Order Number 8912882

Incremental VLSI compaction

Carpenter, Clyde William, Ph.D.

Stanford University, 1989

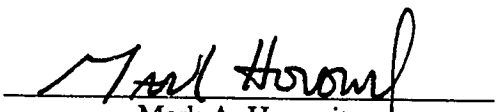
U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

INCREMENTAL VLSI COMPACTION

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

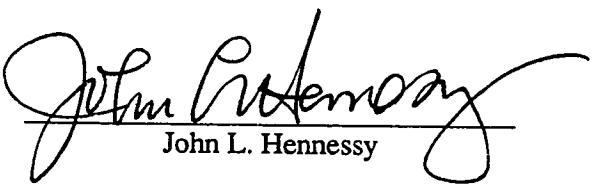
By
Clyde W. Carpenter
November 1988

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.



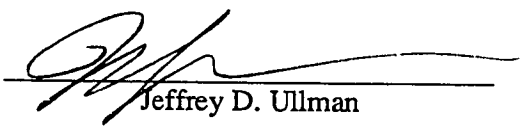
Mark A. Horowitz
(Principal Adviser)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.



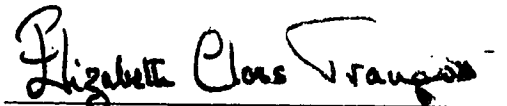
John L. Hennessy

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.



Jeffrey D. Ullman

Approved for the University Committee on Graduate Studies:



Dean of Graduate Studies

INCREMENTAL VLSI COMPACTION

Clyde W. Carpenter, Ph.D.
Stanford University, 1989

Abstract

VLSI compaction is the translation from a high level description of a circuit down to the detailed layout needed for fabrication. A compactor tries to make as compact a layout as possible without violating any design rules. An incremental compactor allows one to edit a schematic or change layout constraints and quickly see the effects of the change.

An incremental compactor has to incrementally generate and solve the constraints needed to enforce the design rules. This dissertation presents an algorithm that uses adjacency lists to generate and incrementally update a minimal complete set of the spacing constraints needed to keep adjacent tiles in a layout from interfering with each other. The base algorithm creates clockwise threaded lists of non-overlapping, fixed-size tiles. The algorithm is complicated by the need to handle wires, overlapping tiles, and various different spacing rules. In near linear time it generates an average of 1.2 spacing constraints per tile. The adjacency lists allow fast, efficient updates when tiles are moved, deleted, or inserted.

This dissertation also presents three algorithms to solve the constraints once they are generated. In addition to minimizing area, these algorithms also minimize the total wire length. One of them calculates the sum of the wire-pull weights on each subtree of a directed spanning tree of active constraints to decide which subtrees need to be moved. This weighted tree provides enough information to make incremental changes in time proportional to the size of the change instead of to the size of the circuit. Wire-length minimization improves the layout but gives compaction a slightly worse than linear expected time.

Acknowledgements

This material is based upon work supported under a National Science Foundation Graduate Fellowship. This work was supported in part by the National Science Foundation under Grant DMC8451822 and in part by the Defense Advanced Research Projects Agency under contracts MDA903-83-C-0335 and N00014-87-K-0828.

Table of Contents

List of Figures

Chapter 1
Introduction

One of the most tedious parts of designing full custom integrated circuits is laying out the artwork needed to fabricate the circuit. The detailed layout has to specify the exact location and size of all the objects required by the fabrication technology to create the desired circuit. VLSI (Very Large Scale Integration) technology has allowed the creation of more and more complex circuits and made layout more and more time consuming. One way to handle this increased complexity is to design circuits at a more abstract level and then automatically translate the designs down to the detailed layout.

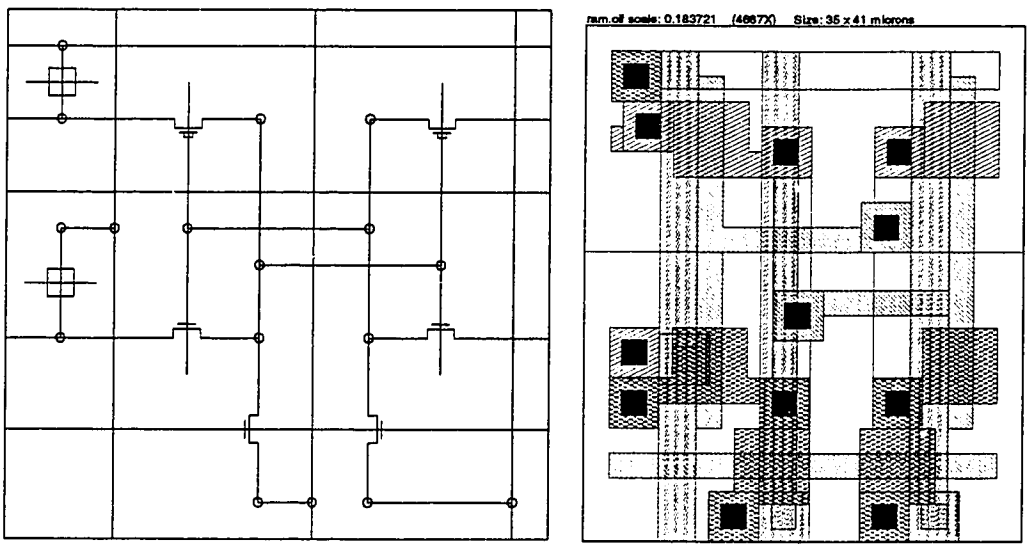


Figure 1-1 RAM Cell Stick Diagram and Layout

Working at a slightly more abstract level allows designers to worry about the relative placement of transistors and wires instead of how to form transistors and how far apart the technology requires things to be spaced. The compaction program takes care of these technology-dependent design rules. An abstract diagram consisting of symbolic transistors wired together with dimensionless wires is called a stick diagram. A stick diagram is much easier to understand and modify than the resulting layout.

Figure 1-1 shows a stick diagram for a six-transistor CMOS static RAM cell and its 250-tile layout. When displayed on a color monitor, the wires are colored to show in which layout layer they run. The stick diagram was converted to a layout by a two-step batch process. First, the compactor created a set of layout-level tiles and generated constraints between the tiles to enforce the design rules. Then it solved the constraints and produced the actual layout.

We want layouts to be as small as possible since smaller layouts are cheaper to manufacture and are generally higher performance. Because of the difficulty of true, two-dimensional area compaction, most compactors instead separately minimize the width and height of a layout. A major problem with these one-dimensional compactors is the lack of control over the interaction between the two dimensions. In batch compaction, if designers do not like a compacted layout or have to change the circuit, they change the circuit description and run everything again in the hope of a better layout. Besides taking a great deal of time, this process often makes it difficult or even impossible to achieve the desired improvement. This dissertation describes algorithms for building an incremental compactor. In incremental compaction, enough information is saved to allow the layout to be quickly updated to reflect changes. For example, when the corners of two tiles catch on each other during a compaction, the designer can simply nudge one of them to break the interlock and produce a better compaction. In batch compaction, the designer can at best rearrange the stick diagram to try to avoid the interlock without creating other, worse problems.

This dissertation is divided into two main parts corresponding to the two main tasks in an incremental compactor: generating and solving the constraints. After giving a brief history of compaction, we describe how adjacency lists can be used to quickly generate and incrementally update a minimal complete set of the spacing constraints needed to keep adjacent tiles in a layout from interfering with each other. These constraints are difficult to handle because they depend on the current tile positions and thus change during compaction. We first describe the adjacency lists algorithm for the simple single-color case and then discuss the enhancements added to handle wires, overlapping tiles, and multiple colors.

The second part of this dissertation describes methods of solving the constraints once we have them. After discussing efficient non-wire-length minimization algorithms, we describe four wire-length minimization algorithms based on the four solution conditions of our weight analogy version of the min-cost max-flow problem.

Minimizing the total wire length is difficult because a tile's placement depends on the total weight of the set of tiles pressing against it, which changes as the tiles move. The first two algorithms work well for batch compaction but do not save enough information to allow incremental updates. The final algorithm, the tree weight algorithm, maintains a directed spanning tree of active constraints. By calculating and storing with each tile the sum of the wire-pull weights on the subtree above it, the algorithm can easily determine which subtrees need to be moved.

After describing these two main tasks, this dissertation ends by giving some implementation details and timings and statistics for a running test compactor. This includes a description of the interaction between the adjacency lists and tree weight algorithms and more detail about the tree operations used by the tree weight algorithm. The tree compactor's performance is compared with several other compactors' on a set of benchmark circuits. It produces better layouts very quickly without using too much memory.

1.1. Background

The earliest computer-aided layout systems were simple drafting programs: layouts were created by digitizing a hand-drawn layout. While this was all right for small circuits, it became tedious as circuits got larger and more complex. One had to worry about creating objects and spacing them far enough apart to not interfere with each other but close enough together to not waste valuable chip area. To make the creation of well-formed objects easier and faster, macros that allowed one-step placement of multiple-tile objects were added to the editors. Batch design-rule checkers were created to check layouts for accidental design-rule violations [Baird 77]. Only recently has this checking been merged with the editors to allow mistakes to be fixed before they propagate too far [Taylor 84].

Even with simple macros and design rule checkers, designers were still responsible for creating artwork without errors and had to be very careful about the space between objects. To try to simplify and speed up the design process, design systems working at a more abstract level were created. Tiles were replaced by symbolic objects; instead of using a macro to place tiles, one just placed a symbolic object on a coarse grid. Williams coined the term sticks [Williams 78] and proposed that compaction should translate a stick diagram, a sketch of the circuit, into a layout. Placing objects on a coarse grid freed the designer from worrying about all the detailed spacing rules. The

first translation systems simply spaced the grid lines using a constant spacing calculated to prevent any possible design-rule violations [Gibson 76]. This wasted a lot of space.

One method to reduce this waste entailed using the shear line algorithm [Akers 70] to post-process the layout. Starting with a layout, a smaller layout was produced by repeatedly compacting along compression ridges. Found by an expensive search and backtrack algorithm, a compression ridge consists of a group of strips of unused space running from one side of the layout to the other. Removing this excess space causes the halves of the layout on either side of the ridge to slide together along shear lines perpendicular to the ridge. The repeated empty-space searches were too slow to be practical [Dunlop 78]. A method that kept track of local information from previous searches was used to speed up the process [Dunlop 80] but did not help much.

Virtual grid systems [Weste 81] used a much faster but less flexible method to reduce the wasted space. They improved the fixed grid method by allowing the grid spacings to vary depending on the objects actually present on each grid line; the largest constraint between two nearby grid lines determined their spacing. The spacing constraints were easy to find since most of them occured between tiles on neighboring grid points. The major problem with these systems was that they linked together groups of tiles that were only related by being on the same grid line and thus unnecessarily increased the layout sizes. Later virtual grid systems [Nyland 87] [Boyer 87a] made trade-offs between allowing more freedom of movement within each virtual grid line and searching for the needed spacing constraints.

More flexible systems [Cho 77] ignored the grid lines completely and instead created a graph of the constraints needed to properly space neighboring objects. The graph was then solved to produce a legal layout. In the next section we discuss the history of generating constraints and in the subsequent section we discuss the history of solving them.

1.2. Generating Constraints

Compaction algorithms operate on the constraints needed to enforce the layout design rules. The stick object definitions prevent ill-formed objects. Therefore a compactor does not have to worry about rules such as poly overlaps of transistors and minimum sizes. Alignment rules tie together the parts of an object and force them to move as a unit. Examples are alignments between a transistor's source, gate, and drain

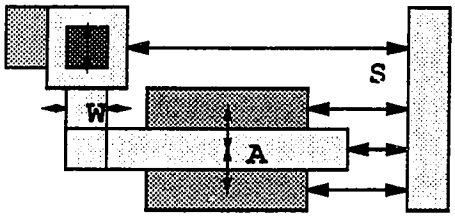


Figure 1-2 Constraints: Alignment, Wire, Spacing

regions (constraints A in Figure 1-2), or between a contact's layers. Wire rules keep wires fully connected to their endpoints, but they also allow wires to slide on wide endpoints. Each wire-endpoint needs one constraint to keep the endpoint's left edge at or to the left of its vertical wire's left edge and another to keep its right edge at or to the right of the wire's right edge (constraints W in Figure 1-2). Finally, the spacing rules provide the margins needed in the fabrication to keep adjacent objects from interfering with each other (constraints S in Figure 1-2). The alignment and wire constraints are easy to handle: they are generated from the circuit description and are invariant during compaction. The spacing constraints are more difficult because they depend on the current x and y positions of the layout tiles.

Many algorithms have been used to generate the spacing constraints. The simplest method is to check for interactions between every pair of tiles. The distance between two tiles is taken to be the maximum of the x and y distances between their edges. Thus an x-direction constraint is needed between two tiles only if the tiles are separated by less than the minimum legal spacing in the y-direction. This algorithm is easy to code but has the drawback that although each comparison is quick, n^2 comparisons are required for an n-tile layout. In Figure 1-3A, half the tests produce constraints (the dark arrows).

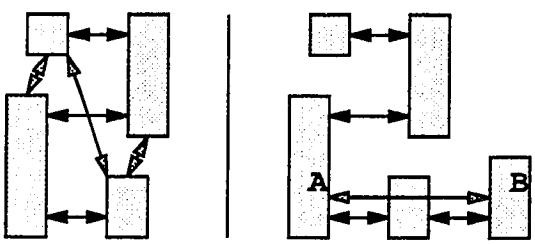


Figure 1-3 A. Every Pair B. Sorted Every Pair

The number of comparisons can be reduced by first sorting the tiles by their bottom coordinates (to generate x constraints). Then each tile can be compared with just the tiles in the horizontal band above it. This gives $O(n^{1.5})$ total comparisons for a roughly square layout [Eichenberger 86]. Besides requiring many comparisons, these two algorithms have another drawback: for normal layouts they produce $O(n^{1.5})$ constraints when, because of transitivity, only $O(n)$ constraints are needed. In Figure 1-3B, the constraint between tiles A and B is unnecessary because of their constraints with the tile between them.

Another way to reduce the number of comparisons and generated constraints is to use intervening groups. Tiles that are rigidly held together are grouped into features. As all the pairs of features are compared, an approximation of the longest path between every pair is created. When constraints with previously compared features require two features to be further apart than the maximum design-rule spacing, there is no need to check the actual spacing constraints between the tiles in these two features. One method [Kingsley 84] keeps track of just one longest path, which results in extra comparisons. Another [Hedges 85] uses a limited-depth search and a square bitmap to store previous results, which requires a great deal of memory. While these are still $O(n^2)$ worst case algorithms, they are relatively fast $O(n^2)$ algorithms. Using the horizontal-band improvement described in the previous paragraph reduces the worst case to $O(n^{1.5})$.

A more complex and theoretically faster algorithm uses shadowing [Hsueh 79a]. The tiles are lexicographically sorted on the (x,y) coordinates of their lower left hand corners. A vertical frontier forms and moves to the right as the tiles are processed. The frontier consists of the processed tiles that could still possibly be seen by a tile to the frontier's right. Constraints are generated from tile(s) in the frontier to each tile before that tile is added to, and shadows part of, the frontier. A tile is shadowed when constraints from it to any possible tile to the right of the frontier are superfluous. This is an elaboration of the basic algorithm used in many design-rule checkers [Baird 77]. Maintaining the frontier can be difficult since a tile often shadows only part of another tile and a large tile can be cut into pieces by the shadows of several small tiles. In Figure 1-4A, tile A is completely shadowed, tile B is partially shadowed, and tile C is shadowed in several sections.

The scan line algorithm [Schlag 82] (sometimes called event driven [Burns 87]) is shadowing seen from the other dimension. A horizontal line is scanned from bottom to

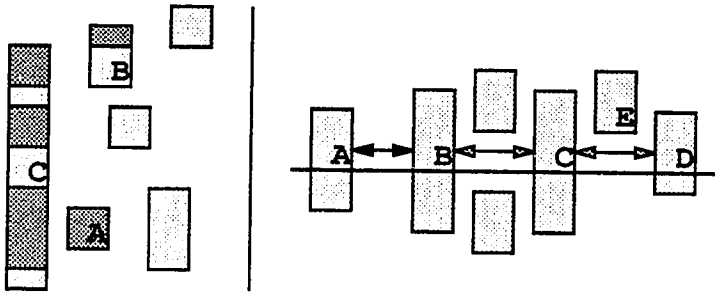


Figure 1-4 A. Shadowing B. Scan Line

top and the set of tiles currently cut by the line is maintained in a balanced tree. Tiles are inserted into and deleted from the tree as the scan line crosses their bottoms and tops, respectively. Neighboring tiles in the tree are visible to each other (along the scan line) and are given spacing constraints. This O($n$lg$n$) algorithm generates at most $3n$ constraints. Visibilities still generate more constraints than are absolutely necessary. In Figure 1-4B, the current scan-line position generates three constraints but only the one between tiles A and B is necessary since the others are blocked by intervening tiles. By caching recent constraints, the algorithm can remove unnecessary constraints during the constraint generation [Lengauer 83]. In Figure 1-4B for example, when the scan line hits tile E, the algorithm realizes that the constraint between tiles C and D can be thrown out.

The above algorithms need to be completely rerun whenever anything is moved or changed. One way to reduce these recomputations is to use the wires to divide the layout into regions [Watanabe 84]. The set of tiles in or on the edge of each region is invariant during compaction since wires remain connected to their endpoints. An every pair algorithm can be used separately on each region. When tiles are moved, only the affected regions need to be redone. One problem with this algorithm is that it runs slowly when regions are large.

The corner-stitching [Ousterhout 84] data structure allows more general, incremental changes. While designed for a layout editor, the routines used to check for design-rule violations are similar to the ones needed to generate the design-rule constraints. Corner stitching adds space tiles to fill the empty spaces in the layout and completely cover the plane. The algorithm keeps the tiles in a canonic form of maximal horizontal strips by cutting tiles horizontally and recombining them, giving preference to width over height. This canonic form is useful in a layout editor since it prevents

fragmentation, but it has some drawbacks for compaction: it loses x-y symmetry and obscures the mapping between higher level descriptions and the layout. Figure 1-5 gives an example of the space tile changes that occur when a tile (shown dark) is added.

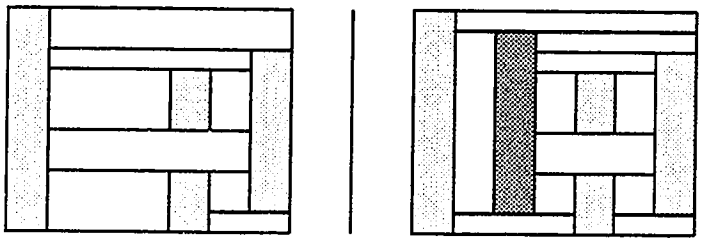


Figure 1-5 Corner Stitching: Tile Addition

A variant of corner stitching combined with shadowing forms the basis for the algorithm described in Chapter 2. The adjacency lists algorithm [Carpenter 87] retains the speed of shadowing and the incremental properties of corner stitching but is tailored to generating design-rule constraints.

1.3. Solving Constraints

CABBAGE [Hsueh 79b] was one of the first systems to generate and then solve constraints. It forces wires to connect to fixed points on endpoints so wire constraints are not needed to keep wires and endpoints fully connected. The spacing constraints form an acyclic graph that is easily solved in linear time to find the critical path across the circuit. The critical path determines the layout's minimum size and the placement of the tiles on the path. CABBAGE uses a scheduling algorithm to place each group of wired-together tiles as low as possible. A group can be placed only after all the groups that could hold it up have been placed -- each group is given a count of the unplaced groups directly below it and when that count is reduced to zero, the group is added to a queue of placeable groups. REST [Mosteller 81] uses the same compaction algorithm but in addition has an elaborate system to allow stick diagrams to be entered using sloppy line segments.

Liao gives an algorithm [Liao 83] to handle the constraint-graph cycles caused by designer-specified maximum spacings. It alternatively performs the scheduling algorithm to satisfy the spacing constraints and a separate, simple pass to satisfy the designer-specified constraints (by moving single groups upwards ignoring the spacing constraints). The algorithm finishes when a simple pass does not move any groups.

The maximum number of passes is bounded by the number of designer-specified constraints in the critical path. More flexible compaction systems have a much larger number of constraint cycles -- caused by the wire constraints needed when wires are allowed to slide on wide endpoints. Not tying wired-together tiles into rigid groups allows smaller layouts to be created. Schiele's compactor [Schiele 83] handles the constraint cycles by repeatedly passing through a list of the constraints to move single tiles up to fix any found constraint violations. Again, the algorithm finishes when a pass does not move anything, but now the maximum number of passes is bounded by the length of the critical path. This gives the algorithm a slow, $O(n^2)$ worst case.

Just compacting a layout to minimum size leaves a range of possible placements for tiles not on critical paths. One way to take advantage of this freedom is to perform wire-length minimization. Shorter wires allow circuits to operate faster and, by bunching things together, can actually allow the creation of smaller layouts. Schiele heuristically minimizes wire length. First, all the lower endpoints (of vertical wires) that are free to move up without moving any other tiles are moved up. Since movements can free other movements, this step loops until no more simple movements are possible. Then a single pass is made through the lower endpoints to move larger sets of tiles. This heuristic not only makes useless simple movements, it also gives up too easily on larger movements. Lakhani uses an improved, event-driven algorithm [Lakhani 87] that continues until all helpful movements have been made.

LAVA [Eichenberger 86] uses a min-cost max-flow network algorithm to minimize each dimension's total wire length. The compaction problem is converted into a linear-programming problem: the constraints become linear inequalities and the wire lengths are embedded in the objective function. The network algorithm first minimizes the wire lengths and then searches through the constraints to find a set of n constraints that enforce a legal layout. This search has an exponential worst case. The algorithm is described in Chapter 3. Tailor's compactor [Marple 88] is similar except it uses a dual network algorithm that first finds a legal layout and then adjusts it to minimize the wire lengths.

Optimal two-dimensional compaction is NP-complete [Sastry 82]. It requires deciding for every pair of neighboring tiles whether it is better for one to be above or to the right of the other. Schlag gives an algorithm [Schlag 83] that starts with just the wire constraints and then recursively adds one possible spacing constraint at a time (from the $O(n^2)$ size set of all possible left-right, up-down constraints). It stops and

backtracks when either a legal layout is created, the current layout becomes larger than the smallest known legal layout, or the current set of constraints becomes infeasible. Given enough time, this simple branch-and-bound search will always find an optimal layout. Watanabe's compactor [Kedem 84] uses a more efficient search over a smaller set of constraints. Instead of starting with infeasible tile positions and adding constraints, it starts with a legal layout and swaps x and y constraints to reduce the critical paths and thus the layout area.

Many approximate two-dimensional algorithms have been devised. Mosteller's 2-D compactor [Mosteller 87] uses Monte Carlo methods to produce curvilinear layouts: wires bend around randomly placed and moved round endpoints. Supercompaction [Wolf 88] is a compromise between one- and two-dimensional compaction. It iteratively recompacts after moving tiles and creating wire jogs to reduce the preferred dimension's critical path. MACS [Crocker 87] uses a more efficient, event-driven approach to incrementally introduce wire jogs along the critical path and to move groups to minimize wire length. ZORRO [Shin 87] does local two-dimensional compaction by allowing lateral movements of elements to try to pack the tiles during repeated compaction steps.

The above algorithms are all batch algorithms: they need to be completely rerun whenever anything is moved or changed. They do not give the designer any control over the interaction between the two dimensions except maybe to specify which dimension has priority and should be compacted first (the compaction in the first dimension interferes with the compaction in the second). An incremental compactor can efficiently propagate changes between the two dimensions. Thus it not only gives designers more control, it also quickly performs multiple one-dimensional steps and makes an ideal basis for two-dimensional compaction. An incremental compactor needs a data structure that stores enough information to allow quick updates. We combine the min-cost max-flow network spanning tree with routines to move weighted subtrees to derive the tree weight algorithm. The algorithm uses a simple batch graph compaction to produce a good estimate of the desired spanning tree. Then a depth-first scan through the tree minimizes the wire length and prepares for incremental changes. This algorithm is described in Chapter 3.

Chapter 2
Generating Constraints

This chapter describes how the set of spacing constraints needed for compaction is generated and incrementally updated. The algorithm presented not only quickly generates the smallest complete set of constraints for a circuit, but also allows one to update the constraints with an effort proportional to the size of the change instead of the size of the circuit when objects are moved, deleted, or inserted. We start with a description of the adjacency lists data structure and its use in the simple single-color case. Then we cover the extensions needed to handle wires, overlapping tiles, and multiple colors. We also give a simple, quick method for initially loading the adjacency lists data structure. We finish with a discussion of the algorithm complexities. The examples in this chapter show generating and updating the set of constraints needed for an x-direction compaction. These constraints are affected by y-direction movements. Using two data structures, one for each dimension, allows the same routines to be used for the symmetric y-x case.

2.1. Adjacency Lists

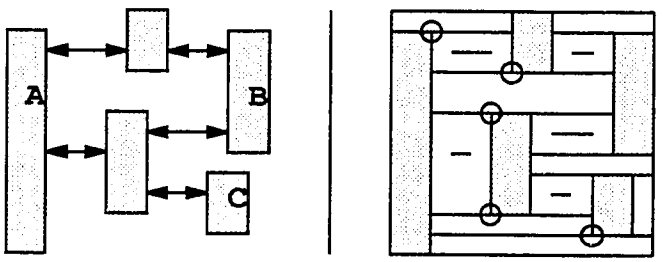


Figure 2-1 A. Adjacencies B. Visibilities

The constraint generation algorithm is based on a strict definition of adjacency. Two tiles are adjacent if and only if moving them in the compaction direction could cause them to hit each other without passing through any other tiles. This is more

restrictive than using visibilities. Because of the intervening tiles in Figure 2-1A, tile A is not adjacent to tiles B and C even though it is visible to them. If we add space tiles to the example (Figure 2-1B), we see that, in the single-color case, the space tiles correspond to visibilities while a subset of the space tiles (marked by dashes) corresponds to our adjacencies. The space tiles form maximal horizontal strips; therefore each space tile's height is determined by neighboring non-space tiles. A space tile's top edge must touch either a non-space tile's bottom edge or one of its top corners (or some combination). Each of a non-space tile's two top corners and one bottom edge can determine the top of at most one space tile. Hence, if there are n non-space tiles there are at most $3n$ space tiles. The space tiles corresponding to adjacencies have both a top and a bottom corner touching non-space tile corners; thus there are at most $2n$ adjacencies. In Figure 2-1B, the space tile corresponding to tile A's top adjacency touches on the top left and bottom right (circled), tile A's bottom adjacency touches on the top and bottom right, and the non-adjacency between tiles A and C touches on only the bottom right.

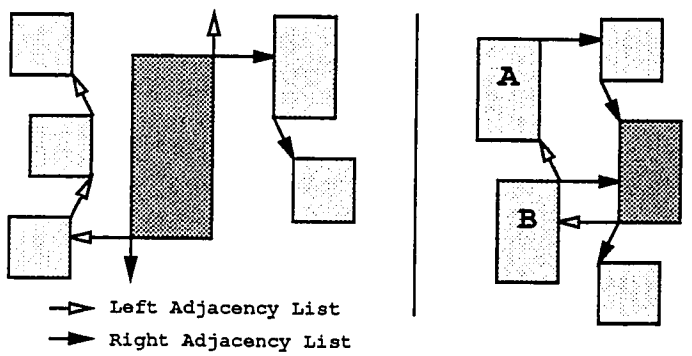


Figure 2-2 A. Adjacency Pointers B. Non-Nil-Ended List

Since there are at most $2n$ planar adjacencies, the adjacency information can be stored using threaded lists. Each tile record has left, right, up, and down pointers. The lists are threaded clockwise; the left pointer points to the lowest left adjacent tile and the up pointers are followed for the rest of the left adjacency list. In Figure 2-2A, the dark tile has three left adjacencies, found by following light arrows. Likewise, the right pointer points to the highest right adjacency and the down pointers are followed for the rest (the dark arrows). Note that these lists are not always nil ended. It is possible for a tile to be the last tile in one tile's adjacency list and the first in another's. The dark tile in Figure 2-2B is the last tile in tile A's right adjacency list and the first in tile B's. The

down pointer of a record is always part of the right adjacency list of the tile pointed to by the record's left pointer. Likewise, the up pointer belongs to the list of the tile pointed to by the right pointer. In the example, the dark tile's left pointer points to tile B so its down pointer continues tile B's, not tile A's, right list.

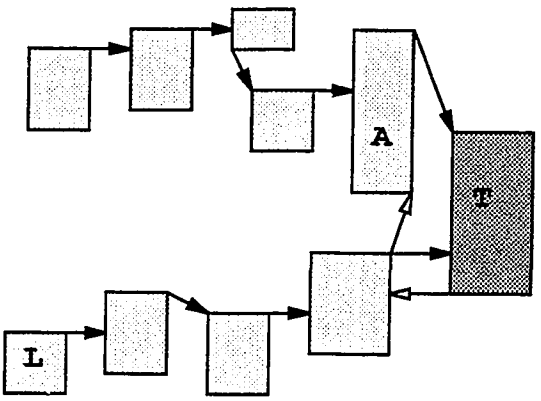


Figure 2-3 Concave Left Turnaround

One common, but non-obvious, operation is to find the tiles above or below a given tile using just the left and right adjacency lists. If tile L in Figure 2-3 were moved upwards we would have to search above it for possible new adjacencies. Since tile L's up pointer is nil we cannot search directly up. Instead we search to the right for a turnaround; then we can search back to the left. Turnaround tile T must be found to travel up or down within the example: the lower half of the tiles have nil up pointers and the upper half have nil down pointers. Right pointers tend to go upwards since they point to top right adjacencies. So, to find a concave left turnaround from below, one just follows right pointers until a non-nil up pointer is found. To find one from above, right and down pointers are followed to find bottom right adjacencies. The turnaround from above occurs when a tile's left pointer does not point back to the previous bottom right tile. In the example, turnaround tile T's left pointer does not point back to tile A.

2.2. Single Color Case

Having defined what adjacency lists are, we now describe the routines used to maintain the adjacency lists for the simple case where there is only one color (type) of tile and the only design rule is that tiles can touch but not overlap. This rule is equivalent to replacing a spacing rule of k with a spacing rule of zero after bloating all the tiles by $k/2$. A frame of special tiles surrounds the layout so that all the normal tiles

will have left and right neighbors. We describe the growing and shrinking of tiles first, as they are the basic operations used to move, delete, and insert tiles. Starting with a simple case makes it easier to explain the basic algorithm; the extensions needed to handle real layouts are described in subsequent sections.

2.2.1. Growing Tiles

Tiles are grown by moving their tops up or bottoms down. The cases are symmetric so we just describe moving tops up. As a tile grows, its path may cut old adjacencies and require them to be deleted. The growing tile may also gain new adjacencies which have to be added to its lists. The main part of the grow routine is a loop that scans counter-clockwise upwards to look for adjacencies that cross the growing tile's path. The scan searches right for a turnaround, goes up once, and then left until a tile is found to the left of the growing tile, that is, we scan around concave left turnarounds (Figure 2-3) to find all the crossing adjacencies, one by one, working upwards until a crossing is found at or above the growing tile's final top.

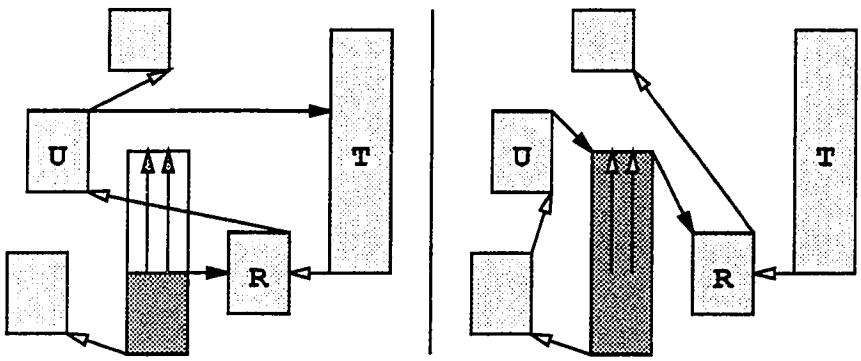


Figure 2-4 Before and After Breaking a Left List

Two cases can occur at the turnaround point: the up tile can be to the left of the growing tile or it can be to the right. If it is to the left, the growing tile has broken into the middle of the turnaround tile's left adjacency list. We remove the up tile from the turnaround's list and add it to the growing tile's left list. In Figure 2-4, up tile U goes from being adjacent to turnaround tile T to being adjacent to the dark, growing tile. Only the relevant pointers are shown. In the other case, the up tile is on right and we have to search for the cut adjacency. Left pointers are followed to find a pair of tiles, one to the right and one to the left of the growing tile's path. If the right tile is the bottom-most tile in the left tile's right list, the right and growing tiles can be made

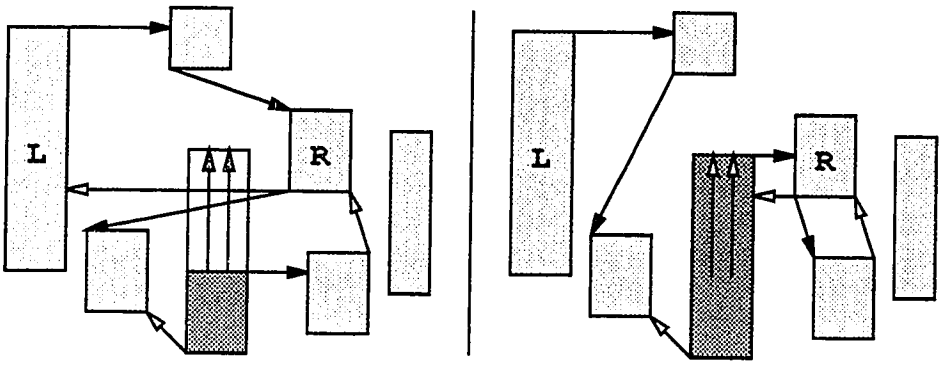


Figure 2-5 Before and After Breaking a Right List

adjacent to each other by simply adding them to each other's adjacency lists. Otherwise, the growing tile has broken into the middle of the left tile's right adjacency list. We remove the right tile from the left tile's right list and add it to the growing tile's right list (tile R changes adjacencies in Figure 2-5).

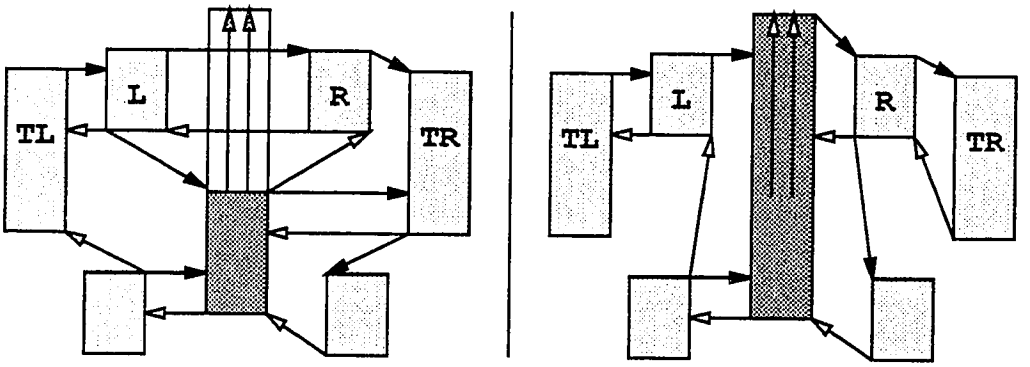


Figure 2-6 Before and After Growing a Tile

Sometimes when a tile is added to the top of a growing tile's adjacency list, that tile shadows the previous top adjacent tile and causes it to be removed from the list. In Figure 2-6, newly adjacent tiles L and R shadow the growing tile's old adjacent tiles TL and TR. This occurs on the right when the growing tile's up pointer is non-nil and on the left when some tile's down pointer points to the growing tile. In the example, tile L points down to the growing tile, which points up to tile R.

After the growing upwards loop is finished, one more adjacency may remain to be found on each side of the grown tile. The grow loop finds all the adjacencies created when the growing path cuts between pairs of tiles, but the grown tile's top corners may also catch on nearby tiles' bottom corners to create new adjacencies. On the right side,

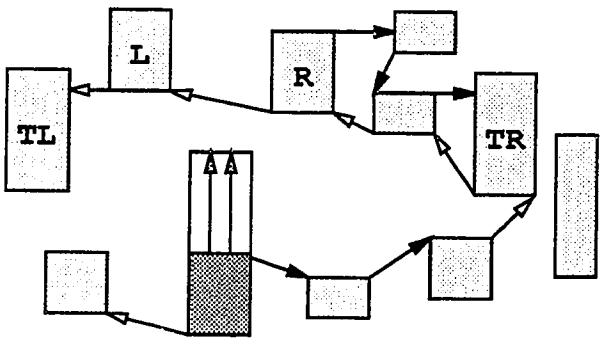


Figure 2-7 Final Search for Top Adjacencies

the top half of the last scanned turnaround is reverse scanned for a tile whose bottom is low enough to be adjacent to the growing tile. A similar search until turnaround is made to the left for a new top left adjacent tile. In Figure 2-7, we find two new adjacencies: the search from tile R finds tile TR and from tile L finds tile TL. If the grown tile were grown up again, further, these final adjacencies would be shadowed (as in the previous paragraph).

2.2.2. Shrinking Tiles

Shrinking a tile is the reverse of growing one. We only describe moving bottoms up since the tops down case is symmetric. As a tile shrinks, it may lose some of its adjacencies and cause new adjacencies to be stitched between tiles on either side of its old position. The shrink routine is a loop that moves the bottom up to drop one tile at a time from the shrinking tile's adjacency lists.

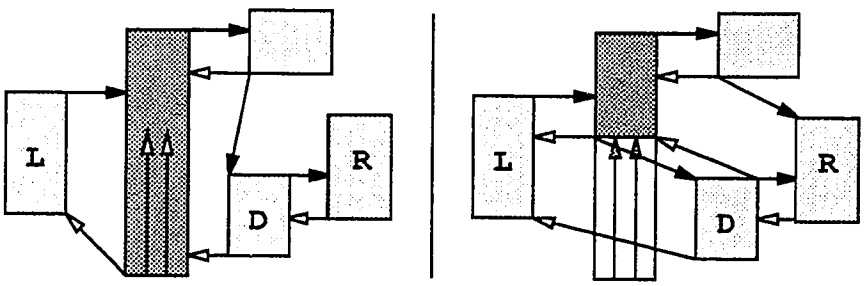


Figure 2-8 Before and After Shrinking a Tile

Two searches are performed as each tile is dropped. When the bottom tile is removed from the shrinking tile's right adjacency list, one search is made to the right of the dropped tile for a tile to replace it in the shrinking tile's right adjacency list and the

other search is made to the left of the shrinking tile for a new top left adjacency for the dropped tile. The search for a new right adjacency follows right pointers until a turnaround is reached or a tile is found whose top is high enough to be adjacent to the shrinking tile's current bottom. The left search likewise follows left pointers looking for a tile with a bottom low enough to be adjacent to the dropped tile. In Figure 2-8, the right search from dropped tile D finds that tile R should now be adjacent to the shrinking tile, while the left search finds that tile L should now be adjacent to tile D. If the shrinking tile shrunk a little more, it would drop its new adjacency with tile R. When a tile is dropped from the shrinking tile's left rather than right adjacency list, symmetric left-up and right-down searches for new adjacencies are used.

2.2.3. Changing Tiles

The grow and shrink routines are used to move, delete, and insert tiles. To move a tile a short distance, we can grow it to make it stretch over its new position and then shrink it to its proper size. To move a tile a longer distance, it is faster to delete the tile and then add it back at its new position to avoid cutting and restitching many adjacencies.

To delete a tile, the tile is first shrunk to zero height so that it will have exactly one left and one right adjacency. The zero height tile is then removed from these tiles' adjacency lists and a simple check decides if the left and right tiles should be adjacent to each other or not. They are made adjacent if the deleted tile was either the only tile in one or both of their adjacency lists, the last tile in both lists, or the first in both lists.

Inserting a tile is the reverse of deleting one. A zero height tile is inserted between two tiles to start its adjacency lists and is then grown (down and up) to its proper size. Locating the two initial adjacencies is a two-step process. Starting at any tile (preferably one nearby), up/right or down/left pointers are followed to find a tile at the same height as the bottom of the new tile. The up pointers move quickly, but if one is nil or jumps over the desired height, the right pointers also tend to go up since they point to top adjacencies. Likewise, the down and left pointers are used to go down. Starting at tile S in Figure 2-9, the height search uses one right and one up pointer.

Once a tile is found at the correct height, the adjacency lists are followed to the left or right to find pairs of tiles at the correct height until a pair is found with one tile on the left and one on the right of the new tile's desired position. The tiles in each pair of tiles

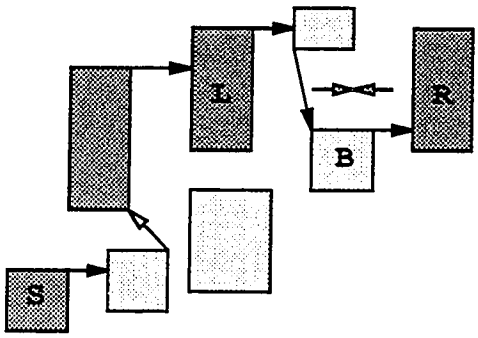


Figure 2-9 Locating Where to Insert a Tile

are visible to each other but need not be adjacent; when none of the tiles in a list are at the wanted height, a search until turnaround is used to go up or down from the listed tile(s) found just below or above the wanted height. In Figure 2-9, the second found pair is the correct one. The final right tile R is found by searching through tile B below the desired height.

2.3. Wires

So far we have discussed one kind of tile, but in layouts there are two very different kinds of tiles: fixed-size objects and wires. In our system, wires connect to fixed-size tiles, not to other wires, with at most one wire connected to each tile's side. An endpoint is made at least as wide as the wire(s) connected to it in order to provide a full connection. When an endpoint is wider than a wire, the wire is free to slide along it. In Figure 2-10, vertical wire V is not rigidly connected to endpoint tile E. Wires could be represented as simple, varying-size tiles but we can take advantage of the fact that their ends are protected -- a horizontal wire never generates any constraints useful in x-direction compactions because it is always completely shadowed by the two endpoints to which it is attached. In Figure 2-10, no tile is needed for horizontal wire H. Thus, while non-wire tiles require two tile records, one for each dimension, we can get by with just one tile record per wire.

If we use two pointers in each tile record, a wire-up and a wire-down pointer, to doubly link wires and endpoints together, we can take advantage of the fact that wires stretch when their endpoints move. Wires themselves never need to be moved. When an endpoint is shrunk away from its wire, the wire grows to follow it. Thus instead of the usual shrink loop, the shrink routine can just see how many of the tiles in the

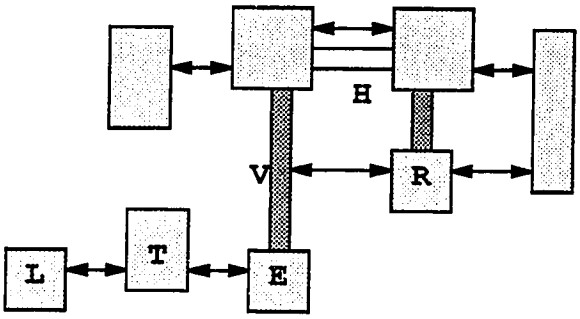


Figure 2-10 Wire Adjacencies

endpoint's adjacency lists need to be moved to the wire's adjacency lists. Endpoints growing towards their wires likewise shrink wires and simply take back some of these adjacencies. Moving endpoint E down and up in Figure 2-10 causes vertical wire V to grow and shrink and gain and lose adjacencies (with tiles T and R), respectively. The wire-up and wire-down pointers can also serve as turnarounds since nothing can pass between a wire and its endpoints. If tile L in the example was grown upwards, the counter-clockwise grow scan could turn around at wire V.

Since a vertical wire is horizontally constrained by its endpoints, it follows, using transitivity, that if a tile is adjacent to one or both of a wire's endpoints, it does not need to also be adjacent to the wire. In Figure 2-10, because of the spacing constraint between tile T and endpoint E and the wire constraint between the left edge of endpoint E and the left edge of its wire V, no spacing constraint is needed between tile T and wire V. Thus we can change the definition of adjacency to say that a tile and a wire are adjacent if and only if the tile is adjacent only to the wire (on that side) and that two wires are never adjacent. In Figure 2-10, tile R's only left adjacency is wire V and the two vertical wires are not adjacent. Note that now, unlike other tiles, a wire's left and right pointers can be nil and its up and down pointers are always nil. Wire V in the example has no left adjacencies.

It is relatively easy to modify the algorithms to handle this definition. A grown tile with more than one adjacency will never be adjacent to a wire, but one extraneous wire adjacency is allowed while a tile is growing so that it can grow past wires. A wire adjacency will be shadowed by any new adjacency or, if necessary, removed after the grow loop. A shrinking tile becomes adjacent to a wire only when nothing else is available; thus a shrinking tile will never lose a wire adjacency. While doing a locate and searching an adjacency list from a wire, if the list is empty or its tiles are all too

high or all too low, the search is continued from the wire's endpoints. To delete a wire, one endpoint is grown towards the other endpoint to zero out the wire. Then the wire is removed and the endpoint is shrunk back. Likewise, to insert a wire, one endpoint is grown towards the other until no adjacencies cross between the endpoints. Then the wire is added between the endpoints and the grown endpoint is shrunk back to its correct size. This causes the proper adjacencies to be transferred from the endpoint to the wire.

In summary, taking advantage of the special properties of wires greatly improves the constraint generation. Using fewer tile records means less memory is required. The wire-up and wire-down pointers, which are needed anyway during compaction to keep wires and endpoints connected, speed up constraint generation and make updates faster. More importantly, since fewer constraints are generated in both dimensions, less time will be required to solve the constraints. The algorithm changes required to achieve these improvements are relatively simple, especially since something has to be done anyway to allow for the varying wire lengths.

2.4. Overlapping Tiles

In VLSI compaction, most of the spacing rules can be relaxed between tiles that are electrically connected since there is no need to keep them from shorting out. Not allowing bloated tiles to overlap causes every wire's two endpoints to be spaced at least their minimum spacing distance apart and thus gives every wire a minimum length equal to the bloat distance. This, for example, would cause wire jogs to always jog at least the spacing distance and would space diffusion contacts far away from transistors. To overcome this problem, we can create netlists specifying which tiles are electrically connected and can thus legally overlap. One simple, but slow, method is to force overlapping tiles to be adjacent to each other. Any wire between two such tiles is unnecessary and is removed and added to a list of negative-length wires. After all the movements are done, this list is checked to see if any of the wires now have positive lengths and should be added back to the graph.

2.4.1. General Case

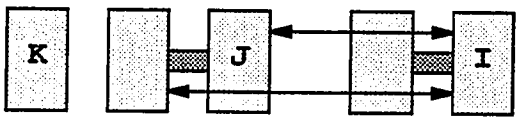


Figure 2-11 Constraints With Pass-Throughs

The most general algorithm would not generate any constraints at all between electrically connected tiles and thus allow such tiles to pass through each other. This flexibility allows a left wire jog to become a right wire jog and vice versa and allows the ordering of taps off of busses to vary. It also makes it much harder to read the constraints from the adjacency lists. To find all the constraints for a tile on net I, we not only have to make a transitive search through adjacent net-I tiles to find the first constraint, say with a tile on net J, we then have to continue the search until we find a tile not on net J. This long search is required because even the furthest away (last scanned) net-J tile could pass through all the intervening net-J tiles and hit the net-I tile after that tile has passed through all the intervening net-I tiles. In Figure 2-11, we have to search all the way from the right-most tile to the left-most tile to find the right tile's two left spacing constraints. While these transitive searches complicate and slow normal shadowing algorithms, they are particularly bad in incremental compaction since constraints may be read from the adjacency lists many times.

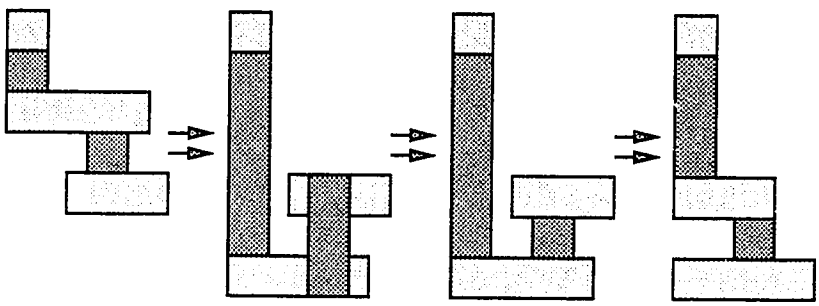


Figure 2-12 Fixing Negative Length Wires

Another problem with letting tiles pass through each other is that negative-length wires are created when one endpoint of a wire passes completely through the other endpoint. Having a wire's upper endpoint below its lower endpoint would confuse the grow and shrink routines. Swapping a negative-length wire's wire-up and wire-down pointers restores its positive length and fixes the problem but, unfortunately, creates a

new problem -- it may try to point an endpoint's wire pointer at two different wires at once. In Figure 2-12, moving the center endpoint down gives the right wire a negative length. Swapping the wire's endpoints causes a problem with the moved endpoint's wire-up pointer. When a pointer conflict occurs, the shorter wire is given preference: the longer wire is disconnected from the endpoint, shrunk, and attached to the other endpoint of the shorter wire. This recurses until a free endpoint is found. A wire or endpoint may have to be moved horizontally slightly to make a full connection when a vertical wire's endpoints are changed. In Figure 2-12, the left, longer wire is shrunk and connected to the new center endpoint, which is moved left slightly to allow a full connection.

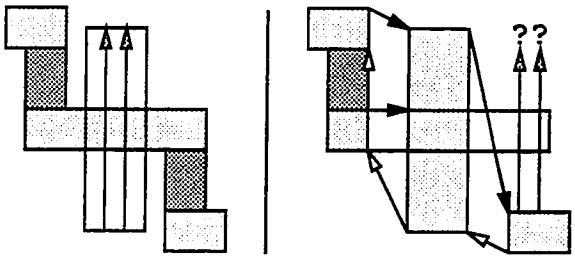


Figure 2-13 A. Insert on Left or Right? B. Cannot Insert Wire

Besides requiring left-right pointer updates, allowing tiles to overlap and cross left-to-right also causes a problem with the function used to decide whether a tile is to the left or right of another tile. Because we use left and right adjacency lists even when two tiles overlap, one tile has to be declared the left so they can be made adjacent. Just comparing left coordinates no longer gives consistent results near wires. When a wire's endpoints are wider than the wire, their left edges do not have to line up. A tile's left edge could be to the left of one endpoint's left edge and to the right of the other's. In Figure 2-13A, it is not clear whether the new tile, which overlaps the center endpoint, should be added on the left or right of the overlapped endpoint since a comparison with the bottom endpoint implies it should be on the left but a comparison with the top endpoint implies it should be on the right. Similarly, if a new wire's desired endpoints are on the opposite sides of a tile, one of them needs to be moved before the wire can be inserted. In Figure 2-13B, the search from the bottom endpoint, on the right, will not be able to find the top endpoint, on the left.

2.4.2. Restricted Case

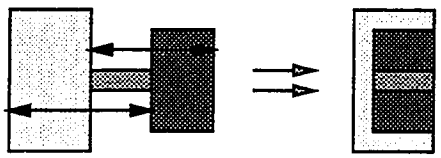


Figure 2-14 Constraints Without Pass-Throughs

In the previous section we saw that allowing tiles to pass through each other causes several problems. With a slight loss in generality we can greatly reduce these difficulties. We allow tiles to overlap but not pass through other tiles; we force the right tile's right edge to be at or to the right of the left tile's right edge and likewise for their left edges. Thus, when two tiles are electrically connected, instead of generating a spacing constraint between the right edge of the left tile and the left edge of the right tile, we generate two zero valued constraints: one between their left edges and one between their right edges (see Figure 2-14). Not allowing left edges to cross left edges, and right over right, fixes many of the pass-through problems: no adjacency searches are needed to determine the constraints, there are no negative-length wire conflicts, and movements never swap left and right adjacencies.

The problem with determining left-rightness is also eliminated except when edges exactly overlap. A simple way to solve this is to define two tiles to be electrically connected if and only if there is a wire directly between them. Then, when two tiles exactly overlap, we can check the wire pointers to determine which is the left and which is the right endpoint. If, when moving several tiles, we always grow the left-most endpoint up first, right endpoints will always grow to the right of any tile they exactly overlap. This allows the grow routine to always break ties in the same direction. To keep from producing temporarily negative-length wires, we break vertical ties so that top endpoints always move up before bottom endpoints. We move right and bottom endpoints down first for the same reasons. Using the direct-connect definition of electrically connected also saves us the trouble of incrementally updating the netlists when tiles are inserted or deleted.

Left and right overlaps in one dimension appear as up and down overlaps in the other. A pointer conflict can occur when top-to-bottom overlapped tiles need to share some of their adjacencies. For example, if three overlapping tiles, on the left, are adjacent to three overlapping tiles, on the right, we may have to generate adjacencies

between each of the left tiles and each of the right tiles -- nine adjacencies in all. The planar adjacency list pointers cannot directly handle so many adjacencies. To resolve this problem, we semi-merge top-to-bottom overlapped tiles -- the semi-merged tiles share one common pair of adjacency lists but still move independently. For more details see Appendix A. Semi-merging makes it slightly harder to read the constraints from the adjacency lists and it can create a few over-restrictive adjacencies. The method works fairly well but our implementation depends on the simple direct-connect definition of electrically connected.

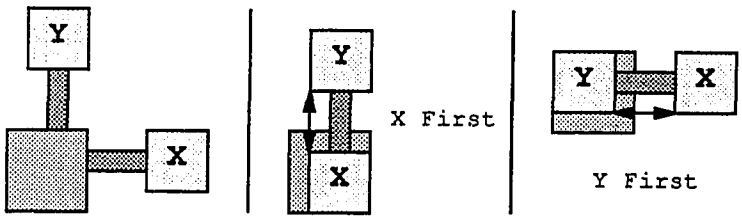


Figure 2-15 Compaction-Order Effects on Overlap

Using the direct-connect definition makes it fairly easy to overlap and merge tiles but, unfortunately, it does not allow all legal overlaps. The most noticeable case is that the three endpoints in an L-shape wire cannot all overlap. The two end endpoints are not directly connected by a wire and thus are not allowed to overlap. Which endpoint will overlap the middle endpoint depends on which dimension is compacted first; the second dimension will see a spacing constraint between the two end endpoints. In Figure 2-15, a *x*-then-*y* compaction reduces the L-shape wire to a vertical wire, while a *y*-then-*x* compaction reduces it to a horizontal wire. While many L-shape wires can be eliminated by changing the stick diagram, doing so is awkward and prevents some useful constructs. The only practical way to fix this problem is to use full netlists and modify the adjacency routines to allow locally unrelated tiles to overlap.

2.5. Multiple Colors

So far we have had just one color (type) of tile and one spacing rule. A real layout needs many types of tiles and many spacing rules. We separate the colors that do not interact into independent planes; each plane consists of a set of tiles connected together by adjacency lists. For nMOS there is a metal plane and a poly-diffusion plane. In a plane with two or more colors, it is unlikely that using constant bloats for each color will satisfy all the spacing rules. Therefore the tiles are stored unbloated and when

comparing two tiles, their colors are used as indices into an array of spacing rules to find the appropriate bloat. Rules with very large spacing distances (p- to n-diffusion in CMOS can bleed across poly) would force us to search through adjacencies to find all the spacing constraints. To prevent these searches, we restrict the rules handled in each plane so that whenever tile L is adjacent to tile M and tile M is adjacent to tile R, no spacing constraint is needed between tiles L and R. Figure 2-16A shows the three-color rule that must apply for all possible color combinations within each plane: the sum of spacing constraints A and C, from tiles L to M and M to R, and the minimum legal size B of tile M must be greater than or equal to any spacing constraint D from tiles L to R. For CMOS, a well plane (with well boundary tiles and duplicated diffusion tiles) must be created to handle the p- to n-diffusion rules.

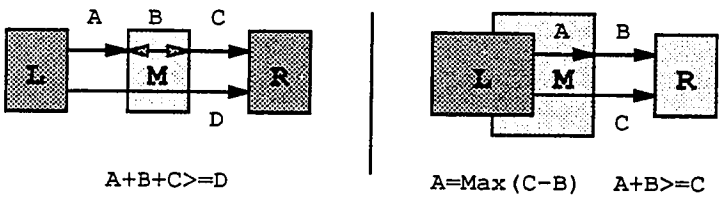


Figure 2-16 A. Spacing and B. Overlap Bleed-Throughs

A similar rule bleed-through problem occurs when tiles overlap. When all the spacing rules were identical, not allowing the right edges of a left tile and an overlapped middle tile to cross guaranteed that no spacing constraint was needed between the left tile and any possible tile to the right of the middle tile. To prevent multiple colors from requiring us to search through overlapped tiles, we increase the constraint between overlapped tiles' right (and left) edges from the previous zero value to a small constant. For each possible pair of overlapping tile colors, we calculate and store in an array the maximum difference between the spacing constraints from the left and the overlapped middle tile to any possible color right tile. In Figure 2-16B, constraint A, the minimum allowable right to right edge distance for tile colors L and M, is calculated using constraint B, from tiles M to R, to make constraint C, from tiles L to R, unnecessary regardless of tile R's color.

2.5.1. Fuzzy Edges

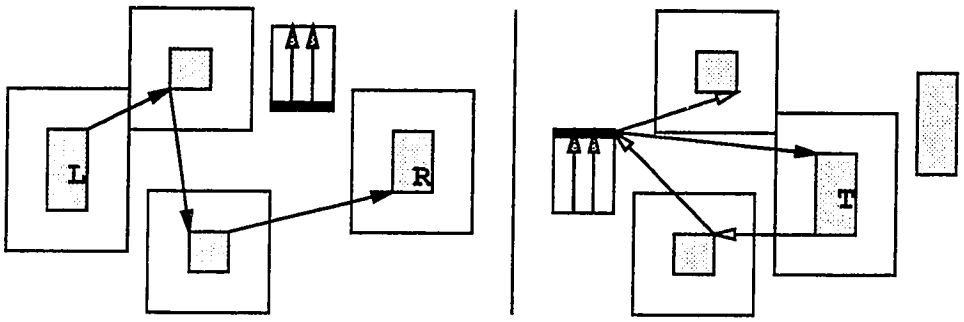


Figure 2-17 A. Locate Off By Bloat B. Shrink Slip Through

The apparent positions of tile tops and bottoms are now fuzzy since the effective bloat of a tile varies with the color of the tile with which it is being compared. This fuzziness causes two problems. The first is minor and occurs when a tile is smaller than the minimum legal size (for that color tile). This can only happen during a tile insertion or deletion. For insertions, the locate to find a pair of tiles to start a new tile's adjacency lists might not be able to find two unbloated tiles at exactly the desired height. In Figure 2-17A, the locate (starting at tile L) returns tiles L and R even though the new tile might not be adjacent to them. After the new tile is inserted and grown to its proper size, the adjacency between it and right tile R may have to be removed (depending on their proper bloat). For deletions, if the shrinking tile in Figure 2-17B dropped its adjacency with tile T, it would be left with no legal right adjacency. The shrinking tile is in the middle of T's left adjacency list; letting it slip over tile T would destroy T's list. Since the shrinking tile is about to be removed, we can just make the shrink loop stop early and leave the shrinking tile adjacent to tile T. If it was not for the three-color rule (Figure 2-16A), normal small tiles could also slip diagonally between adjacencies and cause many problems.

The second fuzzy-edge problem, the around-the-corner problem, is more serious. The adjacency list data structure can only model planar graphs: that is what makes it possible to have just one up and one down pointer per record. But the fuzzy edges make it possible for constraints to diagonally cross. A simple case is shown in Figure 2-18A where four tiles are arranged in a tight square with a diffusion tile above a poly tile on the left and below one on the right. The tiles can be close enough to generate four pairs of constraints since the poly-diffusion spacing rule is smaller than the poly-

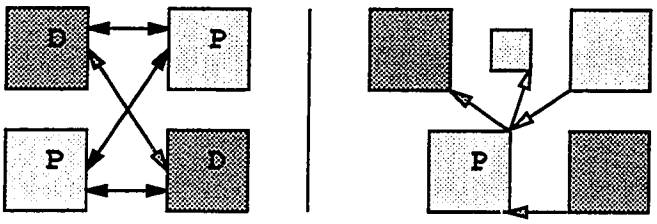


Figure 2-18 A. Crossed Constraints B. Up Pointer Conflict

poly and diffusion-diffusion spacing rules. Adding another tile in the middle of all this can cause an up or down pointer to need to point to two tiles at once. Tile P in Figure 2-18B is in the middle of two left adjacency lists.

Searches for around-the-corner adjacencies are needed after growing and while shrinking tiles to avoid missing any constraints. These constraints are kept in a separate list since they cannot be included in the normal adjacency lists. In a batch compactor, this list is checked when the compaction constraints are generated to see if the constraints are still valid and, if so, if they can now be added to the normal lists. Missing constraints can only occur when tile tops and bottoms are very close; thus a simple test can eliminate or cut short most of the around-the-corner searches. The maximum vertical extent of a search is determined using a precalculated function of the colors' bloat distances. This fudge distance is how far the top of a middle tile must be above the top of a left tile before any right tile high enough to be able to slide horizontally over the middle tile will also be able to slide over the left tile (the middle tile completely shadows the left). Note that this distance is the same as the one used to keep constraints from bleeding through overlapped tiles (Figure 2-16B rotated ninety degrees). The next two sections describe how to modify the grow and shrink routines to handle multiple colors.

2.5.2. Growing Tiles

After we grow a tile, we have to search for a missing adjacency on its left and on its right. In the section on growing tiles upwards we saw that after the growing loop was finished we sometimes had to search left and right-down until turnaround from tiles just above the grown tile to look for possible final adjacencies. The around-the-corner search is the continuation of this search pretending that the first found turnaround tile was removed. On the right side, the turnaround was the last tile in some tile's right

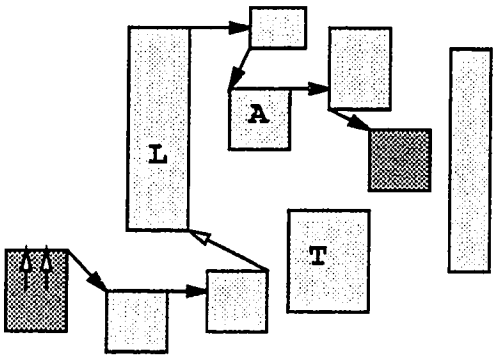


Figure 2-19 A Complex Around-the-Corner Search

adjacency list (tile T in tile L's list in Figure 2-19). We find the second to last tile in that list, if there is one (tile A in the example), and continue searching right-down from it. If we find a tile with a bottom low enough to be adjacent to the grown tile, we have found an adjacency that could have been added if the turnaround tile was not there. In Figure 2-19, the search finds a missing constraint between the two dark tiles. To start a search, right pointers are followed from the grown tile to find the first turnaround. If any of these tiles have tops more than the fudge distance above the grown tile, the grown tile will be shadowed from any problems on this side and we can abort the search. In Figure 2-19, the two tiles scanned before reaching tile T are not high enough to stop the search.

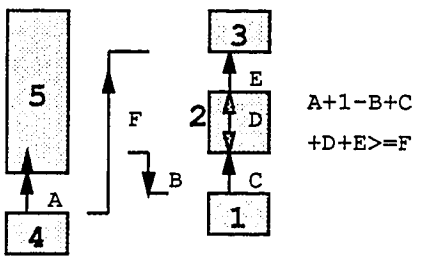


Figure 2-20 Five-Color Rule

Previously we noted that the spacing and minimum size rules in each plane had to obey a function of three tile colors (Figure 2-16A) to prevent bleed-throughs. It turns out that this three-color rule is not quite strict enough to allow the around-the-corner searches to pretend to remove only one turnaround tile. If tiles 3 and 4 in Figure 2-20 could be adjacent, the search would have to not only examine tile 2, tile 5's second from bottom right adjacency, but also tile 3, the third from bottom adjacency. To prevent

this, we make sure tiles 3 and 4 cannot be adjacent: A+1-B+C+D+E>=F must be valid for all possible colorings of five tiles within a plane. All the values are spacing distances except D the minimum size for tile 2. The 1-B occurs because tiles 1 and 5 must be adjacent and B is the spacing rule between those two tiles. This five-color rule does not cause any problem with the CMOS rules as long as the spacing rules are properly divided between the four planes (metal-2, metal-1, active, and well) already required to satisfy the three-color rule.

2.5.3. Shrinking Tiles

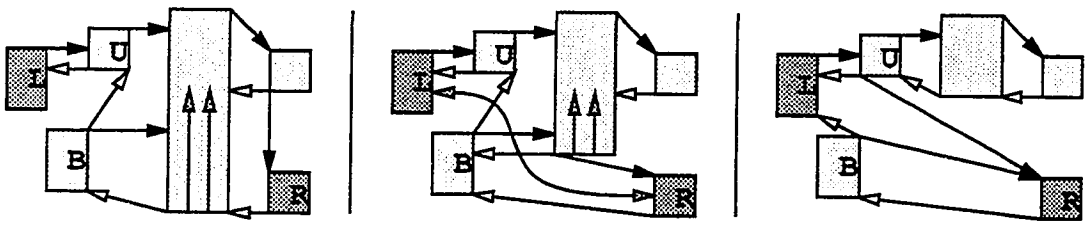


Figure 2-21 Temporarily Cached Missing Constraint

Two major changes have to be made to the shrink routine to handle the around-the-corner problem. First, an extra search is needed after each tile is dropped from a shrinking tile's adjacency lists. When the bottom of a shrinking tile moves up just enough to drop an adjacency on one side (R), it is quite possible that an around-the-corner constraint between the dropped tile and a tile on the other side (L) of the shrinking tile is created. As the shrinking tile continues to shrink and drops the bottom adjacency on this other side (B), it is likely that the constraint can be added directly to the adjacency lists. Therefore we search (left) from the shrinking tile's second from bottom adjacency and cache the missing adjacency, if one, in the hope that it can be quickly resolved. In the first step of Figure 2-21, dropping right dark tile R creates an around-the-corner constraint between it and dark tile L (found by searching from tile U). Dropping the shrinking tile's bottom left adjacency B in the second step allows this missing constraint to be added to the adjacency lists. After the shrinking tile is finished shrinking, if any adjacencies have been dropped from a side, we also have to search for possible missing constraints with the shrunk tile.

The second change to the shrink routine occurs because of the fuzziness of tile tops and bottoms -- the shrinking tile's bottom does not move monotonically upwards. The top of a tile on one side (R) of the shrinking tile may be low enough to not be adjacent

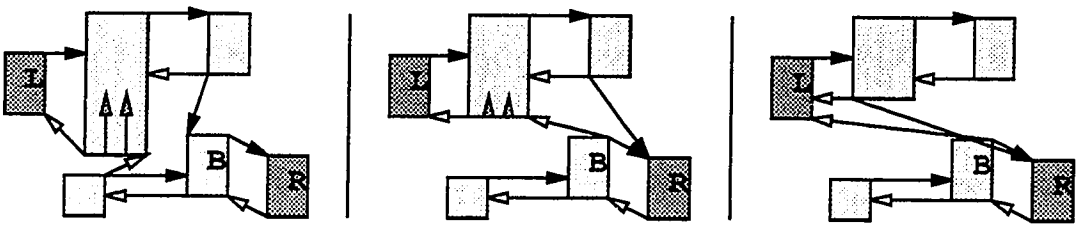


Figure 2-22 Temporary Extra Adjacency

to the shrinking tile's current bottom but still high enough, because of the varying bloats, to be adjacent to a tile on the other side (L) of the shrinking tile. If we just skip the first tile (R), we will miss this unshadowed adjacency (L-R). A simple fix is to not skip any tiles: always make the shrinking tile adjacent to each dropped adjacency's first adjacency. This, in effect, causes the shrinking tile's bottom to sometimes move back down slightly. As each of the adjacent tiles is dropped, its proper unshadowed adjacency, if one, will be found. In the first step of Figure 2-22, we drop the shrinking tile's bottom right adjacency B and make right dark tile R adjacent to the shrinking tile even though it could never possibly hit it. When adjacency R is dropped in the second step, the search to the left of the shrinking tile finds left dark tile L and we make tiles L and R adjacent.

We can do better than the simple, no-skip fix -- many of the tiles can safely be skipped. A dropped adjacent tile will completely shadow from the shrinking tile all the tiles with tops a certain distance below its own. This distance is like the fudge distance except we take the minimum over all possible colors instead of the maximum. Any tile with a top at or below this fudge position can be skipped. This position will monotonically increase during the shrinking loop. Actually, a dropped tile will shadow tiles at various heights depending on their colors. We could store in a shadow array the height below which each different color tile could safely be skipped, but the cost of updating all the entries after every drop is not worth the slight gain from skipping a few more tiles.

A more effective improvement is to store in the shadow array the position of the last tile dropped of each different color. A tile always shadows any tile of the same color with a top at or below its own top. Thus we can skip a tile if its top is at or below the fudge position or if it has been shadowed by a tile of its own color. Except for initializing the array before starting each shrink, this has very little overhead and insures that at most one tile of each color at each height will be checked. Figure 2-23 gives an

example. First, tile A is dropped and the fudge position F is set to slightly below A's top. Tile B's top is above F so it is made adjacent to the shrinking tile and then dropped. Tile B's fudge position is below F so the current fudge position is not changed. Tile C's top is above F but since it shadowed by same color tile B we can skip it anyway. Tile D is likewise shadowed by same color tile A. Tile E's top is above F and above same color tile A's top so it cannot be skipped. Dropping tile E moves the fudge position up slightly.

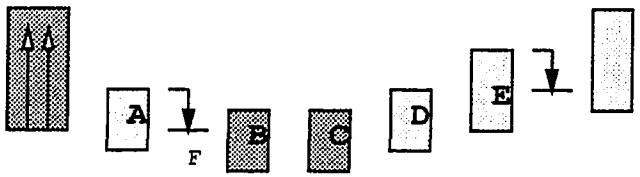


Figure 2-23 Shrink Shadow: Skip Tiles C and D

2.5.4. Summary

Duplicating some of the tiles to create an extra plane to handle large, non-transitive spacing rules is a good compromise between handling all the bloat problems in one plane and eliminating the problems by creating many extra planes (each with a constant bloat). The five-color rule determines which rules can be handled in each plane. Within a plane, the varying bloats cause several problems. Around-the-corner searches are required after growing and while shrinking tiles to keep from missing crossed constraints that cannot be included in the planar adjacency lists. A precalculated function of the bloat distances is used to cut short most of these searches. A constraint cache and a shadow array are used in the shrink routine to help add missing constraints to the adjacency lists and to avoid processing more tiles than is necessary.

2.6. Quick Loading

The adjacency lists data structure could be loaded, given an initial placement, by just inserting every tile. This would use a lot of time just locating where to insert each tile. Sorting the tiles lexicographically on the (y,x) coordinates of their lower left hand corners before inserting them makes the locates fast but creates a long horizontal frontier between the added tiles and the top edge of the frame that is often searched as tiles are grown to their proper sizes. It is better to sort the tiles on (x,y). As they are

added in this order, each tile will have nothing to its right except the right edge of the frame. Thus a simple, quick load routine can be used. The locate reduces to finding the tile in the right edge's left adjacency list just below the new tile's bottom. The growing loop reduces to replacing some of the tiles in that left adjacency list with the new tile and making those tiles the new tile's left adjacencies. We still need to scan until turnaround to look for final top and bottom left adjacencies.

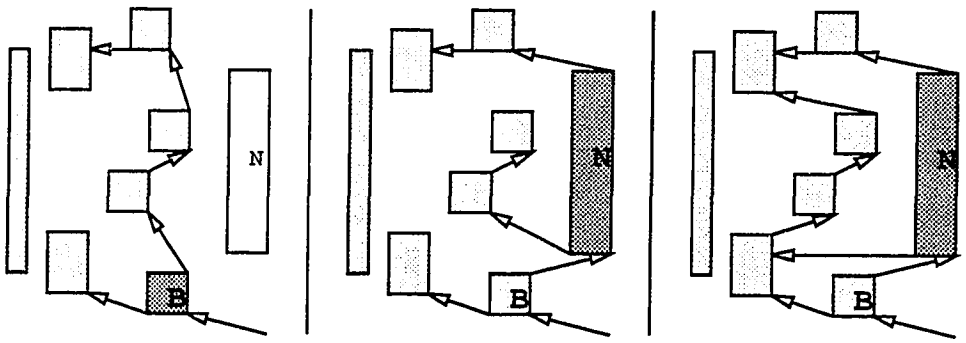


Figure 2-24 Quick Load Steps: Locate, Swap, Search

Figure 2-24 shows the steps of locating dark tile B below new tile N, swapping two tiles out of the right edge's left list, and the final searches from above and below the new tile, both of which find an adjacency in this case. Note that this load routine is actually a fast, simple shadowing routine. The tiles in the right edge's left adjacency list correspond to the shadowing frontier's completely unshadowed tiles and the partially shadowed tiles are found left or left-up from the tiles which shadow them.

Wires are easily loaded. No search for where to add a wire is needed since it begins and ends at its endpoints; we just have to ensure that its endpoints are added first. In the sorted load order, wires are always after their lower endpoints but are often before their upper. If we come to a wire before its upper endpoint has been added, we adjust the list so that the wire will instead be added just after the endpoint. When a wire is added, its left adjacency list will consist of the tiles, if any, between its endpoints in the right edge's left adjacency list. No final searches are needed since wires have no corners to catch on top and bottom adjacencies.

Multiple colors are easily handled: we just need to add leftward searches from the top and bottom of each non-wire tile to look for possible around-the-corner constraints. Overlapping and semi-merged tiles cause more problems: wires and left and right endpoints need to be added in the proper order, endpoints may be partially shadowed

before their wires are added, and the tiles that should be semi-merged are not added in any particular order. See Appendix A for more details. The quick load works very well. Even with all the modifications it is not too complicated. A simple batch compactor could use the load and not even implement the grow and shrink routines. Since the quick load only has to worry about left adjacencies, it should be at least twice as fast as separately inserting and growing each tile.

2.7. Complexities

In this section we examine the run-time complexities for the quick load and for incremental changes. We use n to represent the number of tiles in one dimension's data structure. Note that objects such as transistors and contacts consist of several tiles and that each wire is represented by a tile in only one of the two dimensions. We assume the area of a layout is $O(n)$, that is, the density and size of tiles is independent of n. This assumption is slightly optimistic since larger layouts are more likely to have long wires and maybe even larger driving transistors and more dead space. We also assume that layouts are roughly square, that is, the aspect ratio is independent of n. Together, these two assumptions give us layouts that are $O(n^{.5})$ lambda on a side. Finally, we assume that the number of overlapping tiles semi-merged into each set is small and independent of n. Thus we can ignore layouts, for example, which consist of a few large stacks of tiles and produce a huge number of set to set spacing constraints. The number of different tile colors and spacing rules is determined by the technology and thus is independent of n.

The quick load sorts the tiles on the (x,y) position of their lower left hand corners. Since tile positions are rather dense over a small range, a linear-time radix sort is very efficient. For example, using 256 bins allows sorting, starting with the least significant byte, with just a few passes through the tiles. We assume that the feasible tile positions used to load the data structure produce an initial layout $O(n^{.5})$ on a side. Loading each of the n tiles requires three steps: locate, swap, and final search. The locate of the tile in the right edge's left adjacency list just below each new tile starts from the tile found in the previous locate. Thus the locates for all the tiles with a given x-coordinate (and increasing y-coordinate) are done with a single pass through the right edge's left adjacency list. For a roughly square layout this uses $x = O(n^{.5})$ passes through a $y = O(n^{.5})$ length list for a total $O(xy) = O(n)$ time. Each tile can be added to and removed from the right edge's left adjacency list only once so the total time swapping left

adjacencies is also O(n).

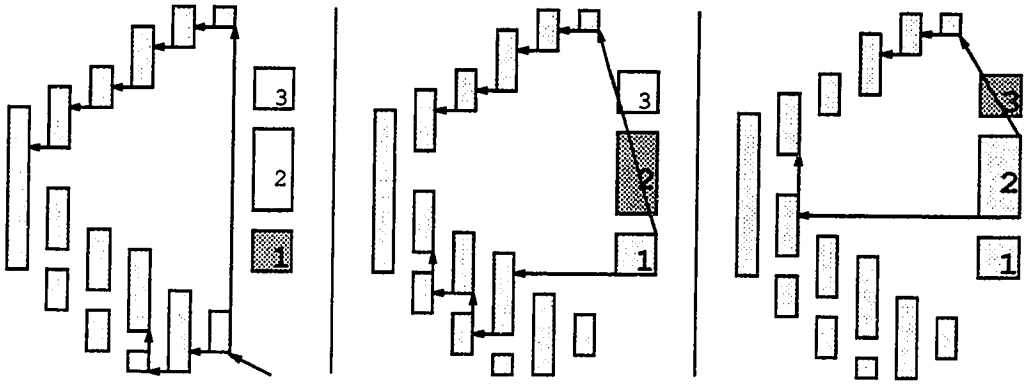


Figure 2-25 Final Searches for Three Tiles

The final searches are trickier. An above search follows left pointers, which point to bottom left adjacencies, until an adjacency or turnaround is found. In a roughly square layout each tile has at most $x = O(n^{.5})$ tiles directly to its left so even if all n above searches reach the left edge, we get a worst case $O(nx) = O(n^{1.5})$ total. A below search follows left and up pointers to scan along top left adjacencies. Since the tiles with a given x-coordinate are added in a bottom to top order, each below search at worst continues where the previous search finished. Thus all the searches for a column of tiles will visit each previously added tile at most once. For a roughly square layout this gives $x = O(n^{.5})$ passes through at most $O(n)$ tiles for a worst case $O(xn) = O(n^{1.5})$ total. Figure 2-25 shows the searches for three tile additions. The first two below searches and the last two above searches find adjacencies. The size of the empty spaces searched around should normally be relatively independent of n so the final searches should have a basically linear expected time.

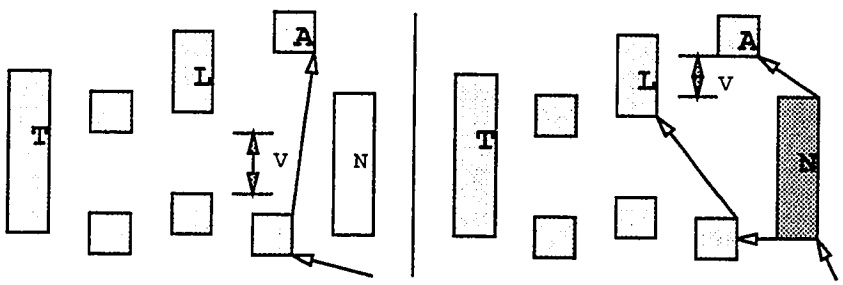


Figure 2-26 Visible Turnaround Range

We can reduce the number of final searches performed in the single-color case. A turnaround tile is located in the gap between each pair of tiles in the right edge's left adjacency list. We keep track of the unshadowed part of each of these turnaround tiles -- the still visible range. A final search is needed if and only if the top of a new tile is above the visible range or the bottom is below it. In Figure 2-26, new tile N's top is below tile A's bottom but above turnaround tile T's visible range V. Thus a left search is needed from tile A to find the adjacency with tile L. The overhead of storing these ranges is probably not worth the normally slight gain from skipping a few searches. For multiple colors it is definitely not worth the trouble of storing and constantly updating the visible range of each turnaround for every possible tile color. However, if we were only worried about theoretical worst case times, keeping track of the turnaround visibilities would be worth-while. Every search would then find an adjacency and in effect shadow an area proportional to the number of left pointers followed. Doubly linking the adjacency lists would provide direct top left adjacency pointers and thus eliminate the below search's need to follow up pointers. Together this would give the final searches an O(area) worst case time (even without any of our assumptions).

None of the model extensions affect the load time complexity. Adding a wire tile is faster than adding a non-wire tile. Handling the semi-merge order problem complicates the code and can produce $O(m^2)$ effects to merge a set of m tiles, but because the m's are small they do not affect the time complexity. The around-the-corner searches are so restricted that their worst case time is no more than the final search time. Thus the quick load has a basically linear expected time and an $O(n+n+n^{1.5}) = O(n^{1.5})$ worst case. Keeping track of turnaround visibilities and doubly linking the lists would reduce the worst case to O(area) but give a slower expected time.

An incremental move has an expected time roughly proportional to the number of changed adjacencies which is roughly proportional to the distance the tile is moved. The worst case for growing a tile is the sum of two effects. For a tile reaching across a layout, each of $y = O(n^{.5})$ counter-clockwise scans to find new adjacencies may scan through $x = O(n^{.5})$ tiles. And each of these y found adjacent tiles may have to be removed from the end of an initially $y = O(n^{.5})$ length adjacency list. Thus the worst case to grow a tile is $O(y(x+y)) = O(n)$. In Figure 2-27, the growing loop scans through the entire array of nine tiles and moves tiles 1, 2, and 3, in that order, from the end of tile L's right adjacency list to the beginning of the growing tile's right list.

Shrinking a tile causes searches similar to the load's final searches. The right-

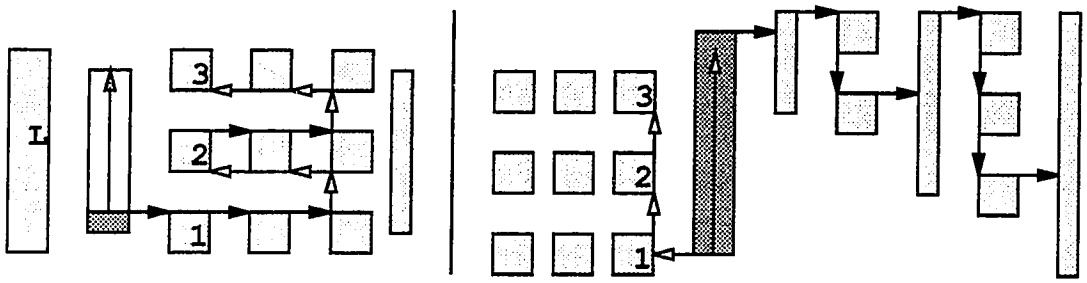


Figure 2-27 Grow and Shrink Worst Cases

down search after dropping each of the at most $l = O(n)$ tiles on the left of a shrinking up tile can visit the at most $r = O(n)$ tiles on its right for an $O(lr) = O(n^2)$ worst case. Using the shrink shadow array (as in Figure 2-23) has little effect on the expected time but reduces the number of tiles dropped to at most $y = O(n^{.5})$ for an $O(yr) = O(n^{1.5})$ worst case. Doubly linking the adjacency lists requires a lot of space but would reduce the number of right tiles searched after each drop to at most $x = O(n^{.5})$ for an $O(yx) = O(n)$ worst case. Since neither effect is very important in the expected case, we use a shadow array but not double links. In Figure 2-27, the shrinking tile drops three left adjacencies: 1, 2, and then 3. As each is dropped, most of the right tiles are searched through for a possible new right adjacency for the dropped tile. Without the shadow array, all nine left tiles would cause right searches. The doubly-linked lists' bottom right adjacency pointers would have allowed the searches to scan only the bottom-most right tiles.

2.8. Summary

In this chapter we described how to use adjacency lists to generate and incrementally update the set of spacing constraints needed for compaction. The adjacencies correspond one-to-one with the at most $2n$ needed spacing constraints and thus can be efficiently stored using clockwise threaded left and right lists. These lists allow us to quickly grow and shrink tiles to move, delete, or insert them. Although the grow and shrink routines have to handle several special cases, each test and action is fairly simple. The other single-color routines are also relatively straight forward.

Simple modifications allow the algorithm to take advantage of the special properties of wires: each wire only needs to be represented in one of the two dimensions, wires stretch to grow and shrink as their endpoints are moved, and tiles

cannot pass between a wire and its endpoints. More importantly, since vertical wires are horizontally constrained by their endpoints, wires generate fewer spacing constraints than non-wire tiles: a tile adjacent to a wire and its endpoint only needs the adjacency with the endpoint.

Handling overlapping tiles is more difficult. The general case of allowing all tiles on the same netlist to pass through each other causes many problems: it makes it hard to read the spacing constraints from the adjacency lists and requires extra updates when tiles do pass through each other. To ease these difficulties, we allow tiles to overlap but not pass through each other. We also semi-merge tiles that overlap top-to-bottom to keep from producing constraints that cannot be including in the planar adjacency lists. To make it easier to merge/unmerge sets and to decide on the left-right and bottom-top order of exactly overlapping tiles, we define two tiles to be electrically connected if and only if there is a wire directly between them. This over-simplifying definition is not very good since it can prevent many legal tile overlaps.

The final enhancement to the algorithm allows it to handle multiple colors. Multiple planes handle independent layers and the five-color rule determines which spacing rules can be handled in each plane. The effective bloat of a tile depends on the color of the tile with which it is being compared. This causes several problems: the worst being that crossed constraints may be generated that cannot be included in the planar adjacency lists. Special around-the-corner searches have to be added after growing and while shrinking tiles to find these constraints. Most of these searches are short since the problem can only occur when tile tops and bottoms are very close. The shrinking case is particularly bad since multiple colors can cause the shrink routine to process many tiles that it would otherwise just skip. A constraint cache and a shadow array help mitigate the shrink problems.

We showed that the adjacency lists can be quick loaded by first sorting the tiles on (x,y). As the tiles are added in this order, each tile will have nothing on its right except the right edge of the frame. Most of the left adjacencies are taken from that edge's left adjacency list. Searches are needed for possible final top and bottom adjacencies and for around-the-corner constraints. The quick load has a basically linear expected time and, because of the final searches, an $O(n^{1.5})$ worst case. This compares to growing or shrinking one tile in an expected time roughly proportional to the distance moved and a worst case $O(n)$ grow or $O(n^{1.5})$ shrink. If we were only worried about worst case times we could reduce the $n^{1.5}$'s to n's by preventing unnecessary searches.

Chapter 3
Solving Constraints

In the previous chapter we showed how to generate and incrementally update the set of spacing constraints needed for compaction. In this chapter we examine how to use these constraints to actually perform incremental compactions. First we give efficient algorithms for simple, non-wire-length compaction. Then we describe a weight analogy for wire-length minimization and use it to derive four wire-length minimization algorithms. Each algorithm sets a different three of four necessary solution conditions true and then corrects the final one. We look at the algorithm complexities, their correspondence to min-cost max-flow network algorithms, and some practical improvements. Two basic operations are required to perform incremental changes: the first improves a layout by taking advantage of the empty space created when tiles are moved or deleted and the second creates enough room to legally insert new tiles. One of the wire-length minimization algorithms, the tree weight algorithm, maintains enough information in a weighted spanning tree to efficiently perform these operations.

3.1. Simple Compaction

Before we delve into the intricacies of wire-length minimization, we look at how to make incremental changes under the simpler group and graph compaction models. To introduce the models, we first describe suitable, efficient batch compaction algorithms. Each compaction step in these one-dimensional compaction algorithms determines a dimension's minimum size by finding a critical, longest path across the circuit. The examples in this section show compactions trying to move tiles down as far as possible.

The group compaction model forces wires to connect to fixed points on endpoints; each group of wired-together tiles must be placed and moved as a unit. The spacing constraints for this model form a directed acyclic graph. Batch compaction is done with a linear-time scheduling algorithm: each group is placed as low as possible in a

topological-sort order. To ensure proper placement we need an order where each group is placed after all the groups that could hold it up have been placed. One method is to mark all the groups unplaced except for the bottom edge (at zero) and then use a depth-first scan down from the top edge in which we recursively place each group after we place all of the groups directly below it [Aho 83]. Another method is to calculate for each group the number of unplaced groups directly below it [Knuth 73]. Starting with the bottom edge, when we place a group we decrement the count in all of the groups directly above it and add those with zero counts to a queue of placeable groups. These algorithms only work on acyclic graphs: a cycle causes the first to never stop and the second to stop too soon.

The graph compaction model is more complex: it allows wires to slide on wide endpoints to produce smaller layouts. The resulting negative wire constraints cause cycles in the constraint graph. Since every tile in a cycle depends on every other tile in the cycle, there is no topological order in which to place the tiles. Compaction is not difficult, however, given an initial feasible solution to the constraints. Instead of solving for tile positions we solve for how far down each tile can move. The constraints for this movement problem are the slacks in the constraints for the position problem [Edmonds 72]. Since a feasible solution has all nonnegative slacks we can use a simple shortest path algorithm [Dijkstra 59]. Starting with the bottom edge, when a tile is moved we add all the unmoved tiles directly above it to a priority queue sorted for minimum downward-movable distance ignoring constraints to unmoved tiles. The distance a queued tile can move will later decrease if some of these ignored constraints are more restrictive. By always moving the least-movable tile, we know that none of the unmoved tiles will be in the way since they will all later be moved at least as far. This is physically equivalent to sweeping the bottom edge up and shoving tiles as they are hit.

One way to generate the needed feasible solution is to place the objects far enough apart to satisfy the spacing constraints and then make any small adjustments needed to satisfy the wire constraints. If the problem has a solution with wires connected to fixed points on endpoints, the group model's topological sort can be used to create a feasible solution. From this initial solution, the maximum distance a tile can be moved down by the shortest path algorithm is relatively small. Instead of using a standard n-node, $O(n \lg n)$ time priority queue (a heap [Williams 64], a balanced binary tree embedded in an array, sorted so that each node has greater priority than its children), we use an

$O(n+b)$ time bin priority queue (a queue with an array of b bins [Dial 69], one linked-list header for each possible priority). For a roughly square layout, a priority queue with $O(n^{.5})$ bins gives an $O(n+n^{.5}) = O(n)$ compaction time.

Although it cannot happen with most stick editors, specifications with cyclic group graphs require a depth-first search [Tarjan 72] to separate out the strongly connected components and a worst case $O(k^2 \lg k)$ general shortest path algorithm on each k-tile cyclic subgraph [Johnson 77]. The depth-first search numbers the tiles (in preorder) and recursively finds the smallest numbered tile of each remaining component. The shortest path algorithm alternates between a fixed-order topological sort to correct the positive spacing constraints and a priority-queue sort to correct the negative wire constraints. The maximum number of $O(k \lg k)$ passes is bounded by the number of times the critical path alternates from wire to spacing constraint -- $O(k)$ in a k-tile subgraph. Starting from the bottom edge of a solution with violated wire constraints, we could instead use a breath-first simple queue of the tiles found to need moving. Then each step is faster, but the maximum number of steps is bounded by the number of wire constraints in the critical path.

3.1.1. Incremental Compaction

Incremental changes are relatively easy under the group and graph models since there is never any question of which way to move tiles. When a tile is inserted and there is not enough room for it in the layout, the tiles that it overlaps are moved up, out of the way. An upward movement may create new overlaps and thus cause more tiles to move up. When a tile is deleted, the remaining constraints of the tiles that it was holding up are checked to see if any tiles are now free to move down (to create a smaller layout). A downward movement may slacken active constraints and thus allow more tiles to move down. An efficient incremental algorithm will move each tile at most once per incremental change.

Under the group model, insertions and deletions can be done using a priority queue sorted for lowest current group position. By always moving the lowest group that needs moving, we know that all the groups below it are in their final position and thus never move a group more than once. Moving the groups in the numbered order after deleting the dark tile in Figure 3-1 moves tile 5 directly to its correct position. To move m bounded-size tiles a maximum distance d we can use a priority queue with worst case $O(m)$ or $O(m+d)$ bins for deletions or insertions, respectively, and a count of groups still

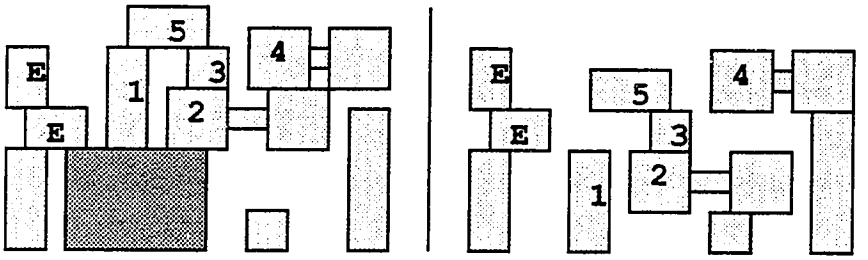


Figure 3-1 Group Deletion Movement Order

needing moving. Inserting or deleting wires can merge or split groups so we have to be able to merge and split constraint-graph nodes.

If we knew before hand which *m* groups an incremental change would move, we could perform incremental compactions with the group's batch O(*m*) topological sort. We do know that the groups that move after deleting a tile will always be a subset of the graph of active constraints rooted at and above that tile -- the groups held up by the deleted tile (E and the numbered groups in Figure 3-1). But the overhead of marking this subgraph with its *e* extra groups makes the resulting O(*m*+*e*) topological sort relatively slow: many nonmoving groups may be processed.

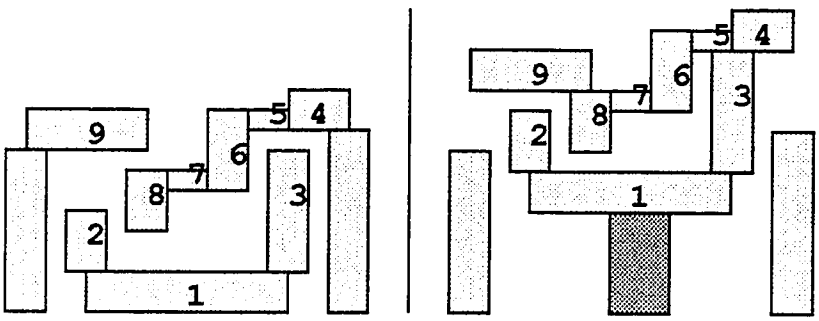


Figure 3-2 Tile Insertion Movement Order

Under the graph model, insertions, which force tiles to move up, can be done using a priority queue of tiles sorted for maximum distance that we know each tile must be moved. By always moving the tile that needs moving the farthest, we know that a movement will not hit and force one of the just moved tiles to move again (since they have already moved at least as far as the moving tile). Moving the tiles in the numbered order in Figure 3-2 moves tile 9 directly to its correct position. As before, the distance a tile must move can change while the tile is in the queue. When a tile moves up, it forces all the tiles in the subgraph of active constraints above it to move with it (tiles 5

to 8 with tile 4 in the example). Therefore we can save time by moving whole subgraphs without putting all of their tiles through the queue. Even better, since almost all insertions cause short maximum movements d we can use a priority queue with d bins to move m tiles in $O(m+d)$ time.

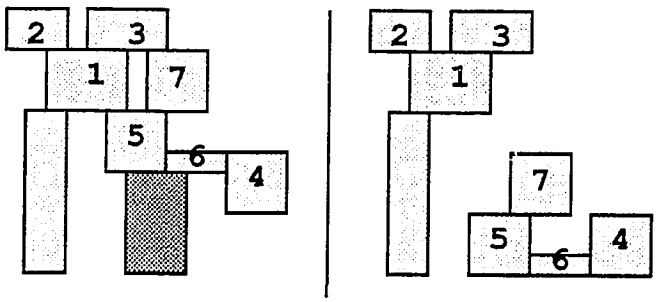


Figure 3-3 Tile Deletion Movement Order

Deletions, which allow tiles to move down, are more difficult. While one violated constraint forces a tile to be raised, all of the constraints holding up a tile must be slack before it can lowered. The tiles in a cycle of active constraints will never move down since each seems to be held up by another. Tiles 4 to 6 in Figure 3-3 will slowly ratchet down because of their constraint cycle. We solve this problem as we did in the batch algorithm -- by ignoring constraints between unmoved tiles. The tiles in the subgraph of active constraints above a tile are the only ones that might move after the tile is deleted. We mark those tiles unmoved (the numbered tiles in the example), leave the others marked moved, and add all the unmoved tiles directly above a moved tile to a priority queue. Then, as in the batch algorithm, we sort for minimum movable distance and always move the least-movable tile (the numbered order in Figure 3-3). We can again move whole subgraphs without putting all of their tiles through the queue. Even though deletions are more likely to cause long movements than insertions, for maximum movements d we can still use a priority queue with d bins to move m tiles after checking c nonmoving tiles (the first three in the example) in $O(m+c+d)$ time.

3.1.2. Summary

In this section we have described a couple of simple compaction algorithms. A linear-time topological sort creates an optimal solution for the groups created when wires are connected to fixed points on endpoints. From a feasible solution, a linear-time shortest path algorithm (using a bin priority queue sorted for minimum constraint slack)

further compacts a layout by allowing wires to slide on wide endpoints. Incremental group deletions and insertions can be done in $O(m)$ and $O(m+d)$ time, respectively, to move m tiles a maximum distance d by always moving the lowest-positioned group that needs moving. Incremental graph insertions can also be done in $O(m+d)$ time by always moving the tile that needs moving up the farthest. Incremental graph deletions, done by always moving down the least-movable tile, are slightly more difficult because they require checking possibly many nonmoving tiles.

3.2. Wire-Length Minimization

Compacting a circuit to its smallest size only determines the placement of tiles in critical paths; the rest of the tiles have a range of possible placements. Instead of just minimizing the placement height of each tile as in the previous section, we use the freedom to minimize the sum of the wire lengths, suitably weighted by type (layer) of wire. Adding a heavy weight dummy wire between the top and bottom edges will minimize the layout's height. Wire-length minimization improves layouts and makes it easier for designers to specify incremental changes. By bunching tiles together, it also leaves larger holes and allows subsequent compaction steps to do a better job.

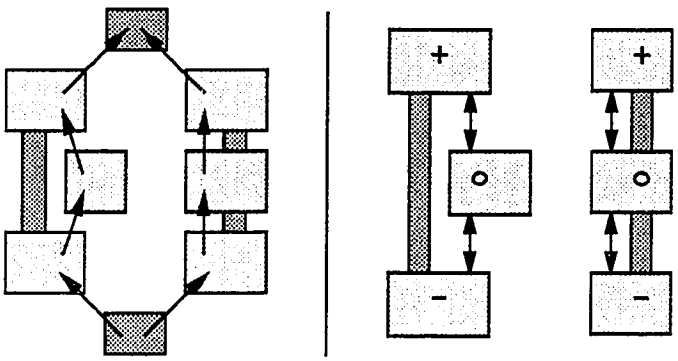


Figure 3-4 A. Flow and B. Weight Analogies

We can use variations of min-cost max-flow network algorithms to perform wire-length minimizations. In the flow analogy, the constraints carry nonnegative flow from their lower to upper nodes. The cost per unit flow traveling through a constraint arc is the negative of the constraint value. Wires force flow from their lower to upper endpoint nodes. To preserve the conservation of flow at each node, we add dummy arcs from the source or drain to supply or consume the forced flows (the arcs from and to the dark blocks in Figure 3-4A). A tile's position is the price (the dual variable), the

negative of the savings if one less unit of flow had to flow to its node. Optimal tile positions correspond to a min-cost flow [Eichenberger 86].

We are going to use a more direct analogy. Each tile is given a weight equal to the sum of the weights of the wires pulling it down minus the sum pulling it up (see Figure 3-4B); positive-weight tiles want to move down, negative up. A vertical wire is represented by adding its weight to its top endpoint and subtracting an equal weight from its bottom endpoint. Wires have the same effect as springs pulling their endpoints towards each other. Constraints correspond to fixed length struts between tile edges. Each constraint has a stress, caused by holding positive weight up and an equal negative weight down, that is equal to the flow through the constraint in the flow analogy. Constraints can be slack, active (exactly satisfied), or violated. An optimal solution has four necessary and sufficient conditions: each tile's weight is balanced by the forces on it, the slack constraints must have zero stress (since they are not holding any tiles apart), there are no violated constraints, and each active constraint's nonnegative stress pushes with an equal force up on its upper tile and down on its lower tile.

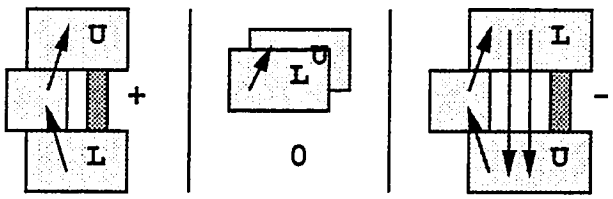


Figure 3-5 Possible Wire Lengths and Stresses

In the previous chapter we decided not to allow tiles to pass through each other. Thus there will always be constraints keeping the upper endpoint of a vertical wire at or above the lower endpoint; we do not have to worry about wire-length minimization preferring negative-length wires over zero-length wires. It would be fairly easy to modify the weight analogy to handle pass-throughs. A relative force between $-w$ and w should not pull a weight w wire's endpoints apart. Since a stress of w is initially put between the two endpoints, this range corresponds to a stress between 0 and $2w$. To hold this stress for zero-length wires, we add a zero-value constraint directly between each wire's endpoints that can hold a maximum stress of $2w$. It is legal for such a constraint to be unstressed and slack, stressed and active, or fully stressed and violated. Figure 3-5 shows the stress distribution for the three possible wire-length cases: the zero constraint has 0, w, or $2w$ stress for wires with positive, zero, or negative lengths, respectively. We get the same effect more simply by just swapping a wire's upper and

lower endpoint pointers when its zero constraint is overstressed. In our physical model, this corresponds to a strut failing and being replaced by one with switched top and bottom connections.

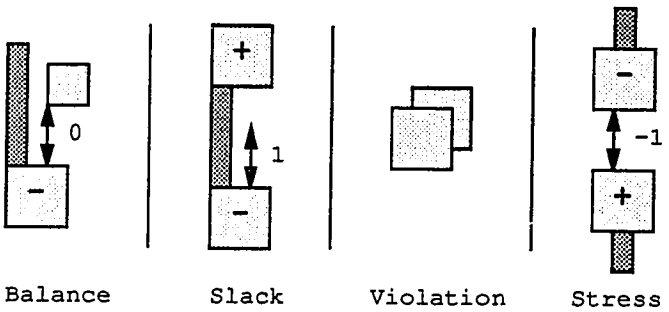


Figure 3-6 Four Minimization Algorithm Types

Wire-length minimization is more difficult to solve than simple graph compaction because now a tile's placement depends on the total weight of the set of tiles pressing against it, which changes as the tiles are moved. We will describe four batch compaction algorithms to solve this problem. Each is based on setting three of the four solution conditions true and then correcting the final one. Figure 3-6 shows the four cases: the first algorithm has to fix the tile weight-force balances, the second any stressed slack constraints, the third any violated constraints, and the fourth any negative stresses. All but the third start with a graph compaction to generate an initial solution. Placing all the tiles as low as possible reduces batch wire-length minimization to determining how far up to move each tile. We describe the algorithms in a more-intuitive to less-intuitive order. The first two are based on common network algorithms to which we have added several enhancements to take advantage of compaction's special properties. The third is the standard simplex network algorithm and the fourth, the tree weight algorithm, is a new, efficient algorithm that is easily modified to perform incremental changes.

3.2.1. Balance Algorithm

The balance algorithm starts with all the stresses set to zero. Thus only zero-weight tiles are balanced by the forces on them. We calculate and store with each tile its current imbalance, its weight. We could fix the imbalances one at a time, but it is better to simultaneously fix them. We repeatedly move the remaining negative-weight tiles upwards as a group the smallest distance needed to hit a zero or positive tile. The

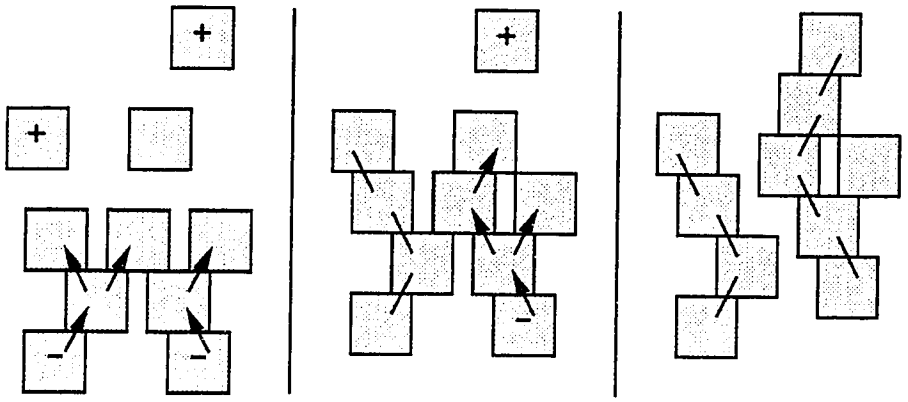


Figure 3-7 Two Balance Algorithm Tile Movements

tiles' imbalances and thus their desire to move are reduced as positive weight cancels negative weight. In Figure 3-7, the first movement satisfies the left negative tile and the second movement the right. In each step, we mark all the negative tiles and then breath-first recursively mark all the tiles reachable by traversing up active constraints. A back pointer from each marked tile is used to store the reverse of the path traversed to reach it so that when a positive tile is found, the stress caused by its weight can be added to all the constraints in the path back to a negative tile. If the positive weight is greater than the absolute value of the negative weight, only enough stress to satisfy the negative weight is added to the path.

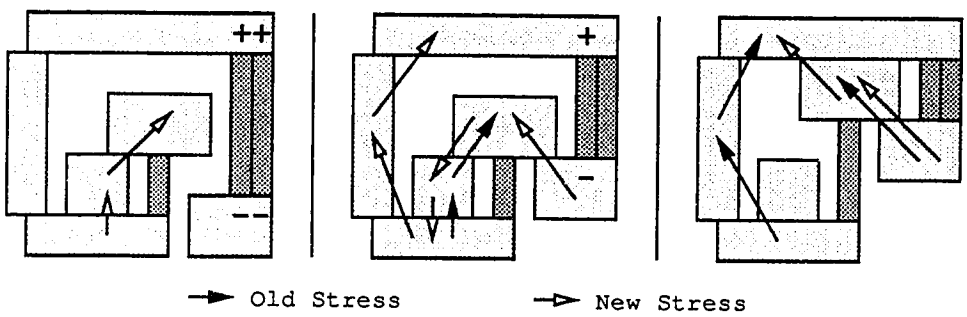


Figure 3-8 Stress Reduction: Three Added Stress Paths

After we traverse up all the active constraints, we move all the marked tiles upwards enough to hit something. This produces new active constraints which are traversed to mark more tiles. We continue moving and marking to find positive tiles. Once a constraint is stressed, it is possible that because of other wires we will later want

to reduce that stress. Therefore, instead of only traversing up active constraints we also have to traverse down constraints with positive stress. In Figure 3-8, the light arrows show newly added stress paths and the dark arrows show old stress. In the example's second step, the added stress reduces the stress in the two stressed constraints that are traversed down. Since stress cannot be reduced below zero, a path's least stressed such constraint can restrict the amount of added stress to less than would otherwise be added. Thus in Figure 3-8 a third step, a third path, is needed to distribute the final unit of stress. Movements will never violate constraints or slacken stressed constraints since we traverse up all reachable active constraints and down all reachable stressed constraints before moving any tiles.

When a negative tile's weight is reduced to zero it no longer wants to move and is no longer a valid end for back-paths. When a constraint's stress is reduced to zero it is no longer a valid step in the path that traversed down it. We have to unmark all the tiles that were marked traversing paths from that tile or constraint and are not reachable by other paths. A simple solution is to unmark all the tiles and begin a fresh marking. Each such pass in a general graph reduces the total negative weight w by at least one and consists of alternatingly marking at least one more of the at most n unmarked tiles and moving the at most n marked tiles. This gives an $O(wnn) = O(n^3)$ total worst case since w is $O(n)$ for bounded-weight wires. This corresponds to the out-of-kilter algorithm with the entire flow initially in an added artificial arc and kiltered by redistribution into flows each through two dummy arcs and some intervening constraint arcs [Fulkerson 61].

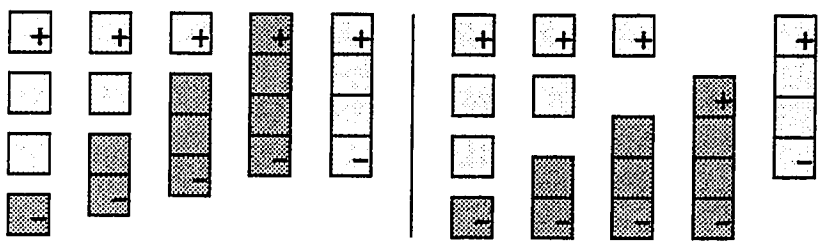


Figure 3-9 $O(m^2)$ Repeated Verses $2m$ Total Tile Movements

We can make two modifications to the balance algorithm to improve its expected running time. First, instead of making more constraints active by repeatedly moving all the marked tiles upwards we just move each tile down once as it is marked and up once as it is unmarked. In the left half of Figure 3-9 we move one, then two, then three tiles,

while in the right half we move one tile at a time except in the final step. We keep all the unmarked tiles directly above marked tiles in a priority queue sorted for minimum slack in constraints to marked tiles. By always marking and moving down the tile with the minimum slack, we mark tiles and make constraints active in the same order as before. When tiles are unmarked, they are moved up the same distance the last tile was moved down; each tile is moved up at least as far as it was moved down and at least the just found positive tile ends up unmoved. The tiles arrive at the same positions as if they had been repeatedly moved upwards. Each of at most w passes in a general graph now consists of marking at most n tiles while making at most $2n$ tile movements. Using a priority queue with s bins gives an $O(w(n+2n)+s) = O(n^2)$ total worst case since in a roughly square layout the maximum constraint slack s is $O(n^{.5})$. This corresponds to the algorithm that starts with zero flow and repeatedly adds flow along the min-delta-cost remaining path [Edmonds 72].

The second modification is to just unmark the tiles that need unmarking instead of repeatedly unmarking all the tiles. We could reconstruct the paths traversed from a newly zeroed-weight tile or unstressed constraint using the back-path pointers, but it is easier to remember the paths by keeping the marked tiles in a forest of trees with one tree per negative-weight tile. Each tree contains the tiles that were marked by traversing from its root, its negative tile (see Figure 3-7). Thus, when a tile's negative weight is satisfied or a constraint is unstressed we can just unmark the tile's tree or the constraint's disconnected subtree. This saves us from continually remarking all the remaining negative-weight tiles and their trees. To use both a queue and the trees, we have to update the queue when we unmark a tree or subtree. Simply, we can remove from the queue any tiles directly above the tree or subtree and then add to the queue any just dequeued or just unmarked tiles directly above a still marked tile. The worst case remains $O(n^2)$ since it is possible to mark many tiles during each pass even when being clever about remarkings.

3.2.2. Slack Algorithm

Instead of starting with no stress as in the balance algorithm, the slack algorithm starts by putting the stress caused by each wire into the constraint directly from the wire's lower to upper tile (zero-value artificial constraints are added where needed). Thus all the tile weights are balanced, but some of the stress may be in slack constraints. We use a (possibly sorted) list of stressed slack constraints to remove the stress from

one slack constraint at a time. Each stress removal creates a pair of unbalanced positive-weight and negative-weight tiles. As before, we mark and move the negative tile up until it cancels with the positive tile. This algorithm has basically the same worst case times as the balance algorithm: O(*wnn*) or O(*wn+wd*) depending on whether or not a queue is used. Fewer tiles will be moved at once since there is only one negative tile at a time. A queue to prevent repeated movements is not as necessary and the one tree to help unmarking is not very useful. This corresponds to the out-of-kilter algorithm starting with a distributed flow -- the stressed slack constraints are the out-of-kilter arcs [Fulkerson 61].

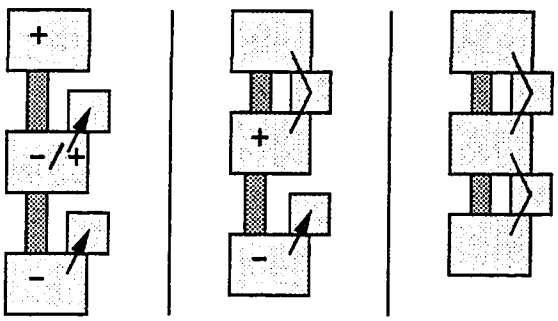


Figure 3-10 Two Slack Algorithm Tile Movements

We can improve the slack algorithm by fixing all the stressed slack constraints simultaneously, much as we moved all the negative weights together in the balance algorithm. Instead of putting a wire's stress in a slack constraint, we give a negative weight to the wire's lower endpoint and an equal positive weight to its upper endpoint. The interior endpoints in a series of wires can have both a positive and a negative weight: wire pulls do not directly cancel as they did in the balance algorithm (see the right 0 tile in Figure 3-4). Tiles with any negative weight are marked to be moved up. Thus once tiles start moving, a mixed tile's positive weight will not be distributed until after its negative weight is satisfied and it stops moving. The first movement in Figure 3-10 satisfies the middle tile's negative weight. Then, in the second step, the bottom tile can move up, hit the middle tile, and satisfy the remaining weight. Again, we use a queue to prevent repeated movements and trees to help unmarking.

When a tile-marking traversal runs across an old, unmarked tree, instead of traversing and marking the tree's tiles one at a time, we can save time by marking large sections of the tree in a single step -- being careful not to traverse down any of the tree's unstressed constraints. This helps, for example, when one of the early traversals finds a

mixed-weight tile. Instead of unmarking the thus satisfied traversal tree and later remarking it (in a traversal from the mixed-weight tile), we can just immediately merge the tree with the mixed-weight tile's tree. Another optimization is to allow the first traversals (before any tiles are moved) to only go up active constraints, not down stressed constraints. This will quickly perform the easy stress distributions, the distributions through the original non-slack constraints. Then the rest of the distributions are done using the normal tile traversals and movements. Layouts often have many long series of short wires, such as power busses and clock lines. The slack algorithm sees separate short wires where the balance algorithm sees long wires with intervening zero-weight tiles. We will show that this gives the modified slack algorithm a speed advantage.

3.2.3. Balance and Slack Timings

In this section we take a closer look at the timings for the balance and slack algorithms. The greatest distance d that a tile can move without increasing the size of a layout is bounded by the height of the layout. Thus in a roughly square layout, the maximum effort expended moving tiles is $O(nd) = O(n^{1.5})$ when every tile moves up as far possible one lambda at a time. In these algorithms, unfortunately, the movements are not the limiting factor: a weight distribution might not move any tiles. The maximum effort expended not moving tiles is w passes each fixing one unit of negative weight after marking at most n tiles for an $O(wn) = O(n^2)$ time. This gives an actual $O(nd+wn) = O(n^2)$ total worst case for all these algorithms.

More interesting than the worst case times are the expected times. Let us assume that to find a path of length p from a tile, we mark about $O(p^2)$ tiles: in a two-dimensional layout there are, in some sense, that many tiles within range. Moving the marked tiles without using a priority queue might cause $O(p)$ repeated movements for a total $O(p^3)$ movement, while with a queue gives an all at once $O(p^2)$ movement (as in Figure 3-9). Let us also assume that the length of the path for a wire's stress is proportional to the final length of the wire. The expected time using a queue and trees is thus O(sum of the square of the wire lengths). Designers try to minimize the number of long wire runs in a layout. Let us estimate that the balance algorithm sees about $O(n^{0.3})$ wires that reach across most of the layout, length $O(n^{0.5})$, with the rest length O(1). The long wires each take $O((n^{0.5})^2) = O(n)$ and the short $O(1^2)$ for a total $O(n^{0.3}n+(n-n^{0.3})1) = O(n^{1.3})$ expected time -- n times the number of very long wires.

Since the slack algorithm sees a series of wires as separate short wires, we can estimate that it sees only $O(n^{0.2})$ long wires for an $O(n^{1.2})$ expected time.

3.3. Tree Compaction

In the next two algorithms we restrict the graph stress to a directed spanning tree of active constraints. This corresponds to only looking at the corners of the linear problem space. Instead of associating stress with constraints, we associate subtree weights with tiles. This weight is the sum of the wire pulls on all the tiles in the subtree rooted above each tile. The only tree path into a subtree goes through its root so our new tree weight is equal to the old stress in the connecting constraint -- positive if the constraint is holding the subtree up, negative if holding it down. If we knew the correct spanning tree we could calculate all the stresses and tile positions with a simple tree traversal and thus save the effort of finding an initial feasible solution and distributing the individual stresses. Unfortunately, fixing one part of an estimated tree often breaks other parts of the tree.

3.3.1. Simplex Algorithm

The simplex [Dantzig 65] algorithm, unlike the others in this chapter, does not start with an initial graph compaction. It uses the spanning tree to avoid finding an initial feasible solution. At each step it only ensures that the constraints in the spanning tree are not violated. The algorithm is run twice: first to create a properly stressed spanning tree and then to fix any violated constraints. The first run starts with all the constraint values set to zero, all the negative individual-weight tiles placed at plus one, and all the other tiles at minus one. Artificial constraints are added to hold the stresses needed to balance the tile weights. The first run moves all the tiles to the origin to fix the violated zero constraints and thereby converts the height one tree of artificial constraints to a legally stressed spanning tree. Then a simple tree traversal calculates, using the correct constraint values, the correct tile positions ignoring constraints not in the tree. The second run fixes any of these ignored constraints that are violated.

Each run makes many passes through the list of constraints. When a violated constraint is found, say the worst violation, it is added to the spanning tree. This creates a cycle that is broken by circulating enough stress to unstress a constraint so that it can be removed from the tree. This is actually done by adding weight to the violated

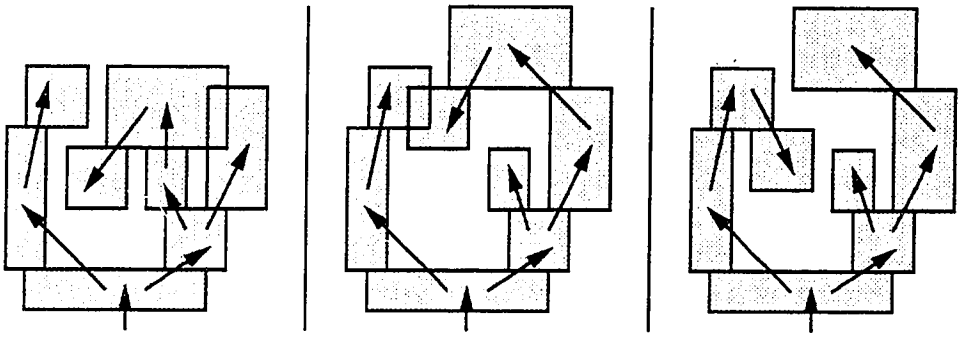


Figure 3-11 Two Simplex Constraint Violation Fixes

constraint's lower tile and subtracting an equal weight from its upper tile to try to pull them apart. This weight change is propagated back through the tiles' ancestors until it cancels at their highest common one (easily found by storing with each tile its tree height). The nearest zero-weight subtree is disconnected to form a tree fragment. This fragment is moved to exactly satisfy the violated constraint, reconnect the fragment, and rebalance the weights. This fix may create new constraint violations, which may cause some of the tiles to later be moved back. In Figure 3-11, two tiles are moved up to fix the top right overlap. This causes a new overlap, on the left, that is fixed by moving one of the tiles back down. These possible oscillations create an exponential worst case.

This algorithm corresponds to the simplex algorithm for min-cost max-flow in a network [Kennington 80]. The first run creates a max-flow spanning tree and the second run does lots of movements to reduce it to min-cost. Other methods such as exponentially increasing the weights or constraint values will not help since our weights and constraint values are small. The expected time, though still not a lot worse than linear, is not very good. The main problem with the simplex algorithm is that, unlike the others, it does not make use of the shortest path algorithm to find a good starting point. Instead, it starts off solving the hard part of the problem, namely, finding a max-flow, a legal distribution of the stress. Then it has to maintain the distribution while it moves tiles up and down to find a feasible solution. Even if we were more clever about creating the max-flow or not over moving tiles, the other algorithms still have the advantage of getting the final tile positions almost for free while distributing the stress.

3.3.2. Tree Weight Algorithm

Instead of using the spanning tree to avoid finding an initial feasible solution, we could use it to initially distribute the stress. One idea would be to use the dual simplex algorithm. Starting with a feasible solution, each step only insures that the stress in the spanning tree is consistent (forces balance weights). Repeated passes are made to find and fix improper (negative) stresses. This is an improvement over the simplex algorithm since it starts with a better initial solution, but it still has the drawback of requiring multiple passes that may over move tiles and then have to move them back. The tree weight algorithm is far more efficient: it fixes the spanning tree in a single pass by being careful to never create new improper stresses and never over moving tiles.

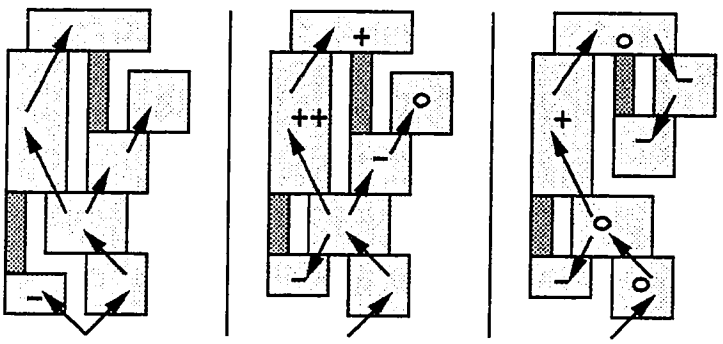


Figure 3-12 Two Tree Weight Negative Subtree Movements

We modify the initial graph compaction so that each tile remembers which tile stopped its downward movement. We use the thus produced spanning tree as an estimate of the final tree and calculate the stresses, the tree weights, using a depth-first tree scan that recursively adds to each tile its children's weights as they are determined. Some of these weights may be negative, but by construction all the constraints in the initial spanning tree hold subtrees up. Since a negative-weight subtree needs to be held down, not up, we take its erroneous connecting constraint out of the spanning tree. The subtree, now a tree fragment, is moved up enough to make a constraint active. This constraint is used to reconnect and hold down the fragment. We have to add the subtree's negative weight to any of its new ancestors whose weights have already been calculated. In Figure 3-12, the first calculated weight (bottom left) is negative so that tile is moved up and reattached. After calculating +, ++, and 0 subtree weights, another negative weight is found. The two-tile subtree is moved and the 0, +, and ++ weights

are reduced to -, 0, and +. Note that the directions of some of the parent pointers in a moved fragment may be reversed. Calculating two more zero weights finishes the compaction.

If a weight redistribution would drive some of the weights past zero, only enough weight to zero some subtree is propagated. We disconnect the nearest such subtree and repeat (moving the newly created fragment) until enough positive weight is found to stop the upward movement (as in Figure 3-8). Since negative-weight subtrees are fixed as they are found, it is likely that movements will hit tiles whose weights have not yet been calculated. We assume that an uncalculated weight will be enough to stop a movement. When it is, we have fixed the tree without changing any ancestors' weights. Otherwise, the stopped tiles will later move again. An alternative would be to first calculate and cut off all the negative-weight subtrees. Moving all the fragments upwards as a group would stop these repeated movements but would also force us to update the weights of all the new ancestors of each moved fragment.

The tree weight algorithm has the same worst case time as the balance and slack algorithms. We use, as before, a priority queue to stop repeated movements. A tree fragment is broken off when a negative weight is calculated. As it moves up, it will accumulate at most n tiles before it either hits a tile with an uncalculated weight and stops or it hits a tile to which it can redistribute some of its weight (which changes at most n weights). The first case happens at most n times since there are at most that many calculated negative weights to stop and the second case at most w times since there is at most that much stress to redistribute for an $O(nn+w(n+n)) = O(n^2)$ total worst case.

We can get the same order worst case using a one-bin queue. When this bin, representing the current minimum slack between marked and unmarked tiles, is empty we make another pass through the marked tiles to find the remaining minimum slack and fill the bin. Since each extra pass occurs after the marked tiles in effect move to take up the old minimum slack, the maximum overhead is limited by the maximum number of tile movements, $O(n^{1.5})$ as before. Instead of filling a bin, entries can be found on demand; instead of using a bin, a pointer to the last found entry can remember where to start the search for the next entry. Besides being much simpler, a pointer is fast since we usually use very few queue entries.

The expected time is again much better than the worst case. The tree weight

algorithm depends on starting with a good estimate of the final spanning tree. Our estimate should be good: all the tiles in the critical paths or with positive weights are correct and many of the rest are close. The tree height should be near the layout height, $O(n^{.5})$. Many of the fragment movements will hit tiles with as yet uncalculated weight and thus spend no time redistributing weight. The tree weight algorithm should perform as well as, or slightly better than, the balance and slack algorithms. They all perform approximately the same number of tile movements, but the balance and slack algorithms have to distribute each individual wire stress -- even for wires that cause little or no tile movement. The tree weight algorithm only worries about stress caused by negative-weight subtrees. This savings is partially balanced by the extra overhead of fixing the estimated spanning tree.

3.3.3. Incremental Compaction

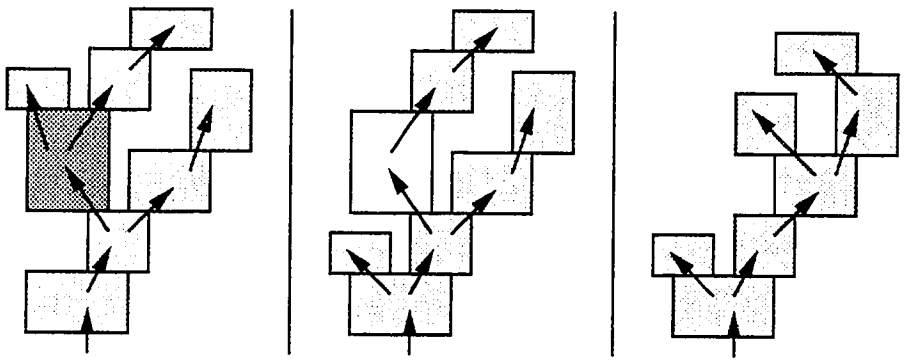


Figure 3-13 Two Subtrees Orphaned by a Tile Deletion

The tree algorithms can be adapted for incremental compactions. The spanning tree tells us which tiles should be moved and in which direction. To delete a tile we remove it from the tree. Removing a tile's constraints will cause its children, if any, to be broken off into tree fragments. We remove the constraints one at a time and move and reattach each fragment as in the tree weight algorithm except now fragments have negative or positive weights and thus move up or down. The tile deletion in Figure 3-13 causes two fragments to move down: first a one tile fragment and then a two tile fragment that breaks apart. By carefully breaking ties when creating tree fragments, the incremental compaction can be biased downward so that a subtree will never be left with a negative-zero weight (say after a wire holding it up is deleted). Negative zeroes

would slow the worst case time and force us to distinguish between positive and negative zero. Alternatively, we could remove all the zero-weight arcs from the tree. This would require maintaining the resulting forest of zero-weight trees but would allow each tree to float without pressing against other trees.

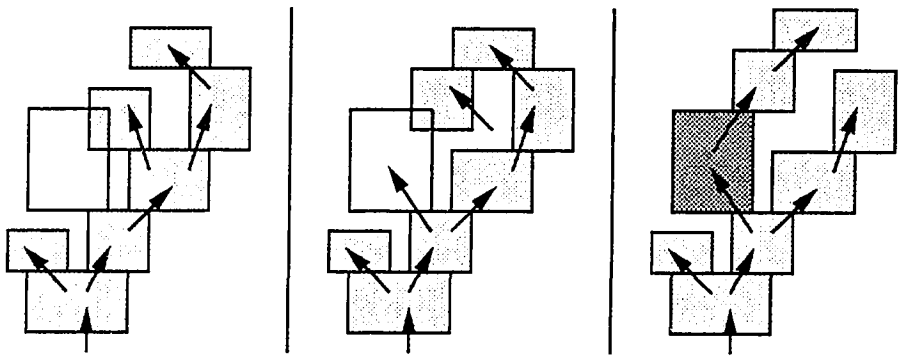


Figure 3-14 Two Movements Caused by a Tile Insertion

To insert a tile we first find a place to attach it to the spanning tree. Some of the constraints with the new tile will be violated if there is not enough room for it in the layout. Violated constraints are fixed as in the simplex algorithm -- by propagating enough weight back through ancestors to break off tree fragments except now we make sure that movements do not violate other constraints; when we move a fragment we find its least-slack constraint and reattach it there. We repeat until enough movement occurs to exactly satisfy the violated constraint, reattach the fragment, and rebalance the weights. This insures that tiles are never moved further than necessary and keeps us from having to search for violated constraints. Some time can be saved by not reorienting parent pointers in a fragment until its final position is found. The tile insertion in Figure 3-14 causes one tile to move up; this tile hits and causes another tile to move.

Incremental changes have an $O(m+c+p+d)$ expected time to move m tiles a maximum distance d while checking c nonmoving tiles and propagating weight changes through p tiles on paths back to highest common ancestors (to redistribute the weights of moved subtrees). Even when two tiles are physically close, their highest common ancestor in the tree could be far away. Because we have to find a possibly $O(n)$ distant ancestor every time a tree fragment is reattached, we get an $O(n^2)$ worst case to redistribute a unit of weight even when no tiles actually move. There can be at most n in a row movements that distribute no weight since each such upward or downward

movement gains or loses, respectively, at least one tile. If we merged long branches into balanced binary trees (dynamic trees [Sleator 85]) we could reduce the maximum amortized height of our tree to O(lg$n$). While this would reduce the worst case for multiple changes, individual changes would have the additional overhead of maintaining the balance. Since the weight redistribution is normally a small part of an incremental change, the dynamic tree complexity is probably not worth the effort.

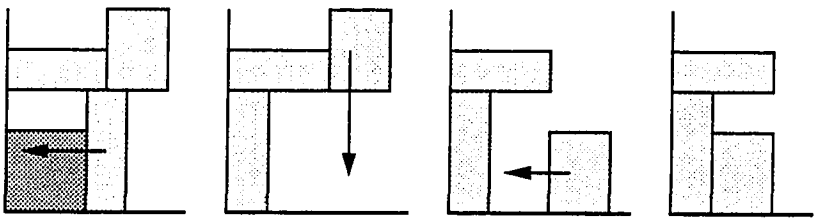


Figure 3-15 Three X-Y Steps from One Change

There is an important interaction between the two dimensions. Say that we have a list of tiles that were moved left-right. We update the adjacency lists for this motion. This can cause some of the arcs in the up-down spanning tree to no longer correspond to actual constraints. The invalid parent pointers belong entirely to the moved tiles or their children (as in tile deletion). We go through a list of these tiles to remove any invalid parent constraints from the tree and fix each resulting fragment as before. This might move some tiles up-down. We use a list of these moved tiles to update the adjacency lists; this time possibly corrupting the left-right spanning tree. We continue alternatingly updating the adjacency lists and spanning tree for each dimension until we reach a fixed point. Deleting the dark tile in Figure 3-15 causes a left, a down, and another left movement. This corresponds to an infinite number of x-y compaction steps almost for free. The movements will usually quickly stop since each step decreases the tile heights, the total weighted wire length, or even the size of the layout.

3.4. Summary

Wire-length minimization improves a layout by taking advantage of the range of possible placements of tiles not on critical paths. Using an initial graph compaction to place each tile as low as possible reduces wire-length minimization to determining how far up to move each tile. The balance algorithm moves as a group all the tiles with more upward than downward wire pull. The slack algorithm moves as a group all the tiles with any upward wire pull. Starting with the tiles we want to move, we traverse up

active constraints and down stressed constraints to find the tiles that must move with them and to find tiles to stop the movements. Both algorithms have an $O(n^2)$ worst case: they distribute individual stresses through active constraints to fix tile weight-force balances or take stress out of slack constraints. We estimate that these algorithms, when using a priority queue to prevent repeated movements and a forest of trees to help unmark tiles, have an O(n times the number of very long wires) expected time. This gives the slack algorithm a speed advantage since the balance algorithm often treats a series of short wires as one long wire.

The simplex algorithm restricts the wire stress to a directed spanning tree of constraints and uses the tree to avoid calculating an initial feasible solution. This, unfortunately, trades a linear-time graph compaction for a possibly exponential number of movements to fix violated constraints. The tree weight algorithm, on the other hand, uses the spanning tree to initially distribute the stress. An initial graph compaction creates a spanning tree. Subtrees with more upward than downward total wire pull are disconnected from the tree, moved up to find something to hold them down, and reattached. Since negative-weight subtrees are moved as the weights are calculated, relatively few ancestor tree weights have to be updated. Instead of a priority queue, we use a previous-entry pointer to reduce repeated movements. The tree weight algorithm should be faster than the other algorithms since it does not have to distribute each individual stress and it never over moves tiles.

The spanning tree allows us to make incremental changes by telling us which way to move which tiles. Deleting a tile can disconnect subtrees from the tree; the fragments, one at a time, are moved up or down depending on their total wire-pull weight. Inserting a tile may force some subtrees to move to create enough room for the new tile. Moving tiles in one dimension may cause some of the arcs in the other dimension's spanning tree to no longer correspond to actual constraints. Removing these arcs creates tree fragments whose movements may disturb the first dimension's spanning tree. Alternating dimensions to reach a fixed point allows us to incrementally perform the equivalent of an infinite number of x-y compaction steps almost for free. In addition to checking possibly many nonmoving tiles, incremental changes with wire-length minimization also require changing possibly many ancestor tree weights to redistribute the weights of moved subtrees. The expected case gives fast, efficient incremental changes -- proportional to the size of the change instead of the size of the circuit.

Chapter 4
Implementation

In the previous chapters we described efficient algorithms to generate and solve the constraints needed in an incremental compactor. In this chapter we look at how these algorithms were integrated into a working compaction system. We start with an overview of the tree compaction system. Then we describe the technology file and the operations used to convert a stick diagram into a set of tiles and then into mask data. After we describe the tile data structure, which contains all the fields needed by the adjacency lists and tree weight algorithms, we give more details on the actual implementation of the batch and incremental compaction algorithms. The compaction system was instrumented to record the amount of time spent in the main compaction steps and the number of times various smaller operations were executed. We finish with an examination of these results.

4.1. Tcmp

The tree compaction system, Tcmp, starts by reading in a technology data file. This file contains the information necessary to translate stick objects into tiles and tiles into CIF rectangles. It also provides the spacing rules and wire weights needed for compaction. The spacing distances are used to fill the various arrays of minimum and maximum spacings and spacing differences needed by the adjacency lists algorithm. Then we are ready to read in a circuit's stick file. The stick files are created by Edc, a stick editor. Tcmp and Edc are separate systems but they should be merged together. In Edc, wires and transistors are first-class objects, which can be placed and moved, but wire endpoints are not: their positions, sizes, and types (single color, contact, or via) are determined by the wires connected to them. Tcmp, therefore, not only has to convert wires and transistors into tiles, it also has to create the proper endpoint tiles and link them with the wires. Edc prevents illegal wire crossings and connections; thus Tcmp can assume that a stick file corresponds to a valid stick diagram.

Once the tile data structure is loaded with a circuit, we can start compacting. The compaction steps are done in a twisted order to minimize the number of constraint generations required to compare the timings and results of group, graph, tree, and incremental compactions. We alternately generate and solve constraints: *y*-constraint, *y*-group, *x*-constraint, *x*-group, *x*-graph, *y*-constraint, *y*-graph, *y*-tree, *x*-constraint, *x*-tree. Then we use the incremental routines to alternately fix trees and update adjacencies; we start with *x*-fix and continue until a tree fix causes no tile movements. This is a good test of the incremental routines but, unfortunately, a bad way to batch compact: the *y*-graph and *y*-tree steps between *x*-graph's creation of the *x* spanning tree and *x*-tree's use of the tree so thoroughly damage the tree that the incremental fix takes much longer than just doing a third batch compaction step.

To test the interactive features of incremental compaction, Tcmp will, optionally, display the compacted layout and allow the designer to move tiles around. A straightforward user interface allows designers to pick a tile and specify which direction (left, right, up, or down) they want it to move. The compactor adds or subtracts enough weight, in the proper dimension, to break off a tree fragment containing the tile. The fragment is moved (possibly zero distance) until a weight is found to stop the movement. Just modifying tree weights, however, will not move tiles on critical paths. A more powerful set of commands is needed to create a complete system. Various layout overlays can be displayed to help the designer: lambda grid, *x* or *y* spanning tree, *x* and *y* current critical paths, or a highlight of all tiles on any *x* or *y* critical path. When the designer is finished, Tcmp saves the layout in CIF format.

4.2. Technology File

The technology file describes how to translate between stick, tile, and CIF formats and it provides the spacing rules and wire weights needed for compaction. It is divided into two parts: the first describes the colors (pseudo-layers) used in the design and the second describes the stick objects. The first part uses a triangular array to specify the minimum spacing rule between each color that belongs in the same plane. Each color has a list of the CIF rectangles that a tile of that color should generate and the minimum legal size of these rectangles. There are three types of CIF layers: normal filled rectangle, arrays of small spaced rectangles for contact cuts, and a filled closed wire polygon for wells. The second part of the technology file describes the possible objects and the legal contacts. There is a list of tile descriptors for each object. Simple-object

tiles have just a color and the possible connection directions. Tiles for objects such as transistors also have size and relative placement (within the object) fields. Sizes have an absolute component and a component relative to the object's size attribute. For example, a transistor with size Width:Height has touching source, gate, and drain tiles with Wx2, W+4xH, and Wx2 sizes, respectively, and a well tile, size WxH+4. The gate tile maps to a poly rectangle and the well tile to a diffusion rectangle.

The technology data is processed for use by the compactor. The wire weights and minimum spacings are stored in arrays. The wire-weight array also stores which color wire can connect to each non-wire color. For each CIF layer we store its name, type, minimum size, and a list of the colors that map to it. For each object we store its name, type, minimum width and height attributes (where appropriate), and a list of its tiles. For each tile we store its color, possible connection directions, and the absolute and relative components of its width and height and its horizontal and vertical offsets. An object's minimum width and height attributes are derived from the minimum legal sizes of its tiles. A tile's offset from the center of its object is derived from the tile size and stacking information.

The ctop array is set to map from color to plane and is used to load each tile into the proper adjacency lists plane. The plane divisions are easily derived from the spacing rules since every pair of colors in the same plane must have a spacing rule but there cannot be any rules between colors in independent planes. The spacing rules are also used to calculate the arrays of distances needed by the adjacency lists algorithm. The maximum spacing rule for each color, used by the load locates, is put in the mspace array. For every pair of colors, the over array is given the maximum difference of their spacings to any third color. This gives the minimum overlapping edge to edge spacing rule. For every color, the fudge array is given the maximum difference of the color's and any second color's spacings to any third color. This is used to decide when tiles are shadowed and when around-the-corner searches can be cut short. The nudge array is given the absolute value of the minimum of these spacing differences and is used to decide which tiles a shrink can safely skip.

So that wires can be distinguished from single color vias, used for right angle kinks, we split each wire color into two colors: a wire color and a via color. A special pair of colors that always bloat by zero is created for use in each plane's surrounding frame. A frame consists of four unit-size corner tiles connected by four wires. Thus, in each dimension and plane, the normal tiles are surrounded by horizontal wires to the left

and right and corner tiles above and below. Finally, the contact definitions are used to set the via array to map from old object and touching object to new object. For example, a poly wire object connected to a metal object creates a poly-metal contact object. This array is needed to determine an endpoint's proper type from just the connecting wire information.

After we load the technology data we are ready to load a circuit. We use the technology data to map each stick object into a set of tiles. Each set's tiles are linked together by their alignment pointers and placed at a multiple of their object's coarse grid position (to give an initial placement for use by the constraint generation and the group compaction). Wire endpoint objects are not explicitly given in the stick files -- they are just numbered. Thus we have to create the proper objects and link them with the wires. As we create the wires, we summarize the connections to each endpoint by using the via array and storing the accumulation for the *i*th endpoint in the *i*th even tile record's height field. After the stick file is completely loaded, a pass is made through the wires to create the proper endpoint objects and link them with the wires. For multiple-tile endpoints, each wire must be connected to the proper side of the proper tile (in the proper plane). To make this possible, each object's tile(s) are temporarily arranged in lists corresponding to the four possible wire connection directions (using the as yet unused adjacency pointers).

Finally, the endpoint tiles are sized to be at least as wide as their widest connected wire and the wire lengths are adjusted so that wires just touch their endpoints. The frame is sized to just bound the circuit's tiles. The circuit's terminals are located at the top and bottom of each dimension. To keep all the planes and terminals aligned, the alignment pointers are used to link together all the lower left frame corners and bottom terminals and likewise for all the upper right frame corners and top terminals.

After a circuit is compacted, a CIF file is written. The tiles are first sorted by color into lists. Each CIF layer's name is written followed by a list of the rectangles needed in that layer. This is easily done using the list of colors that map to each layer and the lists of colored tiles. Ordering the rectangles by tile instead of by layer would require a lot more file space because of the constantly changing current layer attribute. Rectangles are not generated for nonpositive length wires but we make no effort to merge the rectangles of overlapped tiles or of any other touching tiles. To create all the required rectangles, we have to process all the tiles (wire and non-wire) in one dimension and all the wires in the other.

4.3. Tile Data Structure

The data fields needed by the adjacency lists and tree weight algorithms are combined into a single structure. The records in this structure describe the state of each tile: its position, adjacencies, weight, and location in the spanning tree. For a non-wire tile, there is an even-numbered tile record for up-down compaction steps and an odd record for left-right steps. A horizontal or vertical wire has only an even or odd record, respectively. Each record contains a tile's color and the current lambda coordinates of its top, bottom, left, and right edges. Top and bottom edges in one dimension are equivalent to right and left edges in the other. This allows the same routines to operate on the tile records in both dimensions. It also allows us to store tiles' old positions in one dimension and new positions in the other while we move batches of tiles. A global variable stores the current compaction direction so that the routines will know whether to look before or after a non-wire tile record for its companion record.

The adjacency lists algorithm uses seven pointers per tile record. Four pointers store the clockwise threaded adjacency lists. The left and right pointers point to a tile's bottom left and top right adjacent tiles and the down and up pointers continue these tiles' right and left adjacency lists, respectively. Two pointers are used to doubly link wires with endpoints. A wire's wire-up and wire-down pointers point to its two endpoints and their wire-down and wire-up pointers, respectively, point back to the wire. The final pointer, the alignment pointer, is used to circularly link the tiles in each object's set of tiles. In a one-tile object, the tile's alignment pointer just points back to the tile. The left, right, up, and down pointers give the spacing constraints needed to space adjacent tiles. The wire-up and wire-down pointers give the wire constraints needed to keep wires and endpoints fully connected. And the alignment pointers give the alignment constraints needed to force the tiles in each object to move as a unit.

The tree weight algorithm stores its directed spanning tree in a doubly-linked depth-first order list of tiles with the aid of a tree height field. The root of each dimension's tree, at the frame's lower left corner, has a height of zero. Starting at a height i current tile, if the next pointer points to a height $i+1$ tile then that tile is the current tile's first child. Otherwise, the tile is the next child of one of the current leaf tile's ancestors. The last leaf tile is linked with the root to make a circular list. This method of storing a tree makes it easy to scan all of the tiles in a subtree. Starting at a height i subtree's root, we just follow next pointers until we find a tile with height i or less. This will be the first tile that is not a descendant of the subtree's root. The list is

doubly linked so that we can remove a tile or subtree from the spanning tree without having to search for the tile whose next pointer needs adjusting. Such searching would ruin our $O(n^2)$ worst case compaction time.

There is also a parent pointer that, combined with the tree height field, makes it easy to find a pair of tiles' highest common ancestor. We follow parent pointers from the higher of the tiles until we have two tiles with the same height. Then we follow their parent pointers until they meet at the desired ancestor. A tile's set of children tiles should be a subset of the tiles to which it has constraints. The parent pointers, therefore, also allow us to find all of a tile's valid children without following next pointers: we just check the tiles in its constraining set for parent pointers pointing back to it.

The tree weight algorithm uses three more fields. Each subtree's weight is stored in its root's weight field. This signed weight is the sum of the direct wire pulls on a tile and the weights of all its children tiles. A subtree's tiles must be marked when they are broken off the spanning tree so that we can check constraints between subtree and main-tree tiles while ignoring subtree to marked subtree constraints. Instead of a separate boolean field, we can just encode a temporary mark into the sign bit of a nonnegative field such as the tree height or top edge coordinate. We can use the same trick to mark which tiles still need processing during batch compaction. Finally, when non-wire tiles are moved in one dimension, their positions and adjacencies need to be updated in the other. The select pointer is used to create a linked list of the moved tiles. There is a global pointer to the last tile added to this reverse-order list and a dummy trailer record so that all listed tiles will have a non-nil select pointer. Consuming one dimension's list creates a new list for the other dimension.

In total there are 11 pointers and 7 numeric fields per tile. With 32-bit pointers and 16-bit integers this gives 58 bytes per wire tile and 116 bytes per non-wire tile pair.

4.4. Compaction

To compare the various compactions models, our compactor does group, graph, tree, and incremental compactions. For the group compaction we use a flattened recursive algorithm. Compacting leftward, we group the tiles into groups as we recurse to the left and we place the groups as we recurse back to the right. Starting at the right edge, we use one pair of pointers: L and R. R steps through the tiles in a group and L steps through the left adjacencies for each R tile. When L comes across a tile not yet in

a group, it follows (breath-first) the alignment, wire-up, and wire-down pointers to find and circularly link together all the tiles that belong in the group. This L group must be placed before the R group can be; a pointer in tile L saves R, R is moved to L, and L moves leftward. When R finishes stepping through a group, all the group's left neighbors will have been placed so a second pass is made to find the group's left-most legal position. Then R is moved rightward back to the saved value and finishes processing that group. We number the groups so that intra-group spacing constraints (say between parts of a transistor) can be ignored. A dummy, single-tile group is created just left of the bottom left corner to stop the leftward recursion and the right-most group saves a pointer to itself to stop the rightward recursion.

The graph compaction is simpler, though slightly slower, because we do not have to create the groups. Starting with the group solution, we use a shortest path algorithm to reduce the constraint slacks and produce a smaller layout -- the tiles are moved in a least-movable-first order. A doubly-linked bin queue with dummy headers is created with the number of bins equal to the current width (leftward compaction) of the layout. All the tiles are placed in the maximum bin and the lower left corner tile is moved to the zero bin. A counter, starting at zero, steps through the bins and a pointer steps through the entries in each bin. Each entry is moved the counter distance to the left and its alignment, wire-up, wire-down, and right adjacencies are checked to see if its placement restricts any tile's movement. Any tiles thus restricted are requeued into lower bins. A tile's current bin value (slack), required for the comparison, is stored in the tile's height field. There is no need to remove entries from the queue or worry about which tiles have moved or not. When a tile is queued we set its parent pointer to remember which tile restricted its leftward movement and when a tile is finally moved we use this pointer to set the tree next, previous, and height fields (for use in the tree compaction).

The tree compaction starts with the tree created by the graph compaction. A weight is added to the upper right corner to keep the wire-length minimization from enlarging the layout. L and R pointers depth-first step through the tree to calculate the subtree weights and repair the negative subtrees. L starts at the tree root, R is set to L.next, and we loop. When R's parent is not L, L is a tree leaf or an interior node whose children have all been processed. Thus we can finish calculating L's weight by adding its direct wire pulls, if any. If the result is negative and L's parent constraint is not an alignment or equal-width wire constraint (constraints that can hold positive or negative weight), we break off L's subtree and move it upwards. Otherwise we just add

L's weight to its parent. In either case, we set L to its (old) parent and continue processing parents until L reaches R's parent. Now R's weight needs to be calculated before L's can be. If R is the root of a subtree that has already been processed (and moved and attached at L), we just scan R forward to find the next unprocessed tile. Otherwise we move L and R to R and R.next. The loop continues until L reaches the tree root's nil parent.

The routine to move a negative-weight subtree calls four subroutines. First it calls snap to disconnect the subtree from the main tree and make it into a separate, circularly-linked fragment with a nil root parent pointer. Snap also sets the snapped tiles' mark bits. The move routine then calls delta, cut, and graft in a tight loop. Delta finds the remaining minimum constraint slack between the marked fragment and the unmarked main tree -- the distance the fragment has to move to hit the tree. Delta is passed the old minimum slack and a pointer to a fragment tile. Starting at the tile, if it finds a tile with the old minimum, it can immediately return since the new minimum cannot be smaller than the old minimum. Otherwise, it will scan through the whole circularly-linked fragment to find the new, larger minimum. The previous-entry pointer is a compromise between using a full priority queue and constantly checking all the fragment's constraints. Delta returns the minimum slack and the (first) corresponding pair of fragment and hit main-tree tiles. Since semi-merged tiles can produce violated over-restrictive adjacencies, delta is careful not to return a negative slack, which would move tiles the wrong way and possibly produce real violated constraints.

Cut determines if the weight needed to stop the fragment's movement can be distributed along the path through the two hitting tiles without illegally changing the sign of any subtree weights. Cut makes two scans. First it scans down any processed parents of the hit main-tree tile to look for the smallest positive weight smaller than the absolute value of the subtree's remaining negative weight. Then its scans down the branch from the fragment's hit tile to its root's nil parent looking for the greatest negative weight greater than this weight (closer to zero). Cut returns the weight that can be distributed and, if restricted, which tile and branch limited it. In case of ties, the first scan picks its first found tile, the second scan picks its last found tile, and the second scan wins over the first. This order prevents negative-zero weights and, as we will shall see, corresponds to moving as few tiles as possible. Cut can return immediately when a zero weight is found during the first scan since the second scan, which has priority, looks at negative weights and thus cannot find a zero. It is important to the $O(n^2)$ worst

case time that we return early and not repeatedly make useless scans through the fragment.

If cut returns a nonzero weight, the move routine adds the negative weight to the first branch, subtracts it from the second branch, and reduces the subtree's remaining undistributed weight. Then it handles the three possible cut cases. If all the weight has been distributed, it grafts the fragment back onto the main tree and returns. Otherwise, it cuts the weak, restrictive branch by setting the returned tile's parent pointer to nil and then grafts the cut tiles onto either the fragment or the main tree, depending on the case. The last-found smallest-weight fragment tile has the highest priority since it cuts off the most fragment tiles. The last-found main-tree tile has the lowest priority since it adds the most tiles to the moving fragment.

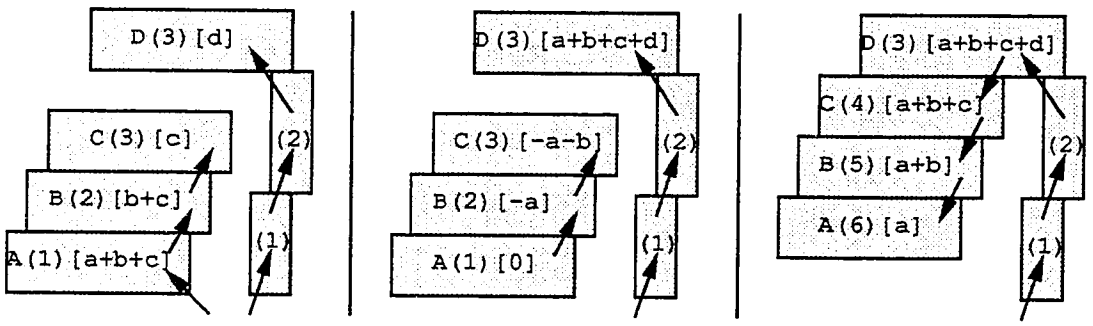


Figure 4-1 Graft Effects on (Height) and [Weight]

Graft is passed two tiles, the weight of the subtree that contains the first tile, and how far the first tile needs to move to hit the second tile. When cut tiles are grafted to the fragment, we mark them and move them down so they press against the fragment. When they are grafted to the main tree, we unmark them and move them up to their correct, final positions (there is no need to update positions for zero-distance moves). This prevents the fragment tiles from being repeatedly moved upwards. When the first tile, F, is not the root of the cut subtree, we have to flip parent pointers in order to graft tile F onto the second tile, P; we make tile P tile F's parent, set P and F to F and F's old parent, and loop until F becomes nil. At each step, we remove F's subtree from its containing subtree, if one, and splice it into P's subtree. We add *i*=P.height-F.height+1 to update the heights of the first step's tiles and *i*+=2 in subsequent steps. The first F gets the passed subtree weight and subsequent F's get the negative of the previous F's old weight. Figure 4-1 shows the effects on three cut tile's (heights) and [weights]: weight is first redistributed and then the cut is grafted. The individual weights of tiles

A-D are *a-d* and are all positive except for *a*, which satisfies *b+c < -a <= b+c+d*.

4.4.1. Incremental Compaction

The subroutines to incrementally move subtrees are slightly more complex. Delta is modified to find the minimum slack for either upward or downward movements. Graft is modified to create a list of all the non-wire tiles that it moves a nonzero final distance. We also need a new, more complex cut routine: it scans four spanning-tree branches instead of just two. One extra branch comes from having to subtract a subtree's weight from its old ancestors. In batch compaction, this step was unnecessary since subtrees were disconnected before their weights had been added to their ancestors. The other extra branch comes from two-way movements: removing a constraint between two tiles will unbalance them and may cause both of them to move towards each other. The first tile's movement is as before, but when it stops and the second tile is broken off, this tile might not be the root of its containing fragment. Thus the weight distribution in a fragment can no longer just go from the hit tile to the root.

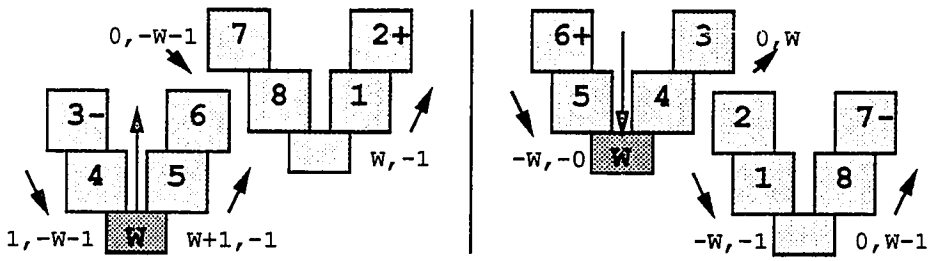


Figure 4-2 Cuts: Negative and Positive Fragments

Cut has to search for the highest common ancestor of the hit tile and the unbalanced tile in both the fragment and the main tree. As it scans these two pairs of branches, it looks for the tile which most restricts the desired weight distribution. Figure 4-2 shows two examples: one with a negative-weight fragment and one with a positive-weight fragment. The dark tiles are the roots of the fragments, which want to move up or down, respectively. Tile positions in each subtree correspond to tree height, not relative tile placement. In the first example, we distribute negative weight W between tiles 3 and 2 through tile 6 which just hit tile 7. We add negative weight to the tiles in branches 3-4 and 7-8 and subtract negative weight from the tiles in branches 5-6 and 1-2. The positive example is similar. The tiles are numbered in the priority order used to break smallest-weight ties. This order corresponds to moving as few tiles up

and as many tiles down as possible as required to prevent negative-zero weights. Figure 4-2 also shows the inclusive range of weight values that have to be checked for each branch. Cut can again return early when it finds a zero weight; in the negative case the main tree must be scanned first and in the positive case the fragment must be first.

The incremental move routine has to handle the five possible cut cases, corresponding to weight distribution restrictions in the four numbered branches in Figure 4-2 or to a full weight distribution. In the negative example, the no restriction and the 5-6 and 7-8 branch cases are the same as the batch move's: a graft and return or a cut and graft. In the 3-4 case, the weak link is snapped and the rest of the fragment is grafted to the main tree -- the fragment, with a new root, continues moving. In the 1-2 case, the whole fragment is grafted to the main tree and the weak link in 1-2 is snapped. The brand new fragment has a positive weight and thus moves in the opposite direction. While it is moving, any zero weights in the equivalent of the positive example's 5-6 branch will be negative zeroes. The range of weights checked in that branch includes (negative) zero so all zeroes will be cut or flipped before the fragment stops moving. The positive to negative fragment switch is similar, by symmetry, except that zero cannot occur in the negative 3-4 branch. The various W and W off-by-one range boundaries allow zero-weight fragments to continue moving down to find active constraints but prevent them from moving up.

After a batch of left-right movements, graft will have created a list of other-dimension non-wire tiles that need moving up-down (wires only exist in one dimension so do not need such updates). The list is divided into a list for down (left) movements and a list for up (right) movements -- any tiles whose positive and negative moves cancel are ignored. The up list is sorted highest-top first and the down list is sorted lowest-bottom first. Growing and shrinking tiles in this order prevents temporary tile pass-throughs. Exactly overlapping top and left endpoints are moved up before their bottom and right endpoints and down after. After fixing the adjacency lists, we fix the spanning tree. A list of shortened wires (wire-up from moved-up tiles, wire-down from moved-down tiles) and both their and the moved tiles' children is made. The tiles in this and the moved lists are checked to see if their parent pointers correspond to still valid constraints. The subtrees with invalid parents are incrementally moved, one at a time, to find valid parents. While we check the lists in one dimension, graft creates a new list of moved tiles in the other. We alternate between the dimensions until a spanning-tree fix does not move any tiles.

There is another interaction between graft and tree fix. Graft can flip parent pointers: tile F and its parent P can be switched to P and its parent F. If F's parent was invalid before the switch, P's parent will be invalid after the switch. Thus, if tile F is in the list to be checked and tile P is not, graft has to add P to the list. It is easy to modify the incremental move routine to handle tile insertion. To fix a constraint violated by a new tile, we tell the move to distribute a very large weight between the illegally overlapped tiles but put a limit equal to the overlap on the total distance subtrees can be moved. To insert a wire, we have to add a tile in one dimension (after forcing proper endpoint alignment) and distribute a weight between the two endpoints in the other. The weight distribution is like the incremental move except that in the first step there is no fragment to worry about -- just the two main-tree branches.

4.5. Results

To test the tree compaction system we ran it on a range of examples. To compare it with other compactors we used circuits from a benchmark suite [Boyer 87b]: a couple full adders, a channel routing, and various size multipliers. The circuits were hand converted from virtual-grid plots into sticks and entered using the Edc stick editor. The topology was unchanged except for slight modifications required because in Edc, wires are not allowed to cross transistor symbols or run parallel on other wires. Figure 4-3 compares the cpu and memory usages and the resulting layout sizes. The benchmark areas were converted to lambda units and their run times were doubled to compensate for our Micro-Vax 3200 (3 MIPS) and their Vax 8650. We used a slightly different set of design rules but it should have little effect on the lambda areas. The benchmark areas included protruding well regions so we similarly bloated our areas. The Tcmp time is the time actually used by our test compactor; using a better sequence of compaction steps, as discussed below, would, for example, reduce the time for the largest circuit, the 8x8 multiplier, from 151 to 99 CPU seconds.

The tree compactor, Tcmp, does very well. It is much faster than most of the benchmark compactors. The closest in speed are MULGA, a simple virtual-grid compactor, and DASL, a split-grid compactor. Our compactor uses a reasonable amount of memory -- about 111K plus the tile array. Symbolic's virtual-grid compactor and SPARCS, a graph-based compactor, have similar space requirements for some of the examples. Our compactor compacts flattened circuits while most of the other's take advantage of the hierarchy in the larger circuits. We produce the smallest circuits.

CELL *k* System	AREA XxY=A Lambda	Norm	CPU Sec	Norm	Memory KByte	Norm
AFAVG 1*k*						
Tcmp	85 x 94 = 7990	1.00	1.8	1.0	166	1.0
DASL	99 x 102 = 10064	1.26	3.2	1.8	379	2.3
MACS	95 x 97 = 9151	1.15	10	5.6	1390	8.4
MULGA	103 x 108 = 11088	1.39	3.2	1.8	243	1.5
SPARCS	105 x 101 = 10536	1.32	16	8.9	372	2.2
Symb.	103 x 103 = 10540	1.32	10	5.6	160	.96
Zorro	86 x 101 = 8624	1.08	1048	582	598	3.6
AFA 1*k*						
Tcmp	96 x 101 = 9696	1.00	1.9	1.0	167	1.0
DASL	102 x 124 = 12648	1.30	2.4	1.3	348	2.1
MACS	95 x 111 = 10550	1.09	18	9.5	1450	8.7
MULGA	115 x 128 = 14763	1.52	2.8	1.5	236	1.4
SPARCS	105 x 120 = 12560	1.30	22	12	356	2.1
Symb.	107 x 126 = 13440	1.39	10	5.3	164	.98
Zorro	94 x 114 = 10678	1.10	860	453	647	3.9
C132 5*k*						
Tcmp	430 x 206 = 88580	1.00	13	1.0	410	1.0
DASL	452 x 222 =100344	1.13	18	1.4	1332	3.2
MACS	418 x 236 = 98648	1.11	82	6.3	3000	7.3
MULGA	567 x 226 =128217	1.45	22	1.7	536	1.3
SPARCS	457 x 226 =103207	1.17	102	7.8	184	.45
Symb.	450 x 220 = 99000	1.12	104	8.0	1040	2.5
Zorro	440 x 215 = 94453	1.07	602	46	1413	3.4
Mul2x2 2*k*						
Tcmp	208 x 151 = 31408	1.00	3.7	1.0	218	1.0
DASL	226 x 172 = 38872	1.24	6.0	1.6	615	2.8
MACS	206 x 168 = 34608	1.10	32	8.6	2050	9.4
MULGA	244 x 169 = 41317	1.32	26	7.0	405	1.9
SPARCS	229 x 170 = 38873	1.24	94	25	215	.99
Symb.	247 x 180 = 44400	1.41	20	5.4	512	2.3
Zorro	208 x 168 = 34944	1.11	1676	453	614	2.8
Mul4x4 10*k*						
Tcmp	371 x 347 =128737	1.00	25	1.0	706	1.0
DASL	421 x 402 =169108	1.31	42	1.7	2292	3.2
MULGA	452 x 410 =185320	1.44	38	1.5	406	.58
SPARCS	433 x 401 =173355	1.35	132	5.3	754	1.07
Symb.	436 x 425 =185445	1.44	108	4.3	840	1.2
Zorro	385 x 385 =148097	1.15	3808	152	741	1.05
Mul8x8 49*k*						
Tcmp	735 x 743 =546105	1.00	151	1.0	2863	1.0
SPARCS	857 x 857 =733878	1.34	178	1.2	2066	.72
Symb.	851 x 901 =766734	1.40	490	3.2	3200	1.1
Zorro	759 x 805 =610727	1.12	23476	155	7754	2.7

Figure 4-3 Benchmark Results

MACS, with wire-length minimization and jog insertion, and Zorro, with zone refinement, do almost as well. Tcmp's advantage comes from performing multiple wire-length minimization steps.

The data for the plots in this chapter come from 14 test runs: an adder, four multipliers (2x2 to 8x8), eight arrays of simple static ram cells (2x2 to 16x16), and a channel routing. Representative CIF plots are given in Appendix C. We define Size s as $(n/1000)^{.5}$ for n-tile circuits -- proportional to the layout width and height. The eight ram cells are sized evenly from 1 to 7.4 ($1k$ to $52k$ tiles). The adder and multipliers have sizes of 1, 1.4, 3.3, 5.2, and 7.1, while the routing has size 2.3. Using arrays of small cells gives a good indication of the practical worst case effects of circuit size on run time. Plot sizes and data values are x and y sums unless otherwise stated.

Figure 4-4 shows how the time is divided among the various parts of the compaction. Tcmp first loads a stick file. Then it generates constraints and does group, graph, tree, and incremental compactions before writing a CIF file. The linear regression of the batch times give (in milliseconds per thousand tiles, k): load = $460k$+140, constraint generation = $165k^{1.07}$, group = $57k$, graph = $89k^{1.02}$, tree = $95k^{1.24}$, and CIF = $450k^{1.02}$. In comparison, a four-pass radix sort takes $29k$. Most of the batch time is spent reading and writing files. The graph compaction step is about 60% slower than the less flexible group compaction step. The tree compaction step takes approximately the same amount of time as the spacing constraint generation. The tree time will not equal the load time until circuits have over $700k$ tiles.

After the batch steps, we perform incremental steps to test the incremental features. The incremental times in Figure 4-4 are for an average of 2.8 tree-fix steps (1.8 adjacency fixes). Almost all the time (msec/ktile) is used during the first incremental step: tree = $105k^{1.62}$ and adj = $175k$ for all steps, tree = $32k^{1.1}$ and adj = 105 average for all but the first steps. The first incremental tree step fixes many invalid parents on the critical path and each such fix starts by breaking off an $O(n)$ size subtree; thus the best we could expect is an overall $O(n^{1.5})$ time. Adding all the test compaction steps together gives $1670k^{1.15}$ total time. Taking out the steps required only to produce the group results (group and an extra batch constraint generation) gives $1460k^{1.16}$. Replacing the first incremental step with a third batch step reduces this time to $1540k^{1.05}$ (doing a compaction with just two batch steps (x-y) gives $1290k^{1.03}$ but produces poorer layouts).

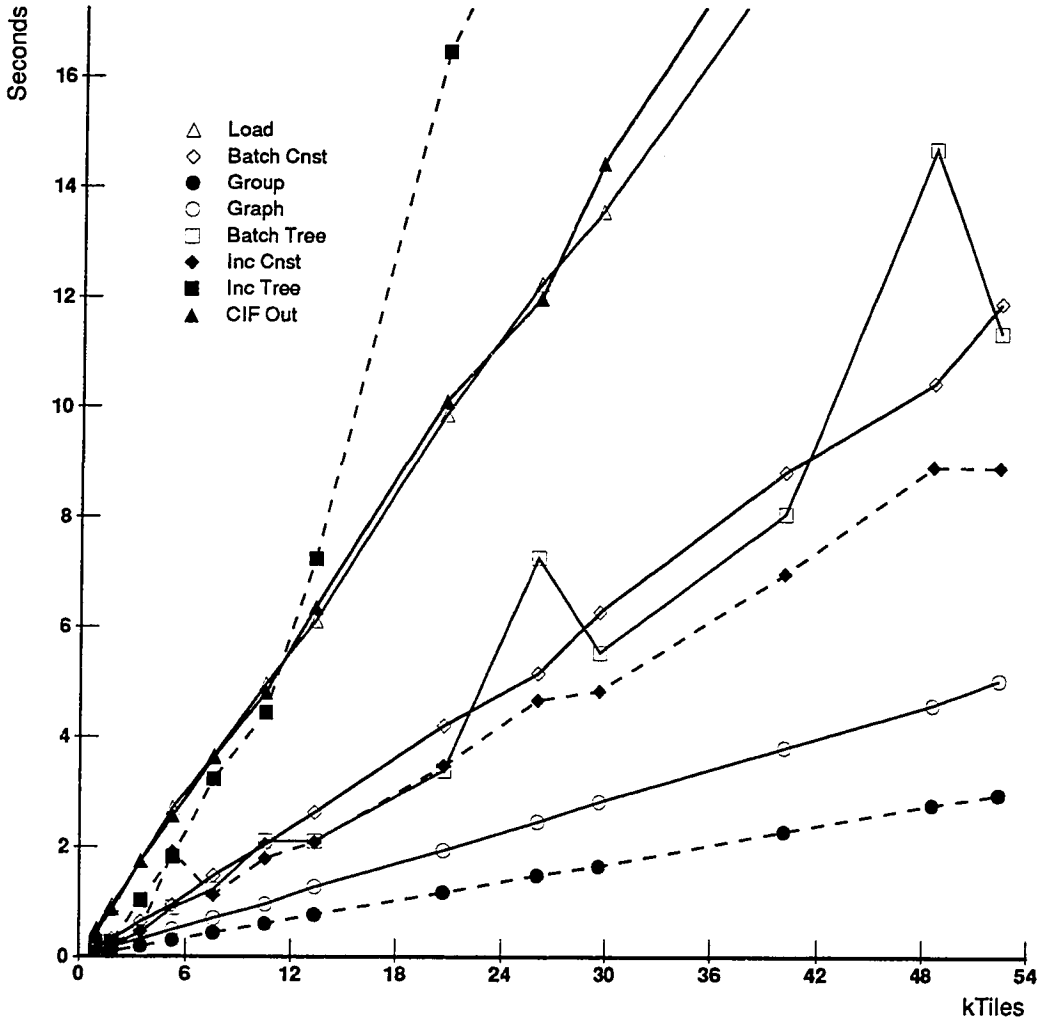


Figure 4-4 Compaction Timings

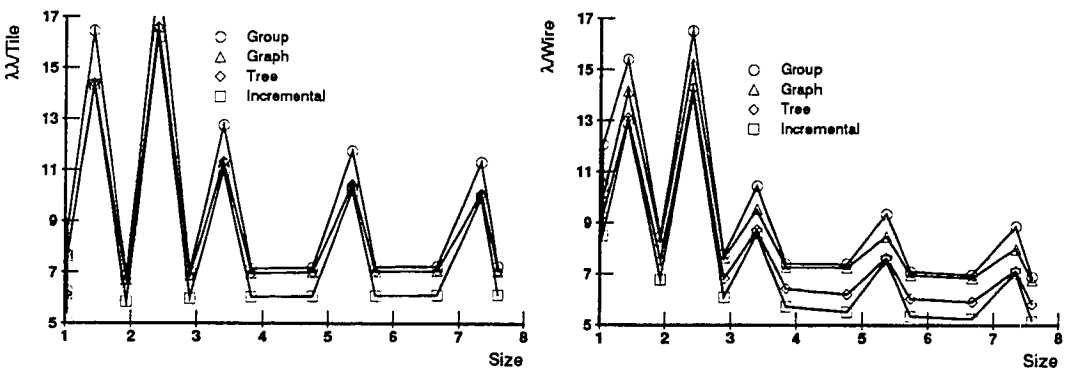


Figure 4-5 Area and Wire Lengths

Figure 4-5 shows the average area per tile and wire length after the group, graph, tree, and incremental steps. The large oscillations in the plot occur because the ram cells are about twice as dense as the other cells. Graph compaction is an improvement over group compaction: 9.7 to 9 λ^2/tile and 9.6 to 9.1λ/wire. Graph and tree have the same area since with the test order they use the same critical path, but tree's wire-length minimization shortens wires by 9% to 8.3λ. The incremental steps improve the layout by more than 6% to give 8.4 area and 7.8 wire length. The average density does not decrease and the average wire length does not increase as these circuits get larger.

4.5.1. Translations

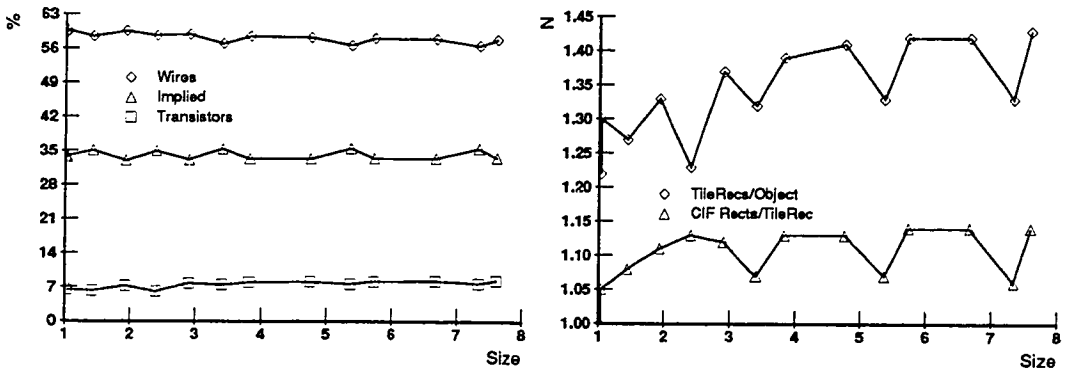


Figure 4-6 Object Types and Translations

About 70% of the single *x-y* tree compaction time is spent in file I/O. The stick file is in a fairly compact form; the only extraneous information concerns wire crossings (used by the stick editor display routines). It contains, on average, 58% wire objects, 7.5% transistor objects, and 34% implied objects. The implied objects are wire endpoints: single color points, contacts, and vias. Not explicitly describing the implied objects speeds up the load: while 80% of the time is spent on fscanf and 12% on translating that information into the tile records, only 8% is spent on creating and linking the implied objects. An average object is translated to 1.35 pairs of tile records (from 0.5 per wire to 4 per transistor). Writing the CIF file is even more I/O bound: 98% of the time is spent in fprinf. An average pair of tile records creates 1.1 CIF rectangles (from zero per duplicated diffusion tile to many per large cuts). There is a per tile average of 6.7 bytes in the stick file and 8.3 bytes in the CIF file. And there are 7.5%/(1.35*2) = 28 transistors per thousand tiles -- 2.8%(4*2) = 22% of the tiles belong to transistors.

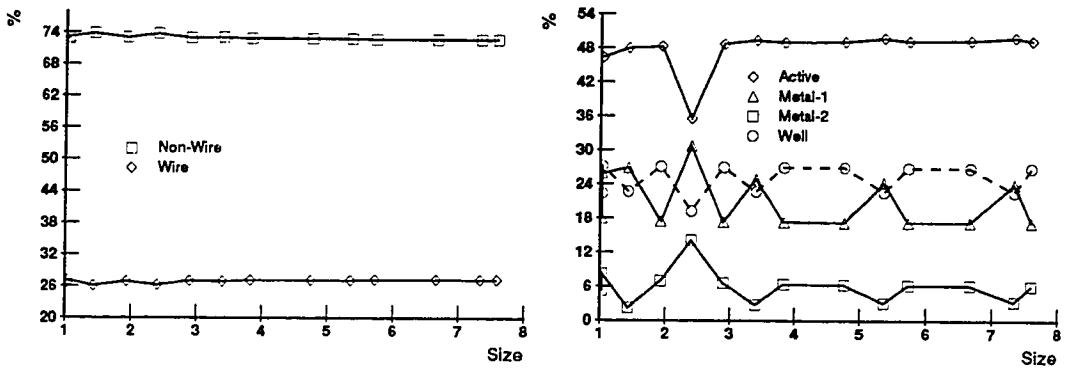


Figure 4-7 Tile Types and Planes

Using only one tile per wire allows the tile array to be 21% smaller -- 27% of the tiles are wires (the percentage of wire tiles is much smaller than the percentage of wire objects since non-wire objects create many more tiles per object). On the other hand, creating an extra layer of diffusion tiles to handle the well spacing rules makes the array 33% larger -- the well layer holds 25% of the tiles, compared to 48% in the active layer and 21 and 6% in the first and second metal layers, respectively. This is a fairly heavy penalty just to keep p- and n-diffusion tiles separate.

4.5.2. Adjacency Lists

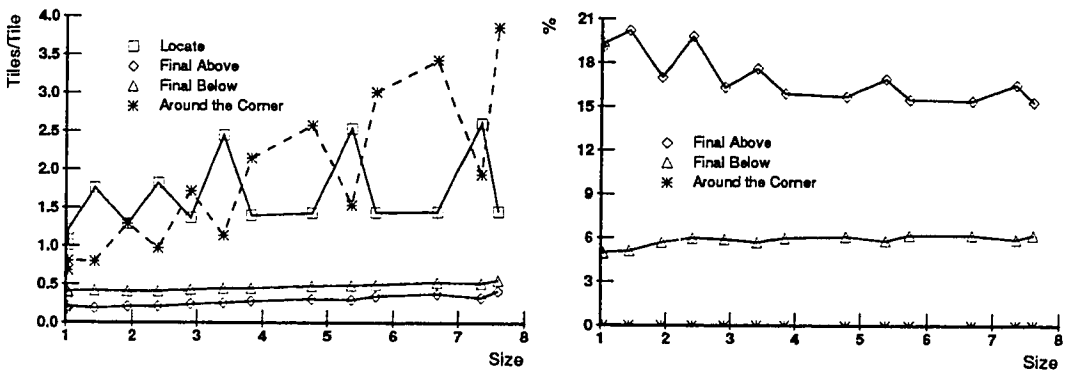


Figure 4-8 Quick Load Search Distances and Successes

Figure 4-8 shows the average number of tiles searched through to quick load a tile and the percentage of loaded tiles for which these searches find adjacencies. An average locate searches through 1.7 tiles and each of the left and left-up around-the-corner searches go through $0.4s+0.35$ tiles. This around-the-corner nonlinearity is caused by searches going along the edges of longer and longer busses in larger and

larger arrays of cells. The final above and below searches are much shorter (0.3 and 0.45 tiles, respectively); the values are small since final searches are not made for wires or for tiles shadowed by other adjacencies. The above searches find adjacencies more often (for 17% of the tiles) than the below searches (5.8% successful) since the above searches are done last and thus always find an adjacency for tiles with no prior adjacencies. The around-the-corner searches find a problem in well under 0.1% of the cases. The total of all the searches for a quick load is $800k^{1.5}+3150k$ tiles. In a 2500-tile circuit, half the searches are around-the-corner searches.

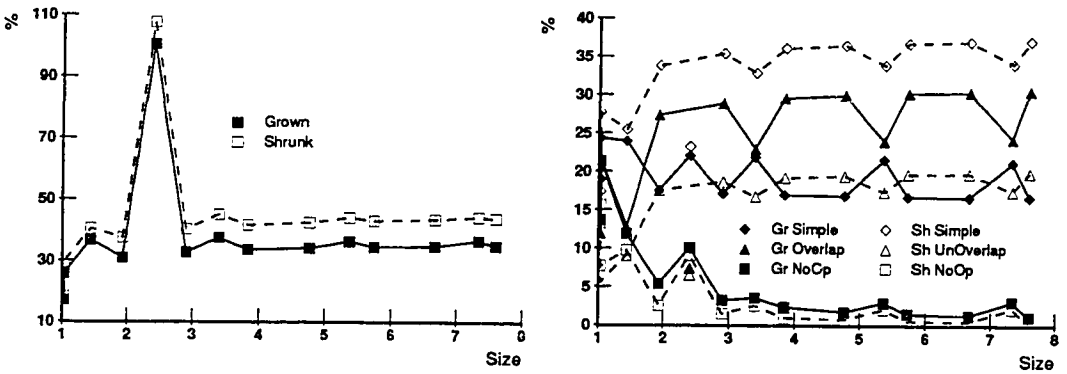


Figure 4-9 Tiles Grown/Shrunk and Types

Figure 4-9 shows that, during the incremental steps, about 37% of the tiles are grown and 44% are shrunk. More tiles are shrunk than grown (6:5) since shrinks that split a semi-merged set sometimes do an extra, zero shrink to properly split the set's adjacencies. The routing cell's movement value (at s=2.3) is large because it is the sum of two large sets of x-y incremental movements. Since only non-wire tiles are moved, at most 73% of the tiles could move per set. The figure also shows the percentages of movements that cause wires to grow or shrink. We save a lot of time by not separately growing and shrinking wires. There are three endpoint grow and shrink cases. In the simple case, adjacencies are just moved from the wire to the endpoint or back (19 and 32% of total grows and shrinks). The overlap/unoverlap case moves adjacencies and also merges/unmerges two semi-merged sets (24 and 16%). In the no-op case, only the amount of overlap within a semi-merged set changes (6 and 4%). This leaves 51 and 48% of tile grows and shrinks using the normal, more expensive loops and searches.

Figure 4-10 gives some statistics for an average grow and shrink. The incremental movements decrease the layout size; thus the average distance a tile is grown or shrunk ($4s$-1λ and $3.2s$-0.6λ) depends on the layout's width and height. The grow loop count is

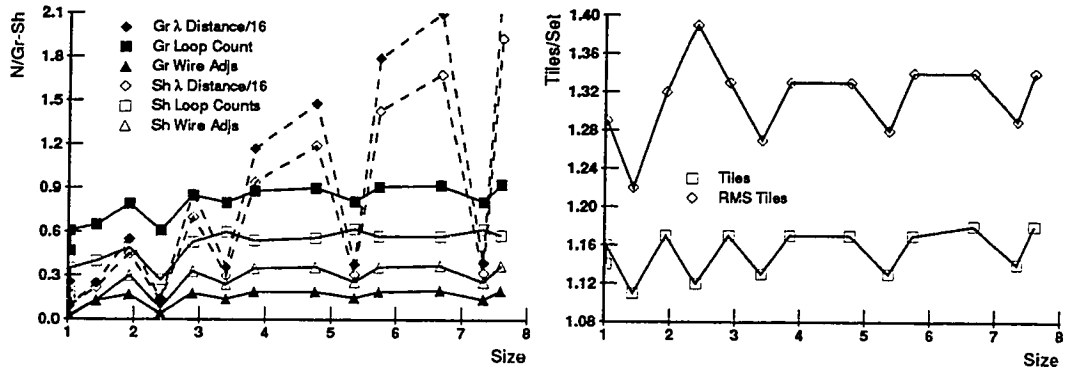


Figure 4-10 Grow/Shrink Stats and Semi-Merges

the number of counter-clockwise grow scans per grow (0.8) and the shrink loop count is the number of dropped adjacencies per shrink (0.5). These counts are small since only about half the movements perform loops (see previous paragraph). The other half of the movements just cause an average (per all moves) of 0.15 and 0.25 adjacencies to move from or to the stretching wire. Thus an average movement changes 0.95 and 0.75 of the moving tile's adjacencies. Also shown in the figure is that, after compaction, the average semi-merged set contains only 1.15 tiles and that the RMS size is a slightly larger 1.3. Thus our assumption that there are not large numbers of large sets was justified. The RMS value is important since two adjacent semi-merged sets containing *i* and *j* tiles will produce *i*j* constraints. Too large a value would give layouts more than 2 spacing constraints per tile.

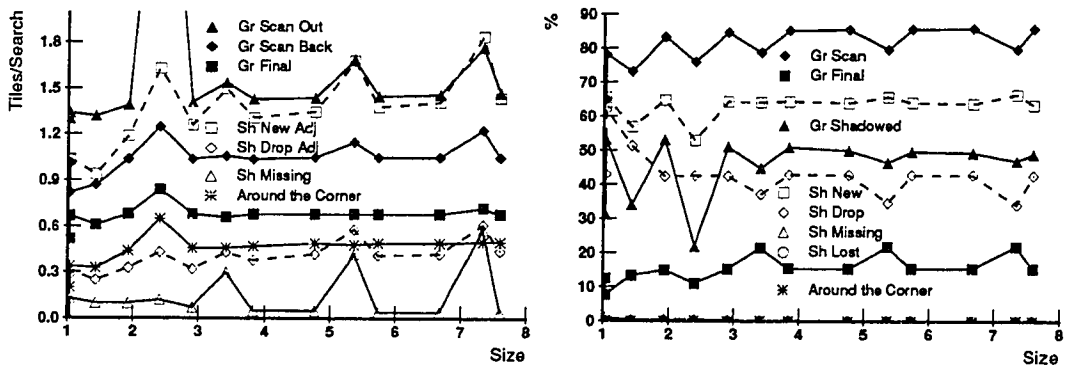


Figure 4-11 Grow/Shrink Search Distances and Successes

Figure 4-11 shows the average number of tiles searched through for grows and shrinks and the percentage of these searches that find adjacencies. Grow scans search out 1.6 tiles to find a non-nil up or down pointer before scanning back 1 tile (the larger, out value includes the up or down tile). An average grow's final left and right searches

together examine 1.3 tiles. When dropping an adjacency from a shrinking a tile, the searches for new adjacencies for the dropped tile and the shrinking tile go through 0.4 and 1.4 tiles, respectively. The first search is shorter since it starts one tile away from the shrinking tile while the second search starts at the dropped tile. The search for missing around-the-corner adjacencies to cache during shrinks is a rare 0.15 tiles per drop and the pair of around-the-corner searches after a grow or shrink together examine 0.9 tiles.

Multiplying the searches by their frequencies gives 4.4 and 2.1 checked tiles per grow and shrink -- 27% of these are around-the-corner searches. Since an average tile move causes 1 grow and 1.2 shrinks, this totals 6.9 searched tiles per move. Even though the grows examine about twice as many tiles as shrinks, grows do not take twice as long since the grow scans are simpler. 80% of the grow scans find a new adjacency and the grow final searches find an adjacency for 15% of the grown tiles. During grows, 45% of the new adjacencies shadow old adjacencies. During shrinks, 63% of the dropped adjacencies are replaced by new adjacencies and 43% of these dropped tiles get replacement adjacencies. Roughly 0.24% of the searches for a temporarily missing constraint to cache and 0.06% of the around-the-corner searches find problems. Almost all the cached constraints are quickly added to the adjacency lists. All together, this means an average grow and shrink actually gains and loses 55%(80%(0.8)+15%)+0.15 = 0.58 and 37%(0.5)+0.25 = 0.44 adjacencies (ignoring semi-merge effects).

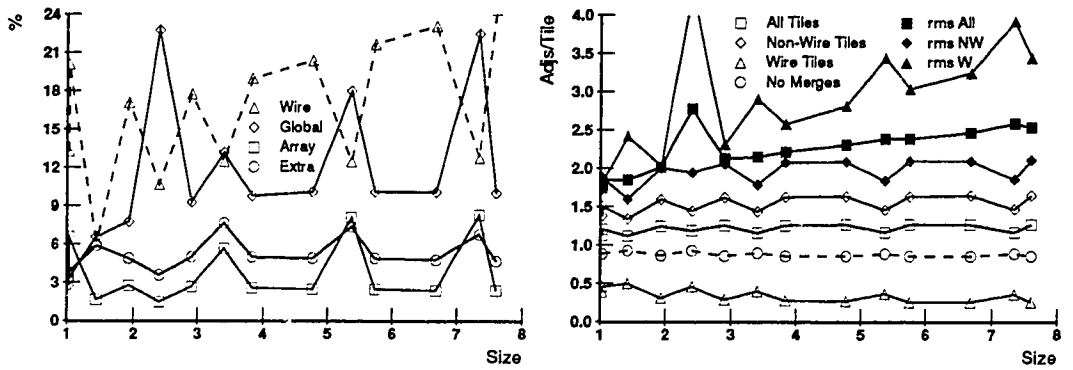


Figure 4-12 Shrink Skip Reasons and Adjacencies

Figure 4-12 shows how well the global position and the shadow array do in skipping the extra shrink adjacency drops caused by multiple colors. When a shrink lengthens a wire, the wire shadows the tiles on the left and the right of the shrinking endpoint from each other. This blocks the fuzziness problem for $1.1s$+12% of the tiles

searched through for new adjacencies for the shrinking tile. The global value that tells when all colors can be skipped allows 1.3s+6% of the searched tiles to be skipped and the shadow array with an entry for each color allows another 4% to be skipped. This leaves 5% of the searched tiles causing extra drops; the rest of the tiles stop searches in any case. Even though the shadow array prevents about 20% of the extra drops, not having it would cause only 4%(1.4*0.5) = 0.028 extra drops per shrink. The shadow array is initialized for the 48% of the shrinks that use the shrink loop and each extra drop would search through about 0.4+0.15 tiles for new adjacencies for the dropped tile. All together, this gives a savings of (0.028*0.55)/48% = 0.032 tiles per array initialization -- we would have done better without the shadow array. In tests, the array slowed the incremental adjacency times by a couple percent.

Figure 4-12 also shows various final average adjacency counts. There are 1.2 adjacencies per tile -- 0.35 per wire and 1.5 per non-wire tile. The RMS values are 0.1s+1.9 per tile, 0.23s+2 per wire, and 1.9 per non-wire tile. Many of the short wires have no adjacencies but many of the long ones have many; thus the relatively small average and large RMS wire values. The frame wires, with their $O(n^{.5})$ length lists, have a large effect on the RMS values. If we count adjacencies by the number of non-nil adjacency pointers, we get 0.9 per tile. The ratio 1.2/0.9 is equal to the RMS size of the semi-merged sets, as expected. In any case, our adjacency lists produce a very good set of constraints to use during compaction. Since the average list length is 1.2, searches do not have to lock far to find top left or bottom right adjacencies. And since the RMS list length is about 2.2, repeatedly removing the last adjacency in a list does not cause much trouble either.

4.5.3. Compaction

The group compaction groups together an average of 9.6 tiles per group and examines 8.9 spacing constraints per group. There are only 0.93 constraints per tile since intra-group spacing constraints are ignored and at first there are no semi-merged sets. The graph compaction examines each graph constraint exactly once: 1.17 spacing constraints, 1.07 wire constraints, and 0.78 alignment constraints per tile. Each wire (27% of the tiles) generates four wire constraints, two with each endpoint. One alignment constraint is checked per tile contained in multiple-tile objects; thus 22% of the tiles belong to single-tile objects. The graph compaction routine keeps the as yet unmoved tiles in a priority queue. When it finds a constraint that further restricts a

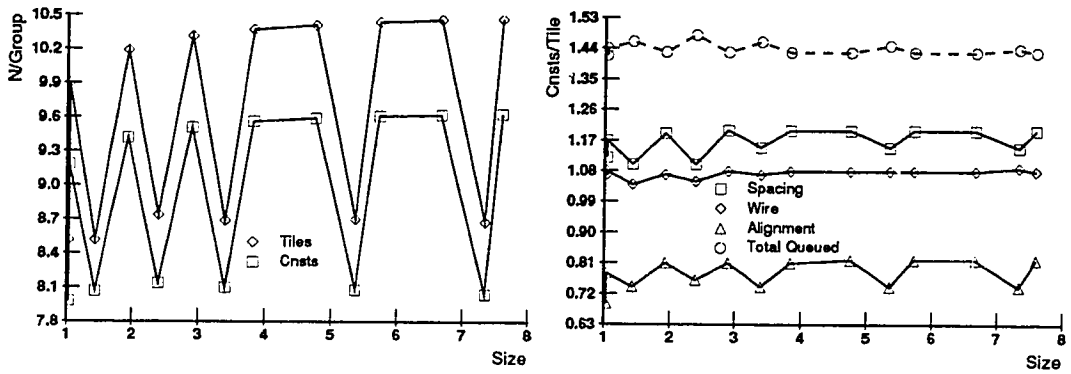


Figure 4-13 Group and Graph Constraints

tile's movement, it requeues that tile into a smaller movement bin. Of the 3 constraints checked per tile, 48% of them are more restrictive than previous constraints: an average tile is queued 1.44 times.

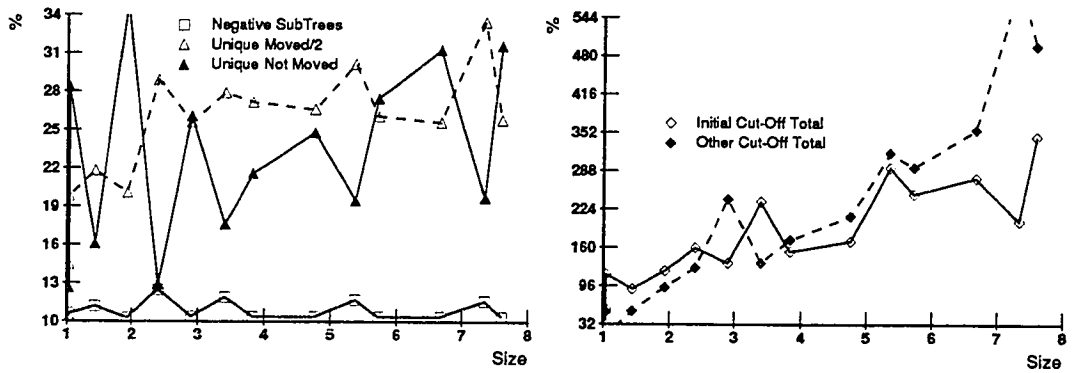


Figure 4-14 Tiles Affected Batch

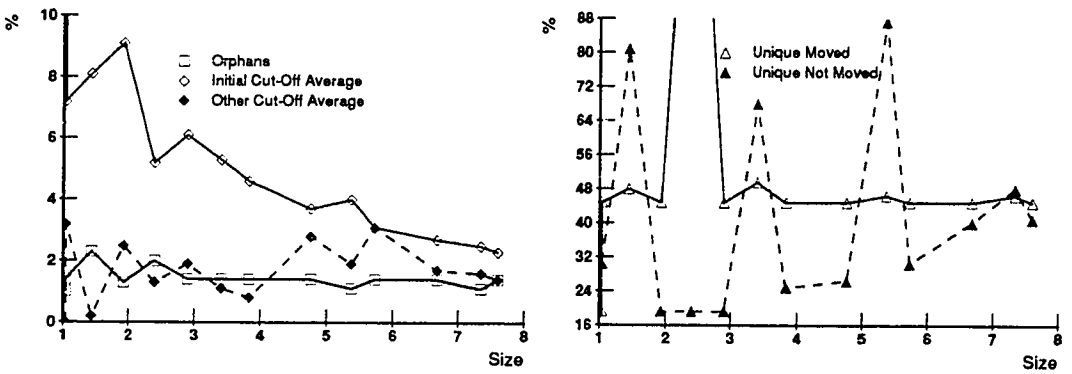


Figure 4-15 Tiles Affected Incremental

Figures 4-14 and 4-15 show the overall effects of the tree compaction. During batch compaction, 11% of the calculated subtree weights are negative and need fixing.

During these fixes, 74% of the tiles are affected: 51% actually move while the other 23% only move a zero distance (counting each tile at most once). In the larger circuits, unfortunately, many of the tiles are cut off (and moved or not) many times. This is the reason the batch compaction has a nonlinear run time. Multiple moves occur when a moving subtree hits and breaks off some already processed tiles or when a subtree is attached to an unprocessed tile that is later moved. Disconnecting the negative-weight subtrees from the main tree cuts off 33s+63% of the tiles (total). While moving, the fragments hit and cut off another 74s-52% of the tiles. During the incremental steps, only 1.4% of the tiles have invalid parent constraints. Disconnecting and moving a subtree to fix a parent cuts off 9.8-1.1s% and then 1.7% of the tiles (average) -- the number of tiles initially cut-off grows slower than n while the other cut-offs are proportional to n. Counting unique cut-off tiles gives 52% moved and 38% not moved. In both batch and incremental steps, the percentage of moved tiles is fairly constant regardless of the circuit being compacted but the percentage of not moved tiles is more erratic.

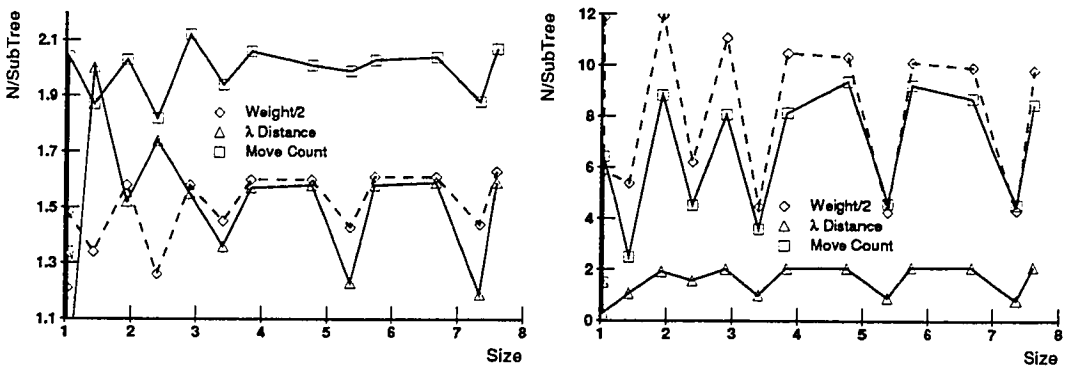


Figure 4-16 Batch and Incremental Stats

Figure 4-16 gives some statistics for an average subtree repair. The average batch repair calculates a negative weight of 3 and moves the subtree up 1.5λ in 2 steps. The average incremental repair moves a weight 16.5 subtree 1.5λ in 6.3 steps. Metal, poly, and diffusion wires have weight 1, 2, and 4, respectively. The average incremental weight is near the 16 units added to the critical path to insure minimum layout height (a small weight was added so it would not swamp out the averages). Thus most of the incremental repairs change the critical path. Dividing by the number of steps shows that average batch and incremental fragment movements move 0.75 and 0.24λ and redistribute weight 1.5 and 2.6 (to either break a branch or stop the fragment), respectively. Thus at least 25 and 76% of the movement steps move a zero distance.

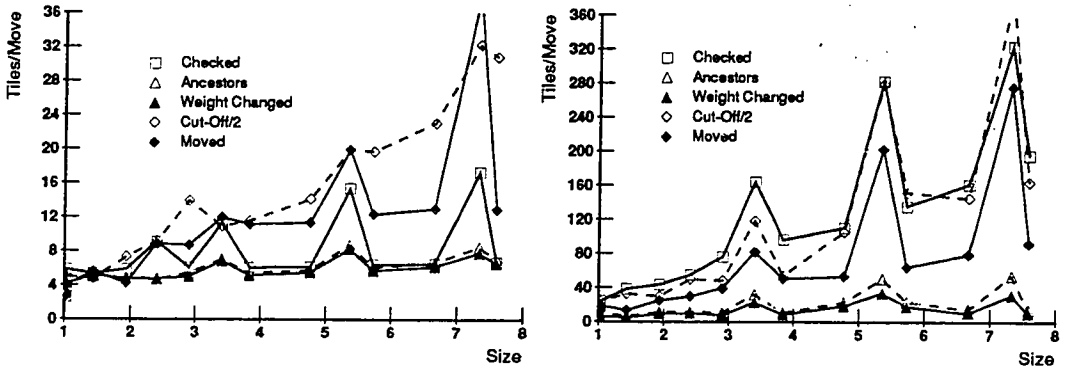


Figure 4-17 Batch and Incremental Moves

Figure 4-17 gives data on the average costs per fragment movement. An average batch step checks the constraints of $0.9s+4.8$ tiles to find the minimum slack and examines $0.6s+3.6$ ancestors to find the most restrictive weight. Since the ancestor search quits early when a zero weight is found, about 96% of the examined ancestors have their weight changed ($0.5s+3.8$). Of the $8.2s-1.6$ tiles that are then cut off, about 40% are moved a nonzero distance ($3s+0.6$). The incremental steps work on much larger fragments. They check the constraints of $36s-14$ tiles and the weights of $4.2s+3.8$ ancestors, but only about 70% of these weights are changed ($2.6s+4.2$). $78s-65$ tiles are cut off and about 50% of these are actually moved ($25s-18$). Note that the incremental steps check the constraints of a relatively large number of tiles and that following the adjacency lists to read the constraints is the most expensive of the movement operations.

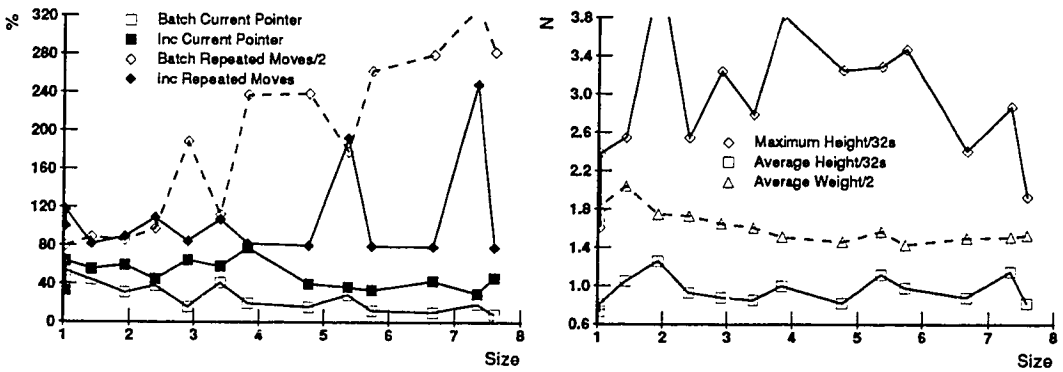


Figure 4-18 Tree Queue Effects and Stats

Finally, Figure 4-18 shows the effects of our improvements to reduce repeatedly checking and moving fragments. Using a current-pointer allows us to perform only 27

and 44% of the constraint checks that would have been required to fill a priority queue, batch and incremental respectively. The current-pointer allows the minimum-slack search to return as soon as the first zero (the minimum possible slack) is found instead of checking the whole fragment. This gain over queuing all the tiles overcomes the loss rechecking fragments during the rare, repeated nonzero moves.

Unfortunately, preventing repeated movements by moving tiles only as they are being marked or unmarked causes $78s$+68% and $7s$+83% of the moves that would have occurred just repeatedly moving the whole fragment. The repeated-move fix makes no difference when a subtree moves exactly once but loses, for example, when a single tile moves up one lambda, hits a big subtree, and then does a zero lambda movement. Repeatedly moving tiles would just move the single tile, but our fix also moves the big subtree down and then back up. The fix is not quite as bad as it looks since it only moves tiles that have to be scanned anyway (to mark/unmark and adjust tree heights) while the repeated moves would require extra whole-fragment scans. Thus our fix helps incremental movements but hurts batch compactions. Figure 4-18 also shows that the final average absolute tree weight is 3.3 and that the average and maximum tree height is proportional to the layout height as expected: $0.95n^{.5}$ and $2.9n^{.5}$.

4.6. Summary

Tree compaction can quickly produce small layouts without using too much memory (58 bytes per tile). Although the batch tree step has an $O(n^{1.24})$ time, the overall time for the test cases, assuming three batch steps followed by the needed number of incremental steps, is about $1.5k^{1.05}$ Micro-Vax 3200 CPU seconds (k=n/1000) since most of the time is spent in file I/O. Graph compaction takes slightly longer than group compaction but produces better layouts. Wire-length minimization reduces the wire length by another 9% and the final, incremental steps reduce the area and wire length by over 6% more. Using implied endpoints makes the load faster by allowing smaller stick files. Wires reduce the number of required tiles, but duplicating diffusion tiles to allow enforcement of the well spacing rules requires extra space.

The adjacency quick load works well: all of its searches are short on average except for the around-the-corner searches, which grow longer as the layout size increases. The semi-merges do not cause too much trouble. Wires help the incremental grows and shrinks: wires are never separately moved and only about half the non-wire moves need grow and shrink loops. About seven tiles are examined and about half an

adjacency is gained and lost per tile move. We used a shadow array to help skip the extra shrink adjacency drops caused by multiple colors. Although this reduces the theoretical worst case, it makes the expected case slightly worse. The adjacency lists algorithm produces a very good set of constraints: about 1.2 adjacencies per tile.

There are almost as many wire and alignment constraints as there are spacing constraints: about 1.1 and 0.8 per tile, respectively. During batch tree compaction, 11% of the calculated weights are negative and need fixing. The average batch repair fixes a negative weight of 3 by moving a subtree up 1.5λ in 2 steps. Many of the incremental repairs change the critical path; they cut off much larger and heavier subtrees and require many steps to redistribute the weight. Thus, even though in our test compactor only 1.4% of the parent pointers are incrementally repaired, it is faster to perform a third batch compaction step before starting the incremental steps. When moving fragments, most of the checked ancestors have their weight changed, but only about half the cut-off tiles are actually moved. Using a current-pointer requires fewer constraint checks than filling a priority queue since all the constraints do not have to be checked in the common, zero movement case. The fix to prevent repeated fragment movements helps the incremental case but, unfortunately, because of the very small number of repeated batch movements, it slows down the batch compaction.

The combination of wire-length minimization and multiple *x-y* compaction steps allows Tcmp to produce high quality layouts. Using the adjacency lists and tree weight algorithms allows it to perform batch compactions very quickly -- in near linear time. While the time and effort spent in incremental compaction by our test compactor appears large, one has to remember that the results were given for O(n) parent fixes, most of which changed the critical path. Even in the largest benchmark circuit, the 8x8 multiplier with 49k tiles, the average time required to fix one parent pointer is only 0.09 seconds and most interactive changes will break very few pointers. By comparison, it would take 43 seconds to recompact the whole circuit and it takes about 24 seconds to draw the entire stick diagram or CIF layout on the screen.

Chapter 5
Conclusions

Compaction allows designers to work at the stick level and not worry about all the detailed design rules. A major problem with one-dimensional compactors is the lack of control over the interaction between the two dimensions. An incremental compactor can quickly update a layout to reflect changes in the stick diagram or layout constraints. Since an incremental compactor efficiently propagates changes between the two dimensions, it can quickly perform multiple one-dimensional steps and it makes an ideal basis for two-dimensional compaction.

This dissertation has presented two algorithms to handle the two main tasks in an incremental compactor: generating and solving constraints. The adjacency lists algorithm quickly generates and incrementally updates a minimal complete set of the spacing constraints needed for compaction. The clockwise threaded lists allow us to move, insert, and delete tiles by growing and shrinking them. The algorithm takes advantage of the special properties of wires to reduce the required time and memory and to reduce the number of constraints generated. Not allowing tiles to pass through each other is a small price to pay to prevent the many pass-through problems. Using a direct-connect definition of electrically connected allows most of the legal tile overlaps. Unfortunately, it can prevent constructs such as L-shape wires from being compacted to their minimum legal size. Overlaps have little effect on the run time but require routines to merge and unmerge the semi-merged sets needed to keep from creating adjacencies that cannot be included in the planar adjacency lists. The semi-merged sets have the drawback of restricting the freedom of movement of tiles within sets since each set of tiles shares a single pair of adjacency lists.

Multiple colors have a much greater effect on the adjacency lists algorithm. When spacing rules can bleed through tiles (violate transitivity), we have to duplicate some of the tiles to put in an extra plane to handle the offending rules. The extra tiles require extra memory and extra time for processing. Within a plane, the varying tile bloats

caused by multiple spacing rules can create crossed constraints that cannot be included in the adjacency lists. Extra, around-the-corner searches are required to avoid missing these constraints. In large circuits, these searches make up a sizable fraction of the searches required to quick load the adjacency lists. The extra searches have a smaller effect on the incremental changes. In fact, the overhead of a simple shadow array to prevent extra shrink adjacency drops was far greater then the savings from preventing about 20% of the extra drops.

The second main algorithm, the tree weight algorithm, solves the constraints to produce a layout. It minimizes wire length by calculating the sum of the wire-pull weights on each subtree of a directed spanning tree of active constraints. The spanning tree allows us to perform the two basic operations needed for incremental changes: improving a layout to take advantage of the empty space created when tiles are moved or deleted, and creating enough room to legally insert new tiles. It tells us which way to move which tiles. To initialize the spanning tree, we first perform a simple graph compaction to create an estimate of the desired tree. The downward compaction reduces batch wire-length minimization to determining how far up to move each tile. A depth-first scan through the spanning tree finds subtrees with more upward than downward total wire pull, disconnects them from the tree, and moves them up to find something to hold them down.

A three-step loop is used to move subtree fragments. First, the distance the fragment must move to hit the main tree is calculated. Then the path through the hitting tiles is checked to see if it can hold the full weight of the fragment. If it cannot, then part of the fragment will break off and stop moving or part of the main tree will break off and start moving. Incremental fragment movements are more complicated than batch movements since they may go up or down and there are four branches (instead of two) that could be broken by the weight distribution.

A compactor has to handle the interaction between the adjacency lists and tree weight algorithms. When a compaction step moves tiles in one dimension, the adjacencies have to be updated in the other. This may cause some of the arcs in that dimension's spanning tree to no longer correspond to valid constraints. Incrementally moving subtrees to fix the invalid parent pointers will cause the first dimension's adjacency lists to need updating. This propagation of changes from one dimension to the other, until a fixed point is reached, allows the equivalent of an infinite number of *x*-*y* compaction steps almost for free.

5.1. Future Work

The major problem with the adjacency lists algorithm is its current dependence on the direct-connect definition of electrically connected. The only practical improvement is to use full netlists. Besides the simple problem of creating and updating the netlists, the algorithm has to be modified to merge and unmerge sets of tiles that, locally, seem to be unrelated. This is more difficult than semi-merging tiles that are directly connected by wires. It also makes it much harder to break left-right or top-bottom position ties when tiles exactly overlap. Such decisions are required to properly load or move a batch of tiles. A lot of time is spent handling multiple colors, but the current compromise between handling all the problems in one plane and eliminating all the problems by creating many extra planes seems to be optimal.

Much work remains to be done on integrating the stick editor and the tree compactor. Much of this involves devising a user interface that allows designers to easily make desired changes. The current compaction system allows one to move tiles by having the system add enough weight to break off a tree fragment. This has several problems. The first, which cannot be eliminated, is that movements cannot just move a single tile a certain distance: they have to move whole subtrees far enough to hit something. The second, related problem occurs when one wants to separate two tiles that are tightly held together in the spanning tree: moving one tile will cause the other to follow. This can be fixed by creating a command to add weight to one tile and subtract an equal weight from the other. The final problem is that the layout cannot be enlarged by just modifying weights: one dimension cannot be reduced at the expense of the other. If the top edge began moving upwards, there would be nothing to stop its movement. One fix would be to let the designer sweep out an area to fill with a dummy tile in order to push tiles apart. A better fix would be to add a command to change a pair of tiles' current x spacing constraint to a y constraint or vice versa.

Another integration problem concerns the mapping from stick diagram to layout (and back). Deleting the proper tiles from a layout when a stick object is deleted is easy. But determining where to insert tiles into a layout when a stick object is created is not as trivial. The problem is especially bad when existing tiles have to be moved to route a straight wire. One also has to decide which commands should be specified on the stick diagram and which on the layout. Since screen refreshes can be very slow, some way to batch layout changes is needed. Otherwise, every small change to a stick diagram might cause half the layout tiles to move a few lambda across the screen.

Appendix A
Semi-Merged Tiles

This appendix gives more details on how we semi-merge overlapping tiles to prevent adjacency pointer conflicts. Left and right overlaps in one dimension appear as up and down overlaps in the other. When the top of one tile overlaps the bottom of another tile, the two tiles may have to share parts of their left and right adjacency lists. In Figure A-1 for example, moving the tiles up and down creates two sets of three overlapping tiles. Spacing constraints should be generated from each of the left tiles to each of the right tiles. These nine constraints cannot be represented in the planar adjacency lists. To handle this we force top-to-bottom overlapping tiles to share all of their adjacencies.

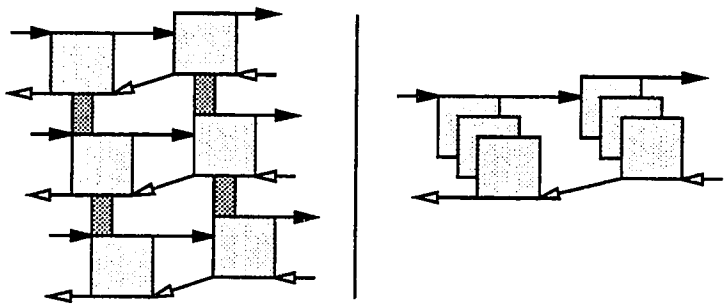


Figure A-1 Semi-Merging Tiles

We semi-merge the tiles so that an overlapped set has one left and one right adjacency list. The bottom-most tile's left pointer points to the start of the set's left adjacency list and the top-most tile's right pointer points to start of the right list. All of the rest of the set's left and right pointers are nil. This makes it easy to determine if a tile is in a merged set and, if so, if it is the set's top or bottom tile. A tile's up and down pointers point to the tiles that it overlaps, above and below. The top tile's up pointer and the bottom tile's down pointer are used to continue any adjacency lists which contain the set. All the right and down pointers pointing at the set point to the set's top tile and, likewise, left and up pointers point to the set's bottom tile. Thus, when

searching for turnarounds we can follow left or right pointers to find non-nil down or up pointers just as before. Figure A-1 shows the left and right pointers before and after the merges are performed.

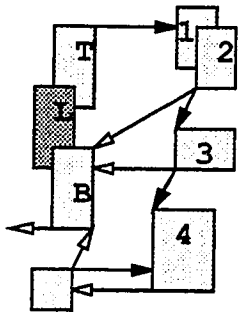


Figure A-2 Tile L's Four Right Adjacencies

Semi-merged tiles make it slightly harder to read the adjacency lists. To find all of a set's right adjacencies we first follow up pointers within the set to find a non-nil right pointer, the pointer to the first right adjacency. We also follow down pointers to find the set's bottom tile, the first tile with a non-nil left pointer. Starting at the first right adjacency, we follow down pointers until we find a tile with a non-nil left pointer that does not point to the set's bottom tile or we find a nil down pointer. If a tile's left pointer points to the set's bottom then, just as always, the tile's down pointer continues the right adjacency list. If a tile has a nil left pointer, it is in a set and the down pointers lead to the set's bottom. Starting at tile L in Figure A-2, we go up and down one to find the set's top and bottom tiles T and B. Tile T's right pointer gives us the first right adjacency, tile 1. It has a nil left pointer so we go down to tile 2. This tile's left pointer points to tile B so we go down to tile 3 and then, likewise, down to tile 4. Its left pointer does not point to tile B so its down pointer is not part of the list and we have found all four of tile L's right adjacencies.

A.1. Growing and Shrinking

It is not very hard to modify the grow and shrink routines to handle the semi-merged sets. Growing up a tile that is not the top tile in a set only changes the amount of overlap. Growing the top tile requires finding the set's bottom since pointers pointing to or from that tile may have to be modified. If a wire's bottom endpoint grows up far enough to overlap its top endpoint, the two endpoint sets will merge. Three adjacency lists on each side must be combined: one from each set and one from

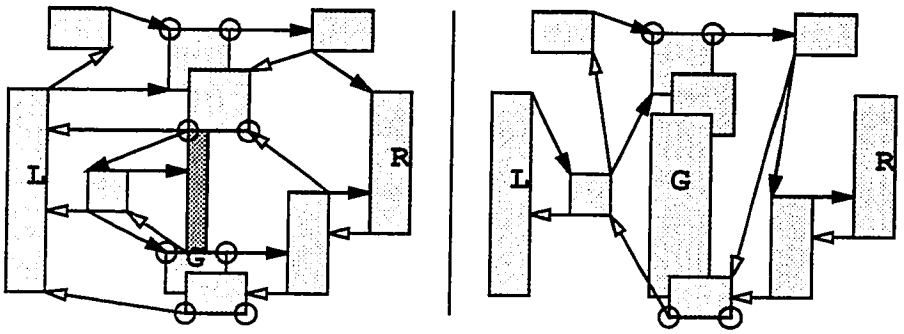


Figure A-3 Vanishing Corners: 8 to 4

the connecting wire. Pointers to and from the wire and its top and bottom endpoints must be modified to account for the combined set's single top and bottom. Adjacencies from each side may be shadowed. This occurs because the two individual sets had the equivalent of eight corners while the one combined set has only four. It is possible that a tile that would have caught on one of these vanished corners cannot hit the combined set because of intervening tiles. In Figure A-3, the relevant corners are circled. Tile R is adjacent to the top set before tile G is grown but is not adjacent to the resulting merged set. Tile L is adjacent to the top and bottom sets before their adjacency lists are merged with the wire's, but afterwards it too is not adjacent to the merged set.

The modifications to the shrink routine are similar. Shrinking up the bottom tile in a set is the same as before except for also modifying pointers to or from the set's top tile. Shrinking any other tile in a set only changes the amount of overlap unless it unoverlaps two tiles. Breaking a set is the reverse of merging two sets: each adjacency list is divided into possibly three lists (the connecting wire's lists may be empty) and then searches are made from the four new corners for possibly unshadowed adjacencies. The lists are divided by splitting them between the two new sets and then shrinking the top of the lower set down zero to transfer the proper adjacencies, if any, from it to the wire. We have to watch for the strange cases where inserting a wire makes the layout smaller by allowing two tiles to overlap and deleting a wire makes the layout larger by forcing two tiles to not overlap. Since a tile adjacent to a wire and its endpoint only gets the adjacency to the endpoint, we also have to make sure that a tile overlaping an endpoint does not also overlap and pass through the endpoint's wire and cause a spacing-constraint violation with a tile on the far side of the wire.

This method of semi-merging tiles can create over-restrictive adjacencies. Each tile in a string of barely overlapping tiles inherits all the set's adjacencies. If the tiles

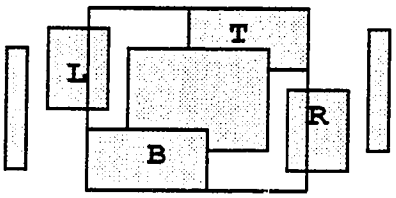


Figure A-4 Violated Over-Restrictive Adjacencies

are offset horizontally, some of the adjacencies created by a merge may be violated. When the three overlapping tiles in Figure A-4 are viewed as one big tile, that tile illegally overlaps tiles L and R. This does not cause much of a problem: we ignore the violations except to make sure that movements do not make them worse. Just throwing out these constraints would be equivalent to breaking a set into two sets without searching from corners for possibly unshadowed adjacencies. Completely ignoring the constraint between tiles L and B in Figure A-4 would allow tile B to slide leftwards and illegally overlap the far left tile -- likewise for tile T and the far right tile. We also have to make sure that the far left and right tiles do not move in to overlap tiles B and T. Note that overlapping tiles can still move independently: they are not glued to each other. In the example, tile B can safely move to the right and tile T to the left.

A.2. Quick Load

The quick load also has to be modified to handle overlapping and semi-merged tiles. Using the direct-connect definition of electrically connected allows us to follow wire connections to make sure that tiles are added in the correct order. We saw before that a vertical wire's addition may have to be delayed until after its upper endpoint's addition. In the case where a horizontal wire's left and right endpoints' left edges exactly overlap, the right endpoint's addition may likewise have to be delayed until after the left's addition to ensure that the right endpoint is properly placed on the right. A left endpoint is found, if there is one, by following two non-nil other-dimension wire-down pointers from the right endpoint. If the left endpoint is also a bottom or top endpoint, its up and down wires also have to be added before the right endpoint. Otherwise, wire additions would have to worry about breaking adjacencies between endpoints on their right and tiles on their left. Adding wires before adding right endpoints will shadow all the left tiles and prevent such adjacencies.

Figure A-5A gives an example of calculating the proper load order. The sort order

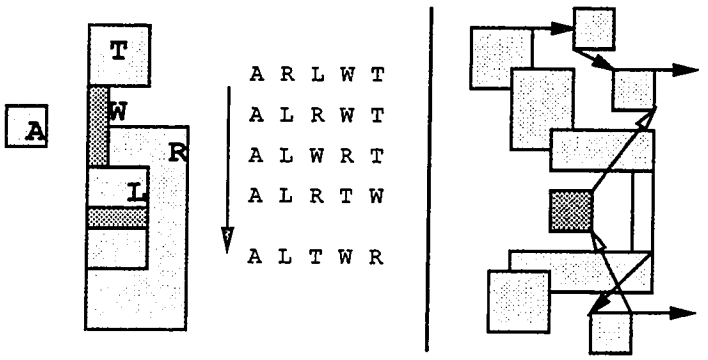


Figure A-5 A. Proper Load Order B. Shadowed Endpoints

would add ARLWT. This would put left endpoint L on the right of right endpoint R. Moving R to after L in the list fixes this problem and gives ALRWT. After adding ALR, tiles A and R would be adjacent and wire W could not be added without somehow breaking this adjacency. By moving R to after W we get ALWRT and keep the A-R adjacency from ever forming. Wire W cannot be added before its top endpoint T is added, so W is moved to after T to get ALRTW. Once again we have the ALR problem and move R to after W to get ALTWR, the correct order.

Overlapping tiles can also cause a wire's endpoint to be partially shadowed before the wire is added. When adding such a wire, we have to follow right or right-down pointers from the lower or upper endpoints, respectively, to find a tile adjacent to the right edge. Then the wire's left adjacencies, if any, can be found. In Figure A-5B, both endpoint sets are partially shadowed. Short searches (the dark arrows) reach the right edge and find the one dark tile to move from the edge's to the wire's left adjacency list.

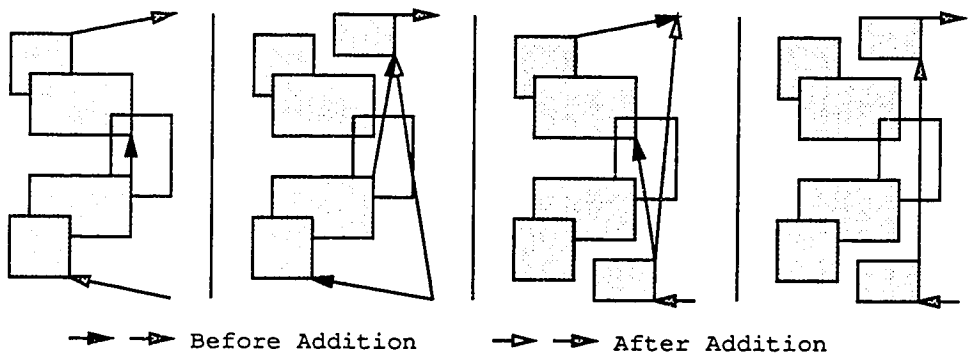


Figure A-6 Four Double Merge Cases

Finally, the tiles that should be semi-merged into a set are not added in any particular order. Thus we have to be prepared for an added tile to merge into a set its bottom overlaps, or into a set its top overlaps, or to cause above and below sets to be merged into one, or even to merge into the middle of an existing set. Besides the problems caused by vanishing corners, this is further complicated by the fact that the effect on the right edge's left adjacency list depends on whether the set(s) the new tile overlaps are in that left list or not. Figure A-6 shows the effects for the four possible cases of merging two sets into one. The dark arrows show the lists before the addition and the light after. There is one case of two adjacencies being merged into one, two cases of one adjacency being shadowed, and one case with no change.

Appendix B
Adjacency Lists

The chapter on generating spacing constraints gave a fairly detailed description of the adjacency lists algorithm. In this appendix we describe the variables used to efficiently store the current state in the top and bottom edge up routines. The two main incremental routines move tile tops or bottoms up. Their parameters are the tile to change and the new edge coordinate. The two routines to move edges down are copies of the up routines with up and right switched with down and left. These edge routines call simple corner routines to check for around-the-corner constraints in each of the four diagonal directions. The corner routines use a shadow height to cut short most searches. They search for a turnaround to bypass and, if not shadowed, search until a constraint or a second turnaround is found. If they find a missing constraint, they add it to a list for later processing. The edge routines are called by the routines which move, delete, or insert tiles.

B.1. Top Edge Up

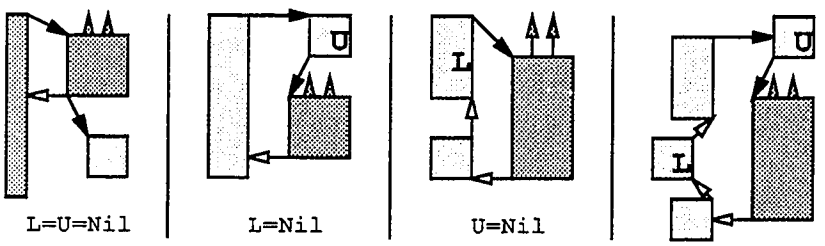


Figure B-1 Four L and U Nil/Non-Nil Cases

The main effect of the top edge up routine is to change the growing tile's left and right adjacencies. Left adjacency changes are more complex since they occur at the ends of adjacency lists. Two pointers are used to store this side's state: L points to the top left adjacency that cannot be shadowed as the tile grows up and U points to the tile whose down pointer points to the growing tile. The four nil/non-nil cases are shown in

Figure B-1. If the growing tile is a wire's top right adjacency then both pointers are nil. If it is not the first tile in its only left adjacency's right list then only L is nil. If it is the first tile in its top left adjacency's right list then U is nil and L points to this top left adjacency. Otherwise, U is non-nil and L points to the growing tile's second from top left adjacency. In every case, L and U point to the tiles whose pointers change when new left adjacencies are added. Because of semi-merged sets, two bottom pointers are needed to handle the right side: one for the growing tile's set and one for its top right adjacency's set.

When a growing tile gains a new left adjacency, either the growing tile's left pointer or the L tile's up pointer is pointed at the new adjacency, depending on whether L is nil or not. If U is nil and L is not, this simply adds a new adjacency. Otherwise, we have replaced an old left adjacency with a new one and have to remove the growing tile from the beginning, middle, or end of that old adjacency's right list (in the L and U nil, L nil and U not, or L and U non-nil cases, respectively). The new left adjacency's bottom right adjacency will always be shadowed. We remove the left tile from the end of this right tile's left list, point U at the left tile's second from bottom right adjacency and point that tile's down pointer at the growing tile. If the left tile had only one right adjacency, we instead remove the left tile from the middle of that adjacency's left list, point L at the left tile and point that tile's right pointer at the growing tile.

Right adjacency changes are simpler since they occur at the beginnings of adjacency lists. When a growing tile gains a new right adjacency we first check to see if the growing tile's up pointer is non-nil or if its right pointer points to a wire. If so, the growing tile's old top right adjacency will be shadowed and we have to remove the growing tile from the beginning or middle of this old adjacency's left list. Likewise, we also check if the new right adjacent tile's down pointer is non-nil or if its left pointer points to a wire. If so, the right tile's bottom left adjacency will be shadowed and we have to remove the right tile from the beginning or middle of this left tile's right list. If U is non-nil when we do this, U is pointing at the right tile which no longer points down to the growing tile. We point U at the tile above the right tile in the left tile's right list if there was such a tile. Otherwise, we nil U and if the left tile is not a wire, we point L at it (it is the growing tile's top left adjacency). In any case, we finish by adding the growing tile and the right tile at the head of each other's left and right adjacency lists.

B.2. Bottom Edge Up

The bottom edge up routine drops bottom adjacencies one at a time from the shrinking tile. The left adjacency list is in the correct drop order but we have to reverse the part of the right list that will be dropped. We change it so that following the down pointers from the second from lowest dropping right adjacency will lead to the second from lowest not dropping right adjacency -- properly handling the cases where there are not that many dropping or not dropping tiles. We use four pointers to keep track of the current bottom adjacencies. L and R point to the about to be dropped tiles and LU and RU point to the next tiles. When L moves to LU, LU moves to its up tile. When R moves to RU, RU moves to its down tile (the one above it). Because of semi-merged sets, two pointers are actually used, one top and one bottom, for each of the shrinking, L, and R tile sets. L's up pointer is kept pointing at LU so that when finished shrinking, the shrunk tile's left pointer can just be pointed at L. On the right side, when finished, either the shrunk tile's right pointer or RU's down pointer is pointed at R, depending on whether RU is nil or not.

We use six variables and two arrays to store the shrinking state. LT and RT give the coordinates the shrinking tile's bottom has to shrink past before the current L and R tiles will drop. One coordinate is updated after each drop. LM and RM mark whether there are any more tiles to drop from each side. They become false when LT or RT move above the final bottom of the shrinking tile or when the shrinking tile's only adjacency on a side is a turnaround tile or wire. LB and RB give the coordinates at or below which any color tile can safely be skipped by (not made adjacent to) the shrinking tile. One coordinate may move upwards after each drop -- depending on the dropped tile's least shadowing effect. LC and RC are arrays of skippable coordinates indexed by tile color. When a tile is dropped, the entry corresponding to its side and color is updated. During searches, an entry is checked only for tiles above LB or RB. LB, RB, and the LC and RC arrays are initialized to zero.

A right search from a dropped right tile has five possible outcomes. The search finds a new adjacency for the shrinking tile if it finds a non-wire tile high enough to be unskippable or if it finds a wire and RU is nil. In these two cases, R is pointed at the new adjacent tile and the shrinking tile is added to the end of this tile's left list. Otherwise, if RU is non-nil, R moves to it, RU and RM are updated, and a cached right around-the-corner constraint can be added to the adjacency lists. Otherwise the shrinking tile is down to a final right adjacency; RM is set false, R is pointed at the wire

or non-wire turnaround tile that stopped the right search, and the shrinking tile is added to the beginning of the wire's or middle of the non-wire's left adjacency list. The left-side search cases are similar except that at a wire turnaround, the shrinking tile is either added to the end of a wire's right list or becomes the only tile in a previously empty list. When we change L or R we also have to find the corresponding top or bottom tile in their semi-merged sets.

B.3. Quick Load

The quick load starts with a list of positioned colored tiles and creates the adjacency lists needed for compaction. It sets the tiles' left, right, up, and down pointers. The load first sorts the tiles on the (x,y) coordinates of their lower left corners. With 16-bit coordinates, this takes a four-pass, 256-bin radix sort. Each plane's frame is modified so that during the load, the two left corners will be adjacent to the right-edge wire. This allows these wires to be used as the load right edges without any first and last adjacency special cases. The sorted tiles are marked not added. Then we scan through the list adding tiles and marking them added. Before adding a wire, we check if its upper endpoint has been added. If not, we move the wire from its current place at the head of the list to directly after that endpoint. Before adding a non-wire tile, we check if its corresponding left endpoint and that endpoint's up and down wires have already been added. If one has not, we move the tile to after it in the list.

The routine to add non-wire tiles is broken into two parts. The first part checks if the tile's bottom overlaps an already added tile and then either merges the new tile into that set or locates the tile just below the new tile in the right edge's left adjacency list. The locate starts from an upper endpoint's already added lower endpoint or, if not one, the cached previously located below tile or, if it is too high, the right edge's first left adjacency. The locate continues upwards in two phases: first using a worst case bloat to get close and then using the actual bloats. From the located tile, a left-up search is done for a possible bottom left adjacency for the new tile. The second part of the add similarly checks if the new tile's top overlaps an already added tile and does the final, left from above adjacency search. It also repairs the right edge's left adjacency list after transferring the proper adjacencies to the new tile's left list. The quick load does not modify the wire-up and wire-down pointers since they are properly set before hand. This means, however, that during the load, wire-up and wire-down pointers pointing to not yet added wires must be treated as nil pointers.

Appendix C
Examples

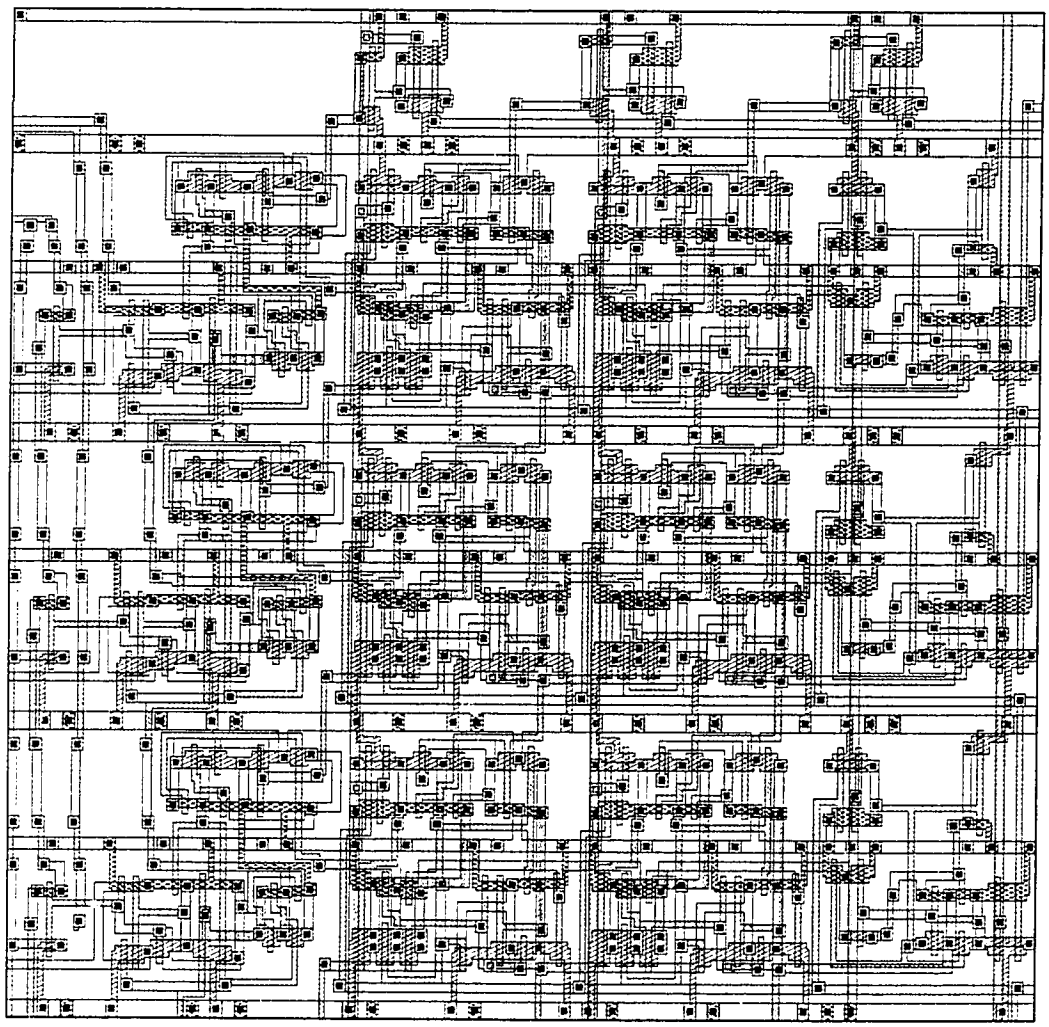


Figure C-1 4x4 Multiplier: 344x336λ, 10.5*k*

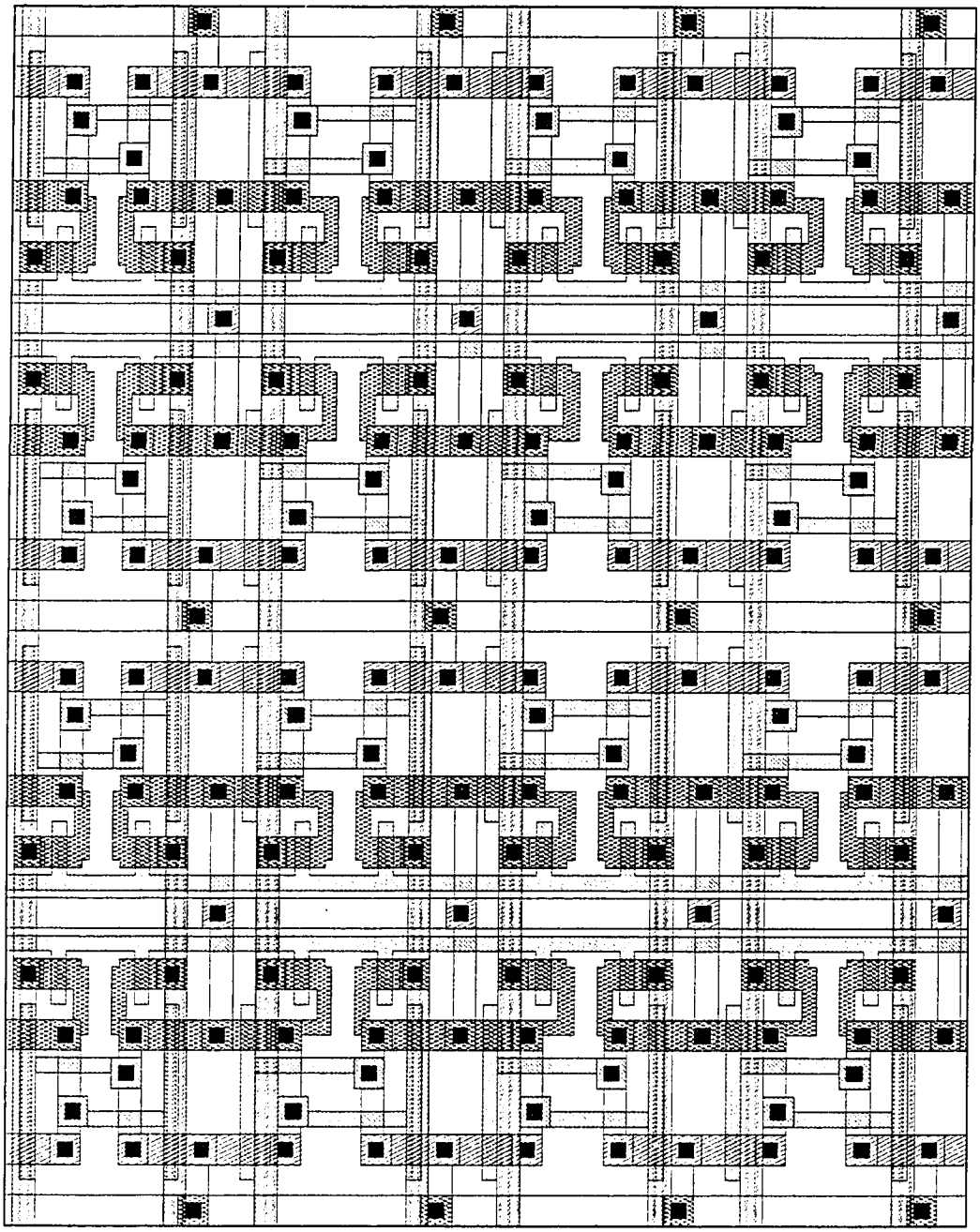


Figure C-2 4x4 Ram: 127x160λ, 3.5*k*

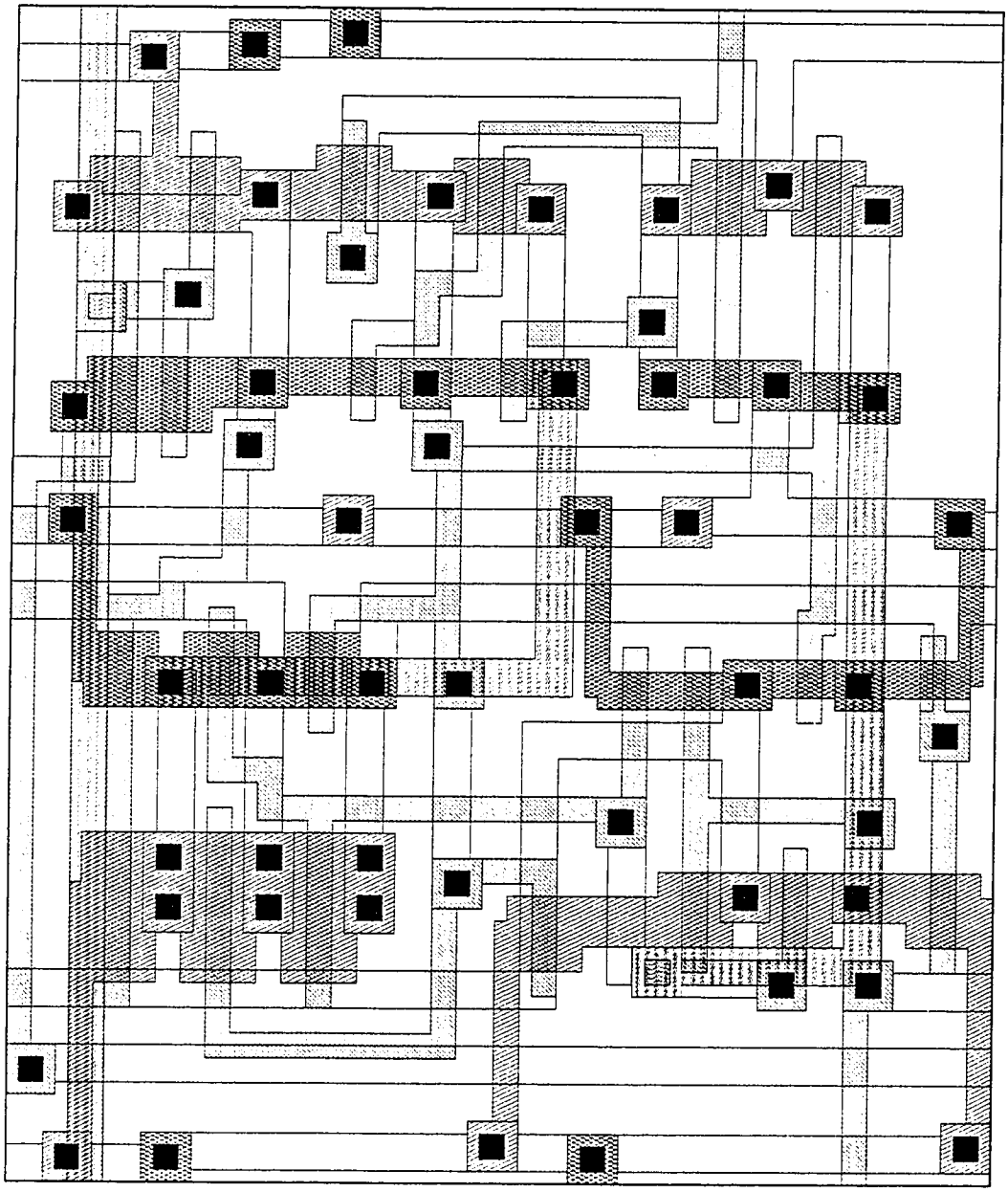


Figure C-3 Full Adder (afavg): 79x94λ, 1*k*

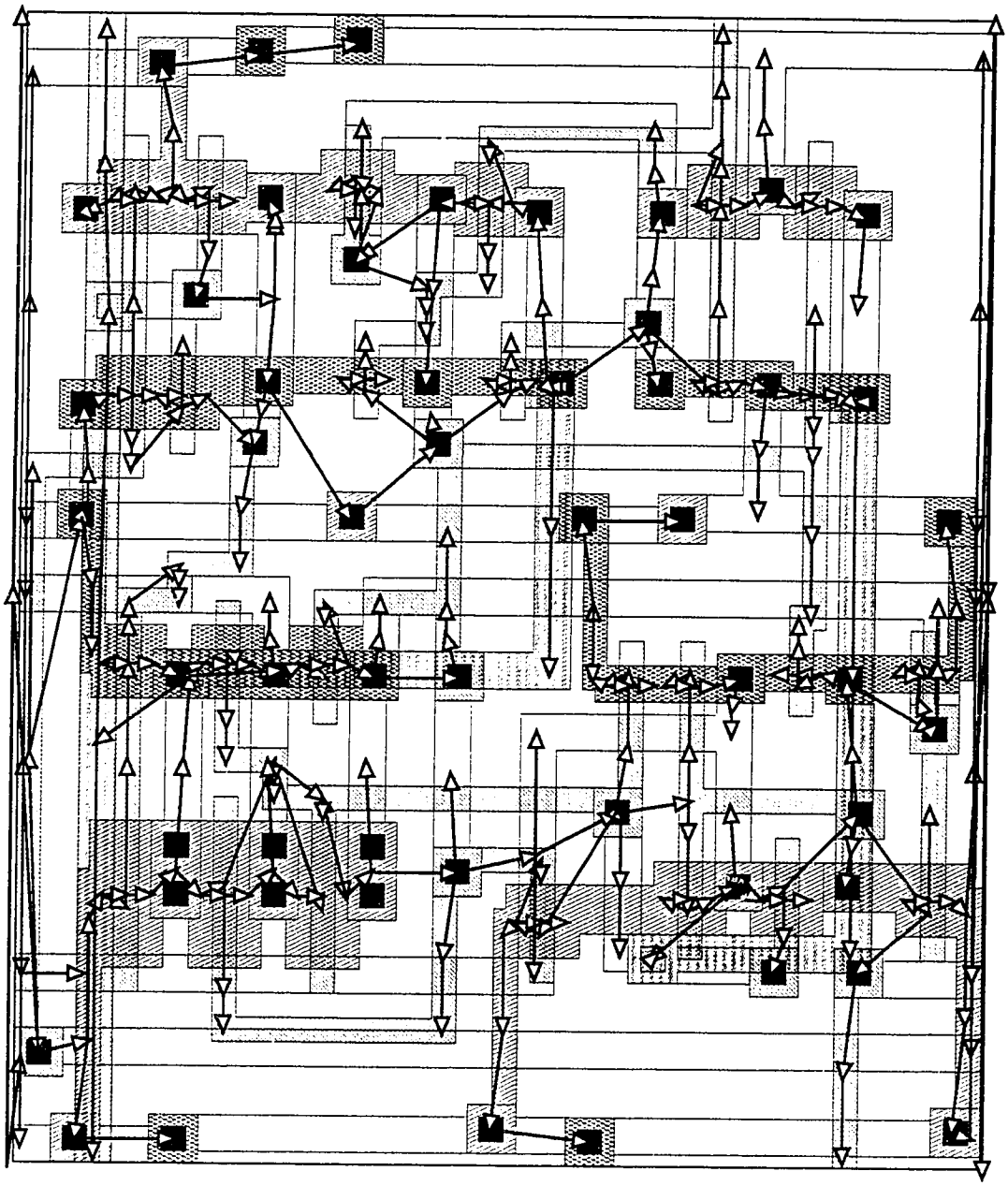


Figure C-4 Adder with Left-Right Spanning Tree

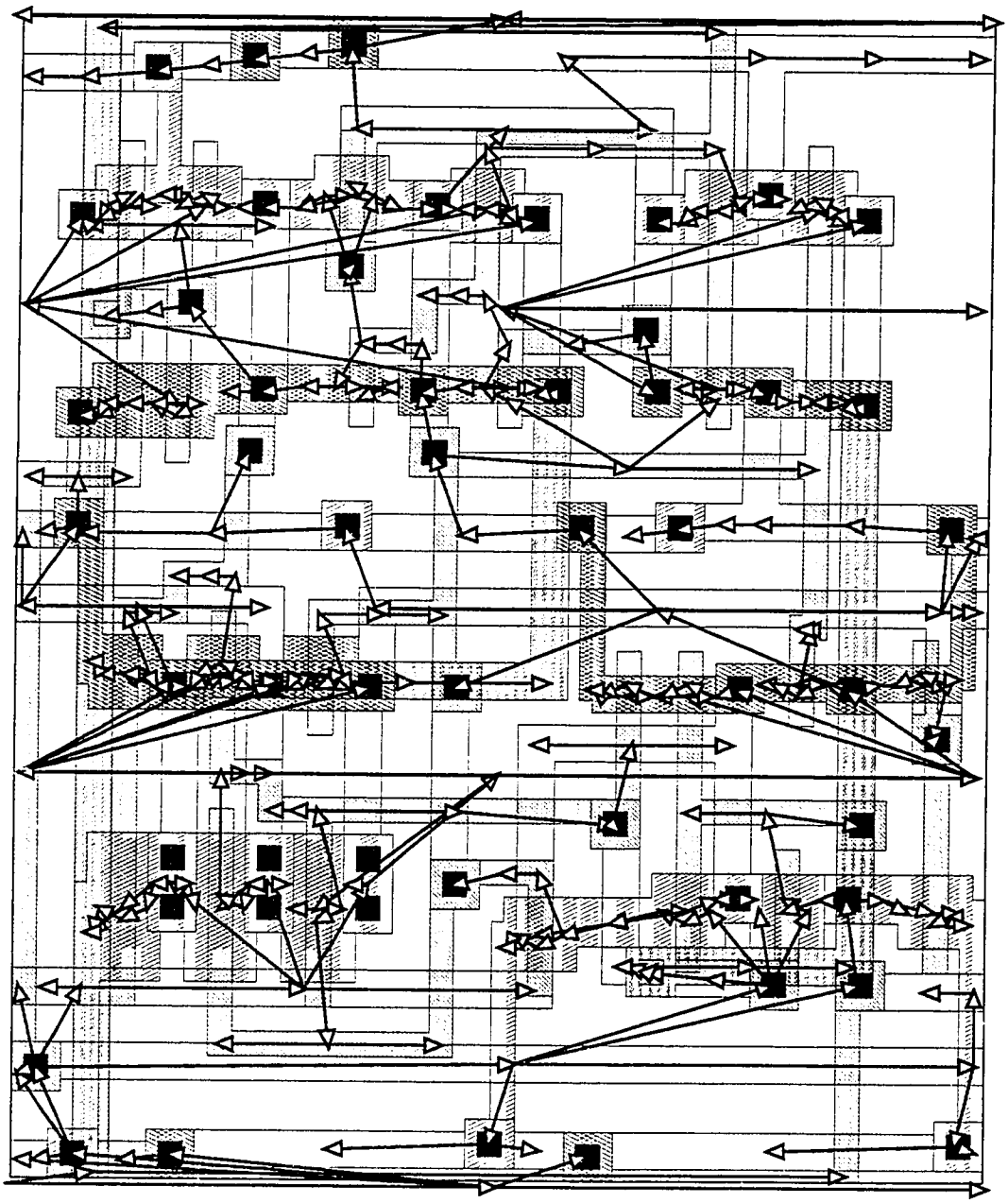


Figure C-5 Adder with Up-Down Spanning Tree

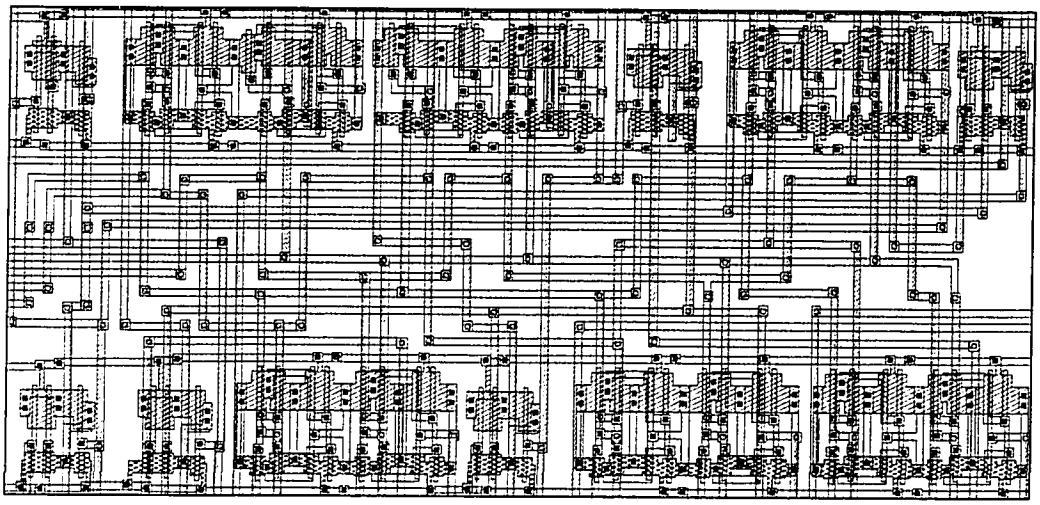


Figure C-6 Routing (C132): 426x201λ, 5.3*k*

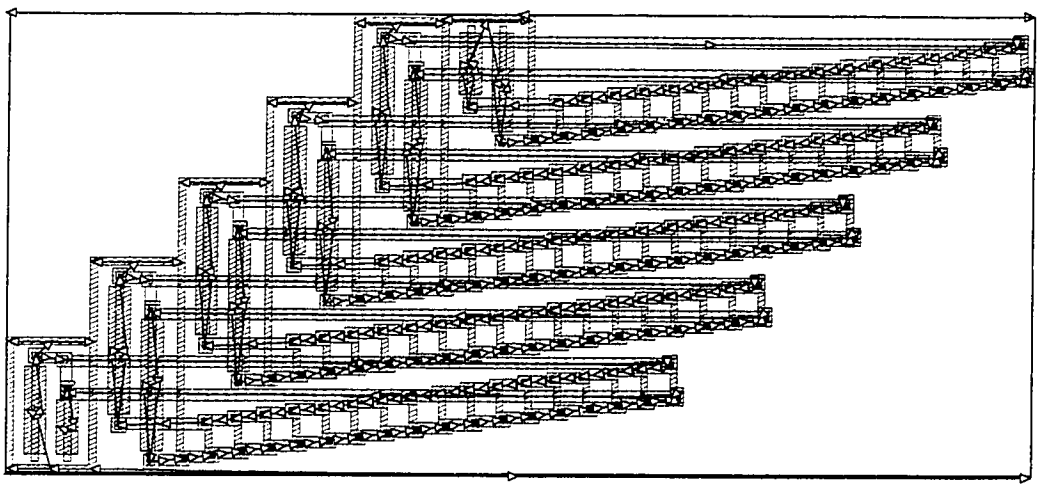


Figure C-7 O(n^2) Worst Case Compaction

Figure C-7 shows an O(n^2) worst case compaction example. The tree compaction converts a two pronged, height $m/2$ tree to a one pronged, height m tree (shown), one tile at a time, without moving a single tile.

References

[Aho 83] Alfred V. Aho, John E. Hopcroft, Jeffrey D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, Reading MA, 1983.

[Akers 70] S.B.Akers, J.M.Geyer, D.L.Roberts. IC Mask Layout With a Single Conductor Layer. In *7th Annual Design Automation Workshop*, pages 7-16, ACM/IEEE, San Fransico CA, June, 1970.

[Baird 77] H.Baird. Fast Algorithms for LSI Artwork Analysis. In *Proceedings 14th Design Automation Conference*, pages 303-311, ACM/IEEE, New Orleans LA, June, 1977.

[Boyer 87a] David G. Boyer. Split Grid Compaction for a Virtual Grid Symbolic Design System. In *Proceedings International Conference on Computer-Aided Design*, pages 134-137, ACM/IEEE, Santa Clara CA, November, 1987.

[Boyer 87b] David G. Boyer. Symbolic Layout Compaction Benchmarks -- Results. In *Proceedings International Conference on Computer Design*, pages 209-217, IEEE, October, 1987.

[Burns 87] J.L.Burns, A.R.Newton. Efficient Constraint Generation for Hierarchical Compaction. In *Proceedings International Conference on Computer Design*, pages 197-200, IEEE, October, 1987.

[Carpenter 87] Clyde W. Carpenter. Generating Incremental VLSI Compaction Spacing Constraints. In *Proceedings 24th Design Automation Conference*, pages 291-297, ACM/IEEE, Miami Beach FL, June, 1987.

[Cho 77] Y.E.Cho, A.J.Korenjak, D.E.Stockton. FLOSS: An Approach to Automated Layout for High-Volume Designs. In *Proceedings 14th Design Automation Conference*, pages 138-141, ACM/IEEE, New Orleans LA, June, 1977.

[Crocker 87] W.H.Crocker, C.Y.Lo, R.Varadarahan. MACS: A Module Assembly and Compaction System. In *Proceedings International Conference on Computer Design*, pages 205-208, IEEE, October, 1987.

[Dantzig 65] George B. Dantzig. *Linear Programming and Extensions*. Princeton University Press, 1965.

[Dial 69] Robert B. Dial. Algorithm 360: Shortest Path Forest with Topological Ordering. *Communications ACM* 12(11), November, 1969.

[Dijkstra 59] E.W.Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik* 1:269-271, 1959.

[Dunlop 78] A.E.Dunlop. SLIP: Symbolic Layout of Integrated Circuits With Compaction. *Computer Aided-Design* 10(6):387-391, November, 1978.

[Dunlop 80] A.E.Dunlop. SLIM: The Translation of Symbolic Layouts into Mask Data. In *Proceedings 17th Design Automation Conference*, pages 595-602, ACM/IEEE, June, 1980.

[Edmonds 72] Jack Edmonds, Richard M. Karp. Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems. *Journal ACM* 19(4):248-264, April, 1972.

[Eichenberger 86] Peter A. Eichenberger. *Fast Symbolic Layout Translation for Custom VLSI Integrated Circuits*. PhD Thesis, Stanford University, April, 1986.

[Fulkerson 61] D.R.Fulkerson. An Out-of-Kilter Method for Minimal-Cost Flow Problems. *SIAM Journal Applied Math* 9(1):18-27, March, 1961.

[Gibson 76] Dave Gibson, Scott Nance. SLIC - Symbolic Layout of Integrated Circuits. In *Proceedings 13th Design Automation Conference*, pages 434-440, ACM/IEEE, San Fransico CA, June, 1976.

[Hedges 85] Thomas Hedges, William Dawson, Y. Eric Cho. Bitmap Graph Build Algorithm for Compaction. In *Proceedings International Conference on Computer-Aided Design*, pages 340-342, ACM/IEEE, Santa Clara CA, November, 1985.

[Hsueh 79a] Min-Yu Hsueh. *Symbolic Layout and Compaction of Integrated Circuits*. PhD Thesis, UC Berkeley, December, 1979.

[Hsueh 79b] Min-Yu Hsueh, D.O.Pederson. Computer-Aided Layout of LSI Circuit Building-Blocks. In *Proceedings International Symposium on Circuits and Systems*, pages 474-477, Tokyo Japan, 1979.

[Johnson 77] Donald B. Johnson. Efficient Algorithms for Shortest Paths in Sparse Networks. *Journal ACM* 24(1):1-13, January, 1977.

[Kedem 84] Gershon Kedem, Hiroyuki Watanabe. Graph-Optimization Techniques for IC-Layout. *IEEE Transactions on Computer-Aided Design* CAD-3(1):12-20, January, 1984.

[Kennington 80] Jeff L. Kennington, Richard V. Helgason. *Algorithms for Network Programming*. John Wiley & Sons, New York, 1980.

[Kingsley 84] Christopher Kingsley. A Hierarchical, Error-Tolerant Compactor. In *Proceedings 21st Design Automation Conference*, pages 126-132, ACM/IEEE, Albuquerque NM, June, 1984.

[Knuth 73] Donald E. Knuth. *The Art of Computer Programming Vol. 1: Fundamental Algorithms*. Addison-Wesley, Reading MA, 1973.

[Lakhani 87] G.Lakhani, R.Varadarajan. A Wire-Length Minimization Algorithm for Circuit Layout Compaction. In *Proceedings International Symposium on Circuits and Systems*, pages 276-279, Philadelphia PA, 1987.

[Lengauer 83] T.Lengauer. Efficient Algorithms for the Constraint Generation for Integrated Circuit Layout Compaction. In *Proceedings 9th Workshop on Graphtheoretic Concepts in Computer Science*, pages 219-230, June, 1983.

[Liao 83] Y.Z.Liao, C.K.Wong. An Algorithm to Compact a VLSI Symbolic Layout With Mixed Constraints. *IEEE Transactions on Computer-Aided Design* CAD-2(2):62-69, April, 1983.

[Marple 88] David Marple, Michiel Smulders, Henk Hegen. An Efficient Compactor for 45 Degree Layout. In *Proceedings 25th Design Automation Conference*, pages 396-402, ACM/IEEE, Anaheim CA, June, 1988.

[Mosteller 81] R.C.Mosteller. REST: Leaf Cell Design System. In *VLSI 81: Very Large Scale Integration*, pages 163-172, Academic Press, New York, 1981.

[Mosteller 87] R.C.Mosteller, A.H.Frey, R.Suaya. 2-D Compaction, a Monte Carlo Method. In *Advanced Research in VLSI*, pages 173-197, Stanford, 1987.

[Nyland 87] L.S.Nyland, S.W.Daniel, D.Rodgers. Improving Virtual Grid Compaction Through Grouping. In *Proceedings 24th Design Automation Conference*, pages 305-310, ACM/IEEE, Miami Beach FL, June, 1987.

[Ousterhout 84] J.Ousterhout. Corner Stitching: A Data-Structuring Technique for VLSI Layout Tools. *IEEE Transactions on Computer-Aided Design* CAD-3(1):87-100, January, 1984.

[Sastry 82] S.Sastry, A.Parker. The Complexity of Two-Dimensional Compaction of VLSI Layouts. In *Proceedings International Conference on Circuits and Computers*, pages 402-406, September, 1982.

[Schiele 83] W.L.Schiele. Improved Compaction by Minimized Length of Wires. In *Proceedings 20th Design Automation Conference*, pages 121-127, ACM/IEEE, Miami Beach FL, June, 1983.

[Schlag 82] M.Schlag, F.Luccio, P.Maestrini, D.T.Lee, C.K.Wong. *A Visibility Problem in VLSI Layout Compaction.* Technical Report RC9896, IBM T.J.Watson Research Center, 1982.

[Schlag 83] M.Schlag, Y.Z.Liao, C.K.Wong. An Algorithm for Optimal Two-Dimensional Compaction of VLSI Layouts. *Integration* 1(2&3):179-209, October, 1983.

[Shin 87] H.Shin, A.L.Sangiovani-Vincentelli, C.H.Sequin. Two-Dimensional Module Compactor Based on Zone-Refining. In *Proceedings International Conference on Computer Design*, pages 201-204, IEEE, October, 1987.

[Sleator 85] Daniel Dominic Sleator, Robert Endre Tarjan. Self-Adjusting Binary Search Trees. *Journal ACM* 32(3):652-686, July, 1985.

[Tarjan 72] Robert Endre Tarjan. Depth First Search and Linear Graph Algorithms. *SIAM Journal Computing* 1(2):146-160, June, 1972.

[Taylor 84] G.S.Taylor, J.Ousterhout. Magic's Incremental Design-Rule Checker. In *Proceedings 21st Design Automation Conference*, pages 160-165, ACM/IEEE, Albuquerque NM, June, 1984.

[Watanabe 84] Hiroyuki Watanabe. *IC Layout Generation and Compaction Using Mathematical Optimization.* PhD Thesis, University of Rochester, 1984.

[Weste 81] N.H.E.Weste. Virtual Grid Symbolic Layout. In *Proceedings 18th Design Automation Conference*, pages 225-233, ACM/IEEE, Nashville TN, June, 1981.

[Williams 64] J.W.J.Williams. Algorithm 232: Heapsort. *Communications ACM* 7(6):347-348, June, 1964.

[Williams 78] John D. Williams. STICKS - A Graphical Compiler for High Level LSI Design. In *AFIPS National Computer Conference Proceedings*, pages 289-295, 1978.

[Wolf 88] Wayne H. Wolf, Robert G. Mathews, John A. Newkirk, Robert W. Dutton. Algorithms for Optimizing, Two-Dimensional Symbolic Layout Compaction. *IEEE Transactions on Computer-Aided Design* CAD-7(4):451-466, April, 1988.