MIT/LCS/TR-372

COMPACTION WITH AUTOMATIC JOG INTRODUCTION

F. Miller Maley

September 1986

Compaction with Automatic Jog Introduction

by

F. Miller Maley

Submitted to the Department of
Electrical Engineering and Computer Science
on May 9th, 1986, in partial fulfillment of the requirements
for the degree of Master of Science

Abstract

This thesis presents an algorithm for one-dimensional compaction of VLSI layouts. It differs from older methods in treating wires not as objects to be moved, but as constraints on the positions of other circuit components. These constraints are determined for each wiring layer using the theory of planar routing. Assuming that the wiring layers can be treated independently, the algorithm minimizes the width of a layout, automatically inserting as many jogs in wires as necessary. It runs in time $O(n^4)$ on input of size n. Several heuristics are suggested for improving the algorithm's practical performance.

The compaction algorithm takes as input a data structure called a *sketch*, which explicitly distingushes between flexible components (wires) and rigid components (modules). The algorithm first finds constraints on the positions of modules that ensure enough space is left for wires. Next, it solves the system of constraints by a standard graph-theoretic technique, obtaining a placement for the modules. It then relies on a single-layer router to restore the wires to each circuit layer. An efficient single-layer router is already known; it is able to minimize the length of every wire, though not the number of jogs.

As given, the compaction algorithm applies only to a VLSI model that requires wires to run a rectilinear grid. This restriction is needed only because the theory of planar routing (and single-layer routers) has not yet been extended to other models. The compaction algorithm's correctness proof elucidates the assumptions on which the algorithm depends, so that the algorithm is easily generalized once the necessary theoretical machinery is in place.

Key words: compaction, constraint solving, jog insertion, VLSI layout, wire routing.

Thesis Supervisor: Charles E. Leiserson
Title: Associate Professor of Computer Science and Engineering

This research was supported in part by a Graduate Fellowship from the Office of Naval Research, and in part by the Defense Advanced Research Projects Agency under contract N00014-80-C-062.

1. Introduction

An automated compaction procedure is an effective tool for cutting the production costs of a VLSI circuit at low cost to the designer because the yield of fabricated chips is strongly dependent on the total circuit area. An effective compaction system also reduces design time by freeing the designer from continual concern over design rules. If excess layout space can be removed automatically, the designer can sketch a layout without making continual efforts to conserve area. For these reasons, compaction algorithms have gained widespread attention in the VLSI literature [4, 5, 9, 11], and have been incorporated into many recent computer-aided circuit design systems [2, 4, 10, 18].

Most compaction algorithms, including the one described here, compress a layout in one dimension only. To reduce both dimensions, the layout is alternately compacted in x and y until no further improvement can be found. Compaction in two dimensions simultaneously is theoretically difficult (in fact, NP-complete), although it may work well in practice [5]. In this thesis, I assume for convenience that the direction of compaction is horizontal.

1.1. *Constraint-based compaction*

Many one-dimensional compaction systems [4,10] use a *constraint-based* technique. The program begins by assigning to each layout component i a variable x_i that represents the x-coordinate of the component's leftmost point. The *design rules* of the fabrication process are then used to derive constraints on the positions of the components. For example, if device i lies to the left of device j, and such devices must remain at least 2 units apart in order to function reliably, the compactor generates a constraint $x_j - x_i \geq 2 + w_i$, where w_i is the width of component i. (We make the usual assumption that components are not allowed to jump over one another.)

The design rules lead naturally to a set of constraints with nice properties. First of all, the constraints are not especially difficult to compute [9]. Second, they are sufficient to guarantee that the compacted layout is legal. Third, they are necessary if components cannot jump over one another. Fourth, the constraints are *simple linear inequalities*: they all can be represented in the form

$$x_j - x_i \geq a_{ij},$$

where x_i and x_j are two of the variables assigned to layout components, and a_{ij} is a constant.

Because of the simple form of the inequalities, they can be solved efficiently by graph-theoretic techniques. One constructs an edge-weighted graph in which the ith vertex represents the variable x_i, and in which an edge of weight a_{ij} from node x_i to node x_j represents the constraint $x_j - x_i \geq a_{ij}$. An assignment to the variables x_i that satisfies all the constraints is then determined by a longest-path computation on the graph. The resulting values specify the optimal positions of the components in the compacted layout. A good introduction to constraint-based compaction may be found in [5]; common algorithms for computing longest paths are discussed in [8]. (Most of the literature discusses the computation of shortest paths, but finding longest paths is equivalent to finding shortest paths when positive edge weights are replaced by negative, and vice versa.)

In the course of my research I stumbled upon an improvement to constraint-based compaction that deserves to be more widely known. If the initial layout satisfies the design rules, then Dijkstra's algorithm can be used to compute longest paths in the constraint graph. The trick is to write all the constraints in terms of *displacements* of components from their original positions, rather than absolute coordinates. If d_i and d_j represent the horizontal displacements of modules i and j from their original positions, and $d_j - d_i \geq a_{ij}$ is a constraint, then the legality of the initial layout means that the inequality $d_j - d_i \geq a_{ij}$ holds when $d_j = d_i = 0$. In other words, the constant a_{ij} is nonpositive. Thus all the edges in the constraint graph have nonpositive weight, which is precisely the precondition of Dijkstra's algorithm. (Usually Dijkstra's algorithm is used to find shortest paths, in which case the edge weights must be nonnegative, rather than nonpositive.) The improvement in worst-case performance is dramatic. To quantify it, I denote the size of a data structure D by $|D|$. If the constraint graph is (V, E), then Dijkstra's algorithm runs in time $O(|E| + |V| \log |V|)$ using Fibonacci heaps [3]. In contrast, the longest-path algorithm of Bellman and Ford, which handles edge weights of both signs, can require $\Omega(|V||E|)$ time.

1.2. *Automatic jog introduction*

In order to perform any sort of compaction, the components of the layout must be differentiated into *modules*, which are fixed in size and shape, and *wires*, which are flexible. Common procedures for generating design rule constraints [4,5,9] assume that wires are simply rectangular regions of variable height or width, and otherwise identical to modules. A vertical wire, for example, would be assigned an x-coordinate during horizontal compaction, and could only be moved rigidly from side to side. But one would often like a previously straight wire to bend around an obstacle during compaction, if the area of the circuit could thereby be reduced.

This problem is not easily overcome. Many systems [4,18] attempt to solve it by allowing the designer to specify *jog points* at which wires may bend. In effect, the wires are broken into subwires at the jog points. Compaction then becomes an interactive procedure in which the designer repeatedly examines the compacted layout, adds more potential jog points, and retries the compaction operation. Other systems [4] attempt to insert jogs automatically, using ad hoc techniques which are not guaranteed to be effective. One technique that will work is to insert a jog point wherever a wire could possibly bend. If the wires are restricted to run in a grid, the number of such jog points can be made polynomial in the size of the input layout, since no wire need bend at a point far from a layout component. This technique, however, consumes large amounts of time and memory, and it does not generalize well to situations in which the grid is absent.

The polynomial-time algorithm presented in this paper has the capability to introduce every jog point that helps to reduce the layout width. It can thus be expected to produce high quality output with little designer intervention. Automatic jog introduction is achieved by treating wires not as solid objects, but only as indicators of the topology of the layout. Constraints between modules no longer express design rules directly; instead, they ensure that there exist paths for the wires, having the given topology, that satisfy the design rules. The new constraints, called *routability* conditions, can be formulated as simple linear inequalities, and solved as usual. When the optimal module placements have been established, the new wire paths are determined by a single-layer router, such as that presented in [6]. That particular router has the advantage of generating no "empty U's," and therefore minimizes wire lengths in the given layout topology.

We need consider only planar compaction problems, as long as wires on different layers can routed independently. Illegal layouts could be generated if there were design rule constraints between wires on different layers; fortunately, there are no problematic constraints in the most common VLSI technologies. In a standard nMOS process with one layer of metal, for example, the polysilicon and diffusion layers can be considered as one layer, or *plane* [14], for routing purposes, and metal the other plane. If transistors are considered to be modules, then the wiring in each plane contains no crossovers. Furthermore, wires on the two planes interact only at contact cuts, which are also represented as modules. Thus one can reduce the problem of layout compaction to a pair of single-layer compaction problems, and compute the constraint systems on each of the two planes. Since some modules extend into both planes, the resulting constraint systems are merged by choosing the most restrictive constraint between every pair of modules. The merged system is then solved normally to place the modules.

The approach to compaction presented here depends on the ability to generate complete routability conditions for a planar layout. Until recently, such conditions were known only for certain channel routing problems [7,16]. The present work is made possible by the theory of planar routing developed in [1] and [6]. At the time of writing (April 1986), this theory considers only a VLSI model that restricts wires to a rectilinear grid. For this reason, I present the compaction algorithm in a grid-based VLSI model. My current research aims to generalize the theory of planar routing to a much larger class of models, including "octagonal" grid-based models as in [17], and models allowing wire segments of arbitrary slope. Once this work is complete, the compaction algorithm will generalize naturally to those models. In fact, my compaction algorithm is an implementation of a more abstract and general compaction technique that works in any model with the properties listed in Section 5.

1.3. *Organization of the paper*

The remainder of this paper is organized as follows. Section 2 states the definitions and theoretical results that underlie the new compaction method. Section 4 details the top level of the compaction algorithm, using a subroutine described and justified in Section 3. The next two sections (5 and 6) prove the correctness of an abstract compaction technique, which is shown in Section 7 to contain my compaction algorithm as a special case. I conclude in Section 8 with some improvements of my compaction algorithm, and a discussion of its practical value.

2. Sketches and planar routability

The principal data structure used by the compaction algorithm is called a *sketch*. A sketch represents one plane of a VLSI circuit, including both rigid objects and flexible interconnecting wires. The algorithms in this paper process only one sketch at a time, without loss of generality. This section defines precisely what I mean by a sketch, and states the theorem from [6] that determines routability conditions for a sketch.

2.1. *Definition of a sketch*

A sketch is an ordered pair (F, W) consisting of a finite set F of *features*, which are points and straight line segments, and a finite set W of *wires*, which are simple paths in the plane. Figure 1 shows an example of a sketch. Modules are represented as collections of features, because for technical reasons, terminals must be separated from other features. The features and wires of a sketch must satisfy the following conditions:

(1) Distinct *components* of the sketch (features and wires) may intersect only at their endpoints.

(2) Distinct wires may not intersect, and no wire may cross itself.

(3) Each wire touches exactly two features, which are single points lying at the endpoints of the wire. They are called the *terminals* of the wire.

(4) Four of the features of the sketch form a *bounding box* around the other components.

When referring to "points in the sketch," we will mean points lying on features in the sketch. Connected groups of features are called the *obstacles* of the sketch. The definitions imply that each terminal is its own obstacle. Features represent the rigid parts of the layout, the modules; wires represent the flexible interconnections. Clearly, a sketch whose wires are well behaved (e.g., consist of line segments) can easily be encoded in a data structure.

2.2. *Legality and routability*

I now define what it means to route a sketch, and what it means for a sketch to represent a legal layout. For more precise definitions, see [6]. A *link* in a sketch

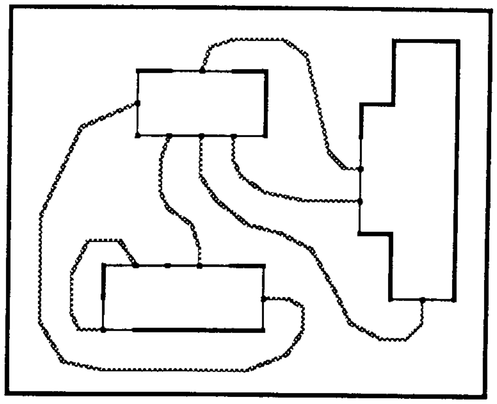

Figure 1. A typical sketch. Dark points and line segments are features, grey lines are wires, and light lines are conceptual module boundaries.

$S = (F,W)$ is a path in the plane that begins and ends on features in F, and intersects no features in between. For example, the wires in W are links in S. Two links in S are ***homotopic*** if they have the same endpoints, and one can be continuously deformed into the other without moving its endpoints or allowing its interior to touch a feature in F. A ***routing*** of S is a sketch (F, W') whose features are the same, and whose wires can be obtained by replacing each wire in W with a homotopic wire. A sketch is said to be ***legal*** if it represents a legal VLSI layout, which for our purposes means that the following conditions hold:

(1) All obstacles and wires lie in the rectilinear grid of unit spacing.

(2) The wires form vertex-disjoint paths in the grid.

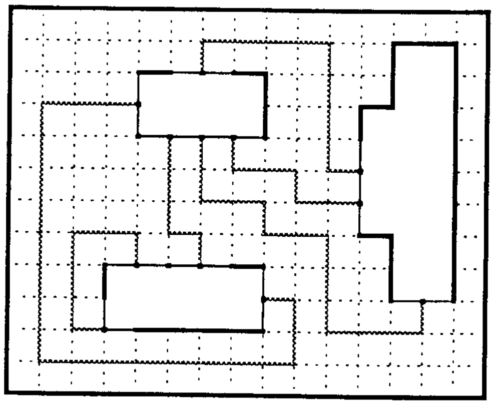

Figure 2. A legal routing of the sketch in Figure 1, using wires of minimum length. Dotted lines represent the routing grid.

A sketch is ***routable*** if it has a legal routing. Using the algorithm in [6], a legal routing of a routable sketch can be found in polynomial time. Figures 1 and 2 illustrate the concepts of legality and routability. The sketch in Figure 1 is illegal because it contains curved wires. Nevertheless, it is routable, and one of its legal routings is shown in Figure 2.

2.3. *Routability conditions*

My compaction algorithm is based on a theorem from [6] that characterizes the routable sketches in terms of the following concepts. If $p = \langle x_p, y_p \rangle$ and $q = \langle x_q, y_q \rangle$ are points in the sketch S, then $\overline{pq}$ denotes the open-ended line segment from p to q. Such a segment is called a ***cut*** if it intersects no features in S. The ***capacity*** of a cut $\overline{pq}$ is the maximum number of wires that can legally cross $\overline{pq}$; in symbols,

$$cap(\overline{pq}) = \max\{|x_q - x_p|, |y_q - y_p|, 1\} - 1 \ .$$

The ***flow*** across $\overline{pq}$, denoted $flow(\overline{pq})$, is the number of crossings of $\overline{pq}$ that are enforced by the topology of the sketch. (See Figure 3.) Crossings of $\overline{pq}$ that can be removed by deforming the wires W do not contribute to the flow. More formally, $flow(\overline{pq})$ is the minimum, over all routings (F, W') of (F, W), of the number of times $\overline{pq}$ is crossed by wires in W'.

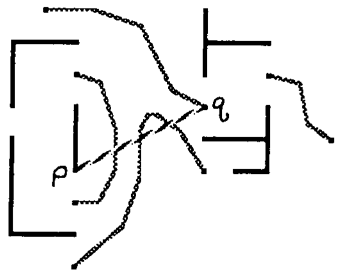

Figure 3. A portion of a sketch with a cut $\overline{pq}$. The flow across $\overline{pq}$ is 1.

The routability of a sketch is completely determined by the flows and capacities of its cuts. Let us say that a cut is ***safe*** [1] if its flow does not exceed its capacity. Then we have the following result.

Lemma 1. [6] A sketch that contains an unsafe cut is unroutable.

More significantly, the converse is true (except when the features of the sketch are illegally placed): a sketch that contains no unsafe cuts is routable. In fact, this statement may be strengthened. A ***critical*** cut $\overline{pq}$ is one such that p is the endpoint of a feature, and q is the closest point on its feature to p. The critical cuts are the only important ones.

Theorem 2. [6] The sketch (F, W) is routable if and only if $(F, \emptyset)$ is legal and every critical cut in (F, W) is safe.

The inequalities $flow(\overline{pq}) \leq cap(\overline{pq})$ for the cuts $\overline{pq}$ of a sketch are called ***routability conditions*** for the sketch. Constraints of this sort will be used by the compaction algorithm to determine the optimal positions for layout features.

3. Computing flows in the sketch

This section describes a procedure used to facilitate the computation of routability conditions for a sketch. As suggested by Theorem 2, the important attributes of a sketch are the flows and capacities of cuts. Capacities are purely geometric quantities, and can be computed from endpoint locations in constant time. In addition, they vary in a regular way with the movement of features during compaction. Flows, on the other hand, are topological quantities, and are relatively difficult to compute. Moreover, they depend in complex ways on the positions of features. Thus to compute flows, we require a data structure that captures the topology of the sketch and that is invariant under compaction. I begin by presenting such a structure.

3.1. *The adjacency graph*

The data structure we use is called the *adjacency graph* of the sketch. Its construction is straightforward, and is illustrated by Figure 4. From a point on the rightmost edge of each obstacle, except the bounding box, a line is drawn rightward until it hits another obstacle. These line segments and rays will be called *hurdles*. Now each wire is replaced by a homotopic wire that intersects as few hurdles as possible, making sure that no two wires cross. The resulting set of objects forms a planar graph: its nodes are obstacles and hurdle/wire crossings, and its edges are pieces of wires and hurdles. The planar dual of this graph, which is actually a multigraph, is the adjacency graph of the sketch. A node of the adjacency graph corresponds to a face of the original graph, and is said to *border on* the points forming the boundary of that face. The adjacency graph does not change during horizontal compaction because hurdles can only slide back and forth, and we will not allow wires or features to cross over one another.

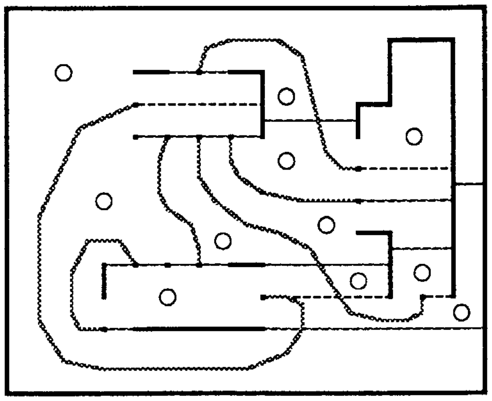

Figure 4. The adjacency graph of the sketch in Figure 1. Dashed lines are hurdles, and circles are nodes of the adjacency graph. Wherever two such nodes are adjacent across a wire or hurdle, there is a "wire edge" or "hurdle edge", respectively, in the adjacency graph. These edges are omitted for clarity. Adjacency across features is not represented in the adjacency graph.

The purpose of the hurdles is to relate links in the sketch to the sketch topology. Consider the sequence of hurdles crossed by a link, in order, together with the directions of crossing. Such a *hurdle sequence* can be put into a canonical form by removing all unnecessary crossings, that is, all places where the link crosses a hurdle and immediately crosses back in the other direction. One can show that two links with the same endpoints have the same canonical hurdle sequence if and only if they are homotopic.

Constructing the adjacency graph of a sketch is not difficult. Let (F, W) be the sketch. One first computes the canonical hurdle sequence of each wire, and forms the graph whose nodes are the sketch obstacles and the necessary hurdle/wire crossings. Since every wire segment could cross every hurdle, this process might require $\Theta(|F||W|)$ time and space, but will probably need much less. (Recall that $|D|$ denotes the size of the data structure D. Thus $|W|$ is not the number of wires, but rather the number of line segments that compose them.) Since the direction of each crossing is known, one has enough information about the embedding of this graph to construct its dual graph. Building the dual graph is straightforward; one simply walks around the faces of the original graph, creating dual nodes and edges as necessary. The time and space taken by this construction are both proportional to the size of the dual graph.

3.2. *Computation of flows*

An appropriate search through the adjacency graph can compute the flow across a cut. To see how, notice that there are two kinds of edges in the adjacency graph: "wire edges," which represent adjacency across a wire, and "hurdle edges," which represent adjacency across a hurdle. A path through the adjacency graph thus has a hurdle sequence determined by the hurdle edges it contains. The following lemma demonstrates the correspondence between the canonical hurdle sequence of a cut and hurdle sequences of paths. By the *length* of a path in the adjacency graph we mean the number of wire edges the path contains.

Lemma 3. Suppose that a cut $\overline{pq}$ of sketch S has hurdle sequence H. Let $\mathit{Paths}(\overline{pq})$ be the set of paths in the adjacency graph of S that begin at a node bordering on p, end at a node bordering on q, and have hurdle sequence H. Then $\mathit{flow}(\overline{pq})$ is equal to the length of the shortest path in $\mathit{Paths}(\overline{pq})$. □

The proof of this lemma requires too much background material to be presented here.

To make use of Lemma 3, we must be able to find shortest paths with given hurdle sequences in the adjacency graph. This involves searching through a subgraph of the adjacency graph.

Definition. The *skeleton* of an adjacency graph G is the graph T consisting of the nodes and wire edges of G.

Lemma 4. The skeleton T of an adjacency graph G is a tree, and it has the following properties.

(1) If h is a hurdle, the set of nodes bordering h from below (or above) is connected in T.

(2) If b and b' are two nodes bordering a hurdle h from below, and a and a' are adjacent to b and b', respectively, across h, then the distance between a and a' in T is equal to the distance between b and b' in T. □

These properties help in proving the correctness the following algorithm, which computes the flow across a cut $\overline{pq}$. We may assume that each hurdle in the hurdle sequence of $\overline{pq}$ is to be crossed from bottom to top.

Algorithm F. (Computes the flow across a cut.)
Input: a cut $\overline{pq}$ with hurdle sequence $\langle h_1, \ldots, h_n \rangle$, the adjacency graph G with skeleton T.
Output: the flow f across $\overline{pq}$.
Local variables: integers i and t, nodes u and v.

```
1. $f \leftarrow \min\{\text{DIST-FROM}(w) : w \text{ borders on } p\}$;
2.    function DIST-FROM($w$);
3.    $t \leftarrow 0$; $u \leftarrow w$;
4.    for $i \leftarrow 1$ to $n$ do
        begin
5.        $v \leftarrow$ a node bordering $h_i$ from below that is closest to $u$ in $T$;
6.        $t \leftarrow t +$ the distance from $u$ to $v$ in $T$;
7.        $u \leftarrow$ the node adjacent to $v$ across hurdle $h_i$;
        end;
8.    $v \leftarrow$ a node bordering $q$ from below that is closest to $u$ in $T$;
9.    return $t +$ the distance from $u$ to $v$ in $T$.
```

In other words: for each node bordering on p, find the shortest path to the first hurdle, cross the first hurdle, find the shortest path from there to the second hurdle, and so on. Breadth-first search can be used to implement lines 5–6 and 8–9. This approach may work well in practice, but its worst-case behavior is poor; it could require $\Omega(n|T|)$ time on a hurdle sequence of length n. Later in this section, I show how to implement Algorithm F more efficiently.

3.3. *Correctness of Algorithm F*

The correctness of Algorithm F follows from Lemmas 3 and 4.

Lemma 5. Let $\overline{pq}$ be a cut in a sketch S, let G be the adjacency graph of S, and let $\langle h_1, \ldots, h_n \rangle$ be the hurdle sequence of $\overline{pq}$. Then Algorithm F, when applied to $\overline{pq}$ and G, outputs $\mathit{flow}(\overline{pq})$.

Proof. The function DIST-FROM computes the length of some path with hurdle sequence $\langle h_1, \ldots, h_n \rangle$ from w to a node bordering q. If w borders on p, then all such paths have length $\mathit{flow}(\overline{pq})$ or greater by Lemma 3. So it suffices to show that DIST-FROM(w) $\leq \mathit{flow}(\overline{pq})$ for some w bordering on p. By Lemma 3, there exists a path π in G that begins at a node bordering p, ends at a node bordering q, and has hurdle sequence $\langle h_1, \ldots, h_n \rangle$ and length $\mathit{flow}(\overline{pq})$. Let π_0 be the portion of this path up to hurdle h_1; for $0 < i < n$, let π_i be the portion of π between hurdles h_i and h_{i+1}; let π_n be the portion of π beyond hurdle h_n. Choose w to be the first node along π.

We consider only the execution of DIST-FROM on w. Define t_i and u_i to be the values of t and u just after iteration i of the loop in lines 2–5; put $t_0 = 0$ and $u_0 = w$. Denote the length of a path α in G by $\ell(\alpha)$. Algorithm F maintains the following invariant:

There is a path α_i in G' from u_i to the origin of π_i such that $\ell(\rho_i) + t_i \leq \sum_{j=0}^{i-1} \ell(\pi_j)$.

In particular, after the loop completes, we have

$$\ell(\alpha_n) + t_n \leq \sum_{j=0}^{n-1} \ell(\pi_j).$$

Let T be the skeleton of G. The concatenation of α_n and π_n, written $\alpha_n \cdot \pi_n$, is a path in T from u_n to a node bordering q. Hence by Lemma 3,

$$\begin{aligned} \text{DIST-FROM}(w) &\leq t_n + \ell(\alpha_n \cdot \pi_n) \\ &= (\ell(\alpha_n) + t_n) + \ell(\pi_n) \\ &\leq (\sum_{j=0}^{n-1} \ell(\pi_j)) + \ell(\pi_n) \\ &= \ell(\pi) = \mathit{flow}(\overline{pq}). \end{aligned}$$

Thus the invariant implies the lemma.

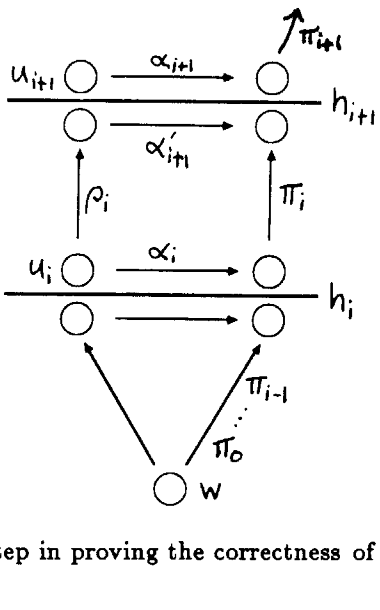

Figure 5. The inductive step in proving the correctness of Algorithm F.

It remains to prove the invariant, which we do by induction on i. The basis case $i = 0$ is trivial. Now assuming the invariant for i, we prove it for $i + 1$. See Figure 5. The path ρ_i represents a shortest path in T from u_i to a node bordering h_{i+1} from below, and the paths α_{i+1} and α'_{i+1} are the shortest paths in T between the indicated nodes. Since T is a tree, the shortest path between two nodes in T is the unique simple path between them.

In particular, α'_{i+1} is the unique simple path between its endpoints. Now by part (1) of Lemma 4, the nodes adjacent to h_{i+1} from below are connected in T, and hence every node along α'_{i+1} is adjacent to h_{i+1} from below. It follows that $\rho_i \cdot \alpha_{i+1}$ is a simple path—the shortest path between u_i and the end of π_i—and thus we have

$$\ell(\rho_i) + \ell(\alpha'_{i+1}) \leq \ell(\alpha_i) + \ell(\pi_i).$$

Combining this inequality with the induction hypothesis, we obtain

$$(t_i + \ell(\rho_i)) + \ell(\alpha'_{i+1}) \leq \sum_{j=0}^{i} \ell(\pi_j).$$

The first term on the right is just t_{i+1}, and part (2) of Lemma 4 shows that $\ell(\alpha'_{i+1}) = \ell(\alpha_{i+1})$. We conclude that the invariant holds for $i + 1$. □

3.4. *Data structures for Algorithm F*

The most time-consuming steps of Algorithm F involve searching through the skeleton T of the adjacency graph. One can speed up these searches by taking advantage of the fact that T is a tree. The first task is to preprocess T so that one can quickly determine the distance between any pair of its nodes, and hence speed up lines 6 and 9 in Algorithm F. The second task is to preprocess T so that one can compute efficiently the closest node in a connected subset of T to a given node. This ability is sufficient to implement lines 5 and 8 of Algorithm F, because the set of nodes bordering a feature or bordering a hurdle from below is connected in T.

This section shows how the adjacency graph G may be preprocessed so that Algorithm F takes $O(\log^2 |G|)$ time per iteration. The preprocessing requires $O(|T| \log^2 |G|)$ time and creates data structures that occupy $O(|T| \log |T|)$ space. One could speed up Algorithm F even further by precomputing the distance between every pair of nodes in T, but only at the cost of $\Omega(|T|^2)$ space.

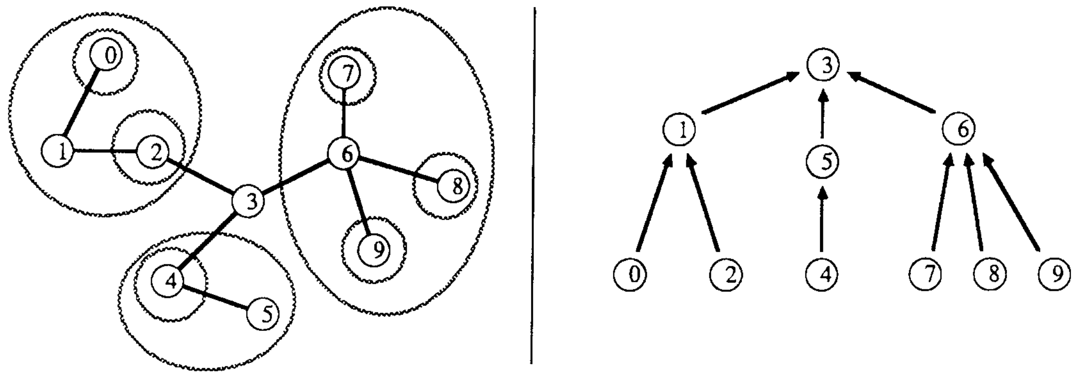

Figure 6. A tree, drawn with solid lines, and its decomposition tree.

The idea is to decompose the graph T recursively, forming a ***decomposition tree*** D on the nodes of T. Figure 6 shows the construction. Let n be the number of nodes in the graph T. The separator theorem for trees [12] implies that T contains a vertex r whose removal disconnects T into subtrees containing at most $\frac{2}{3}n$ nodes each. Moreover, we can find the vertex r in linear time using depth-first search to compute the sizes of subtrees. Now we decompose the subtrees of r recursively, obtaining a decomposition tree for each one. The roots of these trees become the children of r in the decomposition tree D for T. At each stage of the decomposition, the sizes of subtrees are reduced by a constant factor, so D has height $\Theta(\log |T|)$. The recursive construction of D takes $\Theta(|T| \log |T|)$ time because it examines each node and edge in T just $O(\log |T|)$ times. We store with each node of T its distance in T from each of its ancestors in D. These distances can be computed in $\Theta(|T| \log |T|)$ time during the construction of D, and their storage requires $\Theta(|T| \log |T|)$ space.

This information allows one to compute quickly the distance between two nodes of T. One simply finds their lowest common ancestor (LCA) in D, and then adds their distances (in T) from that ancestor. This procedure takes at most $O(\log |T|)$ time; it works because the LCA in D of two nodes in T either equals one of them or separates them in T.

Some extra preprocessing is needed before we can compute closest members of connected sets of T. Let D be the same decomposition tree as above. The LCA in D of a connected set $C \subseteq T$ is a member of C, and we can compute in advance the LCA's of the connected sets that we care about, which are the following:

- For each hurdle, the nodes in G adjacent to that hurdle from below.
- For each feature, the nodes of G adjacent to that feature.

There are $O(|F|)$ such sets in the sketch (F, W). The LCA of each set C can be computed in $O(\log |G| \log |T|)$ steps if some node of C is known, assuming that membership in C can be tested in $O(\log |G|)$ time.* Thus the preprocessing of the connected sets uses $O(|F| \log |G| \log |T|)$ time and $O(|F|)$ space. We also store, for each node y and for each of its ancestors x, the highest vertex in D that is interior to the to the path in T between x and y. In case x and y are adjacent in T, we store *nil* instead. To produce this information requires $O(|T| \log |T|)$ space and $O(|T| \log^2 |T|)$ time.

The following algorithm uses the data from preprocessing in lines 2 and 4.

Algorithm V. (Finds the closest vertex in a connected set $C \subseteq T$ to a node u.) Local variables: nodes v and z.

* In practice, membership in C could probably be tested in $O(1)$ time, since each node of T would probably have bit-vector to specify which hurdles and features it borders. From a theoretical point of view, this approach is no good, because it requires $O(|F||G|)$ bits of storage. If the edge list of each node of G is kept in an appropriate data structure, then it takes at most $O(\log |G|)$ time to determine whether a node is adjacent to a particular hurdle. Similarly, adjacency to features can be tested in logarithmic time. This method loses no asymptotic space efficiency.

```
1. if $u \in C$ then return $u$;
2. $v \leftarrow$ LCA($C$);
3. if $v$ is not an ancestor of $u$ then $u \leftarrow$ LCA($u, v$);
   repeat
4.     $z \leftarrow$ the highest vertex in $D$ on the path in $T$ between $u$ and $v$;
5.     if $z = nil$ then return $v$;
6.     if $z \in C$ then $v \leftarrow z$ else $u \leftarrow z$
   until false.
```

We now check the correctness of Algorithm V. First we show that before every iteration of the loop, the following invariants hold: (a) $v \in C$ but $u \notin C$; (b) one of u and v is an ancestor of the other; and (c) the closest node in C to u is the closest node in C to the original input u. Lines 1–3 serve to establish these invariants. Line 3 does not harm invariant (c), for if v is not an ancestor of u, then the LCA of u and v is on every path between u and C. Hence the closest node in C to u is also the closest node in C to LCA(u, v). Now we check that the loop maintains the invariants. Invariant (a) is maintained by line 6. That line does not affect invariant (b) either, because z, u, and v are all on the same branch of D. Invariant (c) can only be affected if $z \in C$. But in that case, every path from u to C passes through z, because C is connected. Hence the closest node in C to u is also the closest node in C to z. Finally, line 4 outputs the correct node; z being *nil* means that u and v are adjacent, which makes v is the closest node in C to u. Therefore when Algorithm V terminates, it produces the correct answer.

It remains to bound the number of iterations of the loop. We bound it by the height of D, by showing that the height of the vertex z in D decreases by at least one at each step. Let u_0, v_0, and z_0 be the values of u, v, and z at one iteration, and let u_1, v_1, and z_1 be their values at the next iteration. The path between u_1 and v_1 is a subpath of the path between u_0 and v_0, so z_1 is no higher than z_0. Suppose they were the same height. Then $LCA(z_0, z_1)$ would be a higher node separating z_1 from z_0. This node would be on the path between u_0 and v_0, contradicting the definition of z_0. Therefore z_1 is strictly lower than z_0. We conclude that Algorithm V runs in $O(\log |T|)$ iterations. Each iteration requires $O(\log |G|)$ time due to the membership test in line 6. Hence Algorithm V finishes in $O(\log |G| \log |T|)$ time.

4. The compaction algorithm

This section defines mathematically the problem of compaction with automatic jog introduction, and presents a practical algorithm that solves this problem. Because the wiring planes can be treated independently, the compaction algorithm considers only a single plane. It assumes the input is available in the form of a sketch, and that the input sketch is routable.

4.1. *Configuration space*

Let the input sketch be denoted by S. For the purpose of compaction, the obstacles of S must be grouped into ***modules***: collections of features whose relative positions are fixed. The compactor is allowed to choose a horizontal displacement for each module. Such a vector of displacements is called a ***configuration*** of S. The configuration $\mathbf{d} = \langle d_1, \ldots, d_n \rangle$ corresponds to a sketch $S(\mathbf{d})$ in which module i has been shifted right by a distance d_i (or left by a distance $-d_i$). Thus a configuration $\mathbf{d}$ determines how the features of $S(\mathbf{d})$ are to be placed; we shall consider the wires of $S(\mathbf{d})$ shortly. If the sketch S has n modules, then the set of all its configurations is the vector space $\mathbf{R}^n$, and the origin $\mathbf{0}$ of this vector space corresponds to the original sketch.

Using configurations, we can describe how points on modules move during compaction. If p is a point in S, its x and y coordinates will be denoted x_p and y_p, respectively. The module in which p lies will be written $\mu(p)$, so the horizontal position of p in the configuration $\mathbf{d}$ is $x_p + d_{\mu(p)}$. The notation $p(\mathbf{d})$ stands for p shifted by $\mathbf{d}$, that is, the point $\langle x_p + d_{\mu(p)}, y_p \rangle$. We also let $\Delta_{pq}(\mathbf{d})$ be difference in x-coordinates between $q(\mathbf{d})$ and $p(\mathbf{d})$, namely

$$\Delta_{pq}(\mathbf{d}) = (x_q + d_{\mu(q)}) - (x_p + d_{\mu(p)}) \ .$$

To disallow the possibility of modules crossing over during compaction, we restrict attention to a subset of configuration space. Suppose p and q are points in S having the same y-coordinate. If q lies to the right of p, then we only wish to consider configurations $\mathbf{d}$ in which $q(\mathbf{d})$ lies to the right of $p(\mathbf{d})$. So we let $\mathbf{C}(S) \subset \mathbf{R}^n$ be the set of configurations $\mathbf{d}$ such that for all points p and q of S with $p_y = q_y$ and $p_x < q_x$, we have $\Delta_{pq}(\mathbf{d}) > 0$. We call $\mathbf{C}(S)$ the *configuration space* of the sketch S. The configuration space of S is convex, because it is the intersection of convex sets of the form

$$\{\mathbf{d} \in \mathbf{R}^n : d_{\mu(q)} - d_{\mu(p)} > x_p - x_q\}, \qquad p, q \in S.$$

For every configuration $\mathbf{d}$ in $\mathbf{C}(S)$, the hurdles of S transform nicely to $S(\mathbf{d})$. For if a hurdle h in S has endpoints p and q, then the line segment in $S(\mathbf{d})$ between $p(\mathbf{d})$ and $q(\mathbf{d})$ is a hurdle, which we identify with h. Now we can finally make $S(\mathbf{d})$ into a sketch: $S(\mathbf{d})$ is well defined (up to wire homotopy) by requiring that its wires have the same canonical hurdle sequences they had in the original sketch $S(\mathbf{0})$.

4.2. *Problem statement*

The compaction problem is to find a configuration $\mathbf{d} \in \mathbf{C}(S)$ such that $S(\mathbf{d})$ is routable, and can be routed in minimal width. (The *width* of a sketch is the horizontal distance between the leftmost and rightmost points on its features or wires.) As we have stated it, the compaction problem is generally very difficult; in fact, it is NP-complete [11]. The reason is that the routability conditions may not define a convex region of configuration space, and hence the set of acceptable configurations $\{\mathbf{d} \in \mathbf{C}(S) : S(\mathbf{d}) \text{ is routable}\}$ can be very hard to search. For example, consider the sketch in Figure 7. The set of acceptable configurations falls into two components: those in which the upper module lies entirely to the right of the lower module, and those in which the opposite is true. Intermediate configurations correspond to unroutable sketches, and thus the set of acceptable configurations is not convex. In most optimization problems, including compaction, one only expects to search a convex subset of the acceptable configurations in order to achieve a polynomial-time algorithm. The algorithm presented here searches the largest such region that contains the initial configuration, and thus finds the best configuration available to any algorithm of its type.

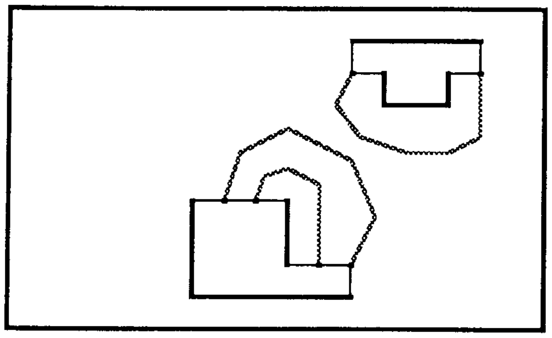

Figure 7. How wires can prevent modules from sliding past one another. If the upper module is allowed to move to the left of the lower one, the set of acceptable configurations is not convex.

4.3. *Algorithm overview*

The basic notion underlying the compaction algorithm is that of a ***potential cut.*** For the purposes of this section, a potential cut is a continuous function that defines for each configuration $\mathbf{d} \in \mathrm{C}(S)$ a line segment between two features in $S(\mathbf{d})$. The line segment may or may not be a cut, depending on the positions of the features in $S(\mathbf{d})$. The configuration $\mathbf{c}$ is said to ***protect*** a potential cut ψ if either $\psi(\mathbf{c})$ is not a cut, or $\psi(\mathbf{c})$ is a safe cut. (Recall that a cut is safe if its flow does not exceed its capacity.) The significance of these definitions lies in a reformulation of Theorem 2 in terms of potential cuts:

Let $S = (F, W)$ be a sketch, and let $\mathbf{c} \in \mathrm{C}(S)$ be a vector in its configuration space. For every endpoint p of a feature in F, and for every other feature Q in F, let $\chi_{pQ}(\mathbf{c})$ denote the line segment from $p(\mathbf{c})$ to the closest point on $Q(\mathbf{c})$. Then χ_{pQ} is a potential cut for S, which we call *critical.* The sketch $S(\mathbf{c}) = (F(\mathbf{c}), W(\mathbf{c}))$ is routable if and only if

(1) the sketch $(F(\mathbf{c}), \emptyset)$ is legal, and

(2) the configuration $\mathbf{c}$ protects every critical potential cut of S.

The compaction algorithm works by finding a subset of configuration space, determined by simple linear inequalities, whose configurations protect every critical potential cut. It thereby ensures that the configuration it chooses satisfies condition (2) above. Condition (1) can be ignored, because for all configurations

$\mathbf{c} \in \mathbf{C}(S)$, the wireless sketch $(F(\mathbf{c}), \emptyset)$ is legal unless its features fail to lie in the grid. This never happens because the initial sketch (F, W) is assumed to be routable, and the compaction algorithm never considers nonintegral displacements for modules. The subspace searched is chosen so as to include the initial configuration. An overview of the compaction technique follows.

The central problem is to find a simple linear inequality that ensures that a potential cut, say ψ, is protected. One would like to use the routability condition $\mathit{cap}(\psi(\mathbf{d})) \geq \mathit{flow}(\psi(\mathbf{d}))$ as a constraint on the configuration $\mathbf{d}$, but for most potential cuts ψ, this constraint is not a simple linear inequality. The difficulty lies not with the capacity of $\psi(\mathbf{d})$, which is determined solely by the geometry of $S(\mathbf{d})$, and depends in a simple way on the displacements d_i. Rather, the quantity $\mathit{flow}(\psi(\mathbf{d}))$ is hard to characterize, because it depends on the relation of the line segment $\psi(\mathbf{d})$ to the topology of the sketch $S(\mathbf{d})$.

The solution is to find a specific configuration $\mathbf{c}$ such that whenever the potential cut $\psi(\mathbf{d})$ is unsafe, its flow is equal to $\mathit{flow}(\psi(\mathbf{c}))$. The constraint $\mathit{cap}(\psi(\mathbf{d})) \geq \mathit{flow}(\psi(\mathbf{c}))$ is then sufficient to protect ψ. Moreover, when this constraint is written in terms of the variables d_i, it becomes a simple linear inequality, because the right hand side is constant. To find $\mathbf{c}$, the algorithm looks for a configuration that minimizes the capacity of ψ, subject to the condition that all critical cuts of smaller vertical span are safe. These shorter cuts force the other features to the side of ψ on which they must lie if ψ is ever to become unsafe. If, in this way, the algorithm finds a configuration $\mathbf{c}$ that does not protect ψ, then the routability condition for $\psi(\mathbf{c})$ is remembered. Otherwise, the potential cut ψ is ignored.

4.4. *Description of the compaction algorithm*

Since critical cuts move in nontrivial ways during compaction, it turns out to be more convenient to consider two simpler types of potential cuts:

- Horizontal cuts incident on feature endpoints.
- Cuts between pairs of feature endpoints.

The constraints generated from these potential cuts turn out to be sufficient to protect all the critical potential cuts.

The horizontal potential cuts are particularly simple because their flows are independent of the configuration. These potential cuts are treated first in order to generate the constraints that prevent features from crossing over one another. A horizontal potential cut is a function ϕ_{pq} of the form $\phi_{pq}(\mathbf{d}) = \overline{p(\mathbf{d})q(\mathbf{d})}$, where p and q are points in the original sketch. Assuming without loss of generality that $x_q > x_p$, Theorem 2 gives the routability condition

$$\Delta_{pq}(\mathbf{d}) \geq \mathit{flow}(\phi_{pq}(\mathbf{d})) + 1, \qquad \mathbf{d} \in \mathbf{C}(S).$$

Since $\overline{pq}$ being horizontal implies that $\mathit{flow}(\phi_{pq}(\mathbf{d})) = \mathit{flow}(\overline{pq})$ for all $\mathbf{d} \in \mathbf{C}(S)$, the constraint is a simple linear inequality

$$d_{\mu(q)} - d_{\mu(p)} \geq (\mathit{flow}(\overline{pq}) + 1) + (x_p - x_q) \tag{1}$$

on the displacements of p and q. The flow across $\overline{pq}$ is easily computed by Algorithm F of the previous section. For each horizontal cut $\overline{pq}$ incident on a feature endpoint, the algorithm computes a constraint of the form (1). Of course, the constraints are maintained as a constraint graph I over the variables d_i.

The second stage of the algorithm concerns the cuts that are not horizontal. Let Φ be the set of potential cuts ϕ_{pq} where p and q are feature endpoints with $y_p \neq y_q$. The *height* of an element ϕ_{pq} of Φ is the quantity $|y_p - y_q|$. This quantity is independent of configuration. Sort Φ in increasing order of height, forming a sequence in which flatter potential cuts precede taller ones. After $\phi_{pq} \in \Phi$ has been processed, the algorithm can ensure that the output configuration protects ϕ_{pq}.

The algorithm examines the elements of Φ in sorted order, and for each one that proves important, it adds an appropriate constraint to the graph I. The constraint for a potential cut $\phi_{pq} \in \Phi$, with $x_q \geq x_p$, is computed thus. First, the algorithm solves the current constraint system I together with the temporary constraint $\Delta_{pq}(\mathbf{d}) \geq 0$, fixing $d_{\mu(p)}$, and minimizing $d_{\mu(q)}$. Call the resulting configuration $\mathbf{c}$. If $\mathbf{c}$ protects ϕ_{pq}, then the constraint set is unchanged; otherwise, the constraint

$$d_{\mu(q)} - d_{\mu(p)} \geq (x_p - x_q) + \mathit{flow}(\phi_{pq}(\mathbf{c})) + 1$$

is added to I. The new constraint is a simple linear inequality derived from the routability condition $\mathit{cap}(\phi_{pq}(\mathbf{c})) \geq \mathit{flow}(\phi_{pq}(\mathbf{c})) + 1$.

After all the potential cuts in ψ have been processed, the constraint system I is complete, and the algorithm solves it using a longest-path algorithm. The resulting configuration is used to build an output sketch, which is then routed using a single-layer router such as Algorithm R in [6]. That particular router has the advantage of being able to minimize the lengths of the wires in the routing.

The compaction algorithm is summarized below. We assume that the left and right edges of the sketch's bounding box compose modules 1 and n, respectively, and that the top and bottom edges of the box are ignored.

Algorithm C. (Compacts a sketch horizontally.)
Input: a sketch $S = (F, W)$ with n modules specified.
Output: the compacted sketch.
Local variables: the points p and q, a configuration $\mathbf{c}$, and the constraint graph I over variables d_i $(1 \leq i \leq n)$.
Subroutines: Algorithm F is used to compute flows in lines 2 and 5; Dijkstra's algorithm is used in lines 4 and 7; a single-layer router is used in line 8.

```
1. Preprocess $S$ as described in Section 3;
```

```
2. Let $I$ be the set of constraints $\{\Delta_{pq}(\mathbf{d}) \geq flow(\overline{pq}) + 1\}$ where $\overline{pq}$ is a horizon-
       tal cut with $x_p < x_q$ and $p$ or $q$ is an endpoint of a feature in $F$.
3. foreach pair of feature endpoints $\{p, q\}$ with $x_p \leq x_q$ and $y_p \neq y_q$, in order of
       increasing height, do
     begin
4.     Find a configuration $\mathbf{c}$ that minimizes $c_{\mu(q)} - c_{\mu(p)}$ while obeying the con-
         straints $I \cup \{\Delta_{pq}(\mathbf{d}) \geq 0\}$;
5.     if $\overline{p(\mathbf{c})q(\mathbf{c})}$ is a cut in $S(\mathbf{c})$ then
6.       if $flow(\phi_{pq}(\mathbf{c})) > cap(\phi_{pq}(\mathbf{c}))$
           then add to $I$ the constraint $\Delta_{pq}(\mathbf{d}) \geq flow(\phi_{pq}(\mathbf{c})) + 1$
     end;
7. Find a configuration $\mathbf{c}$ satisfying $I$ that minimizes $c_n - c_1$;
8. Route the sketch $S(\mathbf{c})$ and output the result.
```

4.5. *Details of the implementation*

• The computation of flows in line 2 is performed using Algorithm F of the previous section. The cuts themselves may be found by any straightforward method, as the algorithm's run time will be dominated by other factors.

• Line 3 requires that pairs of feature endpoints be enumerated in order of vertical separation. Writing down the pairs and sorting them would waste large amounts of space; the following approach is better. First sort the feature endpoints by y-coordinate, and associate with each endpoint the next higher endpoint. Place these pairs in a priority queue, and keep the queue sorted by difference in y-coordinates. At each iteration of the loop (lines 3–6), withdraw the best element $\{p, q\}$ from the priority queue, and process the potential cut ϕ_{pq}. Then find the next endpoint q' above q in y-coordinate, if one exists, and insert the pair $\{p, q'\}$ into the priority queue. This method uses linear space, and no more time than other parts of Algorithm C.

• To solve the constraint system in line 4, it suffices to compute longest paths from the vertex $\mu(p)$. Dijkstra's algorithm can be used for the purpose, because every edge in the graph has weight zero or less. (Normally, Dijkstra's algorithm is used to find shortest paths, and then the edge weights must be nonnegative.) To see why edge weights are nonpositive, consider the case when all the displacements d_i are zero. Using the assumption that the initial configuration is legal, one can prove that it obeys all constraints. Hence if $d_j - d_i \geq a_{ij}$ is a constraint in I, then it holds under the assigment $\mathbf{d} = \mathbf{0}$. The result is that $0 - 0 \geq a_{ij}$, that is, a_{ij} is nonpositive.

• Once the algorithm finds the key configuration $\mathbf{c}$ in line 4, line 5 must determine whether $\phi_{pq}(\mathbf{c})$ is a cut. To do so it tests all features in $S(\mathbf{c})$ for intersection with $\phi_{pq}(\mathbf{c})$.

• Line 6 invokes Algorithm F to calculate *flow*$(\phi_{pq}(\mathbf{c}))$. It requires as input the hurdle sequence of $\phi_{pq}(\mathbf{c})$, which can be found by checking every hurdle that lies between y_p and y_q in altitude. Include only those hurdles of $S(\mathbf{c})$ that cross $\overline{p(\mathbf{c})q(\mathbf{c})}$. The hurdle sequence should, of course, be sorted by y-coordinate, and all crossings must be from bottom to top. Presorting all the hurdles by y-coordinate eliminates the need to sort each individual hurdle sequence.

• In line 7, Dijkstra's algorithm should be used once again, this time computing longest paths in I from module 1, which is the left edge of the bounding box of the sketch. If desired, the designer or design system may add other simple linear inequalities to I, provided that they are all satisfied by the initial layout $S(\mathbf{0})$.

• The configuration $\mathbf{c}$ found in line 7 specifies the optimal compacted sketch, but that sketch must still be constructed at line 8. For the purpose of applying Algorithm R, the single-layer wire-router of [6], it is not necessary to construct a complete sketch $S(\mathbf{c})$, but only to produce something called the ***rubber-band equivalent*** (RBE) of $S(\mathbf{c})$. The features of the RBE are the same as those of $S(\mathbf{c})$, and can be located easily. The wires of the RBE can be found as follows. The set of points not lying on features or hurdles of $S(\mathbf{c})$ is a simply connected region, and its boundary is polygonal (if we allow vertices at infinity). Hence it can be triangulated quickly, and the resulting set of triangles forms a tree under the obvious adjacency relation. We can therefore find for each wire w in $S(\mathbf{c})$ the shortest sequence of triangles that a routing of w could pass through, and apply Algorithm W from [6] to find the wire in the RBE corresponding to w.

4.6. *Complexity analysis*

The worst-case time complexity of Algorithm C is $O(|S|^4)$. (Recall that $|S|$ is the size of the data structure S. If $S = (F, W)$, then $|S| = |F| + |W|$.) We can obtain a more precise bound, however, in terms of $|G|$ and $|I|$. What follows is a line-by-line breakdown of time costs.

(1) According to Section 3, the preprocessing phase takes time $O(|F||W| + |G| \log^2 |G|)$, where G is the adjacency graph of the input sketch (F, W).

(2) Computing constraints for horizontal cuts is no harder than computing them for the other cuts, so line 2 may be ignored.

(3) Enumerating pairs of feature endpoints takes $O(|F|^2 \log |F|)$ time, $O(\log |F|)$ time per pair. This quantity is dominated by other terms.

(4) Line 4 calls Dijkstra's algorithm, which runs in time $O(|E| + |V| \log |V|)$ on a graph (V, E) [3]. Since $|E|$ is $O(|I|)$, and $|V|$ is n, the number of modules, line 4 uses $\Theta(|I| + n \log n)$ time in each of $\Theta(|F|^2)$ iterations.

(5) Line 5 takes $O(|F|)$ time per potential cut, and hence line 6 dominates it.

(6) Algorithm F uses $O(|F| \log^2 |G|)$ time, so line 6 costs $O(|F|^3 \log^2 |G|)$ time in total.

(7) The call to Dijkstra's algorithm in line 7 contributes a neglible quantity to the running time.

(8) Careful analysis shows that the construction and routing of the output sketch requires only $O(|F||G|\log|G|)$ time [6].

Thus the total running time of Algorithm C is

$$O\big(|F||W| + |G|\log^2|G| + |F|^2(|I| + n\log n) + |F|^3\log^2|G| + |F||G|\log|G|\big).$$

Since $|I| = O(|F|^2)$ and $|G| = O(|F||W|)$, this expression yields the claimed bound of $O(|S|^4)$. The only term that exceeds $O(|S|^3\log^2|S|)$ is the term $|F|^2|I|$ due to repeated constraint solving at line 4.

Which part of Algorithm C will dominate in practice is not clear. In the worst case, $|G|$ can be as high as $\Omega(|F||W|)$, if some $\Omega(|W|)$ wire segments intersect $\Omega(|F|)$ hurdles each, and the crossings are not redundant. In most situations, however, $|G|$ should be closer to $|F|$. Making reasonable estimates about the average run time of Algorithm F and the density of the constraint graph I, one can predict that actual performance for the entire operation will probably approach $\Theta(|F|^{3+\epsilon})$ for some small positive value of ϵ.

Space usage is easier to evaluate: the main contributors are the graphs G and I, along with Algorithm R, which may use $O(|F||G|)$ space in the worst case. Thus the worst case bound is $O(|F|^2|W|)$, but none of the data structures of Algorithm C or Algorithm R is likely to approach its maximum size. The actual figure will depend on the number of crossings between wires and certain cuts in the sketch (*e.g.*, hurdles), and will probably look like $\Theta(|F|^{1+\alpha})$ for some constant $\alpha \in (0,1)$.

5. The abstract compaction algorithm

To prove the correctness of Algorithm C, the compaction algorithm, we proceed by way of an intermediate procedure called Algorithm A, the *abstract* compaction algorithm. The name derives from the fact that Algorithm A (which is not really an algorithm at all, but just a mathematical definition) abstracts the essential element of Algorithm C, namely the iterative definition of the subspace of configurations to be searched for a minimum width sketch. Algorithm A defines a sequence $A_0, A_1, \ldots, A_m$ of increasingly restricted subsets of the configuration space. These sets will to correspond to sets of configurations satisfying the constraint system I at different stages of Algorithm C.

This section is devoted to the statement of Algorithm A and its preconditions. The next section demonstrates its correctness by proving the following theorem.

Theorem 6. The output A_m of Algorithm A is the connected component of $\{\mathbf{c} \in \mathbf{C}(S) : S(\mathbf{c}) \text{ is routable}\}$ that contains the initial configuration $\mathbf{0}$.

Finally, Section 7 demonstrates the correspondence between Algorithms C and A, and in particular that A_m is precisely the set of configurations that satisfy the final constraint system I of Algorithm C. Together with Theorem 6, this implies that the constraints generated by Algorithm C are both necessary and optimal, if only convex constraints are allowed. Finally, because Algorithm C finds an optimal configuration among those satisfying the constraint system, it will follow that Algorithm C is correct, and that it finds the best solution available to any algorithm of its type.

There are at least two reasons for taking this abstract approach. First of all, it simplifies the correctness proof by separating the mathematical from the algorithmic concerns. Second, and more important, it clarifies the assumptions on which the compaction algorithm relies. An understanding of these assumptions will allow Algorithm C to be easily generalized.

5.1. *Definitions and assumptions*

For the purpose of discussing Algorithm A, we must use a more technical definition of potential cut. Let P and Q be features in the original sketch S, and let ψ be a continuous function that defines, for each configuration $\mathbf{d}$ in $\mathbf{C}(S)$, a line segment between the features $P(\mathbf{d})$ and $Q(\mathbf{d})$ in the sketch $S(\mathbf{d})$. The function ψ is a potential cut if the position of $\psi(\mathbf{d})$ relative to $P(\mathbf{d})$ and $Q(\mathbf{d})$ depends only on the displacement between $P(\mathbf{d})$ and $Q(\mathbf{d})$, namely $\Delta_{PQ}(\mathbf{d}) = d_{\mu(Q)} - d_{\mu(P)}$ (stretching the notation slightly). In other words, ψ must satisfy the following condition.

> If $\mathbf{d}$ and $\mathbf{d}'$ are any two configurations such that $\Delta_{PQ}(\mathbf{d}) = \Delta_{PQ}(\mathbf{d}')$, then $\psi(\mathbf{d}')$ is equal to $\psi(\mathbf{d}')$ shifted to the right by $d'_{\mu(P)} - d_{\mu(P)}$ units.

Following the terminology of the previous section, a configuration $\mathbf{d}$ is said to *protect* a potential cut ψ if $\psi(\mathbf{d})$ is not a cut, or if $\mathit{cap}(\psi(\mathbf{d})) \geq \mathit{flow}(\psi(\mathbf{d}))$. If p and q are points in the original sketch S, then a typical potential cut is the function ϕ_{pq} defined by $\phi_{pq}(\mathbf{d}) = \overline{p(\mathbf{d})q(\mathbf{d})}$.

The input to Algorithm A is a legal sketch S together with a sequence $\Psi(S) = \langle \psi_1, \ldots, \psi_m \rangle$ of potential cuts of S; the output is a set of configurations A_m. As a precondition of Algorithm A, the potential cuts $\Psi(S)$ must determine the routability of the modified sketches $S(\mathbf{d})$. Specifically, they must have the following property:

> *Routability property.* If $S(\mathbf{0})$ is routable, and for all $\lambda \in [0,1]$ the configuration $\lambda\mathbf{d}$ protects every $\psi \in \Psi(S)$, then $S(\mathbf{d})$ is routable.

The capacities of the potential cuts must also have a special property:

> *Bitonic property.* For each $\psi \in \Psi(S)$, and each line L in configuration space, there is a point $\mathbf{c}$ of L at which the capacity $\mathit{cap}(\psi(\mathbf{c}))$ is minimal, and $\mathit{cap}(\psi(\mathbf{d}))$ is nondecreasing as $\mathbf{d}$ moves away from $\mathbf{c}$ along L.

In principle, my compaction method depends on only one further fact:

> *Ordering property.* Suppose that the following statements hold.
> (1) The configuration $\mathbf{d}$ protects ψ_i for all $i < k$.
> (2) The configuration $\mathbf{d}$ lies on the boundary of the set $\{\mathbf{c} \in \mathbf{R}^n : \psi(\mathbf{c}) \text{ is a cut}\}$.

(3) The cut s is properly contained in the line segment $\psi_k(\mathbf{d})$.
Then s is safe in $S(\mathbf{d})$.

In practice, of course, we also desire that the sequence $\langle\psi_i\rangle$ be computable in polynomial time. As we show in Section 7, the sequence of potential cuts examined by Algorithm C has all these desirable properties.

5.2. *The abstract algorithm*

Before plunging into the algorithm, I shall provide a brief overview. Algorithm A computes a sequence of polytopes in configuration space, each one contained in the last. The configurations in the kth polytope will protect the first k potential cuts in $\Psi(S)$. To process ψ_k, the kth potential cut, the algorithm first determines whether ψ_k is unsafe in any configuration in the current polytope. If not, the algorithm ignores ψ_k. Otherwise, it defines a set of unacceptable configurations in which the capacity of ψ_k falls below some critical value. This set contains all configurations in the current polytope that fail to protect ψ_k. Its complement consists of two half-spaces: one in which the lower endpoint of ψ_k is far to the right of the upper endpoint, and one in which the situation is reversed. Because the initial configuration is always acceptable, it must fall into one half-space or the other; the kth polytope is determined by intersecting the $(k-1)$st polytope with the half-space that contains $\mathbf{0}$. Thus Algorithm A only considers configurations reachable from the initial one; just as in Figure 7, one is not allowed to pass through a region of unacceptable configurations to reach a safe configuration on the other side.

Algorithm A. (Finds the set of acceptable modifications of a sketch.)
Input: a legal sketch S with n modules specified, and a sequence $\langle\psi_1,\ldots,\psi_m\rangle$ of potential cuts of S with the routability, bitonic, and ordering properties.
Output: the configuration set A_m.
Local variables: an integer k, polytopes A_k of acceptable configurations, sets U_k of unacceptable configurations, and inequalities Ξ_k.

```
1.  $A_0 \leftarrow \mathbf{C}(S)$;
2.  for $k \leftarrow 1$ to $m$ do
      begin
3.      if some $\mathbf{c} \in A_{k-1}$ does not protect $\psi_k$ then
          begin
4.          $U_k \leftarrow \{\mathbf{d} \in \mathbf{R}^n : cap(\psi_k(\mathbf{d})) < flow(\psi_k(\mathbf{c}))\}$;
Note: If the endpoints of $\psi_k$ lie on the features $P_k$ and $Q_k$, then $U_k$ has the form
      $\{\mathbf{d} : \Delta^- < \Delta_{P_kQ_k}(\mathbf{d}) < \Delta^+\}$, and either $0 \in (-\infty, \Delta^-]$ or $0 \in [\Delta^+, \infty)$.
5.          $\Xi_k \leftarrow \begin{cases} \Delta_{P_kQ_k}(\mathbf{d}) \geq \Delta^+, & \text{if } 0 \geq \Delta^+; \\ \Delta_{P_kQ_k}(\mathbf{d}) \leq \Delta^-, & \text{if } 0 \leq \Delta^-; \end{cases}$
6.          $A_k \leftarrow \{\mathbf{d} \in A_{k-1} : \mathbf{d} \text{ satisfies } \Xi_k\}$
          end
7.      else $A_k \leftarrow A_{k-1}$; $U_k \leftarrow \emptyset$
      end.
```

Some remarks about Algorithm A are in order.

- The set U_k is defined in terms of an arbitrary configuration $\mathbf{c} \in A_{k-1}$ that fails to protect ψ_k. In the next section, we show that U_k is independent of the choice of $\mathbf{c}$.
- Observe that Ξ_k is a simple linear inequality between $d_{\mu(P_k)}$ and $d_{\mu(Q_k)}$, and hence defines a closed half-space in $\mathbf{R}^n$. Since A_0 is convex, the set A_k is therefore convex for each k.
- In the light of the following results, the definition of A_k in lines 5–6 may be read "A_k is the component of $A_{k-1} - U_k$ that contains $\mathbf{0}$."

6. Correctness of the abstract algorithm

This section proves the correctness of Algorithm A, by which we mean the following theorem.

Theorem 6. The output A_m of Algorithm A is the connected component of $\{\mathbf{c} \in \mathbf{C}(S) : S(\mathbf{c}) \text{ is routable}\}$ that contains the initial configuration $\mathbf{0}$.

In other words, Algorithm A above defines precisely the set of routable configurations that are reachable from the initial one. In the process of proving Theorem 6, we develop some results that will be very useful later on, both in proving Algorithm C correct and in finding correct extensions of it.

6.1. *Body of the correctness proof*

The following lemma is fundamental to the correctness proof. Its proof is lengthy and fairly deep, so we defer it to the end of this section.

Definition. Two configurations, $\mathbf{d}$ and $\mathbf{d}'$, are *equivalent* with respect to a potential cut ψ if the hurdle sequences of $\psi(\mathbf{d})$ in $S(\mathbf{d})$ and $\psi(\mathbf{d}')$ in $S(\mathbf{d}')$ are identical. This relation is written "$\mathbf{d} \approx \mathbf{d}'$ with respect to ψ".

Lemma 7. Let $\mathbf{d}$ and $\mathbf{d}'$ be configurations in $\mathbf{C}(S)$, let L be $\{(1-\lambda)\mathbf{d}+\lambda\mathbf{d}' : \lambda \in [0,1]\}$, and let ψ be a potential cut whose capacity has at most one local minimum on L. Suppose also that whenever $\mathbf{b} \in L$ lies on the boundary of $\{\mathbf{c} \in \mathbf{C}(S) : \psi(\mathbf{c}) \text{ is a cut}\}$, all the cuts contained in the line segment $\psi(\mathbf{b})$ are safe. Then:

(1) If $\mathbf{d}'$ protects ψ but $\mathbf{d}$ does not, then $\mathit{cap}(\psi(\mathbf{d}')) \geq \mathit{flow}(\psi(\mathbf{d}))$.
(2) If neither $\mathbf{d}$ nor $\mathbf{d}'$ protects ψ, then $\mathbf{d} \approx \mathbf{d}'$ with respect to ψ.

Lemma 7 provides us with the following lemma, our main tool for proving Theorem 6. We shall use this lemma frequently.

Lemma 8. (*Potential Cut Lemma*) Suppose $1 \leq k \leq m$, and let $\mathbf{d}$ and $\mathbf{d}'$ be configurations in A_{k-1}.

(1) If $\mathbf{d}'$ protects ψ_k but $\mathbf{d}$ does not, then $\mathit{cap}(\psi_k(\mathbf{d}')) \geq \mathit{flow}(\psi_k(\mathbf{d}))$.
(2) If neither $\mathbf{d}$ nor $\mathbf{d}'$ protects ψ_k, then $\mathbf{d} \approx \mathbf{d}'$ with respect to ψ_k.

According to Lemma 3 of Section 3, the flow across a cut depends only upon the cut's hurdle sequence. Therefore, configurations that are equivalent with respect to ψ_k have equal flow across ψ_k. Statement (2) of Lemma 8 thus implies that any two configurations $\mathbf{d}, \mathbf{d}' \in A_{k-1}$ that fail to protect ψ_k must satisfy $\mathit{flow}(\psi_k(\mathbf{d})) = \mathit{flow}(\psi_k(\mathbf{d}'))$. Thus Lemma 8 fulfills a promise made in the previous section, to show that the sets U_k defined in lines 4 and 7 of Algorithm A are uniquely determined.

The proof of Lemma 8 depends on several facts about the set A_{k-1}. In particular, the lemma makes no sense unless A_{k-1} is well defined. On the other hand, A_k is well defined only if the Potential Cut Lemma holds for A_{k-1}. We must therefore prove Lemma 8 in parallel with the following claim.

Lemma 9. For $1 \le k \le m$, the following statements hold:
(3) The set A_k is well defined by Algorithm A.
(4) The point $\mathbf{0}$ lies in A_k.
(5) Every configuration in A_k protects the potential cuts ψ_1 through ψ_k.

Proof of Lemmas 8 and 9. The proof proceeds by induction on k, with the inductive hypothesis being the conjunction of (3), (4), and (5). A basis for this hypothesis is easily established at $k = 0$: the set A_0 is obviously well defined, $\mathbf{0} \in A_0$ by definition, and condition (5) is vacuously true. So assume $k \ge 1$. The key step is the proof of (1) and (2), in Lemma 8, from the inductive hypothesis.

(1,2) We apply Lemma 7 to the configurations $\mathbf{d}$ and $\mathbf{d}'$ and the potential cut ψ_k. By the bitonic property, the capacity function of ψ_k is minimal on at most one interval of L. And since A_{k-1} is a convex set, the inductive hypothesis implies that every configuration $\mathbf{c} \in L$ protects the potential cuts ψ_1 through ψ_{k-1}. This fact, combined with the ordering property, demonstrates the final assumption of Lemma 7. The conclusion of that lemma is identical to the conclusion of Lemma 8.

(3) For A_k to be well defined, the set U_k defined in line 4 of Algorithm A must have the specific form $\{\mathbf{d} \in \mathbf{C}(S) : \Delta^- < \Delta_{P_kQ_k}(\mathbf{d}) < \Delta^+\}$, for some Δ^- and Δ^+. Recall that U_k includes a point $\mathbf{d}$ if and only if the capacity $\mathit{cap}(\psi_k(\mathbf{d}))$ of $\psi_k(\mathbf{d})$ is less than the constant $f = \mathit{flow}(\psi_k(\mathbf{c}))$. But by the definition of a potential cut, $\psi_k(\mathbf{d})$ depends only on $\Delta_{P_kQ_k}(\mathbf{d})$. Hence it suffices to show that the set

$$\{\Delta_{P_kQ_k}(\mathbf{d}) : \mathbf{d} \in \mathbf{R}^n \text{ and } \mathit{cap}(\psi_k(\mathbf{d})) < f\}$$

is a nonempty open interval (Δ^-, Δ^+). Choose a line L through $\mathbf{c}$ on which $\Delta_{P_kQ_k}(\mathbf{d})$ is not constant. The bitonic property of ψ_k implies that the set $\{\mathbf{d} \in L : \mathit{cap}(\psi_k(\mathbf{d})) < f\}$ is a open interval of L; it is nonempty because it contains $\mathbf{c}$. Since $\Delta P_kQ_k(\mathbf{d})$ is a nonconstant linear function on L, the set

$$\{\Delta_{P_kQ_k}(\mathbf{d}) : \mathbf{d} \in L \text{ and } \mathit{cap}(\psi_k(\mathbf{d})) < f\}$$

is also a nonempty open interval. This is enough, because every value $\Delta_{P_kQ_k}(\mathbf{d})$ is represented by some $\mathbf{d} \in L$.

(4) By the induction hypothesis, $\mathbf{0} \in A_{k-1}$. If every $\mathbf{c} \in A_{k-1}$ protects ψ_k, then $\mathbf{0} \in A_k$ trivially. Otherwise, apply (1) to $\mathbf{0}$ and $\mathbf{c}$. (Because $S(\mathbf{0})$ is routable, $\mathbf{0}$ protects ψ_k by Lemma 1.) So $\mathit{cap}(\psi_k(\mathbf{0})) \ge \mathit{flow}(\psi_k(\mathbf{c}))$, whence $\mathbf{0} \notin U_k$. Because $\Delta_{P_kQ_k}(\mathbf{0}) = 0$, by

definition, we have $0 \notin (\Delta^-, \Delta^+)$. Thus $\mathbf{0}$ satisfies the constraint Ξ_k defined at line 5, and so $\mathbf{0} \in A_k$.

(5) Since $A_k \subseteq A_{k-1}$, every configuration $\mathbf{d} \in A_k$ protects ψ_1 through ψ_{k-1}, by the induction hypothesis; it remains to show that every $\mathbf{d} \in A_k$ protects ψ_k. Suppose that $\mathbf{d} \in A_{k-1}$ fails to protect ψ_k. Then U_k is nonempty, and is defined in terms of some configuration $\mathbf{c}$. By part (2), $\mathbf{d} \approx \mathbf{c}$ with respect to ψ_k, and in particular $\mathit{flow}(\psi_k(\mathbf{d})) = \mathit{flow}(\psi_k(\mathbf{c}))$. Because $\mathbf{d}$ does not protect ψ_k, certainly $\mathit{cap}(\psi_k(\mathbf{d})) < \mathit{flow}(\psi_k(\mathbf{d}))$, and it follows that $\mathbf{d} \in U_k$. But the constraint Ξ_k excludes all members of U_k from A_k. Therefore $\mathbf{d} \notin A_k$. $\square$

From the above lemma, most of Theorem 6 follows quickly. First of all, the initial configuration $\mathbf{0}$ is a member of A_m by claim (4). Second, if $\mathbf{d} \in A_m$, then for all $\lambda \in [0,1]$, the configuration $\lambda\mathbf{d}$ lies in A_m, and hence protects every $\psi \in \Psi(S)$ by claim (5). Therefore by the routability property, $S(\mathbf{d})$ is routable for all $\mathbf{d} \in A_m$. It remains to argue that A_m is a single connected component of $\{\mathbf{d} \in \mathbf{C}(S) : S(\mathbf{d}) \text{ is routable}\}$. To do so, we make use of an elementary topological result. A subset X of a topological space is said to ***surround*** another subset Y if Y lies in the interior of X, and the closure of Y is contained in X. If X surrounds the nonempty set Y, then Y is a connected component of the complement of $X - Y$.

Lemma 10. For $0 \le k \le m$, the set A_m is surrounded by the region

$$X_k = A_k \cup \left(\bigcup_{i=1}^{k} A_{k-1} \cap U_k \right) .$$

Proof. It suffices to show that A_m is closed and X_k is open, because clearly $A_m \subseteq X_k$. First the former: consider the boundary of A_m. It must be contained in the boundary of A_0, together with the set of points that satisfy some constraint Ξ_k with equality. If a point $\mathbf{d}$ lies on the boundary of A_0, then two distinct modules in $S(\mathbf{d})$ intersect, and hence any configuration sufficiently close to $\mathbf{d}$ does not correspond to a routable sketch. Therefore $\mathbf{d}$ cannot lie on the boundary of A_0, and as a consequence, A_m is just the set of configurations that satsify the inequalities Ξ_k: a closed, convex polytope in R^n.

Now we prove by induction on k that X_k is open. The basis case, $X_0 = A_0$, is left to the reader. Let $k > 0$, and consider the nontrivial case when U_k is nonempty. From the definition of X_k we derive $X_k = (X_{k-1} - A_{k-1}) \cup A_k \cup (A_{k-1} - U_k)$, which reduces to $X_{k-1} - (A_{k-1} - U_k - A_k)$. The set $B = A_{k-1} - U_k - A_k$ is the intersection of A_{k-1} with one of the closed half-spaces forming the complement of U_k; it remains to show that B is closed in X_{k-1}. But A_{k-1} is just the subset of X_{k-1} satisfying the constraints Ξ_i, for all $i < k$, so B is X_{k-1} intersected with finitely many closed half-spaces. Therefore $X_k = X_{k-1} - B$ is open. $\square$

Setting $k = m$ in Lemma 10, we find that $\bigcup_{i=1}^{m}(A_{i-1} \cap U_i)$ disconnects A_m from the rest of $\mathbf{R}^n$. Hence the connected component of $\{\mathbf{d} \in \mathbf{C}(S) : S(\mathbf{d}) \text{ is routable}\}$ that contains A_m cannot be a proper superset of A_m, unless it also contains a point in $A_{i-1} \cap U_i$ for some i. But if $\mathbf{d} \in A_{i-1}$ corresponds to a routable sketch, then it protects ψ_i, and statement (1) of the Potential Cut Lemma applies to $\mathbf{d}$ and the

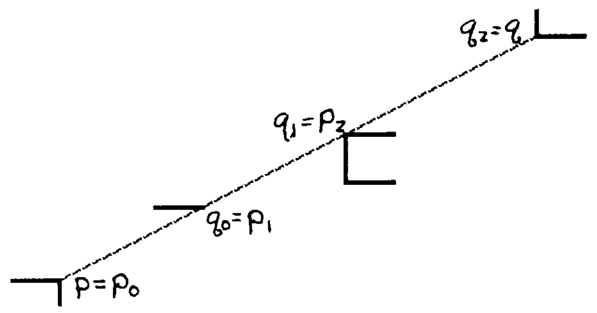

Figure 8. The line segment $s_\lambda = \psi(\mathbf{b}_\lambda)$ just as it ceases to be a cut.

configuration $\mathbf{c} \in A_{i-1}$ used to define U_i. It shows that $cap(\psi_i(\mathbf{d})) \geq flow(\psi_i(\mathbf{c}))$, which means that $\mathbf{d} \notin U_i$. Therefore $\mathbf{d} \in A_{i-1} \cap U_i$ implies that $S(\mathbf{d})$ is not routable. So A_m is precisely equal to the component of $\{\mathbf{d} \in \mathbf{C}(S) : S(\mathbf{d}) \text{ is routable}\}$ that contains $\mathbf{0}$. This completes the proof of Theorem 6, except for Lemma 7.

6.2. *Core of the correctness proof*

We now justify the lemma upon which Theorem 6 is ultimately based. Its proof explains the purpose of the bitonic and ordering properties.

Lemma 7. Let $\mathbf{d}$ and $\mathbf{d}'$ be configurations in $\mathbf{C}(S)$, let L be $\{(1-\lambda)\mathbf{d}+\lambda\mathbf{d}' : \lambda \in [0,1]\}$, and let ψ be a potential cut whose capacity has at most one local minimum on L. Suppose that whenever $\mathbf{b} \in L$ lies on the boundary of $\{\mathbf{c} \in \mathbf{R}^n : \psi(\mathbf{c}) \text{ is a cut}\}$, all the cuts contained in the line segment $\psi(\mathbf{b})$ are safe. Then:

(1) If $\mathbf{d}'$ protects ψ but $\mathbf{d}$ does not, then $cap(\psi(\mathbf{d}')) \geq flow(\psi(\mathbf{d}))$.
(2) If neither $\mathbf{d}$ nor $\mathbf{d}'$ protects ψ, then $\mathbf{d} \approx \mathbf{d}'$ with respect to ψ.

Proof. For $\lambda \in [0,1]$, let $\mathbf{b}_\lambda$ represent $(1-\lambda)\mathbf{d}+\lambda\mathbf{d}'$. As λ varies from 0 to 1, $S(\mathbf{b}_\lambda)$ varies from $S(\mathbf{d})$ to $S(\mathbf{d}')$, and the line segment $s_\lambda = \psi(\mathbf{b}_\lambda)$ is sometimes a cut, and sometimes it crosses features. Denote the flow across s_λ by $f_\lambda = flow(\psi(\mathbf{b}_\lambda))$, and the capacity (or "length") of s_λ by $l_\lambda = cap(\psi(\mathbf{b}_\lambda))$.

We first argue that the set $Z = \{\lambda \in [0,1] : s_\lambda \text{ is a cut}\}$, considered as a subspace of the unit interval, is open. Let s_λ be a cut; say it connects the features P and Q. Because $S(\mathbf{b}_\lambda)$ is compatible with S_0, there is some positive distance between s_λ and every feature but P and Q; no other features can touch the endpoints of s_λ. And since s_λ and the module positions in $S(\mathbf{b}_\lambda)$ are all continuous functions of λ, there is some neighborhood of λ whose points all correspond to cuts. So Z is open, and hence it consists of disjoint intervals, each one open in $[0,1]$. We now focus attention on one of these intervals, call it Λ. If α lies on the boundary of Λ, then s_α is not a cut. The following claim is the crux of the argument.

Claim: For all $\lambda \in \Lambda$, and all α on the boundary of Λ, the configuration $\mathbf{b}_\lambda$ protects ψ unless $l_\lambda < l_\alpha$.

First note that the flow f_λ does not vary as λ moves through Λ. The motion of $S(\mathbf{b}_\lambda)$ is continuous, and ψ is a continuous function, so no features can jump across s_λ while $\lambda \in \Lambda$. Therefore any configuration $\mathbf{b}_\lambda$, for $\lambda \in \Lambda$, is equivalent to any other with respect to ψ. Hence by Lemma 3, f_λ is a constant f_Λ for all $\lambda \in \Lambda$.

Now consider the sketch S_α. At this point, one or more features have just contacted line segment s_α, and hence s_α is broken up into a series of cuts $\overline{p_0q_0}, \ldots, \overline{p_jq_j}$. (See Figure 8.) Because $\mathbf{b}_\alpha$ is on the boundary of the set of configurations that make ψ a cut, all the cuts $\overline{p_iq_i}$ are safe in configuration $\mathbf{b}_\alpha$. Adding up the inequalities that define safety, we find that

$$\sum_{i=0}^{j} cap(\overline{p_iq_i}) \geq \sum_{i=0}^{j} flow(\overline{p_iq_i}) \ .$$

Compared to the flow f_Λ, the right-hand sum cannot be deficient by more than j. Even if each of the j intervening features were terminals, they could only contribute j extra wires to f_Λ. And by the definition of capacity, the left-hand sum is bounded by $l_\alpha - j$. The result is that $l_\alpha \geq f_\Lambda$. By definition, if $\mathbf{b}_\lambda$ fails to protect ψ then $l_\lambda < f_\Lambda$, so $l_\lambda < l_\alpha$.

We now prove the lemma. Both parts of the lemma assume that $\mathbf{d}$ fails to protect ψ, so we may assume that $s_0 = \psi(\mathbf{d})$ is a cut, and that $l_0 < f_0$. Suppose first that $\mathbf{d}$ and $\mathbf{d}'$ are equivalent with respect to ψ. Then $flow(\psi(\mathbf{d}')) = flow(\psi(\mathbf{d}))$. If $\mathbf{d}'$ protects ψ, then also $cap(\psi(\mathbf{d}')) \geq flow(\psi(\mathbf{d}'))$, and the two inequalities together establish (1). Conclusion (2) is trivial if $\mathbf{d} \approx \mathbf{d}'$. Now suppose that $\mathbf{d}$ and $\mathbf{d}'$ are not equivalent with respect to ψ. Then there exists $\lambda \in (0,1]$ such that s_λ is not a cut. Let α be the smallest such value, and consider the interval $\Lambda = [0, \alpha)$. Since $\mathbf{d} = \mathbf{b}_0$ does not protect ψ, the claim implies $l_0 < l_\alpha$. Now by assumption, l_λ as a function of λ has at most one local minimum in $[0,1]$. Because $l_0 < l_\alpha$, the minimum value of l_λ must occur in the interval $(-\infty, \alpha)$. Hence l_λ is nondecreasing on $[\alpha, 1]$, and we have $l_1 \geq l_\alpha \geq f_0$. This proves conclusion (1), because l_1 is $cap(\psi(\mathbf{d}'))$ and f_0 is $flow(\psi(\mathbf{d}))$. Now we prove (2) by showing that $\mathbf{d}'$ protects ψ. If s_1 is a cut, let β be the largest value such that s_β is not a cut. (One must exist, for we are assuming $\mathbf{d} \not\approx \mathbf{d}'$.) Applying the claim to the interval $\Lambda = (\beta, 1]$, we find that $\mathbf{b}_1$ protects ψ because $l_1 \geq l_\beta$. Since $\mathbf{b}_1 = \mathbf{d}'$, this proves statement (2). $\square$

7. Correctness of the implementation

In this section, we build upon the results of Sections 3 and 6 to prove the correctness of Algorithm C, the concrete compaction algorithm. The hard part of the proof is over: Algorithm A, which is an abstract description of the compaction algorithm, is proven correct by Theorem 6 of the previous section. It remains to show that Algorithm C is just a special case of Algorithm A. There are two steps to this process: first, to identify the potential cuts that Algorithm C uses, and show that they satisfy the preconditions of Algorithm A; and second, to prove an explicit correspondence between the quantities computed by the two algorithms. The correctness of the compaction algorithm will then follow from the correctness of its subroutines (Algorithms F and R) along with Theorem 6.

7.1. *Preconditions of Algorithm A*

Our first task is to show that the potential cuts used by Algorithm C satisfy the requirements of Algorithm A, namely the routability, bitonic, and ordering properties. The potential cuts in question are the following.

(1) Horizontal potential cuts $\psi_1, \ldots, \psi_h$, each of the form ϕ_{pq} where either p or q is a feature endpoint.

(2) Diagonal potential cuts $\psi_{h+1}, \ldots, \psi_m$, each of the form ϕ_{pq} where both p and q are feature endpoints.

We may assume that the potential cuts are numbered in the order that Algorithm C examines them, because we know it checks the horizontal ones first. Denote the input sketch by S, and the sequence $\langle \psi_1, \ldots, \psi_m \rangle$ by $\Psi(S)$.

Lemma 11. The sequence $\Psi(S)$ has the ordering, bitonic, and routability properties.

Proof. The sequence $\Psi(S)$ is easily seen to have the ordering property. Let k satisfy $1 \leq k \leq m$, and suppose that $\psi_k(\mathbf{d})$ contains a smaller cut. For $\mathbf{d}$ to lie on the boundary of the set $\{\mathbf{c} \in \mathbf{R}^n : \psi_k(\mathbf{c}) \text{ is a cut}\}$ means that the features interrupting $\psi_k(\mathbf{d})$ must do so at their endpoints, and furthermore that $\psi_k(\mathbf{d})$ is not horizontal. Therefore all the cuts contained in $\psi_k(\mathbf{d})$ are cuts between feature endpoints, and they have smaller height than ψ_k. Hence they appear in the list $\psi_1, \ldots, \psi_{k-1}$.

The bitonic property is also easy to verify, because any potential cut of the form ϕ_{pq} has a convex capacity function. The reason is that $cap(\phi_{pq}(\mathbf{d}))$ is essentially the norm of a vector that is linear in the components of $\mathbf{d}$, namely $(p - q) + (d_{\mu(p)} - d_{\mu(q)})\mathbf{i}$, where $\mathbf{i}$ is the unit vector $\langle 1, 0 \rangle$. Convexity now follows from elementary properties of norms.

Verifying the routability property is somewhat more difficult. Let $\mathbf{d}$ be a configuration such that $\lambda\mathbf{d}$ protects all $\psi \in \Psi(S)$ for all $\lambda \in [0,1]$. By Theorem 2, the Planar Routability Theorem, it suffices to show that $\mathbf{d}$ protects every critical potential cut of S. Consider an arbitrary feature endpoint p of S and another feature Q. The critical potential cut χ_{pQ} between them is defined by $\chi_{pQ}(\mathbf{d}) = \overline{p(\mathbf{d})q(\mathbf{d})}$ where $q(\mathbf{d})$ is the closest point on the feature $Q(\mathbf{d})$ to the point $p(\mathbf{d})$. If the horizontal line drawn through p intersects Q, then χ_{pQ} is always horizontal because Q is either a horizontal or vertical line segment. In this case, χ_{pQ} is equal to one of the potential cuts ψ_i with $1 \leq i \leq h$, so that $\mathbf{d}$ automatically protects χ_{pQ}. If Q is a vertical line segment, then its closest point to p is always the same endpoint of Q, and $\chi_{pQ} = \psi_i$ for some $i > h$. So we may assume that Q is horizontal and is displaced vertically from p. Furthermore, we may assume that $\chi_{pQ}(\mathbf{d})$ does not share an endpoint with $Q(\mathbf{d})$. Then $\chi_{pQ}(\mathbf{d})$ must be a vertical line segment. Now either $\chi_{pQ}(\mathbf{0})$ is also vertical, or there is some configuration $\alpha\mathbf{d}$ such that $\chi_{pQ}(\alpha\mathbf{d})$ is vertical and shares an endpoint with $Q(\alpha\mathbf{d})$. In either case, we have a configuration $\alpha\mathbf{d}$ that protects χ_{pQ}, and such that $cap(\chi_{pQ}(\lambda\mathbf{d}))$ is minimal at $\lambda = \alpha$.

We now suppose that $\mathbf{d}$ does not protect χ_{pQ}, and apply Lemma 7 to the potential cut χ_{pQ} and the configurations $\mathbf{d}$ and $\alpha\mathbf{d}$. The capacity $cap(\chi_{pQ}(\mathbf{d}))$ is a convex function of $\mathbf{d}$, as one may check. Hence it has at most one local minimum on the line segment L between $\alpha\mathbf{d}$ and $\mathbf{d}$. Furthermore, if $\lambda\mathbf{d}$ lies on the boundary of $\{\mathbf{c} \in \mathbf{R}^n : \chi_{pQ}(\mathbf{c}) \text{ is a cut}\}$, then $\chi_{pQ}(\lambda\mathbf{d})$ intersects feature endpoints only, and hence the cuts contained in $\chi_{pQ}(\lambda\mathbf{d})$ are instances of the potential cuts ψ_{h+1} through ψ_m. Since we are assuming that $\lambda\mathbf{d}$ protects every potential cut in $\Psi(S)$, the cuts in $\chi_{pQ}(\lambda\mathbf{d})$ are safe. Thus $\alpha\mathbf{d}$ and $\mathbf{d}$ satisfy part (1) of Lemma 7; we conclude that $cap(\chi_{pQ}(\alpha\mathbf{d})) \geq flow(\chi_{pQ}(\mathbf{d}))$. But $cap(\chi_{pQ}(\alpha\mathbf{d})) \leq cap(\chi_{pQ}(\mathbf{d}))$, because the capacity of $\chi_{pQ}(\lambda\mathbf{d})$ is minimal at $\lambda = \alpha$. Combining these inequalities, we find that $flow(\chi_{pQ}(\mathbf{d})) \leq cap(\chi_{pQ}(\mathbf{d}))$: the configuration $\mathbf{d}$ protects χ_{pQ}. Therefore the potential cuts $\Psi(S)$ have the routability property. $\square$

7.2. *Correspondence between the algorithms*

The final phase of our proof strategy involves showing that the constraints computed by the concrete algorithm define the same space as the constraints Ξ_k defined abstractly. This fact will imply that the compaction algorithm searches precisely the set A_m of acceptable configurations, and correctness will follow quickly. In order to state the correspondence, let C_0 denote the set of configurations satisfying the constraint system I defined at line 2 of Algorithm C, and let C_k denote those configurations satisfying I after the kth iteration on the loop in lines 3–6.

Lemma 12. For all k satisfying $h \le k \le m$, the sets C_{k-h} and A_k are identical.

Proof. Recall that h is the number of horizontal cuts in the sequence $\Psi(S)$. We prove the lemma by induction on k, the basis case being $k = h$. Any configuration in A_h is in $\mathbf{C}(S)$, because $A_h \subseteq A_0$, and also protects the horizontal potential cuts, according to part (5) of Lemma 9. Therefore $A_h \subseteq C_0$. On the other hand, you may check that when the constraint Ξ_k exists, for $k \le h$, it corresponds to the potential cut in I_0 induced by ψ_k. (Here we Lemma 5, which shows the correctness of Algorithm F.) Therefore $C_0 \subseteq A_h$.

For the inductive step, suppose that $C_{k-h-1} = A_{k-1}$. We first draw a correspondence between the configurations **c** found by Algorithms A and C. The key observation is that the configuration **c** found by Algorithm C at line 4 minimizes the capacity $cap(\psi_k(\mathbf{c}))$ over all $t \in C_{k-h-1} = A_{k-1}$. It does so by minimizing the horizontal separation $|\Delta_{pq}(\mathbf{c})|$ between the points $p(\mathbf{c})$ and $q(\mathbf{c})$, since their vertical separation is fixed. (Dijkstra's algorithm is applicable here, because according to Lemma 9, the initial configuration $\mathbf{0}$ satisfies the constraint system.) We wish to argue that if any $\mathbf{d} \in A_{k-1}$ fails to protect ψ_k, then neither does **c**. Suppose to the contrary that **c** protects ψ_k but $\mathbf{d} \in A_{k-1}$ does not. Then by the Potential Cut Lemma, statement (1), we have $cap(\psi_k(\mathbf{c})) \ge flow(\psi_k(\mathbf{d}))$. But $cap(\psi_k(\mathbf{c})) \le cap(\psi_k(\mathbf{d}))$ by the choice of **c**, so $cap(\psi_k(\mathbf{d})) \ge flow(\psi_k(\mathbf{d}))$, and **d** protects ψ_k after all. Thus line 4 of Algorithm C correctly implements line 3 of Algorithm A.

There are now two cases to consider. If the configuration **c** does protect ψ_k, then so do all configurations in A_{k-1}. Therefore Algorithm A sets A_k to A_{k-1}, and Algorithm C does not change I, so we have $C_{k-h} = A_k$ as desired. On the other hand, if **c** does not protect ψ_k, then Algorithm C adds the constraint

$$\Delta_{pq}(\mathrm{d}) \ge flow(\phi_{pq}(\mathrm{c})) + 1$$

to I, where p and q are defined by $\psi_k = \phi_{pq}$ and $x_q \ge x_p$. It remains to show that the above inequality is the constraint Ξ_k defined by Algorithm A. We first evaluate U_k:

$$\begin{aligned} U_k &= \{\mathrm{d} \in \mathbf{C}(S) : cap(\phi_{pq}(\mathrm{d})) < flow(\phi_{pq}(\mathrm{c}))\} \\ &= \{\mathrm{d} \in \mathbf{C}(S) : \max\{|\Delta_{pq}(\mathrm{d})|, |y_q - y_p|\} - 1 < flow(\phi_{pq}(\mathrm{c}))\}\ . \end{aligned}$$

Since $cap(\phi_{pq}(\mathrm{c})) < flow(\phi_{pq}(\mathrm{c}))$, it follows that $|y_q - y_p| - 1 < flow(\phi_{pq}(\mathrm{c}))$, and so

$$U_k = \{\mathrm{d} \in \mathbf{C}(S) : |\Delta_{pq}(\mathrm{d})| < flow(\phi_{pq}(\mathrm{c})) + 1\}\ .$$

For brevity, we consider only the case where p lies on P_k and q lies on Q_k, and not the reverse. A little algebra then shows that $\Delta^+ = (x_p - x_q) + flow(\phi_{pq}(\mathrm{c})) + 1$ and that

$0 \geq \Delta^+$. Hence Ξ_k is the desired constraint

$$d_{\mu(q)} - d_{\mu(p)} \geq (x_p - x_q) + \mathit{flow}(\phi_{pq}(\mathbf{c})) + 1. \ \square$$

We conclude that the configurations that obey the final constraint system I in Algorithm C are precisely those in A_m. (If the design system adds extra constraints to I, some configurations in A_m may be excluded.) Theorem 6, which characterizes A_m, now implies that every configuration obeying I is routable, and that the constraints I are optimal, unless the constraints are allowed to define a disconnected region of configuration space. Finally, line 7 of Algorithm C finds an optimal configuration obeying the constraint system I. The resulting sketch is guaranteed to be routable, and hence Algorithm R, the single-layer router from [6], can regenerate the layout. This completes the proof of correctness of the compaction algorithm.

8. Extensions and discussion

The purpose of this section is to suggest several ways in which the compaction algorithm can be improved, and to discuss its practical value. I regret that I cannot report here on any generalizations of Algorithm C to wiring models involving multiterminal nets, wires of different widths, or sketches with nonrectilinear features and design rules. The reason is that such generalizations would require extending Theorem 2 and Lemmas 1 and 3 to other wiring models, and the theory of planar routing is not yet sufficiently advanced. Nevertheless, preliminary results indicate that many natural extensions of Algorithm C are possible. I hope to report these results, along with the mathematics that justifies them, in my Ph.D. dissertation.

8.1. *Optimizations of Algorithm C*

Both the time and space performance of Algorithm C can be improved by reducing the size of the adjacency graph. One therefore wishes to choose hurdles in such a way as to minimize the number of crossings between wires and hurdles. Although we defined hurdles so that every obstacle has only one hurdle incident on its right, this property is unimportant. The hurdles can be chosen to be any set of horizontal cuts such that the set of points inside the bounding box, but not lying on a hurdle or a feature, is simply connected. Equivalently, if obstacles and hurdles are considered as the nodes and edges, respectively, of a graph, then this graph must be a tree.

A minimum-cost spanning tree algorithm can be used to find a set of hurdles that cross as few wires as possible. Every horizontal cut between different obstacles is a potential hurdle, but we may restrict our attention to horizontal cuts that are incident on feature endpoints. There are at most $O(|F|)$ such cuts, and they can be thought of as the edges of a graph H over the obstacles. The cost of an edge will be the number of crossings of the cut by wires in the original sketch; costs can be computed efficiently using a scanning algorithm as in [6]. The hurdles are chosen to be the edges in a minimum-cost spanning tree of the graph H.

Another way to speed up Algorithm C is to ignore potential cuts that cannot generate constraints. For example, if a potential cut ϕ_{pq} has minimal capacity in the initial configuration, it cannot generate a constraint. We have noticed this already in the proof of Lemma 12; it follows from statement (1) of the Potential Cut Lemma. Therefore, the algorithm need only check potential cuts ϕ_{pq} for which $|x_q - x_p| > |y_q - y_p|$. Second, the lower endpoint of a feature need not be considered in conjunction with feature endpoints above it, and symmetrically for the upper endpoint. Similarly, potential cuts whose position in the initial configuration travels right from the left endpoint of a horizontal feature need not be considered, and symmetrically for right endpoints. The correctness of these optimizations can be proven using the techniques of Lemma 12. Finally, a potential cut ϕ_{pq} with $x_q > x_p$ need not be checked if in all configurations $\mathbf{d} \in \mathbf{C}(S)$ with $\Delta_{pq}(\mathbf{d}) > |y_q - y_p|$, the line segment $\phi_{pq}(\mathbf{d})$ is not a cut.

None of these improvements affect the fact that Algorithm C requires $\Omega(|F|^3)$ time, not just in the worst case, but in almost every case. To reduce this amount, one must avoid considering most of the potential cuts. Most constraints in practice are likely to be local, so one can try to ignore all potential cuts of sufficiently large height. If one solves the constraint system before evaluating all the potential cuts, and the routing algorithm succeeds, then compaction may be terminated. If the routing algorithm fails, more potential cuts must be considered.

8.2. *Summary and conclusion*

The main theoretical contribution of this paper is a polynomial-time algorithm that compacts IC (or PCB) layouts while introducing jogs into wires in an optimal fashion. The power of Algorithm C comes from the elimination of wires as hard objects in the layout, and their replacement by constraints between modules. The use of routability conditions to solve placement problems is not new [7,13,16], but until now, only channel routing problems had been considered. The reason is that the routability of general planar layouts was not adequately understood until very recently [1,6]. To characterize planar routability requires a robust model of a circuit layer, such as the sketch, and a fair amount of theory. In addition, some care is needed to apply routability conditions to the compaction of general sketches; the correctness of Algorithm C is nontrivial.

On the practical side, my compaction method can be expected to produce high-quality layouts with little designer intervention, saving both in chip area and design time. Its primary drawback lies in its use of computational resources. Although there are good reasons to believe that its worst-case performance bounds will not be approached in practice, resource limitations may prevent it from being used to compact large layouts all at once. Algorithm C is amenable to use at all levels of the design, however, so that hierarchical compaction can alleviate much of the resource problem. It also may be suited to use in channel routing, where the number of components is not too great. The idea, which was implemented at Bell Labs (see Acknowledgements) is as follows: the channel is artificially inflated, so that an an ordinary channel routing algorithm, which may have difficulty with crowded channels, may succeed; then a compactor like Algorithm C, with the ability to insert arbitrarily complex jogs, is applied in order to compact the channel back to the proper size.

One important question left open by my research is whether the compaction method embodied in Algorithm C is more efficient in practice than the straightforward algorithm, namely, inserting jog points into each wire where it crosses each horizontal gridline, and solving the resulting constraint system normally. This technique is evidently simpler than that of Algorithm C, and may be more efficient in practice. On the other hand, it should be possible to extend Algorithm C to situations where wires and modules may contain diagonal segments, and grid-based algorithms break down.

Acknowledgements

I would like to thank my advisor Charles Leiserson for many helpful discussions, and for corrections and comments on this paper. The starting point of this research was provided by Prof. Leiserson who, along with his former student Ron Pinter, discovered the connection between routability conditions and compaction with automatic jog introduction. Some special case algorithms they developed were implemented by Andrew Hume at AT&T Bell Laboratories, and used for channel routing. Thanks also to Ron Greenberg for comments on a draft of this paper.

References

[1] R. Cole and A. Siegel, "River routing every which way, but loose," *Proceedings of the 25th Annual Symposium on Foundations of Computer Science* (October 1984), pp. 65–73.

[2] A. E. Dunlop, "SLIP: symbolic layout of integrated circuits with compaction," *Computer Aided Design*, Vol. 10, No. 6 (November 1978), pp. 387–391.

[3] M. L. Fredman and R. E. Tarjan, "Fibonacci heaps and their uses in improved network optimization algorithms," *Proceedings of the 25th Annual Symposium on Foundations of Computer Science* (October 1984), pp. 338–346.

[4] M. Y. Hsueh, *Symbolic Layout and Compaction of Integrated Circuits*, Ph.D. thesis, EECS Division, University of California, Berkeley, CA (1979).

[5] G. Kedem and H. Watanabe, "Optimization techniques for IC layout and compaction," Technical Report 117, Computer Science Department, University of Rochester (September 1982).

[6] C. E. Leiserson and F. M. Maley, "Algorithms for routing and testing routability of planar VLSI layouts," *Proceedings of the 17th Annual ACM Symposium on Theory of Computing* (May 1985), pp. 69–78.

[7] C. E. Leiserson and R. Y. Pinter, "Optimal placement for river routing," *SIAM Journal on Computing*, Vol. 12, No. 3 (August 1983), pp. 447–462.

[8] C. E. Leiserson and J. B. Saxe, "A mixed-integer linear programming problem which is efficiently solvable," *Proceedings of the 21st Annual Allerton Conference on Communication, Control, and Computing* (October 1983), pp. 204–213.

[9] T. Lengauer, "Efficient algorithms for the constraint generation for integrated circuit layout compaction," *Proceedings of the 9th Workshop on Graphtheoretic Concepts in Computer Science* (June 1983).

[10] T. Lengauer and K. Melhorn, "The HILL system: a design environment for the hierarchical specification, compaction, and simulation of integrated circuit layouts," *Proceedings, Conference on Advanced Research in VLSI* (January 1984).

[11] T. Lengauer, "On the solution of inequality systems relevant to IC layout," *Journal of Algorithms*, Vol. 5, No. 3 (September 1984), pp. 408–421.

[12] R. J. Lipton and R. E. Tarjan, "A separator theorem for planar graphs," *SIAM Journal on Applied Mathematics*, Vol. 36, No. 2 (April 1979), pp. 177–189.

[13] R. Y. Pinter, *The Impact of Layer Assignment Methods on Layout Algorithms for Integrated Circuits*, Ph.D. Thesis, MIT Department of Electrical Engineering and Computer Science (August 1982).

[14] W. S. Scott and J. K. Ousterhout, "Plowing: interactive stretching and compaction in Magic," *Proceedings of the 21st Design Automation Conference* (June 1984), pp. 166–172.

[16] A. Siegel and D. Dolev, "The separation for general single-layer wiring barriers," *Proceedings of the Carnegie-Mellon Conference on VLSI Systems and Computations* (October 1981), pp. 143–152.

[17] T. E. Whitney and C. Mead, "An integer-based hierarchical representation for VLSI," *Proceedings of the Fourth MIT Conference on Advanced Research in VLSI* (April 1986), pp. 241–257.

[18] J. D. Williams, "STICKS – a graphical compiler for high level LSI design," *National Computer Conference* (1978), pp. 289–295.

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (*When Data Entered*)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER MIT/LCS/TR-372	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (*and Subtitle*) Compaction with automatic jog introduction		5. TYPE OF REPORT & PERIOD COVERED
		6. PERFORMING ORG. REPORT NUMBER MIT/LCS/TR-372
7. AUTHOR(*s*) F. Miller Maley		8. CONTRACT OR GRANT NUMBER(*s*) DARPA/DOD N00014-80-C-062
9. PERFORMING ORGANIZATION NAME AND ADDRESS MIT Laboratory for Computer Science 545 Technology Square Cambridge, MA 02139		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS DARPA/DOD 1400 Wilson Boulevard Arlington, VA 22209		12. REPORT DATE September 1986
		13. NUMBER OF PAGES 35
14. MONITORING AGENCY NAME & ADDRESS(*if different from Controlling Office*) ONR/Department of the Navy Information Systems Program Arlington, VA 22217		15. SECURITY CLASS. (*of this report*) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (*of this Report*)

Approved for Public Release, distribution is unlimited

17. DISTRIBUTION STATEMENT (*of the abstract entered in Block 20, if different from Report*)

unlimited

18. SUPPLEMENTARY NOTES

19. KEY WORDS (*Continue on reverse side if necessary and identify by block number*)

compaction, constraint solving, jog insertion, VLSI layout, wire routing

20. ABSTRACT (*Continue on reverse side if necessary and identify by block number*)

This thesis presents an algorithm for one-dimensional compaction of VLSI layouts. It differs from older methods in treating wires not as objects to be moved, but as constraints on the positions of other circuit components. These constraints are determined for each wiring layer using the theory of planar routing. Assuming that the wiring layers can be treated independently, the algorithm minimizes the width of a layout, automatically inserting as many jogs in wires as necessary. It runs in time $O(n^4)$ on input of size n.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (*When Data Entered*)

20.

Several heuristics are suggested for improving the algorithm's practical performance.

The compaction algorithm takes as input a data structure called a sketch, which explicitly distinguishes between flexible components (wires) and rigid components (modules). The algorithm first finds constraints on the positions of modules that ensure enough space is left for wires. Next, it solves the system of constraints by a standard graph-theoretic technique, obtaining a placement for the modules. It then relies on a single-layer router to restore the wires to each circuit layer. An efficient single-layer router is already known; it is able to minimize the lenth of every wire, though not the number of jogs.

As given, the compaction algorithm applies only to a VLSI model that requires wires to run a rectilinear grid. This restriction is needed only because the theory of planar routing (and single-layer routers) has not yet been extended to other models. The compaction algorithm's correctness proof elucidates the assumptions on which the algorithm depends, so that the algorithm is easily generalized once the necessary theoretical machinery is in place.