IMA Mathematical Modeling in Industry Workshop 2005: Integrated Circuit Layout Reconstruction

Ian Besse[*], Patrick Campbell[†], Julianne Chung[‡], Malena I. Espanol[§], Mark Iwen[¶], Edward Keyes (Mentor)[‖], Qingshuo Song[**]

August 22, 2005

Abstract

We present a heuristic algorithm for the reconstruction of integrated circuit layout represented by polygons. The algorithm we propose both reduces the number of vertices and recreates original shapes by reconstructing the polygons using only orthogonal and 45 degree lines.

1 Introduction

Integrated Circuits (IC) are designed using a polygon representation of the wiring and transistors layers known as "layout". Layout must conform to strict design rules. Typically the design rules describe the minimum width of any polygon, minimum spacing between adjacent polygons, directionality (normally only vertical, horizontal and sometimes 45 degree lines are allowed.

[*]University of Iowa
[†]Los Alamos National Laboratory
[‡]Emory University
[§]Tufts University
[¶]University of Michigan
[‖]Semicondutor Insights
[**]Wayne State University

Multiple wiring layers are common, with modern ICs having as many as ten wiring levels. Different wiring layers are connected through a contact or "via" level. The via levels must also obey their own set of design rules on size and spacing.

The process of analyzing the design of an integrated circuit involves imaging the various layers and then using image processing to reconstruct the layout of the different layers. Built-in biases, both in the manufacturing process and the analysis process prevent the reconstructed layout from being a faithful representation of the original layout. For example, sharp corners become rounded, line widths may either expand or contract and straight lines become roughened with many false vertices.

The original layout must be recreated from the raw polygon data. There are a number of potential errors that must be avoided in any reconstruction scheme including: creation of shorts between adjacent polygons, creation of a break or "open" in an existing polygon or creation of self intersecting polygons.

In this report, we present a heuristic algorithm to deal with this problem and evaluate the performance of our algorithm on a variety of test problems. The paper is organized as follows: Section 2 includes the basic problem and explains the energy function underlying the problem. Then, Section 3 explains the various details of our algorithm, and some numerical results are displayed in Section 4.

2 The Basic Problem

Given a polygon **P** we would like to approximate it with another polygon **Q** having the following properties:

1. **Q**'s maximum deviation from **P** is small. In other words, we want **Q** to capture **P**'s general shape.

2. **Q** should include as much of the region bounded by **P** as possible. **P** is ultimately a metal line connecting vias. **Q** will continue to intersect all the vias connected by **P** as long as **Q** contains **P**. Hence, we would like **Q** to be a 'fattened' version of **P**.

3. The fewer vertices **Q** contains the better.

4. We prefer that **Q** consists entirely of vertical, horizontal, 45 degree, and 135 degree lines since they are the only types of lines that should appear in standard IC layouts.

To determine what polygons better approximate **P** we utilize an energy function. The IC layout problem then reduces to optimizing the energy of approximating polygons. We next propose an energy function that addresses the four preceding properties.

Given a polygon $P = (x_1, y_1), (x_2, y_2), ...(x_N, y_N)$ and an approximating polygon $Q = (x'_1, y'_1), (x'_2, y'_2), ...(x'_M, y'_M)$ we define the maximum deviation of Q from P as follows:

For any polygon R and point s let $dist(s, R) = min\{dist(s, r)|$ r is on polygon R$\}$. The maximum deviation of Q from P is defined as

$$dev(Q, P) = max\{dist(p, Q)| \text{ p is a point on} P\} \cup \{dist(q, P)|q \text{ is a point on } Q\}.$$

In order to express that Q is better when it deviates less from P we include the following term in our energy function:

$$\alpha * e^{\beta * (dev(Q,P) - \epsilon)}.$$

We also want to bias towards having Q include all of P's interior points as per the second condition above. In order to help express this 'fattening' bias we define $P - Q$ to be the set of points contained in P but not in Q. And, as one might expect, we let $Area(P-Q) =$ the area of the region $P-Q$ in the Euclidean plane. In order to express that Q is better when it includes more of the region bounded by P we include the following term in our energy function:

$$\gamma * e^{\delta * Area(P-Q)}.$$

To represent the desirability of horizontal, vertical, 45 degree, and 135 degree lines we also include the energy function term

$$\tau * \sum_{i=1}^{M} sin^2(4 * sin^{-1}(|y_i - y_{i+1 mod M}| / \sqrt{(x_i - x_{i+1 mod M})^2 + (y_i - y_{i+1 mod M})^2})).$$

Given P, the energy of our approximation Q is then given by

$$Energy(Q) = M + \alpha * e^{\beta * (dev(Q,P) - \epsilon)} + \gamma * e^{\delta * Area(P-Q)}$$

$$+\tau * \sum_{i=1}^{M} sin^2(4 * sin^{-1}(\frac{|y_i - y_{i+1 mod M}|}{\sqrt{(x_i - x_{i+1 mod M})^2 + (y_i - y_{i+1 mod M})^2}})).$$

We are now ready to state our problem: *Given a polygon* $\boldsymbol{P}$ *we wish to find an approximating polygon* $\boldsymbol{Q}$ *such that Energy(Q) is minimized.*

Our solution proceeds by greedily choosing the portion of P's boundary whose approximation by a line yields the greatest decrease in energy. We then 'repair' the boundary and repeat the process on the next best portion of P's boundary. As it will be discussed in the next section, casting the problem as an optimization problem allows the use of fast efficient heuristic approaches (such as greedy pursuit).

3 The GROS Algorithm

A typical layout data set contains millions of polygons. Any simplification algorithm must be computationally inexpensive. Even an approximate solution of the above energy equation is impractical. Certainly, if computational efficiency was not a factor, one could generate an optimal or very near optimal algorithm for attaining the minimal energy. However, since this algorithm needs to be practical in a business context, we must choose to forego some mathematical exactness and use a more heuristic solution.

To address the issue of fidelity, we first designate a fidelity factor, a maximum allowable deviation that a simplified edge can have from an original edge. This ϵ depends heuristically upon the scale of the features in the polygon. Deviations of a polygon edge of less than $\epsilon/2$ are considered to be inconsequential.

To satisfy the angular energy in the energy function we adopt the rule that we always repair (whenever repair is possible) with segments that are only horizontal, vertical or 45 degree diagonal.

To reduce the vertex energy term we adopt a rule which replaces long chains of vertices by a single two vertex segment provided all the replaced vertices are within our fidelity metric ($\epsilon/2$) of the proposed segment.

Additionally, intuition dictates that a greedy approach to decimation be employed. Thus, before the repair portion of the algorithm begins, a search for the longest sequence of vertices satisfying our tolerance conditions is conducted and the identified sequence is the sequence which is repaired first. Our algorithm then proceeds in a clockwise direction from that point on-

ward. Ultimately, an algorithm which leaps from the longest to the next longest such sequence would be preferable as the number of vertices would then follow the path of greatest descent. With these considerations in mind, we proceed with a detailed explanation of the GROS algorithm.

3.1 Finding the Safety Factor

Not only is it important to ensure faithful reconstruction of the original polygon edges but also we must be careful of its neighboring polygons. Joins between the original polygon and its neighbors must be avoided. The amount by which a polygon edge may be moved without joining to another polygon is a crucial part of any reconstruction algorithm.

Given a particular polygon, the process of finding a proper safety factor S begins by first finding the neighbor polygons. This is done by finding the minimum and maximum x and y values of the polygon and creating bounding boxes around each polygon. Then we pad them to find its neighbors. Then compute intersection between each box and its surrounding boxes and only between them find polygon distance.

3.1.1 Minimum distance between non-convex polygons

In this subsection, we assume self-intersection does not happen for simplicity. To proceed, we define the minimum distance between two disjoint polygons as

$$dist(P, Q) := \min_{p \in P, q \in Q} dist(p, q) \tag{1}$$

We introduce a naive approach for calculating S, so that the polygons do not intersect to each other within the S change.

1. Given a maximum S, and a finite set of polygons Ω,
2. For each polygon P, find

$$\epsilon_1 = \min_{Q \in \Omega} dist(P, Q), \tag{2}$$

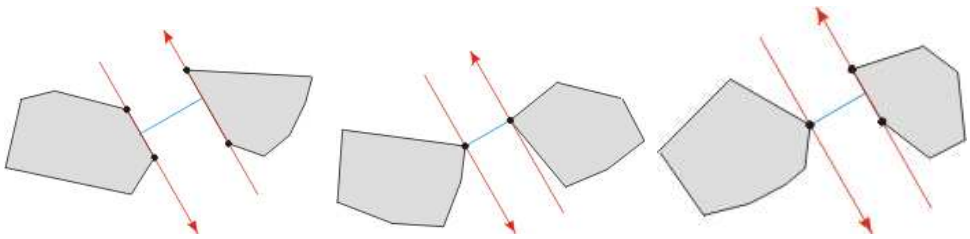
3. Set the S for polygon P as $\S_p = \min\{\S_1, \S\}$.

Therefore, the safety factor calculation comes up with calculation of $dist(P, Q)$ for arbitrary disjoint polygons P and Q. The minimum distance between two

polygons P and Q is determined by anti-podal pair between the polygons. As there are three cases for anti-podal pairs between polygons, three cases for the minimum distance can occur:

1. The vertex-vertex case, see Figure 1 (a),
2. The vertex-edge case, see Figure 1 (b),
3. The edge-edge case, see Figure 1 (c).

In other words, the points determining the minimum distance are not necessarily vertices.



(a) vertex–vertex case (b) vertex–edge case (c) edge–edge case

Figure 1: Three cases for anti-podal pairs between polygons

If all the given polygons are convex, then we can apply an algorithm based on the *Rotating Calipers*. However, there are many non-convex polygons involved in our circuit layout problem. To calculate $dist(P, Q)$, we need to calculate the minimum distance from each line in P to each line in Q pairwisely, and take the minimum. That is,

$$dist(P, Q) = \min_{l \in P, k \in Q} dist(l, k), \tag{3}$$

where $l \in P$ and $k \in Q$ are edges in corresponding polygons.

Using the algorithm above, the computational cost for finding $dist(P, Q)$ is $O(mn)$, provided that P and Q have m and n vertices, respectively. If the set of polygons Ω has N vertices altogether, the total computational cost for finding S is $O(N^2)$, which is relatively expensive. For example, if we have $N = 10^4$, then the cost is roughly 10^8. It turns out improving the algorithm is extremely important.

3.1.2 Neighborhood approach

To reduce the computational cost, we want to replace (2) by

$$\S_1 = \min_{Q \in N(P)} dist(P, Q) \tag{4}$$

where $N(P)$ is the neighborhood of P. The basic idea is that, we only calculate the distance to those polygons in the *neighborhood of* P.

Before precisely presenting the definition of $N(P)$, we need to introduce several notations. Let p_x and p_y be $x-$ and $y-$ coordinates of vertex $p \in P$. Also let

$$x_m(P) = \min_{p \in P} p_x, \qquad y_m(P) = \min_{p \in P} p_y,$$

and

$$x_M(P) = \max_{p \in P} p_x, \qquad y_M(P) = \max_{p \in P} p_y.$$

Construct rectangle $P^{\S}$ with diagonal $(x_m(P) - \S, y_m(P) - \S)$ and $(x_M(P) + \epsilon, y_M(P) - \S)$. In such a way, $P^{\S}$ gives a rectangle which is enlarged by $\S$ from each side of convex hull of P.

The most important property of this simplification of polygons is that a complex object is represented by a limited number of bytes. Although a lot of information is lost, neighborhood rectangles of spatial objects preserve the most essential geometric properties of the object, that is, the location of the object and the extension of the object in each axis. For realistic profiles of data and operations, the gain in performance is quite considerable.

Now we define $N(P)$ as following:

$$N(P) = \{Q \in \Omega : P^{\S} \cap Q^{\S} \neq \Phi\}. \tag{5}$$

To implement (4), we need to use following fact: Two rectangles $P^{\S}$ and $Q^{\S}$ intersect to each other if and only if it satisfies

$$x_m(P^{\S}) < x_M(Q^{\S}) \ \& \ x_m(Q^{\S}) < x_M(P^{\S}) \tag{6}$$

$$y_m(P^{\S}) < y_M(Q^{\S}) \ \& \ y_m(Q^{\S}) < y_M(P^{\S}). \tag{7}$$

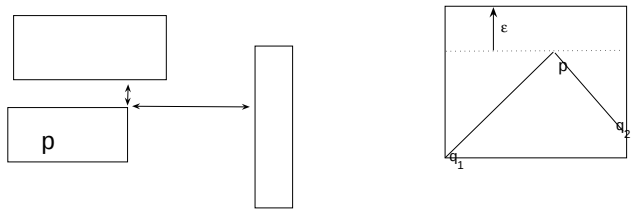
In the last subsection, we calculate $dist(P, Q)$ by finding distance from line to line pairwisely. However, it is very often that some lines in P are far apart from Q. To save consuming cost for $dist(P, Q)$, we replace (3) with

$$dist(P, Q) = \min_{l \in N(Q), k \in N(P)} dist(l, k) \tag{8}$$

where l and k are edges in P and Q respectively. We can use (6) and (7) to verify if $l \in N(Q)$ for all $l \in P$.

3.1.3 Tolerance for each vertex in direction

In Figure 2 (a), the safe move distances to the right upwards differ considerably. It is therefore beneficial to allow safety factors for each vertex in each direction. We therefore need 4 directions to consider for each vertex: *up, down, left, right.*



(a) (b)

Figure 2: various Safety Factors for each direction each vertex

For illustration, we explain the case in Figure 2 (b). Suppose we want to adjust vertex p upwards.

1. construct upward neighborhood rectangle $p^{\S}$ for p containing adjacent edges.
2. calculate $\S_1$ similar to (4).

$$\S_1 = \min_{i=1,2,l\in N(p)} dist(l, q\bar{p}_i) \tag{9}$$

In the above, l is line segment, such that,

$$l \in N(p) := \{l \in \Omega : l \cap p^{\S} \neq \Phi\}.$$

The advantage of this method is that we can avoid self-intersection in our work.

3.2 Finding the Optimal Starting Sequence

Before we run the repair algorithm, we run a routine to find the longest sequence of vertices which satisfies the following two conditions: 1) the pairwise distance between the vertices must be less than a fidelity tolerance ϵ in either the x or y direction, and 2) the sequence of Euclidian distances between the first vertex and each subsequent vertex, must be monotonically increasing. That is, we want the longest sequence, $< v_1, v_2, \ldots, v_n >$ such that

$$\|v_i - v_j\|_\infty < \epsilon$$

for all $i, j \leq n$, and

$$\|v_i - v_1\|_2 \leq \|v_{i+1} - v_1\|_2$$

for $i = 1, \ldots, n$.

We choose the vertex at the beginning of the sequence as the starting point of our repair routine because the sequence is presumably relatively straight, and contains no switchbacks.

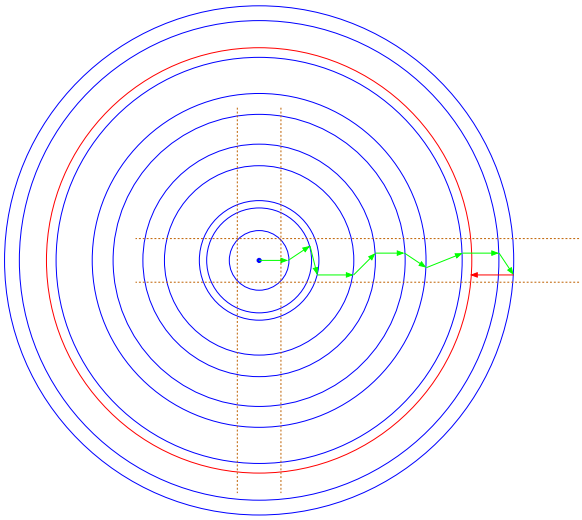


Figure 3: A sequence satisfying our criteria

3.3 Repair Algorithm

Once the longest sequence of vertices satisfying our vertical and horizontal tolerance conditions has been identified, the repair of the polygon may commence. Ultimately, the algorithm for identifying the longest viable sequence

of vertices will search in all four directions, but ours does not currently support this feature. Thus, the repair algorithm as it currently stands uses only the information about the location of the longest viable sequence to determine a starting vertex and conducts its own search for a viable sequence from there. In order to understand the repair algorithm, the diagram in Figure 4 may be helpful. The diagram shows an initial vertex in the center and

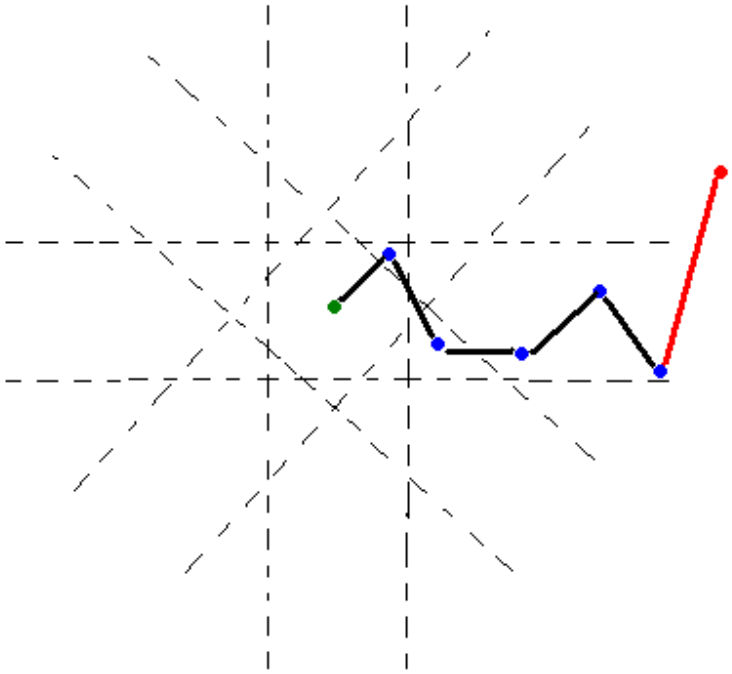


Figure 4: Bounding bands enclosing a sequence of vertices.

the sequence of consecutive vertices extending from it. As one progresses from one vertex to the next, one or more of the four bounding bands may be exited. Once this occurs, the algorithm raises a flag to indicate the direction of the violation. As soon as a flag has been raised at least once for each of the bounding bands, the we have reached a stop-and-repair condition.

If at the first step, the vertex adjacent to the initial vertex immediately exits all four of the bounding bands, the algorithm then checks to see if the offending vertex has exited horizontal and vertical bounding bands which are twice as wide as the original. If it has, then we make no repair to the segment and proceed using the offending vertex as the new initial vertex.

If the offending vertex is within twice our tolerance in either the vertical or horizontal directions the algorithm generates the appropriate horizonal or vertical line midway between the two vertices and generates two new vertices that are the projections of each vertex onto that line. The algorithm resumes its search for viable sequences using the offending vertex as a new initial vertex.

If the offending vertex is not adjacent to the initial vertex, then the previous, non-offending vertex is then considered the terminal vertex in what is a viable sequence. The algorithm then proceeds to conduct a implication of that sequence based on a number of conditions. If the last bounding band to be flagged is either of the diagonals, then the simplification is merely to decimate all vertices whose indices are strictly between those of the initial and terminal vertices. In the horizontal and vertical directions, the simplification follows what we call the "always fatten" rule. Note that in both the horizontal and vertical cases, we may replace the sequence with two edges, one vertical and one horizontal. Such a replacement is not unique, however. We have two choices for generating the middle replacement vertex as shown in the Figure 5

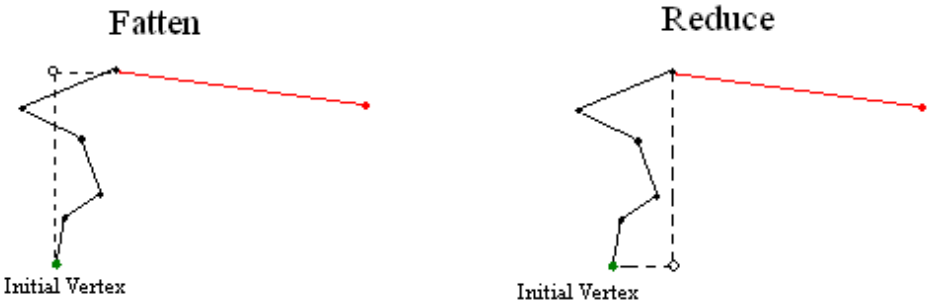


Figure 5: The figure on the left results in greater area for our simplified polygon.

The algorithm generates the vertex which results in the capture of a greater portion of the interior of the polygon. This choice is an effort to reduce the likelihood of narrowing a polygon to zero width in some spots and also to reduce the likelihood of self-intersection.

During this process of simplifying sequences, redundancies may have been created and "hanging chads," vertices which stick out on a line segment from the polygon, may be left behind. The final process in the repair algorithm

is then to search once around the polygon to identify and eliminate identical consecutive vertices, and once more to eliminate the "chads." This new polygon in matrix form is then passed to the display shell and superimposed upon the original polygon.

4 Numerical Experiments

We perform an experimental investigation of the performance of our algorithm on real data of circuit layouts and measure the success of our algorithm using two criteria.

1. By comparing the number of vertices before and after reconstruction, we can numerically estimate how well our algorithm decimated useless and unnecessary vertices. By reducing the number of vertices, we reduce future storage requirements and computational cost.

2. The second criteria is a heuristic one. Since we would like to maintain fidelity to the original data, we visually compare polygons to check how the algorithm altered the polygons.

In our experiments, we use Matlab 7.0.

4.1 Details of the experimental process:

Now that we understand the basic GROS algorithm, we explain the choreography of the process from raw data retrieval to presentation of results.

1. First, we get the raw data and put the polygons into cell format.

2. Each data set contains multiple polygons, so for each polygon, we:

   - Find the neighbors and compute a tolerance.
   - Run the GROS algorithm on the polygon.

3. Once all polygons are simplified, we store all new polygons in both cell and matrix format.

4. Finally, we display the results by overlaying the old and new polygons to visually compare the polygons.

4.2 Some Good results

Running the GROS algorithm on a plethora of real test data polygons, we found that, for the most part, our algorithm did an excellent job at eliminating unnecessary vertices and squaring off jagged edges and corners. We encountered many successful reconstructions, but in this section, we only look at two such examples. In Figure 6, the yellow polygons with blue vertices represent the original polygon data. Especially evident in the 45 degree polygons, we see an excess of vertices. To compare the original with the new polygons we overlay the original polygons with the red simplified polygons, which have red dots as the vertices.

The success of our algorithm is evident in both cases, orthogonal and 45 degree angles, but we see that in Figure 6, the polygon containing the 45 degree angle had a 61 % reduction in vertices on that polygon alone. We will see that in larger cases that contain 45 degree angles, the reduction can be extremely significant.

4.3 Analysis/Comparison

- In the previous subsection, we looked at smaller cases of polygons to gage how well our algorithm performed. However, many of the actual data polygon sets contain hundreds to thousands of polygons and thousands to millions of vertices. To illustrate the power of our algorithm, we compare the before and after images of one of the larger sets of polygons. Figure 7 shows the original polygons and the simplified polygons. The blue stars represent the polygon vertices. There are 3912 vertices for the original data and 333 vertices for the simplified data, which is a significant decrease in storage. In addition, we kept the features of the original polygons.

- Another factor that can affect our results is the tolerance. In this section we consider the tradeoff between tolerance size and percentage decrease in the number of vertices. We run our algorithm on example data, and we compare the results with respect to the tolerance. We only measure with respect to the first criteria, comparing the original number of vertices and the new number of vertices. Our results are displayed in Figure 8 and we can see the incremental change in behavior as the tolerance increases.

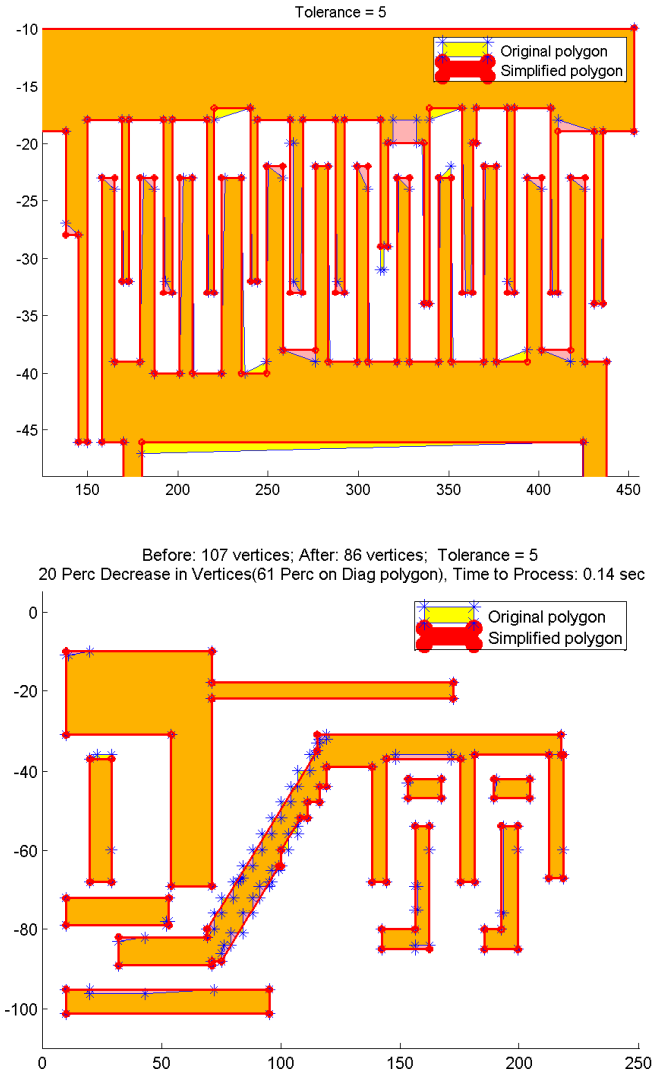


Figure 6: Good reconstructions

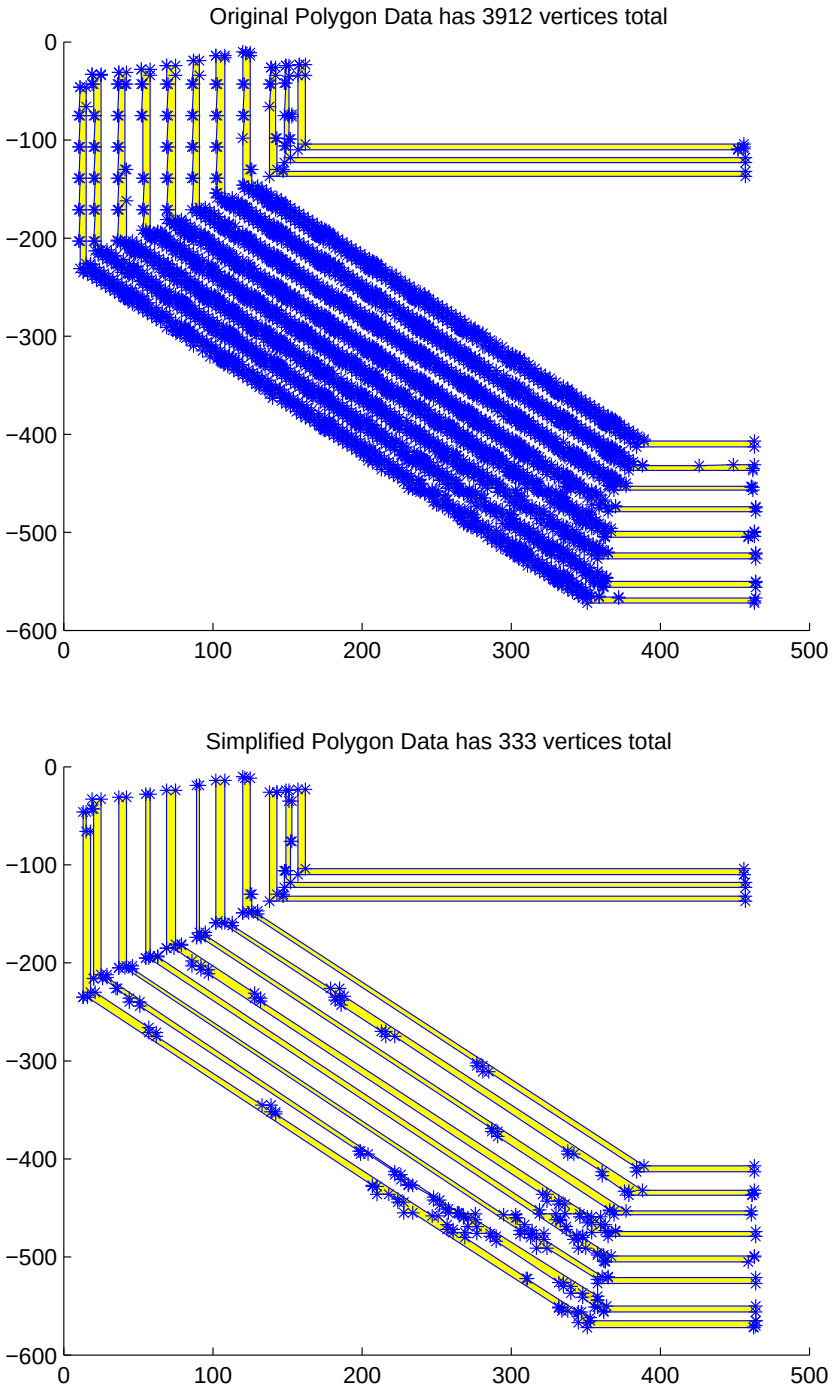


Figure 7: Before and After

Though we see in Figure 8 that a higher tolerance can result in less vertices, we must be cautious of tolerances which are too large. We have observed that the fidelity decreases as the tolerance increases, so it is a tradeoff between both.

Tolerance	# vertices before	# of vertices after	Percentage
1	23838	18435	23%
2	23838	14486	39%
3	23838	7721	68%
4	23838	6435	73%
5	23838	5476	77%
6	23838	5122	79%
7	23838	4872	80%

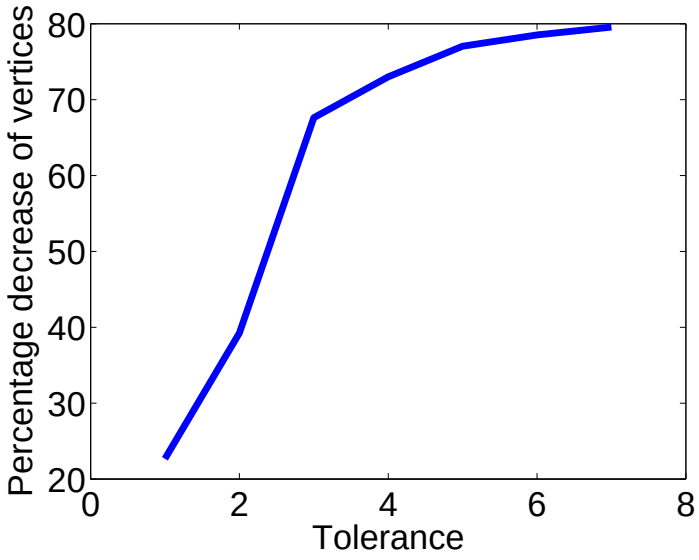


Figure 8: Tolerance vs. Percentage decrease in vertices

4.4 Some problems we encountered

Though our algorithm repaired many polygons as expected, there were some problems that we encountered. With our current approach, there exists the potential of creating self-intersections. For example, with a tolerance too large on a 45 degree polygon, we may get polygons like that shown in Figure 9.

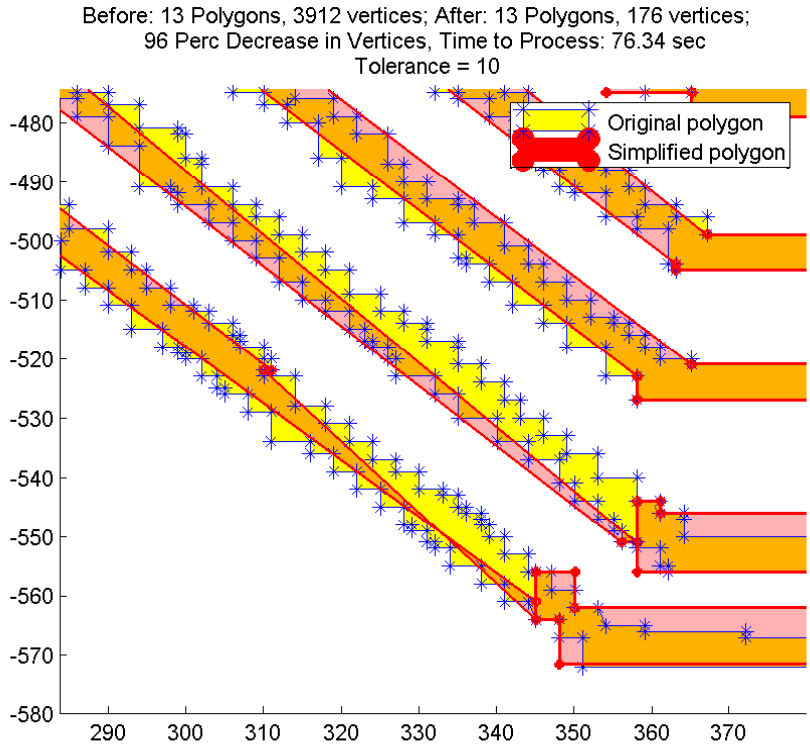


Figure 9: A Self intersection on a 45 degree polygon

Also, depending on the starting line segment, it is possible to create self intersection "holes" like that shown in Figure 10.

5 Conclusions

We presented an heuristic algorithm to reconstruct an integrated circuit layout. We have shown the details of the implementation: the computation of a fidelity factor, the greedy approach to vertex decimation and the rules associated with the repair of an edge. The results generated by our algorithm, both numerical and graphical, provide evidence that our approach is quite successful.

Aside from relatively few instances in which our algorithm produces anomalous results, the algorithm seems to converge to a solution which significantly reduces the energy of the polygon sets. In many cases, the number of vertices, the primary factor in determining image file size, is significantly reduced from the original number. Visual inspection of the simplified polygons indicates that in most cases a high degree of fidelity has been maintained and once-ragged edges have been rendered quite crisp.

Future work may resolve the problems with self-intersection and the emergence of anomalous holes in the polygons. In addition, a more effective method of determining the fidelity factor could not only speed up the computations, but if a vertex-specific variable tolerance could be determined, it would likely result in a great increase in fidelity.

6 Acknowledgements

The authors would like to acknowledge the support from the Institute of Mathematics and its Applications. We would also like to thank Steve Begg, Senior Software Developer, Martin Strauss, University of Michigan, and Suresh Venkatasubramanian, AT&T Labratory.

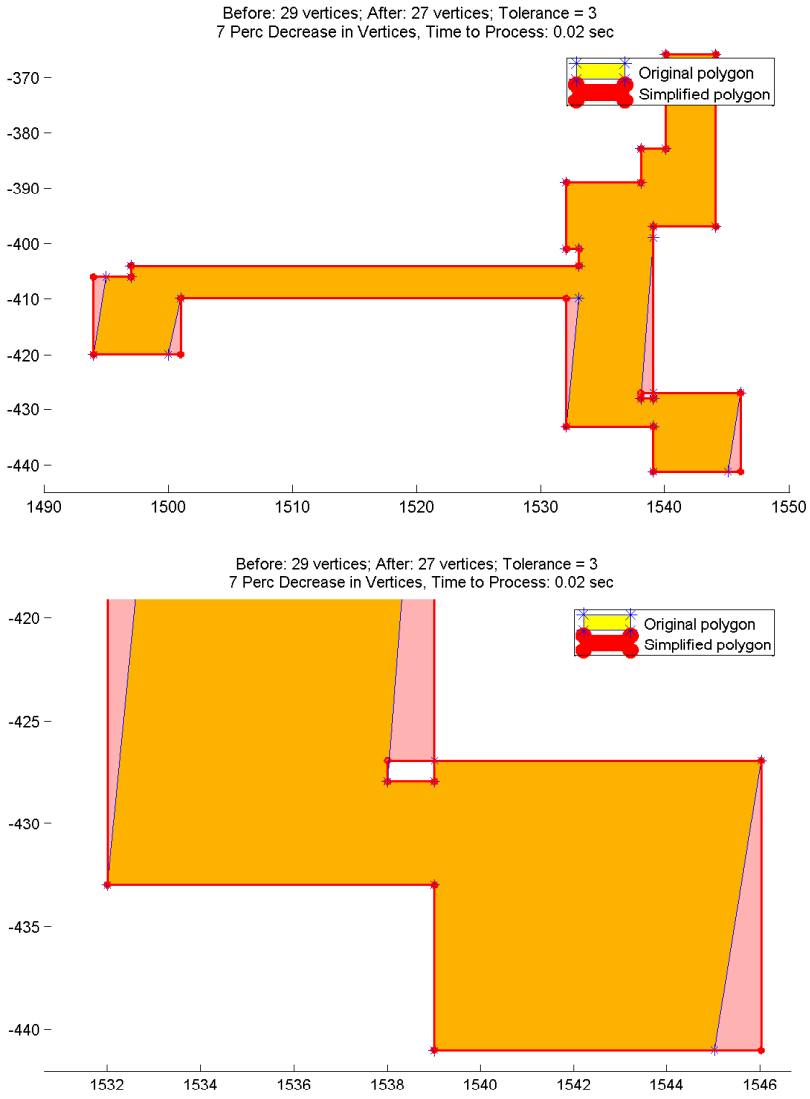


Figure 10: Self intersection hole