EFFICIENT SOLUTION OF SYSTEMS OF ORIENTATION CONSTRAINTS

Joseph L. Ganley

Cadence Design Systems, Inc.
ganley@cadence.com

ABSTRACT

One subtask in constraint-driven placement is enforcing a set of orientation constraints on the devices being placed. Such constraints are created in order to, for example, implement matching constraints or enforce regularity among members of an array of devices. Here we present an efficient algorithm for solving systems of discrete orientation constraints. The algorithm handles overconstraints by selectively relaxing constraints until the remaining set can all be simultaneously enforced. The algorithm runs in linear time in the absence of overconstraints.

1. INTRODUCTION

Part of the constraint-driven placement problem is finding a set of orientations for the devices being placed that satisfies certain orientation constraints. In this paper we present an efficient algorithm that solves this problem. It has been implemented in the rectilinear metric for four types of orientation constraints, and can be easily extended to other metrics and other types of orientation constraints.

For clarity of exposition, we focus on the rectilinear model, in which only horizontal and vertical lines are allowed, since this is the model we use in our implementation and under which most integrated circuits are fabricated. (Note, however, that our algorithm is applicable to any metric in which the orientation of a device can be described by a finite sequence of independent rotations and/or reflections.) In the rectilinear model, there are eight possible orientations of a device, which are shown in Figure 1 (we use the standard Cadence nomenclature to describe them).

With respect to these orientations, the following constraints are defined:

- A single device has FIXED ORIENTATION set to one of the eight orientations.
- Every device in a set of two or more must have the SAME ORIENTATION.
- A pair of devices must have MIRRORED ORIENTATION about either the x- or y-axis.
- Every device in a set of two or more must have either the SAME OR MIRRORED ORIENTATION (if mirrored, then about either the x- or y-axis).

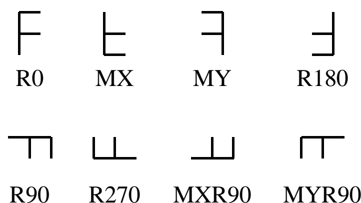

Figure 1. The eight possible orientations.

Here we present an efficient algorithm that uses graph coloring to solve the following problem: given a set of devices and a set of constraints on these devices, where each constraint is one of the four described above, choose an orientation for each device such that all the constraints are satisfied (if possible; see below). If one is only concerned with the first three of these types of constraints, then a simple iterative algorithm that propagates orientations out from the devices with fixed orientations suffices to solve this problem. However, not only is the coloring algorithm more efficient than such an iterative algorithm, but it also handles SAME OR MIRRORED ORIENTATION constraints. The iterative technique is insufficient to handle such constraints; details are given in the next section.

Sometimes all constraints cannot be satisfied simultaneously. For example, suppose two devices are members of both a SAME ORIENTATION and a MIRRORED ORIENTATION constraint; clearly these two constraints cannot be simultaneously satisfied. Such a condition is called an *overconstraint*, and it is handled by *relaxing* (i.e. not enforcing) one or more of the constraints involved in the overconstraint, such that the remaining constraints can be simultaneously satisfied. The coloring algorithm, unlike the previous iterative algorithm, allows total control over the selection of which constraints to relax in order to resolve overconstraints.

2. PREVIOUS WORK

To our knowledge, the literature contains no previous work on this problem.

In the first release of our placement product, solving orientation constraints was accomplished using a simple iterative algorithm. This algorithm proceeds as follows: Initially the orientation of every device is set to *undefined*. Next the orientations of those devices on which there is a FIXED ORIENTATION constraint are set appropriately. The algorithm then repeats the following *orientation propagation* operation: find an as-yet-unprocessed constraint, some of whose members' orientations have already been set, and set the orientations of the remaining devices in the constraint appropriately. If no such constraint exists, then pick a device whose orientation is undefined and set its orientation arbitrarily, and repeat the orientation propagation loop.

As mentioned above, this algorithm is sufficient to handle the

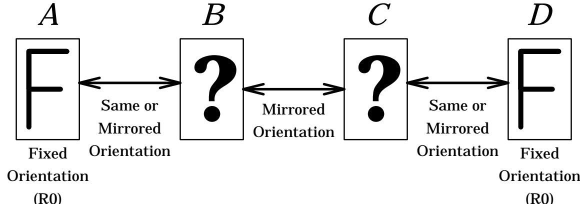

Figure 2. A Test Case for the Iterative Algorithm.

first three types of constraints, but it faces a problem when presented with a SAME OR MIRRORED ORIENTATION constraint. Within the local context of the constraints and devices being considered in each orientation propagation step, there is insufficient information with which to choose whether to enforce a SAME OR MIRRORED ORIENTATION constraint as a SAME ORIENTATION constraint or as a MIRRORED ORIENTATION constraint. In our implementation, one of the two was arbitrarily chosen; however, this can lead to situations where no solution is found to a system of constraints that does, in fact, have a solution satisfying all constraints.

Consider Figure 2, which illustrates a set of constraints on four devices A, B, C, and D.

The algorithm begins by setting devices A and D to orientation R0. However, it must then decide whether to enforce each of the SAME OR MIRRORED ORIENTATION constraints as a SAME ORIENTATION constraint or as a MIRRORED ORIENTATION constraint. Due to the MIRRORED ORIENTATION constraint between B and C, the two SAME OR MIRRORED ORIENTATION constraints must be enforced oppositely—one as a SAME ORIENTATION constraint and the other as a MIRRORED ORIENTATION constraint—but there is no way to determine this while considering only the constraints involving devices whose orientations have already been determined.

One can easily devise a strategy to deal with this particular test case, but we can construct a similar but longer chain of SAME ORIENTATION, MIRRORED ORIENTATION, and SAME OR MIRRORED ORIENTATION constraints, terminated at the ends by FIXED ORIENTATION constraints, to defeat any such ad hoc strategy.

The iterative algorithm also deals poorly with overconstraints. The way they are handled in our implementation was that if the orientation of a device is already set, and a constraint now being considered would force that device to have an orientation different from its current one, then there is an overconstraint and the current constraint is relaxed.

The iterative algorithm has time complexity $O((n + m)^2)$, where n is the number of devices and m is the number of constraints. This running time could probably be improved by the use of suitable data structures, but in light of our new algorithm this hardly seems worthwhile.

3. THE COLORING ALGORITHM

The key observation leading to our coloring algorithm is that the orientation of a device in the rectilinear metric can be represented by the independent application or nonapplication of three operations:

- Rotate the device 90 degrees counterclockwise (R90).
- Mirror the device about the x-axis (MX).
- Mirror the device about the y-axis (MY).

If two or more of these operations are applied, then they must be applied in this order; i.e., R90 is applied before MX or MY, and MX is applied before MY. These operations, if applied, start from the default orientation R0. Thus, each of the eight possible orientations can be precisely and uniquely described by choosing whether

Orientation	R90?	MX?	MY?
R0	No	No	No
MX	No	Yes	No
MY	No	No	Yes
R180	No	Yes	Yes
R90	Yes	No	No
R270	Yes	Yes	Yes
MXR90	Yes	Yes	No
MYR90	Yes	No	Yes

Table 1. Operations applied to produce each orientation.

or not to apply each of these operations. Table 1 shows, for each of the orientations shown in Figure 1, which of these operations are applied to produce that orientation.

We will now construct three graphs. A *graph* $G = (V, E)$ consists of a collection V of *vertices* and a collection E of *edges*, where each edge has two vertices u and v as endpoints. Such an edge is denoted (u, v). Edges are undirected, so (u, v) and (v, u) denote the same edge between vertices u and v. If edge (u, v) is in E, then we say that u and v are *adjacent* and are one another's *neighbors*. The graphs will be colored using two colors: red and green. A coloring is valid if, for every edge (u, v), vertices u and v have different colors. Note that this implies that if two vertices u_1 and u_2 are adjacent to a vertex v—i.e. (u_1, v) and (u_2, v) are both edges in E—then u_1 and u_2 must have the same color.

We will now construct three graphs, denoted R, X, and Y, which correspond to the three orientation components R90, MX, and MY. In each graph, there is initially a vertex for each device, and there are initially no edges. For a given device, the corresponding vertices in R, X, and Y are v_R, v_X, and v_Y, respectively (if there is no danger of confusion, then these three vertices are collectively referred to as v). We are going to color each of these three graphs using two colors, red and green, such that adjacent vertices have different colors. After the graphs have been successfully colored, if a vertex v_R in R is green, then the R90 operation is to be applied to the corresponding device; if v_R is red, then the R90 operation is not to be applied. Similarly, the colors of the vertices in X and Y indicate whether the MX and MY operations, respectively, are to be applied to the device corresponding to each vertex.

Initially, the color of every vertex is undefined. A FIXED ORIENTATION constraint is implemented for the device corresponding to vertex v by setting the colors of vertices v_R, v_X, and v_Y to the appropriate values to implement the orientation as prescribed in Table 1.

We will now add edges and intermediate vertices between the vertices corresponding to devices involved in each constraint, such that a valid 2-coloring of each graph translates to an orientation for each device such that all constraints are satisfied. This is accomplished using the fact that adjacent vertices must have different colors and pairs of vertices each connected to an intermediate vertex must have the same color.

A SAME ORIENTATION constraint is implemented as follows. If the constraint involves more than two devices, then this operation is applied pairwise between an arbitrarily chosen device A in the constraint and each additional device in the constraint (clearly if all devices have the same orientation as A, then they all have the same orientation). For each such pair, let u and v be the vertices corresponding to the pair of devices. Intermediate vertices p_R, p_X, and p_Y are added to R, X, and Y, respectively. Edges (u_R, p_R) and (p_R, v_R) are added to R, and similarly edges (u_X, p_X) and (p_X, v_X) are added to X and edges (u_Y, p_Y) and (p_Y, v_Y) are added to Y. In this way, u_R and v_R are forced to be colored the same color, as are u_X and v_X as well as u_Y and v_Y. Thus, the

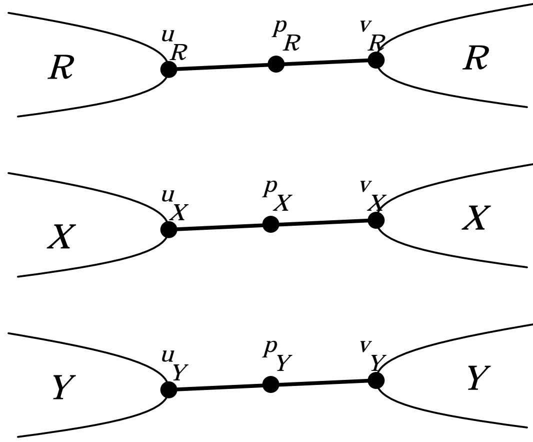

Figure 3. The vertices and edges added for a SAME ORIENTATION **constraint.**

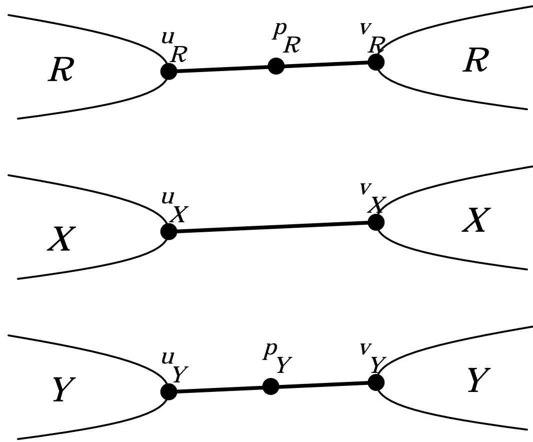

Figure 4. The vertices and edges added for a MIRRORED ORIENTATION **constraint about the x-axis.**

devices corresponding to u and v are forced to have the same orientation. This configuration is shown in Figure 3.

A MIRRORED ORIENTATION constraint is implemented as follows. Assume without loss of generality that the constraint is with respect to the x-axis. A MIRRORED ORIENTATION constraint must include exactly two devices; let u and v be the vertices corresponding to these two devices. Intermediate vertices p_R and p_Y are added to R and Y, respectively. Edges (u_R, p_R) and (p_R, v_R) are added to R, and edges (u_Y, p_Y) and (p_Y, v_Y) are added to Y. A single edge (u_X, v_X) without an intermediate vertex is added to X. In this way, u_R and v_R are forced to have the same color, as are u_Y and v_Y. The vertices u_X and v_X are forced to have different colors. Inspection of Figure 1 and Table 1 shows that this forces the devices corresponding to u and v to have mirrored orientation about the x-axis. This configuration is illustrated in Figure 4.

The implementation of a MIRRORED ORIENTATION constraint with respect to the y-axis is similar, but with X and Y reversed throughout. That is, we add vertex p_R and edges (u_R, p_R) and (p_R, v_R) to R, we add vertex p_X and edges (u_X, p_X) and (p_X, v_X) to X, and we add edge (u_Y, v_Y) to Y.

The final constraint is the SAME OR MIRRORED ORIENTATION constraint. Again, assume without loss of generality that the constraint is about the x-axis. This type of constraint can include more than two devices; as with the SAME ORIENTATION constraint, the following operation is applied pairwise between an arbitrarily chosen device A in the constraint and each additional device in the constraint (again, if all devices satisfy the constraint with respect to A, then they must satisfy it with respect to one another). For each such pair, let u and v be the corresponding pair of vertices. We again add intermediate vertex p_R and edges (u_R, p_R) and (p_R, v_R) to R and add intermediate vertex p_Y and edges (u_Y, p_Y) and (p_Y, v_Y) to Y. No edge is added between u_X and v_X. In this way, vertices u_R and v_R must have the same color, as must vertices u_Y and v_Y. Vertices u_X and v_X may either have the same color, in which case the devices corresponding to u and v will have the same orientation, or they may have different colors, in which case the devices corresponding to u and v will have mirrored orientation about the x-axis (again, this can be verified by examining Figure 1 and Table 1). This configuration is illustrated in Figure 5.

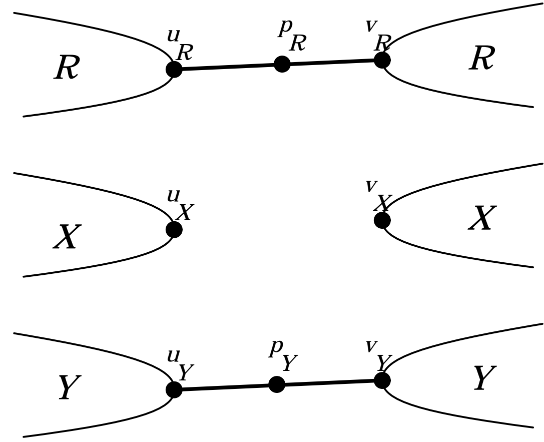

Figure 5. The vertices and edges added for a SAME OR MIRRORED ORIENTATION **constraint about the x-axis.**

Again, a SAME OR MIRRORED ORIENTATION constraint about the y-axis is similar, but with X and Y reversed throughout. That is, we add vertex p_R and edges (u_R, p_R) and (p_R, v_R) to R, and we add vertex p_X and edges (u_X, p_X) and (p_X, v_X) to X.

We have shown how to construct graphs R, X, and Y such that a valid 2-coloring of each graph translates to an orientation for each device such that all constraints are satisfied. What remains to be shown is how to compute such a coloring, and how to deal with overconstraints.

In the absence of overconstraints, a 2-coloring of each graph can be easily computed in time linear in the number of vertices and edges in the graph. One way to do so is by using a depth-first search (see Cormen, Leiserson, and Rivest [1, pp. 377-485], for a description of depth-first search, and see Kozen [2, p. 119], for an explanation of 2-coloring using depth-first search). First, choose a vertex v whose color has already been set due to a FIXED ORIENTATION constraint. If no such vertex exists, then choose an arbitrary vertex v and assign it the color green (in the latter case, there are at least two valid solutions with opposite colors, so the choice of color for the starting vertex is irrelevant). We add a field $\mathsf{parent}(u)$ to each vertex u; the contents of this field will be the vertex that was visited before u. Now, apply the following procedure starting from v.

Color(v):
For each neighbor u of v not already colored:

1. Assign u the opposite color from v
2. Set $\mathsf{parent}(u)$ to v
3. **Color**(u)

In the absence of overconstraints, when this procedure completes it will have produced a valid 2-coloring of the graph.

An overconstraint can be recognized if, in Step (1) above, a neighbor u of v has already been assigned the same color as v. In this case, the graph edges corresponding to the constraints in the overconstraint either form a cycle, or else they form a path whose endpoints are vertices whose colors were assigned due to a FIXED ORIENTATION constraint. Using the parent field, we trace backward from v until we either encounter u again or encounter a vertex whose color was assigned due to a FIXED ORIENTATION constraint. One of these two events must occur, since if we do not encounter a vertex whose color was assigned due to a FIXED ORIENTATION constraint, then we must find a cycle starting and ending at u, or else parent(v) would be equal to u.

Having found the overconstraint, we form a set of constraints corresponding to the edges that the algorithm traced, plus the edge (u, v). We pass this set of constraints to a procedure that relaxes one of the constraints, and then we start all over, build the graphs R, X, and Y, and attempt to 2-color them again. This procedure is repeated once for each overconstraint until all overconstraints have been resolved and we produce a valid 2-coloring of each graph. The criteria used by the procedure to decide which constraint to relax are independent of the algorithm; i.e., the algorithm works correctly regardless of which constraint the procedure chooses to relax. Note that this differs from the way overconstraints are handled by the iterative algorithm, which simply relaxes the constraint it is currently considering at the time the overconstraint is detected.

Once a valid 2-coloring has been computed, it is simple to translate the colors of each trio of vertices back into an orientation for the corresponding device, using the translations given in Table 1. When this is done, each device will have been assigned an orientation such that all constraints that were not relaxed due to overconstraints are satisfied. Each coloring attempt requires $O(n+m)$ time [2], where n is the number of devices and m is the number of constraints (which is within a constant multiple of the number of edges in the graphs). This process, as well as the procedure that decides which constraint in an overconstraint to relax, is repeated once each time an overconstraint is found. Thus, if there are c overconstraints and $f(n, m)$ is the time required by the overconstraint resolution procedure, then the total time required is $O(c(n + m + f(n, m)))$. A typical overconstraint resolution procedure requires at most $O(n + m)$ time itself, which makes the total time required equal to $O(c(n + m))$.

In the actual implementation, there is no need to construct three separate graphs. Instead, one can compute a single graph, each of whose vertices contains three separate colors and three separate sets of edges (one for each of R, X, and Y). In addition, there is no need to explicitly add the intermediate p vertices; rather, one can simply set a flag on the appropriate edges such that they are treated as though p were present.

In addition, it might be possible to reconstruct only the portions of the graphs involved in an overconstraint when one is found, rather than reconstructing the graphs entirely and 2-coloring them from scratch. However, there are typically few or no overconstraints, and the algorithm runs in linear time for each overconstraint, so this optimization is probably not cost-effective.

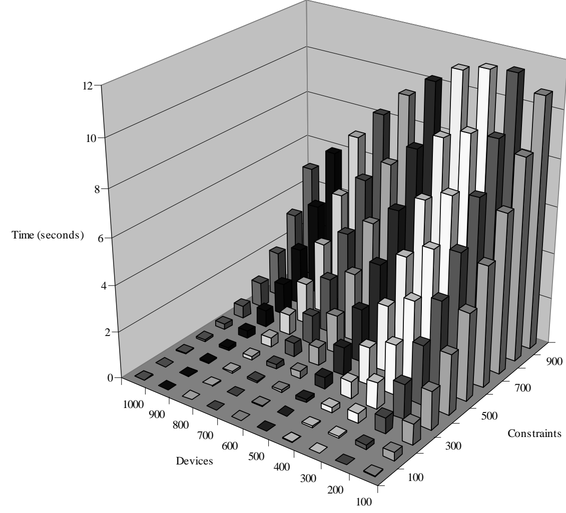

Figure 6. Running times for the given numbers of devices and constraints.

4. EXPERIMENTAL RESULTS

Our coloring algorithm has been implemented as part of a Cadence placement product. Experimental results of running the algorithm on a number of randomly generated test cases are shown in Figure 6 (detailed numerical data available on request).

The test cases are randomly generated so as to avoid the most degenerate types of overconstraints: constraints between two devices that both have fixed orientation, and multiple constraints of different types between the same pair of devices. Each type of constraint is equally probable. Each data point is the result of running 1000 tests. Note that because our test cases are randomly generated, despite being generated as to avoid simple types of overconstraints, they still contain far more overconstraints than would a typical real design. The majority of the running time of the algorithm on this test data is spent dealing with these overconstraints, as evidenced by the fact that for large numbers of constraints the running time actually decreases with increasing numbers of devices, due to fewer overconstraints.

We can get more realistic measurements by running on test cases with the number of constraints equal to half the number of devices. In such test cases, we see approximately one overconstraint for every 50 constraints, whereas for the first set of data we often see one or more overconstraints for every two constraints. The results of running on these more realistic test cases are shown in Figure 7.

5. CONCLUSIONS

We have presented an efficient algorithm that uses graph coloring to solve an important subproblem in constraint-driven placement: solving systems of orientation constraints. This algorithm is more efficient than its predecessor, handles overconstraints more effectively, and finds solutions in difficult cases that defeated the previous algorithm.

While we have discussed only the four constraints that were interesting in our application, this technique can enforce any arbitrary constraint that involves a fixed, same, or different relation among the three orientation components R90, MX, and MY. For example, one application that arose after the initial implementation of this algorithm was complete was in the placer's handling of *fences*. One can define a rectangular region called a fence, and then require that certain devices be placed within the fence. In

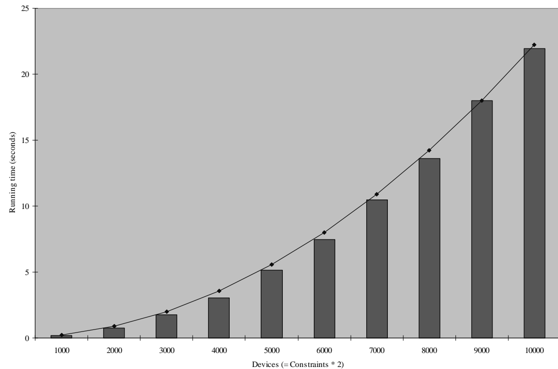

Figure 7. Running times with the given numbers of devices and half that many constraints. The bars indicate the actual running time, and the line is a curve of $O(c(n+m))$ fit to the actual data.

some cases, if the fence and a device that it will contain are both long and thin, then the device may only fit if it is oriented such that its long dimension matches that of the fence. In other words, we would like to enforce for this device that the R90 component of its orientation be fixed, while the MX and MY components are free to vary. This is easily handled by our algorithm.

In addition, the technique can be easily generalized to handle more general sets of orientations, as long as they can be uniquely described by a discrete set of independent rotation and reflection operations.

6. ACKNOWLEDGMENTS

Thanks to Enrico Malavasi and Kathy Jones for many helpful comments. The algorithm described here is patent pending.

REFERENCES

[1] CORMEN, T. H., LEISERSON, C. E., AND RIVEST, R. L. *Introduction to Algorithms*. MIT Press, Cambridge, Massachusetts, 1990.

[2] KOZEN, D. *The Design and Analysis of Algorithms*. Springer-Verlag, 1992.