding empty columns, we consider the total number of columns spanned by w to be variable; but $l(w)$ and $r(w)$ are fixed. The following conditions must be satisfied by our metric:

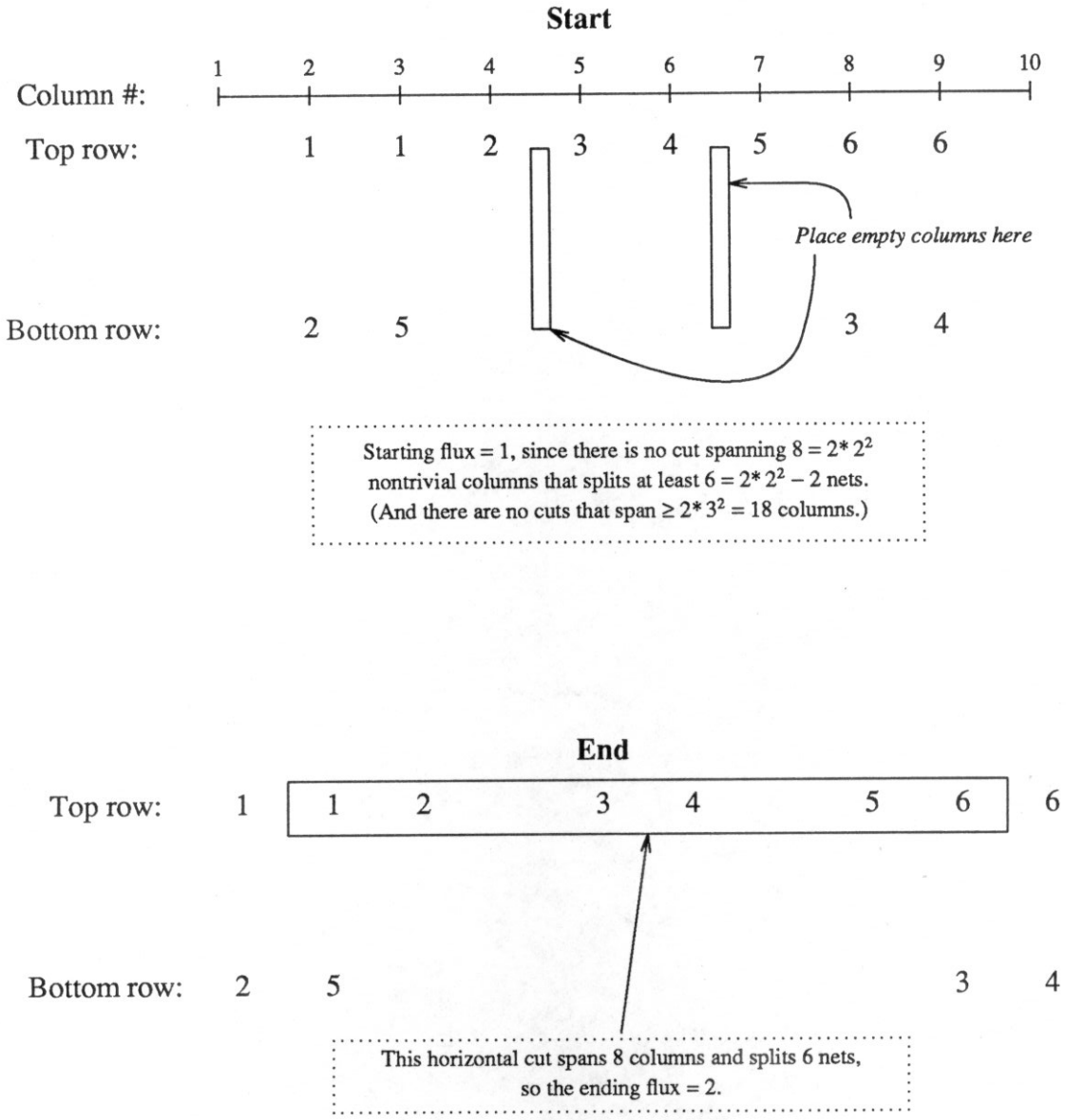


Figure 6-2 Example where adding empty columns increases the *flux*

(1) No sub-window of w may have its metric value increased by the addition of empty columns; that is, we require the metric to be monotonic (in window size).

(2) We must also be able to calculate how many extra columns must be added to a given window in order to lower the metric value for that window to a given target value.

(3) We must be able to compute the window's *critical extent* — the region within the window in which the extra columns must be added in order to achieve the target value.

The above conditions do not absolutely rule out the possibility of a window whose extent is not contiguous, but we do not know of any channel width metric that bounds the width for non-contiguous windows.

There may be a number of metrics that satisfy our conditions above. However, the following metric, derived from flux, is the only one we know of that does satisfy the conditions, can be decreased by adding empty columns, and provides a useful lower bound for channel width. We shall define a revised flux metric, *smooth-flux*, which may be calculated for any horizontal cut.

Define:

C = *the horizontal cut (window) under consideration.*

n = *the number of nontrivial columns in* C.

e = *number of empty spaces (positions not containing a terminal) within* C.

S = *number of split nets within* C *(does not include trivial nets).*

U = *number of unsplit (one-sided) nets in* C.

R = *number of redundant terminals in* $C = n - e - S - U$

Note that R counts the repeated terminals for the nets in S and U.

Definition:

Smooth-flux of the cut C, $smooth\text{-}flux(C)$, equals the smallest possible ω that satisfies the following equation:

$$\omega e + \omega(\omega+1) + (\omega-1)(U+R) \geq S \qquad (6.1)$$

The smooth-flux of a channel is the maximum of the smooth-flux values of all the

windows within the channel. Note that equation (6.1) is basically identical to the equation defining *flux*, except for the inclusion of the term $(\omega-1)(U+R)$. Solving the quadratic inequality above, we find the smooth-flux to be

$$f = \frac{-(e+U+R+1) + \sqrt{(e+U+R+1)^2 + 4(U+R+S)}}{2} \tag{6.2}$$

Or, since $S+e+U+R = n$,

$$f = \frac{-(n-S+1) + \sqrt{(n-S+1)^2 + 4(n-e)}}{2}$$

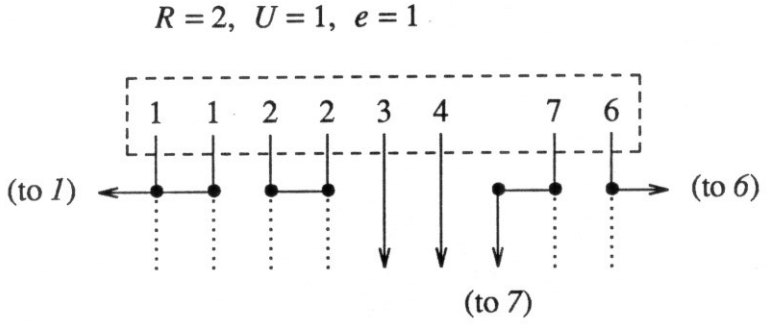
This measure bounds from below the number of tracks needed to route all the split nets in C. The following analysis holds for either a horizontal cut on the top or a horizontal cut on the bottom. The argument, similar to that in [Brown] and [Baker], bounds the number of split nets that may be routed into the correct column by using a particular track.

track 1 :

Two split nets may connect via track 1 to points outside the cut, adding 2 more columns available for routing (see nets *1* and *6* in figure 6-3). Also, e split nets may be routed into the e columns under the free spaces in C (see net *7* in figure 6-3). Finally, we might connect all unsplit nets and all redundant terminals. This would add $U+R$ more columns available for routing split nets on all remaining tracks (see nets *1* and *2* in figure 6-3).

track 2 :

For each available column, we may possibly connect together the terminals of a split net. There are (from step 1 above) at most $e+2+U+R$ columns available. In addition, 2 more split nets may be connected to points outside the cut, thus freeing up two more columns.



6 free columns after track 1 routing

Figure 6-3 Example routing on track 1

etc.

So, on track i, we may connect at most $e + 2i + U + R$ split nets, except on the first track, on which we may connect at most $e + 2$ split nets. Thus, the maximum number of split nets connected on ω tracks is:

$$\sum_{i=1}^{\omega} e + \sum_{i=1}^{\omega} 2i + \sum_{i=2}^{\omega} (U+R) =$$

$$\omega e + \omega(\omega+1) + (\omega-1)(U+R)$$

Since all split nets must be connected, this number must be greater than or equal to the total number of split nets, S, hence smooth-flux constitutes a lower bound.

We may compute the number of extra columns needed to lower the smooth-flux value from ω to t by simply using the definition. That is, we may substitute t for ω and re-write equation (6.1) above to find the total number of free columns needed:

$$e' \geq \frac{S - t(t+1) - (t-1)(U+R)}{t}$$

The number of columns we must add is simply $e' - e$. Since the smooth-flux value for a window v does not depend on the positions but rather on the *number* of split nets, free columns, unsplit nets, and redundant terminals, we may place the extra empty columns anywhere strictly between $l(v)$ and $r(v)$.

As to the question of whether or not adding an empty column to a window v can raise the smooth-flux, the answer is no.

Lemma 6.0:

For any placement of terminals within a channel, adding an empty column can not increase the channel's smooth-flux.

Proof:

To see that this is the case, consider any *new* window v' created by the addition of empty columns. Clearly, v' is equivalent to an original window v, plus some empty space(s). By equation (6.2) and since the only change is an increase in e, smooth-flux(v') ≤ smooth-flux(v).

□

Therefore, we may ignore the *new* windows if we wish to lower the value for smooth-flux.

6.2. The Flux Reduction Algorithm

Our problem is the following: Given an instance of a channel with top and bottom terminals, an appropriate channel width metric, and an integer T, how many empty columns must we add to reduce the channel width metric to T, and where do we place the empty columns?

In this section, we define some additional terms, present our algorithm, and give an example of the use of the algorithm.

6.2.1. Formal Problem Definition

We begin by restating the problem we wish to solve in the following, equivalent terms.

SATISFACTION OF WINDOW DEMANDS [*SWD*]

Input: Problem instance P consisting of

- a set of *windows* $W = \{w_1, w_2, \ldots, w_n\}$.
- each window $w \in W$ has an *extent* $(l(w), r(w))$, where $l(w)$ and $r(w)$ are positive integers, $l(w) < r(w)$. Of the two endpoints, $l(w)$ is the *start* of the extent and $r(w)$ is the *end* of the extent.
- each window $w \in W$ has a *demand* $\delta(w)$.

Objective:

- We are allowed to place columns at positions within the windows' extents. We may place a column at any non-integer position. No two columns may be placed in the same position.
- If at least $\delta(w)$ columns are placed within the extent of w, we say that the demand of w is *satisfied.*

Question:

- Where do we place columns so that the demand of every window in W is satisfied and so that we use as few columns as possible?

The correspondence between *SWD* and the problem of decreasing a channel width metric by adding columns is readily seen. Here, $w_{channel}$ is a window derived from an instance of the problem of decreasing the channel metric to a certain value by adding empty columns; $w_{channel}$ is a horizontal cut on either the top row or the bottom row. Then, w_{SWD} is the window corresponding to $w_{channel}$ in the instance of the *SWD* problem:

$$\textit{window } w_{SWD} \leftrightarrow \textit{window } w_{channel}$$

$$\textit{extent } (w_{SWD}) \leftrightarrow \textit{critical extent of } w_{channel}$$

$$\textit{demand } \delta(w_{SWD}) \leftrightarrow \textit{number of columns needed by } w_{channel}$$

We note that the size of an instance may expand in going from the channel realm to the *SWD* realm, since the windows are not explicitly given in the former problem. In Section 6.4, we show that this expansion is at most quadratic.

We note that we may re-write SWD as an interval graph problem. The transformed problem is an instance of *Weighted Clique Cover*:

- Create vertex-weighted graph $G_{wt} = (V, E, wt)$.

  (1) For each window w, create node $v_w \in V$.

  (2) For each node v_w, let the node's weight be $wt(v_w) = \delta(w)$.

  (3) For each pair of windows w, u whose extents overlap, create an edge $(v_w, v_u) \in E$ between the corresponding vertices.

- Objective:

  Assign non-negative values to the cliques of G_{wt} so that for a given node $v \in V$, the sum of the values on all cliques containing v is greater than or equal to $wt(v)$. Minimize the sum of the values over all cliques.

We know of no previous work on the general Weighted Clique Cover problem; however, the case where every vertex's weight equals 1 is simply the problem of *Partition into Cliques*, or finding a minimal-sized set of cliques that covers the set of vertices. The problem Partition into Cliques is known to NP-complete for general graphs [Garey2]. For the class of comparability graphs (a more general class than interval graphs), the problem is solvable in polynomial time [Golumbic]. Our contribution is a solution of the Weighted Clique Cover problem with arbitrary vertex weights for interval graphs.

6.2.2. Terminology for *SWD*

Before we proceed, we shall define some terms that we need in order to describe our algorithm and its proof.

Define:

- *Interval I*:

for integers j and k, with $j < k$, a segment (j, k) such that there are endpoints of windows at both j and k, but there are no window endpoints at any position between j and k. We denote $l(I) = j, r(I) = k$.

- *Critical Interval*:

  an interval I such that $l(I) = l(u)$ for some $u \in W$ and $r(I) = r(v)$ for some $v \in W$. Intervals u and v may possibly be identical.

- $\rho(I)$:

  where I is an interval. This is the set of windows whose extents contain the extent of interval I ,i.e., $w \in \rho(I)$ if $l(w) \leq l(I)$ and $r(w) \geq r(I)$.

- $I > J$:

  where I, J are intervals, means that $\rho(I) \supsetneq \rho(J)$; we say that I ''dominates'' J.

- $\eta(x)$:

  where x is either an interval or a window, is the number of columns placed within x's extent, for some assignment of columns.

- *Column Assignment:*

  a possible solution to a problem instance. It tells how many columns are placed in each critical interval (we show later, in Lemma 6.1, that we do not have to place columns in noncritical intervals). An assignment is given as a sequence of numbers $A = \{\eta(I_1), \eta(I_2), \ldots, \eta(I_M)\}$, where $I_1, I_2, \ldots, I_M$ are all the critical intervals, from left to right.

- *(Total) Cost:*

  The (total) cost of a column assignment A is $c(A)$, the total number of columns used by A 's constituents, namely $c(A) = \sum_{k=1}^{M} \eta(I_k)$.

6.2.3. The Algorithm

Here is our algorithm to solve *SWD*. The idea is to place columns in the critical intervals. Later, in Lemma 6.1, we prove that we don't have to place columns in noncritical intervals. We start with the leftmost critical interval and proceed with each critical interval in turn. As we consider a critical interval, we place enough columns in it so that the window demand is satisfied for any window whose extent includes the present critical interval, but whose extent does not include any critical intervals yet to be considered.

Greedy algorithm:

Let $W = \{w_1, \ldots, w_n\}$ be the set of windows, where $w_i = (l(w_i), r(w_i))$;

$\{\delta(w_i) \mid 1 \le i \le n\}$ is the set of window demands.

Calculate

$I = \{I_1, \ldots, I_M\}$ set of critical intervals, where $I_i = (l_i, r_i)$ and $l_i < l_{i+1}$.

Initialization:

$W_0 = W$

$U_0 = \varnothing$

$\eta_G(I_0) = 0$, where I_0 is a dummy variable

$\delta_0(w) = \delta(w)$, for *all* $w \in W$

for $i = 1$ *to* M **do**

$V_i = \{ v \in W \mid v \in \rho(I_i)$ and $v \notin \rho(I_j)$ for $j > i \}$

$U_i = \rho(I_i) - V_i$

Initialize $\eta_G(I_i) = 0$

end

for $i = 1$ *to* M **and** while W_{i-1} is nonempty **do**

(1) Update window demands at i^{th} iteration,

$$\delta_i(w) = \begin{cases} \delta_{i-1}(w) & , \text{if } w \notin U_{i-1} \\ \max \{ 0, \delta_{i-1}(w) - \eta_G(I_{i-1}) \} & , \text{if } w \in U_{i-1} \end{cases}$$

(2) Compute $\eta_G(I_i)$, the number of columns to be placed in I_i,

$$\eta_G(I_i) = \max_{v \in V_i} \{\delta_i(v)\}$$

(3) Update the current set of windows,

$$Wi = W_{i-1} - V_i$$

end

We shall denote the column assignment produced by the greedy algorithm as $G = \{\eta_G(I_1), \ldots, \eta_G(I_M)\}$.

6.2.4. Complexity of the Greedy Algorithm

We now analyze the run-time for the greedy algorithm.

(A) The initialization:

Sort the endpoints of windows in W and scan the sorted endpoints from left to right in order to compute I. This takes $O(n \log n)$ time.

Let P denote the total size of all $\rho(I_i)$, i.e., $P = \sum_1^M |\rho(I_i)|$. It takes $O(P)$ time to compute V_i, U_i, and $\rho(I_i)$, for $i = 1$ *to* M.

(B) The main loop:

Steps (1), (2), and (3) all take $O(P)$ time.

Thus, the complexity of the Greedy algorithm is $O(P + n \log n)$ time. Clearly, P is bounded above by $M * n$. M is the number of critical intervals, which must be less than n, so $P=O(n^2)$. In the worst case, P is of order n^2; therefore, the running time of the algorithm is $O(n^2)$.

6.2.5. Example Usage of the Greedy Algorithm

Below is an example of the usage of the greedy algorithm. We show the input and the running of the algorithm for an example where the number of critical intervals is three.

Input:

$W = \{w_1, w_2, w_3, w_4\}$

Window	Extent		Demand
	Start	End	
w_1	1	3	3
w_2	2	4	6
w_3	3	7	8
w_4	5	8	4

(See figure 6-4)

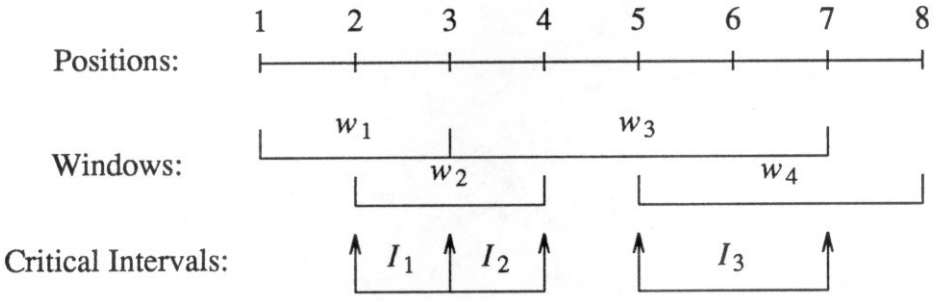


Figure 6-4 Example Input to Greedy Algorithm

The critical intervals are $I_1 = (2,3)$, $I_2 = (3,4)$, and $I_3 = (5,7)$.

Iteration 1: ($i = 1$)

$W_0 = \{w_1, w_2, w_3, w_4\}$

Demands are $\delta_1(w_1) = 3$, $\delta_1(w_2) = 6$, $\delta_1(w_3) = 8$, $\delta_1(w_4) = 4$

We find that $V_1 = \{w_1\}, U_1 = \{w_2\}$ and so $\eta_G(I_1) = 3$. We place 3 columns within I_1, then we continue with the algorithm.

Iteration 2: $(i = 2)$

$W_1 = \{w_2, w_3, w_4\}$

Demands are $\delta_2(w_2) = 3$, $\delta_2(w_3) = 8$, $\delta_2(w_4) = 4$

We find that $V_2 = \{w_2\}, U_2 = \{w_3\}$ and so $\eta_G(I_2) = 3$. We place 3 columns within I_2, then we continue with the algorithm.

Iteration 3: $(i = 3)$

$W_2 = \{w_3, w_4\}$

Demands are $\delta_3(w_3) = 5$, $\delta_3(w_4) = 4$

We find that $V_3 = \{w_3, w_4\}, U_3 = \varnothing$ and so $\eta_G(I_3) = 5$. We place 5 columns within I_3, then we are finished since $i = M$.

Totally, we have used 11 columns (which is optimal), and the assignment is 3, 3, 5.

□

6.3. Proof of Optimality

6.3.1. Three Properties

We want to prove that our greedy algorithm finds an optimal solution to *SWD*. First, we present three properties of solutions for the *SWD* problem, which we use implicitly in our proofs.

Properties:

(1) Given a problem instance P, a solution must exist if all window demands are finite.

(2) We only need to specify in which interval to place a column. The actual numerical position is unimportant.

(3) Given 3 consecutive intervals I_1, I_2, and I_3, if $w \in \rho(I_1)$ and $w \in \rho(I_3)$, then $w \in \rho(I_2)$.

Proof:

Property (1) is added for completeness, to show that a solution to our problem exists. We may simply place exactly $\delta(w)$ columns in the extent of all windows $w \in W$. This may be done since there are an infinite number of positions within any window.

Property (2) comes from the fact that we do not restrict where we may place a column. Placing a column at a particular position will only affect one interval; and for a given interval, any two columns placed within the interval will affect exactly the same set of windows, namely $\rho(I)$. So, from now on, we shall only specify the interval in which we are placing a column.

Property (3) ensures that a window may not "skip over" the extent of an interval. By hypothesis, $l(I_1) \leq l(I_2)$ and $r(I_3) \geq r(I_2)$. Since by definition of ρ, $l(w) \leq l(I_1) \leq l(I_2)$ and $r(w) \geq r(I_3) \geq r(I_2)$, $w \in \rho(I_2)$.

□

6.3.2. Critical Intervals

So far, we have constrained the solution set so that columns must be placed only in intervals, and there is a limited number of intervals — at most $2*|W| - 1$, since there are only $2*|W|$ window endpoints. But we can even further limit the intervals that we must consider to the set of critical intervals. The upper bound on the number of critical intervals is $|W|$, since there are only $|W|$ window *start* points (or *end* points).

Lemma 6.1:

For any instance of the *Satisfaction of Window Demands* problem, there is an optimal solution for which columns are placed only in critical intervals.

Proof:

First, observe that the operator ''>'' defines a partial ordering on the set of intervals, by the definition of ''>'' and since the operator ''$\supsetneq$'' defines a partial ordering on a set of sets.

Second, assume we are given two intervals I and J, with $I > J$. If a column is placed in J, we may shift it to I; and following the shift, the column still affects all the windows that it originally affected. The reason for this is that a column placed in interval J affects only those windows that intersect J, namely $\rho(J)$. If the column is moved to interval I, it now affects the set of windows $\rho(I) \supsetneq \rho(J)$.

Given the two observations above, we conclude that we only need to place columns in *maximal* (with respect to ''>'') intervals. The rest of this proof shows that all maximal intervals are critical intervals.

Assume that an interval $J = (j,\ k)$ is not a critical interval. Let $I = (i,\ j)$ be the interval immediately to the left of J, and let $K = (k,\ m)$ be the interval immediately to the right of J. There are 2 cases to consider:

CASE ①: $k = l(v)$, for *some* $v \in W$; $k \neq r(u)$, for *any* $u \in W$

Any point x in window v satisfies $x > l(v) = k$, so v does not intersect interval J; that is, $v \notin \rho(J)$. On the other hand, $l(v) = k < m \leq r(v)$, since there are no window endpoints between k and m; thus, $v \in \rho(K)$.

Consider any window $w \in \rho(J)$. By definition, $l(w) \leq j < k \leq r(w)$. But since $k \neq r(w)$ and since there are no window endpoints between k and m, $k < m \leq r(w)$. In other words, $w \in \rho(K)$.

So, we see that $\rho(K) \supsetneq \rho(J)$; or $K > J$.

CASE ②: $j = r(v)$, for *some* $v \in W$; $j \neq l(u)$, for *any* $u \in W$

With arguments similar to those for case ①, we can show that $v \in \rho(I)$, but $v \notin \rho(J)$. Also, for any $w \in \rho(J)$, $w \in \rho(I)$. So, we find that $I > J$.

The only intervals that do not fall into either class ① or class ② are those that are critical intervals. And since intervals in class ① and ② are dominated by adjacent intervals, the only intervals that are not dominated are critical intervals; thus, all maximal intervals must be critical intervals. We have already shown that we only need to place columns in maximal intervals, so we conclude that we only need to place columns in critical intervals.

□

6.3.3. Performance of the greedy algorithm

We now wish to show that the column assignment produced by the greedy algorithm satisfies all window demands.

Lemma 6.2:

The assignment $G = \{ \eta_G(I_1), \ldots, \eta_G(I_M) \}$ constructed by the Greedy Algorithm satisfies all window demands.

Proof:

Consider a window w whose extent includes critical intervals I_s through I_t, i.e. $r_{s-1} \le l(w) \le l_s < r_t \le r(w) \le l_{t+1}$. (See figure 6-5)

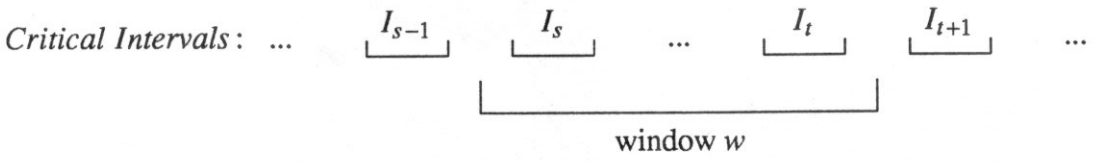


Figure 6-5 Window extending from I_s to I_t

Observe the following :

(i) $w \in V_t$, *since* $w \in \rho(I_t)$ *but* $w \notin \rho(I_{t+1})$

(ii) The number of columns placed in w is

$$\eta_G(w) = \eta_G(I_s) + \cdots + \eta_G(I_t) = \sum_{k=s}^{t} \eta_G(I_k).$$

(iii) From step (1) of the greedy algorithm,

$$\delta_s(w) = \delta_{s-1}(w),$$

$$since\ w \notin \rho(I_{s-1}) \supseteq U_{s-1}$$

Similarly,

$$\delta_k(w) = \delta_{k-1}(w), \text{ for } 1 \le k < s$$

Therefore,

$$\delta_s(w) = \delta_{s-1}(w) = \cdots = \delta_0(w) = \delta(w)$$

From (i) above, $w \notin V_k$, for $k \neq t$. So from step (1) of the greedy algorithm, the window demand for w from the $s+1^{st}$ to the t^{th} iteration is

$$\delta_{k+1}(w) = \max \{ 0, \delta_k(w) - \eta_G(I_k) \},$$

$$since\ \ w \in \rho(I_k) - V_k = U_k\ ,\ \text{for } s \le k < t$$

i.e.,

$$\delta_{k+1}(w) \ge \delta_k(w) - \eta_G(I_k),\ \text{for } s \le k < t \tag{6.1}$$

Combining equation (6.1) and (iii) gives

$$\delta_t(w) + \eta_G(I_{t-1}) + \cdots + \eta_G(I_s) \ge \delta_s(w) = \delta(w) \tag{6.2}$$

But from step (2) in the greedy algorithm,

$$\eta_G(I_t) = \max_{u \in V_t} \{\delta_t(u)\} \ge \delta_t(w)\ , \tag{6.3}$$

Therefore, (ii), (6.2), and (6.3) give

$$\eta_G(w) = \sum_{k=s}^{t} \eta_G(I_k) \ge \delta(w)\ \text{, for } all\ w \in W.$$

Hence, all window demands are satisfied.

□

6.3.4. Transformation of solutions

The following lemma shows that the greedy construction of solutions is justified. We shall use it later to derive one optimal solution from another optimal solution, in order to obtain one in the form we need.

Lemma 6.3:

For any two solutions,

$$H = \{ \eta(I_1),...,\eta(I_M) \}$$

$$H' = \{ \eta'(I_1),...,\eta'(I_M) \}$$

where there is a $t < M$ *s.t.* $\eta(I_j) = \eta'(I_j)$ for $1 \le j < t$ and $\eta(I_t) < \eta'(I_t)$ [i.e., H and H' agree for the first $t-1$ intervals, and H' has more columns at the t^{th} interval],

We can find another solution,

$$H'' = \{ \eta''(I_1),...,\eta''(I_M) \}$$

such that

$$\begin{cases} \eta''(I_j) = \eta'(I_j) = \eta(I_j) \ , \text{for } j < t \\ \eta''(I_t) = \eta(I_t) \\ cost(H'') = cost(H') \end{cases}$$

Proof:

Intuitively, we may take solution H' and *ship* the extra number of columns, $\eta'(I_t) - \eta(I_t)$, in interval I_t to interval I_{t+1} without changing the cost and validity of the solution. We can show this by constructing H'' as follows:

$$\eta''(I_j) = \eta'(I_j) \ , \text{for } j < t$$

$$\eta''(I_t) = \eta(I_t)$$

$$\eta''(I_{t+1}) = \eta'(I_{t+1}) + \eta'(I_t) - \eta(I_t)$$

$$\eta''(I_k) = \eta'(I_k) \ , \text{ for } k > t+1$$

There is no change in cost:

$$cost(H'') = \sum_{i=1}^{M} \eta''(I_i)$$

$$= \sum_{j=1}^{t-1} \eta'(I_j) + \eta(I_t) + \eta'(I_{t+1}) + \eta'(I_t) - \eta(I_t) + \sum_{k=t+2}^{M} \eta'(I_k)$$

$$= \sum_{i=1}^{M} \eta'(I_i) = cost(H')$$

Validity of H'':

We may classify a given window to be one of three different types, depending on the window's position relative to critical intervals I_t and I_{t+1}. The three types are

(1) $r(w) \le l(I_{t+1})$

Window w lies entirely to the left of interval I_{t+1}.

(2) $l(w) \ge r(I_t)$

Window w lies entirely to the right of interval I_t.

(3) $l(w) \le l(I_t) \le r(I_{t+1}) \le r(w)$

Window w contains the extents of intervals I_t and I_{t+1}.

(See figure 6-6).

We shall now show that for any window w, $\eta''(w) \ge \delta(w)$; therefore, H'' is also a solution. Recall that $\eta(w) \ge \delta(w)$ and $\eta'(w) \ge \delta(w)$, since H and H' are both solutions. We assume that window w contains all critical intervals from I_p to I_q, $p \le q$, and no other critical intervals.

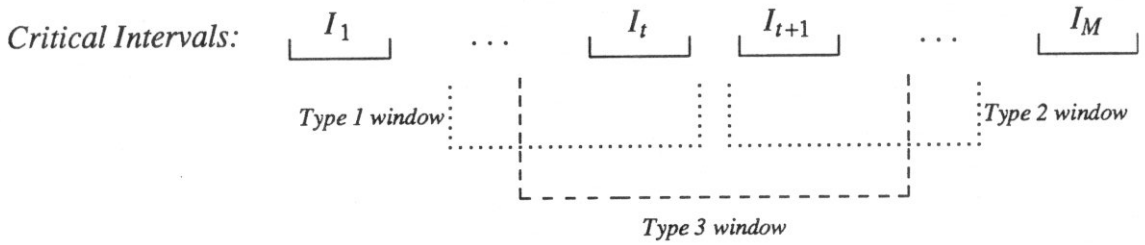


Figure 6-6 Three different types of windows

Type 1 window:

$$\eta''(w) = \eta''(I_p) + \cdots + \eta''(I_q) \quad , \text{for } q \le t$$

$$= \eta'(I_p) + \cdots + \eta'(I_{q-1}) + \eta(I_q)$$

$$= \eta(I_p) + \cdots + \eta(I_q) = \eta(w) \ge \delta(w) \quad , \textit{since } \eta'(I_j) = \eta(I_j) \text{ for } j < t$$

Type 2 window:

$$\eta''(w) = \eta''(I_p) + \cdots + \eta''(I_q) \quad , \text{for } p > t$$

$$[\textit{case (i)}] \quad = \eta'(I_p) + \cdots + \eta'(I_q) = \eta'(w) \ge \delta(w) \quad , \text{if } p > t+1$$

$$[\textit{case (ii)}] \quad = \eta'(I_{t+1}) + \eta'(I_t) - \eta(I_t) + \eta'(I_{t+2}) + \cdots + \eta'(I_q)$$

$$= \eta'(w) + (\textit{positive value}) > \eta'(w) \ge \delta(w) \quad , \text{if } p = t+1$$

Type 3 window:

$$\eta''(w) = \eta''(I_p) + \cdots + \eta''(I_t) + \eta''(I_{t+1}) + \cdots + \eta''(I_q)$$

$$= \eta'(I_p) + \cdots + \eta(I_t) + \eta'(I_{t+1}) + \eta'(I_t) - \eta(I_t) + \cdots + \eta'(I_q)$$

$$= \eta'(w) \ge \delta(w)$$

□

6.3.5. The Main Result

Finally, we are ready to prove our main result.

Theorem 6.1:

The column assignment $G = \{ \eta_G(I_1),...,\eta_G(I_M) \}$ constructed by the Greedy Algorithm is an optimal solution.

Proof:

Assume there is another solution H such that

$$cost(H) = \sum_{i=1}^{i=M} \eta'(I_i) < \sum_{i=1}^{i=M} \eta_G(I_i) = cost(G)$$

Then by Lemma 6.3, we may transform H into another solution $H' = \{ \eta'(I_1),...,\eta'(I_M) \}$ such that $cost(H') = cost(H)$ and H' matches G for as long as possible. Thus, we know that

$$\eta'(I_j) = \eta_G(I_j) \quad , \text{for } j < t$$

and, since $cost(H') < cost(G)$,

$$\eta'(I_t) < \eta_G(I_t) \quad , \text{for } \textit{some } t$$

Consider the column assignment for critical interval I_t [step (2) of greedy algorithm]:

$$\eta_G(I_t) = \max_{w \in V_t} \{\delta_t(w)\}$$

Let $v \in V_t$ be a window that achieved the maximum, i.e. $\eta_G(I_t) = \delta_t(v)$. Let the intervals intersected by v be $I_x, \ldots, I_t$, for some $x \le t$. Then, by reasoning analogous to that in the proof of Lemma 6.2,

$$\delta(v) = \delta_x(v) = \sum_{j=x}^{t} \eta_G(I_j) = \eta_G(v)$$

Then,

$$\eta'(v) = \sum_{j=x}^{t} \eta'(I_j)$$

$$= \sum_{j=x}^{t} \eta_G(I_j) + (\eta'(I_t) - \eta_G(I_t))$$

$$< \eta_G(v) = \delta(v) \quad , \textit{since } \eta'(I_t) - \eta_G(I_t) < 0.$$

But this means that v's demand is not satisfied by H', which contradicts the validity of H' as a solution. By Lemma 6.2, G is a solution. We have shown that there is no solution with cost less than that for G; thus, we conclude that G is an optimal solution.

□

6.4. Notes on the *smooth-flux* metric

There are a couple of observations that help reduce the time complexity to lower the smooth-flux to a given target value. One simple point, which applies to any other appropriate metric, is that we only have to consider a window if its metric value is greater than the target value. If the value is less than or equal to the target value, we may just leave that window alone.

An important point in lowering the smooth-flux is that we may disregard any window w that contains an empty space in either its leftmost or its rightmost column. To see why this is so, consider the largest subwindow w' of w such that:

- w' contains all the terminals contained in w
- w' contains terminals in both its leftmost and rightmost columns (see figure 6-7).

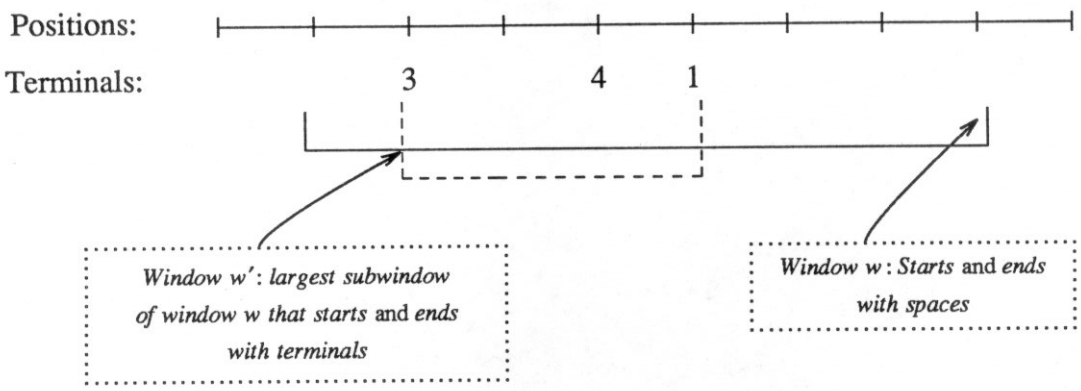


Figure 6-7 Important subwindow

From w to w', the values for U, R, and S remain unchanged, but the value for e has decreased. From equation (6.2), the change in smooth-flux with respect to e is:

$$\frac{\partial f}{\partial e} = \frac{1}{2}\left\{-1 + \frac{(e+U+R+1)}{\sqrt{(e+U+R+1)^2 + 4(U+R+S)}}\right\}$$

Since $(e+U+R+1) \le \sqrt{(e+U+R+1)^2 + 4(U+R+S)}$, $\frac{\partial f}{\partial e} \le 0$; so if we decrease e, f increases. In other words, the smooth-flux value for w' is greater than or equal to the value for w. And since any empty column we place within w' will also lie within w, we may decrease the smooth-flux of w by decreasing the smooth-flux of w'.

Now, we shall compute the time complexity of calculating the starting value of smooth-flux for a problem instance. Let N be the maximum of the number of top row terminals and the number of bottom row terminals. As a preliminary step we order each row of terminals by position; this takes $O(N \log N)$ time. Since we only need to consider the windows that begin and end with terminals (but we need to consider both top windows and bottom windows), there are at most $2 * \binom{N}{2} = N(N-1)$ relevant windows. We scan through the relevant windows by continuously fixing the left endpoint (at some terminal's position) and varying the right endpoint (at another terminal's position). While doing this, determine the values n, S, U, and R $[e = n-(S+U+R)]$ for each

window. The update time for each window is constant, since a window with only one terminal is immediate and all other windows only have to increment or decrement each value by at most 1 from the values for the previous window. With these values, we can compute the smooth-flux for each window. Totally, this takes $O(N^2)$ time.

For a given channel and a target value T, the total time required to reduce the smooth-flux of the channel to T is $O(N^4)$, since there are $O(N^2)$ windows.

6.5. Extensions and future work

Noting that we know of only one example of an appropriate, non-trivial metric for our algorithm, it would be interesting to find other useful metrics that belong to the same class. Also, there are a number of possible extensions and modifications for our problem of reducing a given cost metric. Here are two very closely related problems:

(1) The extra columns may be constrained to be placed within a set of *allowable* line segments. This situation might arise if the terminals are divided into *components* that are rigid. In such a case, we may expand or contract the amount of space only *between* components.

(2) We may be given a fixed number of extra columns, with the task of placing these columns in order to reduce the cost metric as much as possible.

The problem (1) above can be solved by our own greedy algorithm, with the modification that for each *allowable* segment, instead of considering only the critical intervals, we only consider placing columns in the maximal (with respect to ">") intervals that intersect the *allowable* segment. Problem (2) can be solved by applying our greedy algorithm and using a binary search on the target value for the cost metric. Since the value for smooth-flux, like the value for flux, is bounded by $O(\sqrt{n})$, where n is the total number of nets, we only need to consider at most $O(\log n)$ target values. The total number of windows is bounded by N^2, where N is the maximum number of terminals on the top or bottom rows; the number of critical intervals is bounded by N. From our

earlier figure of $O(M * N)$ for the greedy algorithm, where M is the number of critical intervals, the total running time is $O(N^3 \log n)$. It is not clear that this is the optimal method for solving (2), though. More research might yield a better algorithm. It may also be possible to decrease the running time of the greedy algorithm by finding a better data structure for the $\rho(I)$'s.

An interesting extension of our problem is the following open problem:

(3) Instead of adding *columns*, we may add arbitrary *spaces* on the top or bottom row, as long as the total number of columns does not exceed a given limit. This situation might represent a channel where the terminal ordering is fixed, but the terminals are otherwise flexible.

Problem (3) is similar in flavor to the problems examined in [Gopal] and [LaPaugh3]. This extension adds substantial complexity when *smooth-flux* is considered because trivial nets can be created.

Chapter 7
Conclusions

We have presented and discussed a wide range of problems concerned with certain placement and routing aspects of an automatic logic compiler.

The problem of ordering the gates to be implemented has led us into the realm of graph theory, where we translated our problem into the task of finding a good linear arrangement of a graph's vertices. Here, each vertex represents either a gate or an input signal. We considered two goals for our circuit — minimize the propagation delay and minimize the area.

To model the concept of propagation delay, we specify certain critical paths in the graph representation, each critical path corresponding to a signal propagation path through the gates. The dilation of a critical path estimates propagation delay caused by a signal's traveling between gates. Since we assume that the structures of the gates are known, we have a good idea of the propagation delay through each gate. Thus, we may place a limit on a critical path's dilation; if the dilation is less than or equal to the limit, the corresponding signal path satisfies the propagation delay limit.

While the problem of Linear Arrangement with Critical Paths was already known to be NP-complete in the general case, we proved that it is still NP-complete even if the input graph is restricted to be a tree and the critical paths are restricted to be ''proper''. Likewise, we showed that the problem is NP-complete for ''trees'' if the input graph is a directed acyclic graph and the linear arrangement is constrained to be topological. We were able to show, however, that certain restrictions on the undirected and directed

problems are solvable in polynomial time. Specifically, the undirected version becomes polynomial if we restrict all critical paths to share a common root node and to be node-disjoint (except for the root), and the paths may not cross over the root in the linear arrangement. The directed version is polynomial if we bound the number of critical paths or the number of critical path sources (or sinks).

To the goal of delay propagation minimization, we added the goal of area minimization. Since our target arrangement in the graph realm is a linear arrangement, we translated area minimization to the problem of cutwidth minimization. We realized that the problem of cutwidth minimization is NP-complete, even without the critical path constraints, so we confined our efforts to heuristics and restrictions on the problem of Mincut Linear Arrangement with Critical Paths.

We presented a heuristic for the cutwidth minimization problem on undirected graphs, without critical paths. The heuristic, based on finding and integrating together good layouts for the biconnected components, was shown to have two NP-complete steps. We then showed that the directed version of the cutwidth minimization problem on trees, without critical paths is solved by a simple algorithm. For the problem where the input graph is restricted to be a tree and the critical paths are node-disjoint and consist of only a single edge, we found an algorithm that finds a solution within one of optimal. We also studied a restricted case of the directed version of the cutwidth minimization problem with critical paths. The problem restricts the input graph to be a directed ''tree'' and allows only one critical path. Although we were not able to find an algorithm to solve the problem, we were able to characterize how a solution might look. Also, we showed how to solve the problem if there were a certain bound on the critical path limit.

At this point, we assumed an ordering of the gates, and we stepped away from the realm of graph theory to ponder the goal of placing circuit elements within an array of elements in order to minimize the area needed to realize the circuit. We showed that this problem is NP-hard, even if there is only one gate. So, we concentrated on heuristic

algorithms to solve the problem.

We presented some issues concerning how to solve the problem of Cluster Placement in an Array of Gates. These include deciding how many horizontal tracks to use, how to account for signals that pass through but do not otherwise affect a gate, and how to estimate the number of vertical columns required to route the circuit. We then described the following heuristics, which we chose to study or use: Greedy, Fuzzy Partitioning, Randomized Iterative Improvement, Kernighan-Lin Iterative Improvement, and Simulated Annealing. The Greedy algorithm was used to find starting configurations and was not meant to be competitive. Finally, we presented the results of running the four competitive heuristic algorithms on six different input files. The data were not absolutely conclusive about the relative merits of the different heuristics, but they did indicate that Simulated Annealing will take too long for large problem instances; the other heuristics perform adequately, but with shorter run times.

Next, we considered the effect of limiting the size of the array and fixing the positions for some of the terminals. Specifically, what happens if there are only two gates (rows)? We proceeded to show that for certain restrictions on cluster or signal ordering and size, the problem has a polynomial-time algorithm. Next, we added the further restriction that the terminal positions on the top gate (row) are fixed. This restriction may arise from an algorithm that finds placements for an array in one channel at a time. We extended a result by Atallah and Hambrusch to show that the problem is NP-complete for the case where all clusters have size one and the bottom terminals may be placed anywhere within the length of the channel. Then, we described an heuristic algorithm that provides a lower bound on the channel density and divides the problem into smaller subproblems. For each such subproblem, each net may have at most one top terminal and one bottom terminal. We presented an algorithm for the subproblem that finds a placement that achieves the optimal possible channel density. We were not able to say if the placement can also achieve the optimal possible channel width.

The final problem we tackled is the question of how flexibility of a channel in the horizontal direction may be translated into a saving of space in the vertical direction. Specifically, we assumed a fixed vertical alignment of the terminals in a channel. The only change that may be made is the adding of empty columns into the channel. In this situation, density may not be changed, so we use another lower bound, related to flux, to estimate the change in channel width due to adding columns. We presented an algorithm that computes how many columns to add and where to place them in order to reduce the value of our lower bound to a given target value.

7.1. Future Work

There are a number of tasks that remain to be done. One direction is to extend our work in certain ways. For example, we considered separately the two goals of finding a good linear arrangement of the gates and placing the terminals within an ordered array of gates. It is possible to consider the two subproblems at the same time; thus, an iterative improvement algorithm might look like this:

> Start with an ordering of the gates. A *move* is defined as in Chapter 4, but in addition, let a move be able to swap the positions of a pair of gates (or maybe let the gates be ''split'' apart, with the appropriate penalty in the cost function).

Using a strategy of moving clusters and gates, we may even create a multi-dimensional array.

One possible extension of our work is to find a better measure than cutwidth to bound the area required. A problem with cutwidth on a linear arrangement is that it does not take into account that each gate has a height. A better measure places a weight on each node. The weight for a node equals the total number of signals that form a transistor in the gate corresponding to the node. The definition of cut would then be revised to take into account the node weights. The problem of mincut linear arrangement of trees

with node weights is solved by a minor modification of Yannakakis' arrangement algorithm for trees [Yannakakis]. One other possible change concerns multiple arcs emanating from the same node. Since in the Weinberger array realm, only one track is needed for a given signal, at a cut we need to count only one of the arcs emanating from a given node. The rest may be ignored.

Another possible extension of our work is to consider our problems in the context of other technologies and routing models. We have been assuming that the wiring in the channels is done in a 2-layer Manhattan style. Current technologies have multiple metal layers, as well as polysilicon and diffusion. Since a metal layer may overlap with any other layer, more than two layers can be allowed for Manhattan channel routing, although the routing is typically restricted to two metal layers. Also, work has been done on channel routing models that do not restrict wire segments on a certain layer to be always vertical or always horizontal. These models include the following: knock-knee, which allows wires on different layers to share a corner [Rivest1] [Preparata]; unit-vertical-overlap, which allows wires on different layers to overlap in the vertical direction for a single unit segment [Gao]; and unrestricted, in which wires may overlap arbitrarily [Hambrusch] [Brady].

We have considered a number of heuristics analytically and experimentally. To further study these heuristics, we would integrate them into an automatic logic compiler. We could then compare the results obtained with those for established compilation tools.

In conclusion, this dissertation has studied problems of linear arrangement and terminal placement. Not surprisingly, the more general cases of these problems were determined to be NP-complete, and we have settled for solving restricted cases. While it is interesting and useful to know the complexity of restricted cases, it is necessary to have methods to find good solutions for the general problems. An avenue of future research in this area is to develop algorithms to find near-optimal solutions for the problems. In particular, it might well be possible to find such algorithms for the problems of Mincut

Linear Arrangement and Linear Arrangement with Critical Paths. The mixed problem of Mincut Linear Arrangement with Critical Paths might be a bit hard for such a method; and the problem of terminal placement in an array of gates, with or without clusters, seems much too intricate for a near-optimal algorithm. For the latter problem, it would seem best to try to find a heuristic that performs as well as Simulated Annealing, but doesn't take so much time.

References

[Atallah1] Atallah, Mikhail J., and Susanne E. Hambrusch, "On Bipartite Matchings of Minimum Density," *Journal of Algorithms,* vol. 8, 1987, pp. 480-502.

[Atallah2] Atallah, Mikhail J., and Susanne E. Hambrusch, "Optimal Rotation Problems in Channel Routing," Dept. of Comp. Science, Purdue Univ., Tech. report CSD-TR-467, January 1984.
Baker, Brenda S., Sandeep N. Bhatt, and Frank Thomson Leighton, "An Approximation Algorithm for Manhattan Routing," *SIGACT,* ACM, 1983, pp. 477-486.

[Brady] Brady, M., and D. J. Brown, "Optimal Multilayer Channel Routing with Overlap," *4th M.I.T. Conf. on Advanced Research in VLSI,* M.I.T. Press, 1986, pp. 281-298.

[Brayton] Brayton, Robert K., Gary D. Hachtel, Curtis T. McMullen, and Alberto L. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis,* Kluwer Academic Publishers, Boston, 1984.

[Brown] Brown, Donna J., and Ronald L. Rivest, "New Lower Bounds for Channel Width," *Proc. CMU Conference on VLSI Systems and Computations,* 1981.

[Bryant] Bryant, R. E., "MOSSIM: A Switch-Level Simulator for MOS LSI," *Proc. of the 18th Design Automation Conference,* July 1981, pp. 786-790.

[Bui] Bui, Thang Nguyen, and Sing-Ling Lee, "On the Mincut Bipartite Arrangement Problem," *Proc. of the International Conf. on Computer-Aided Design,* IEEE, 1987, pp. 466-469.

[Burstein] Burstein, Michael, "Hierarchical Channel Router," *Proc. 20th Design Automation Conference,* IEEE, 1983, pp. 591-597.

[Chinn] Chinn, P. Z., J. Chvatalova, A. K. Dewdney, and N. E. Gibbs, "The Bandwidth Problem for Graphs and Matrices — a Survey," *Journal of Graph Theory,* **6** , 1982, pp. 223-254.

[Gao] Gao, S., and S. Hambrusch, "Two-Layer Channel Routing with Vertical Unit-Length Overlap," *Algorithmica,* vol. 1, no. 2, Springer-Verlag, 1986, pp. 223-232.

[Garey1] Garey, M. R., R. L. Graham, D. S. Johnson, and D. E. Knuth, ''Complexity results for bandwidth minimization,'' *SIAM Journal Appl. Math.*, **34** , 1978, pp. 477-495.

[Garey2] Garey, Michael R., and David S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness,* W. H. Freeman and Company, San Francisco, 1979.

[Gavril] Gavril, Fanica, ''Some NP-Complete Problems on Graphs,'' *Conference on Information Sciences and Systems,* The Johns Hopkins University, 1977.

[Golumbic] Golumbic, M. C., ''The complexity of comparability graph recognition and coloring,'' *Computing,* **18** , 1977, pp. 199-208.

[Gopal] Gopal, Inder S., Don Coppersmith, and C. K. Wong, ''Optimal Wiring of Movable Terminals,'' *IEEE Trans. on Comp.,* **C-32** , no. 9, IEEE, Sept. 1983, pp. 845-858.

[Greene] Greene, Jonathan W., and Kenneth J. Supowit, ''Simulated Annealing without Rejected Moves,'' *IEEE Transactions on Computer-Aided Design,* **CAD-5** , No. 1, IEEE, January 1986, pp. 221-228.

[Gurari] Gurari, E. M., and I. H. Sudborough, ''Improved Dynamic Programming Algorithms for the Bandwidth Minimization Problem and the Min Cut Linear Arrangement Problem,'' Technical Report, Dept. of Electrical Engineering and Computer Science, Northwestern University, Evanston, IL, 1982.

[Hambrusch] Hambrusch, S., ''Using Overlap and Minimizing Contact Points in Channel Routing,'' *Proc. 21st Annual Allerton Conf. on Comm., Control, and Comp.,* 1983, pp. 256-257.

[Hashimoto] Hashimoto, A., and J. Stevens, ''Wire Routing by Optimizing Channel Assignment within Large Apertures,'' *Proc. 8th Design Automation Workshop,* IEEE, 1971, pp. 214-224.

[Hollis] Hollis, Ernest E., *Design of VLSI Gate Array ICs,* Prentice-Hall, Inc., 1987.

[JohnsonD] Johnson, David S., Cecilia R. Aragon, Lyle A. McGeoch, and Catherine Schevon, ''Optimization by Simulated Annealing: an Experimental Evaluation (Part I),'' *unpublished manuscript.*

[JohnsonS] Johnson, S. C., ''Code Generation for Silicon,'' *Proc. Tenth ACM Symposium on Principles of Programming Languages,''* ACM, 1983, pp. 14-19.

[Kernighan] Kernighan, B. W., and S. Lin, "An Efficient Heuristic Procedure for Partitioning Graphs," *The Bell System Technical Journal,* February 1970, pp. 291-307.

[Kirkpatrick] Kirkpatrick, S., C. D. Gelatt, Jr., and M. P. Vecchi, "Optimization by Simulated Annealing," *Science,* **220** , May 13, 1983, pp. 671-680.

[Kobayashi] Kobayashi, Hideaki, and Charles E. Drozd, "Efficient Algorithms for Routing Interchangeable Terminals," *IEEE Transactions on Computer-Aided Design,* **CAD-4** , No. 3, IEEE, July 1985, pp. 204-207.

[Lageweg] Lageweg, B. J., J. K. Lenstra, and A. H. G. Rinnooy Kan, "Minimizing maximum lateness on one machine: computational experience and some applications," *Statistica Neerlandica,* **30** , 1976, pp. 25-41.

[LaPaugh1] LaPaugh, Andrea S., "Algorithms for integrated circuit layout: An analytic approach," Ph.D. dissertation, M.I.T. Lab. Computer Science, Nov. 1980.

[LaPaugh2] LaPaugh, Andrea S., and Mihalis Yannakakis, *personal communication,* 1987.

[LaPaugh3] LaPaugh, Andrea S., and Ron Y. Pinter, "On Minimizing Channel Density by Lateral Shifting," *Proceedings of the International Conf. on Computer-Aided Design,* September, 1983.

[Lin] Lin, William W., Susan S. Yeh, and Andrea S. LaPaugh, "A Weinberger Array Generator," Princeton University technical report CS-023, January 1986.

[Mata] Mata, Jose, "A Methodology for VLSI Design and a Constraint-Based Layout Language," Ph.D. dissertation, Princeton University Dept. Computer Science, October 1984.

[Mayo] Mayo, Robert N., John K. Ousterhout, and Walter S. Scott, editors, "1983 VLSI Tools: Selected works by the original artists," Report No, UCB/CSD 83/115, University of California at Berkeley Computer Science Division, March 1983.

[Monien] Monien, B., and I. H. Sudborough, "Min Cut is NP-Complete for Edge Weighted Trees," *Proc. 1986 International. Conf. on Automata, Languages, and Programming,* **226** , Springer Verlag's Lecture Notes in Computer Science, 1986, pp. 265-274.

[Ohtsuki] Ohtsuki, T., ed., *Layout Design and Verification,* volume 4 of *Advances in CAD for VLSI* series, North-Holland, 1986.

[Ousterhout1] Ousterhout, J. K., "Crystal: A Timing Analyzer for mNOS VLSI Circuits," *Third CalTech Conference on Very Large Scale Integration,* ed. Randal Bryant,

Computer Science Press, Inc., 1983, pp. 57-69.

[Ousterhout2] Ousterhout, J. K., G. T. Hamachi, R. N. Mayo, W. S. Scott, and G. S. Taylor, "Magic: A VLSI Layout System," *Proceedings of the 21st Design Automation Conference,* 1984, pp. 152-159.

[Papadimitriou] Papadimitriou, C. H., "The NP-completeness of the Bandwidth Minimization Problem," *Computing,* **16** , 1976, pp. 263-270.

[Preparata] Preparata, F. P., and W. Lipski, "Optimal Three-Layer Channel Routing," *IEEE Trans. on Computers,"* **C-33** , 1984, pp. 427-437.

[Rivest1] Rivest, Ronald L., A. Baratz, and G. Miller, "Provably Good Channel Routing Algorithms," *1981 CMU Conf. on VLSI Systems and Computations,* Oct. 1981, pp. 158-159.

[Rivest2] Rivest, Ronald L., and Charles M. Fiduccia, "A "Greedy" Channel Router," *Proc. 19th Design Automation Conference,* ed., IEEE, 1982, pp. 418-423.

[Rowen] Rowen, Christopher, "Multi-Level Logic Array Synthesis," Stanford University Computer Systems Laboratory, Tech. Report No. 85-279, July 1985.

[Sabety] Sabety, Theodore M., David E. Shaw, and Brian Mathies, "The Semi-automatic Generation of Processing Element Control Paths for Highly Parallel Machines," *Proc. 21st Design Automation Conf.,* IEEE, 1984, pp. 441-446.

[Sauris] Suaris, Peter R., and Gershon Kedem, "Standard Cell Placement by Quadrasection," Duke Univ., technical report CS-1986-34, 1986.

[Schlag] Schlag, Martine D. F., Ellen J. Yoffa, Peter S. Hauge, and C. K. Wong, "A Method for Improving Cascode-Switch Macro Wirability," *IEEE Transactions on Computer-Aided Design,* **CAD-4** , No. 2, IEEE, April 1985, pp. 150-155.

[Sechen] Sechen, Carl, and Kai-Win Lee, "An Improved Simulated Annealing Algorithm for Row-based Placement," *Proc. of the ICCAD,* IEEE, Nov. 1987, pp. 478-481.

[Sethi] Sethi, Ravi, "Complete Register Allocation Problems," *SIAM Journal on Computing,* **4** , 1975, pp. 221-248.

[Simonson] Simonson, Shai, "Routing with Critical Paths," Univ. of Illinois at Chicago, technical report, 1987.

[Siskind] Siskind, Jeffrey, Jay R. Southard, and Kenneth W. Crouch, "Generating Custom High Performance VLSI Designs From Succinct Algorithmic Descriptions," *1982 Conference on Advanced Research in VLSI,* M.I.T., 1982, pp. 28-39.

[Southard] Southard, Jay R., Antun Domic, and Kenneth W. Crouch, "Report on the Lincoln Boolean Synthesizer," *Digest of International Conf. on Computer-Aided Design,* IEEE, Sept. 1983, pp. 192-193.

[Szymanski] Szymanski, Thomas G., "Dogleg Channel Routing is NP-Complete," *IEEE Trans. on Computer-Aided Design,* **CAD-4** , no. 1, January 1985, pp. 31-41.

[Terai] Terai, Masayuki, "A Method of Improving the Terminal Assignment in the Channel Routing for Gate Arrays," *IEEE Trans. on Computer-Aided Design,* **CAD-4** , no. 3, July 1985, pp. 329-336.

[Ullman] Ullman, Jeffrey D., *Computational Aspects of VLSI,* Computer Science Press, Inc., 1984.

[Weinberger] Weinberger, Arnold, "Large scale integration of MOS complex logic: a layout method," *IEEE Journal of Solid-State Circuits,* **SC-2** , no. 4, Dec. 1967, pp. 182-190.

[Widmayer] Widmayer, P., and C. K. Wong, "An Optimal Algorithm for the Maximum Alignment of Terminals," *Information Processing Letters,* **20** , North-Holland, 1985, pp. 75-82.

[Yannakakis] Yannakakis, Mihalis, "A Polynomial Algorithm for the Min Cut Linear Arrangement of Trees," *Proc. 24th Annual Symposium on Foundations of Comp. Science,* ed., IEEE, 1983, pp. 274-281.

[Yoshimura] Yoshimura, T., and E. S. Kuh, "Efficient Algorithms for Channel Routing," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems,* **CAD-1** , no. 1, Jan. 1982, pp. 25-35.