Notes by Rich Vuduc

- Notes regarding the homework assignment
    1. Representing Delaunay triangulations
    2. Incremental Delaunay
    3. Ruppert meshing
    4. Surface approximation
- Intro to VLSI Physical Design
    1. Overview of the VLSI process
    2. The role of algorithms in physical design
- Summary
- References

1 Some notes regarding the homework assignment

Here we discuss a few implementation details that are relevant to a few of the recent algorithms which we have considered.

1.1 Representing a Delaunay triangulation

There are a number of application-dependent approaches to storing the mesh triangulations, such as the one shown in Figure 111. Below, we discuss three possibilities: neighbor graph, quad edge, and triangle representations.

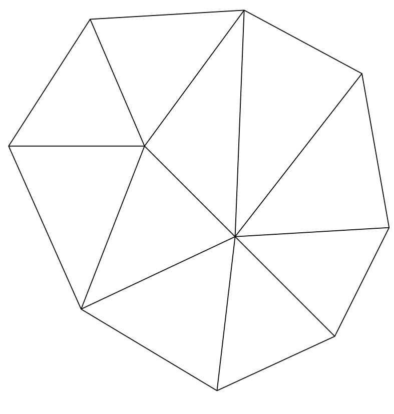

Figure 111: A sample mesh.

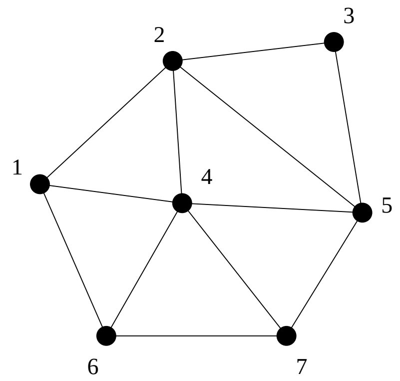

Figure 112: In the *neighbor graph* representation, we interpret the triangulation as a graph and store each vertex with a list of its neighboring vertices.

1.1.1 Neighbor graph

In the *neighbor graph* representation, we treat the nodes of the mesh as vertices in a graph, as shown in Figure 112. There, for each vertex we would store its adjacent vertices:

vertex	neighbors
1	{ 2, 6, 4 }
2	{ 1, 4, 5, 3 }
3	{ 2, 5 }
4	{ 1, 6, 7, 5, 2 }
5	{ 2, 4, 7, 3 }
etc.	...

It is generally helpful to keep each list of adjacent vertices ordered (in the above example, we have stored them in counterclockwise order, but you could use any ordering that is convenient for your application).

1.1.2 Quad edge

Instead of using only vertices for our primary representation, we could store edges instead. One clever way to store edges is called the *quad edge* data structure (Guibas and Stolfi, ACM Trans. Graphics, 4(2), 1985)[2]. This data structure is defined along with an "edge algebra" which allows one to traverse a triangulation easily. In this representation we keep a structure for each edge which stores the 4 adjoining edges as we travel clockwise and counter clockwise from each endpoint of the edge. For example in Figure 113 the four edges adjoining edge 4 are 3, 8, 1 and 5, and the four edges adjoining edge 1 are 3, 4, 5 and 2. In fact the data structure keeps a pointer to one of the 4 fields within each of these adjoining edges. This makes it possible to easily circle around a vertex or an edge. The table below shows an example of our internal representation for the mesh shown in Figure 113. The value after the dot represents which of the 4 entries in the edge is being pointed to.

[2] This was the format used by Garland and Heckbert [1] in their original work. However, in comments to Prof. Blelloch they mentioned that this representation was not particularly well-suited to their application.

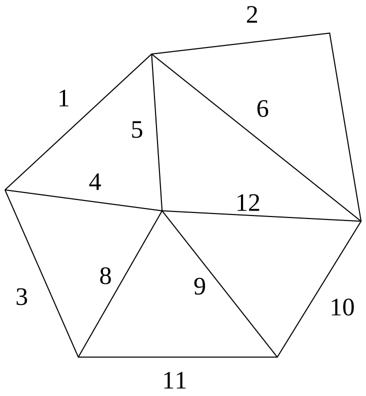

Figure 113: For the quad-edge format, we store each edge along with the 4 adjoining edges (when traveling clockwise and counterclockwise).

edge	neighboring edges
1	{ 2.0, 5.1, 4.2, 3.1 }
	...
4	{ 5.0, 8.3, 3.2, 1.1 }
5	{ 12.2, 4.3, 1.0, 6.1 }
	...
8	{ 4.0, 9.1, 11.2, 3.3 }
9	{ 10.0, 11.3, 8.0, 12.1 }
	...
12	{ 6.0, 10.3, 9.2, 5.1 }
	...

To circle around a vertex we start at an even field of an edge structure (.0 or .2) and follow the pointers, and to circle around a face (triangle) we start at an odd field (.1 or .3). For example, to circle around the vertex in the middle starting at edge 4 we simply start by following 4.0 and continue around touching edges 5, 12, 9, 8 and back to 4 (more specifically 5.0, 12.2, 9.2, 8.0, 4.0). Similarly to circle around triangle (1,4,5) we can start from 4.3 and visit 1.1, 5.1 and back to 4.3.

The algebra on the quad-edge data structure suggested by Guibas and Stolfi uses the three operations `next`, `rotate`, and `symmetric` which take a pointer and return a new pointer. The `next` operation circles around a vertex or triangle, the `rotate` operation switches from a vertex to a triangle (the one that shares the edge, next edge, and vertex) or vise versa, and the `symmetric` operation switches to the vertex at the other end of the edge. Assuming each quad-edge is an array of 4 "pointers" each consisting of a pointer to another quad-edge and an index within the quad-edge, then for a pointer `(e,i)` the operations are defined as:

```
next(e,i) = e[i]
rotate(e,i) = (e, i+1)
symmetric(e,i) = (e, i+2)
```

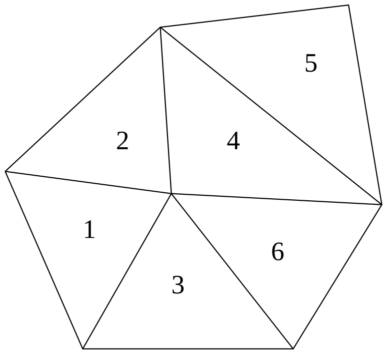

Figure 114: Mesh representation using triangles. Triangle 1 shares sides with triangles 2 and 3.

where + is defined modulo 4.

1.1.3 Triangles

Lastly, we turn to what is perhaps the most natural representation: *triangles*. With each triangle, it may be useful to store the 3 other with which it shares a side. So, for the example shown in Figure 114, our table of data might look something like:

triangle	side-adjacent triangles
1	{ 3, 2, □ }
2	{ 1, 4, □ }
3	{ 6, 1, □ }
4	{ 6, 5, 2 }
5	{ 4, □, □ }
6	{ 4, 3, □ }

In addition to the neighboring triangles it is typically useful to store the vertices of each triangle. In this representation the function `opposite(p,T)` which given a triangle `T` and a vertex (point) `p` on that triangle returns the triangle opposite that point is often useful. For example, in Figure 114 given the leftmost point and triangle 2, `opposite` would return triangle 4.

1.2 Incremental Delaunay

Recall that in the incremental Delaunay algorithm of previous lectures, we are given a triangulation to which we wish to add an additional point. The suggested approach is based on a depth-first search style algorithm. Figure 115 illustrates the action of the following pseudocode to insert a point p into an existing triangulation.

```
function Insert(p, R)
    // p a point
    // R an existing triangulation

    T = the triangle containing p
    (T1, T2, T3) = the triangles created by splitting T by p
    Patch(p, T1)
    Patch(p, T2)
    Patch(p, T3)
end

function Patch(p, T)
    T' = Opposite(p, T)
    if InCircle(p, T')
        (T1, T2) = Flip(T, T')
        Patch(p, T1)
        Patch(p, T2)
end
```

The function Flip flips two adjacent triangles so the crossedge changes.

1.3 Ruppert Meshing

In this section we discuss two issues to consider when implementing Ruppert's meshing algorithm [2].

The first relates to how one finds encroaching points. First, recall that a point p *encroaches* upon a segment s if p lies within the diametrical circle defined by s (see Figure 116). To determine which points encroach upon s, it is sufficient to check endpoints of all Delaunay edges that come out of an endpoint of the segment. In a practical implementation, one would probably need a data structure that facilitates moving from point to point within the triangulation; one possibility would be a data structure that keeps neighboring triangles with each point.

The second issue relates to the creation of *phantom small angles*, which is best explained by an example. In particular, suppose that our input polygon is the one shown in Figure 117 (*top*). Recall that an initial step in computing the triangulation requires that we compute the convex hull of the input polygon; this could result in the creation of *phantom small angles* such as the one shown in Figure 117 (*bottom-left*). An easy solution is to first create a bounding box around the input polygon, as shown in Figure 117 (*bottom-right*).

1.4 Surface Approximation

One important step in the surface approximation algorithm discussed in a previous lecture is to find the largest error in a newly added triangle. This requires examining every point within the triangle. One technique for doing so might be to bound the triangle under consideration by a rectangle first, then scan the pixels within the triangle row- or column-wise. For each

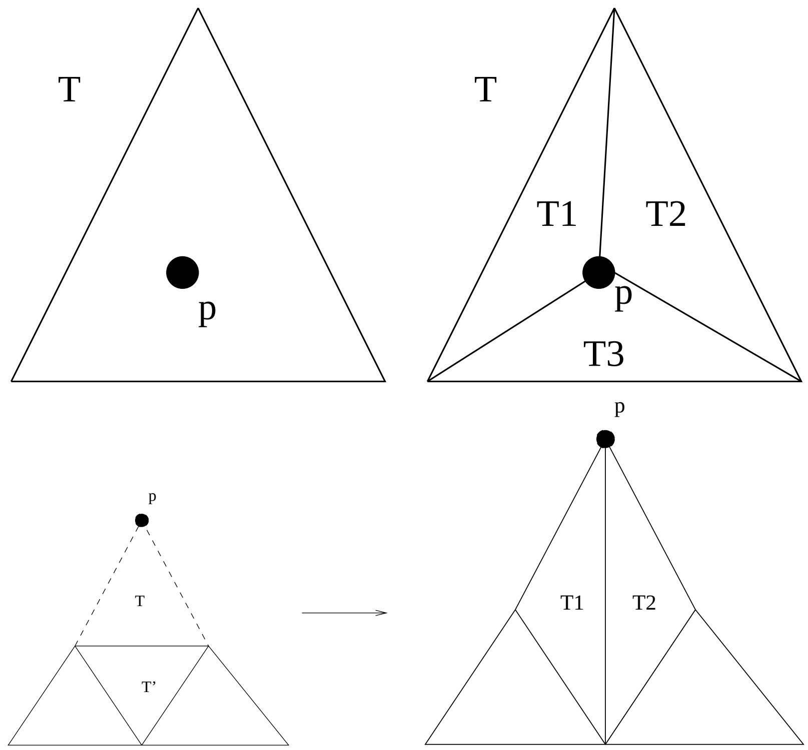

Figure 115: (*Top-left*) We wish to add the point p to an existing triangulation, and p happens to fall within the triangle T. (*Top-right*) Adding p creates the smaller triangles T_1, T_2, T_3. (*Bottom-left*) We consider the action of the `Patch` subroutine at some point in the recursion for the point p and some other triangle T (i.e., not necessarily the same as above). (*Bottom-right*) Result after executing `flip`(p, T').

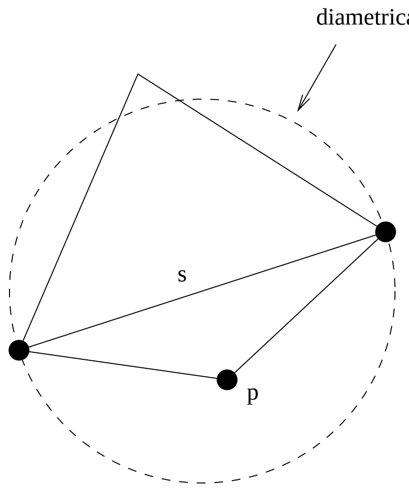

Figure 116: In the situation shown, we wish to see if there are any points in the triangulation that encroach on the segment s (e.g., p).

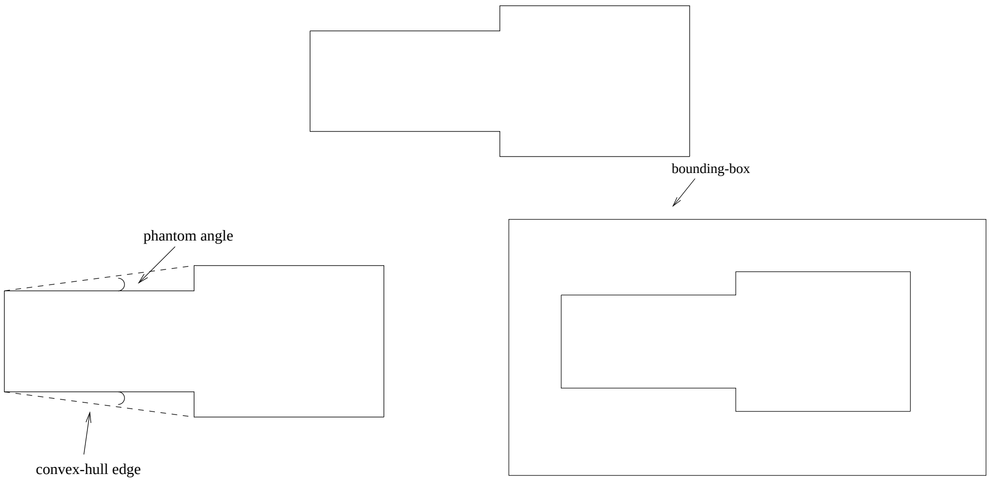

Figure 117: (*Top*) Input polygon. (*Bottom-left*) Initial convex-hull computation results in creation of tiny "phantom small angles." (*Bottom-right*) One easy solution is to placing a bounding box around the figure, and then triangulate this new figure.

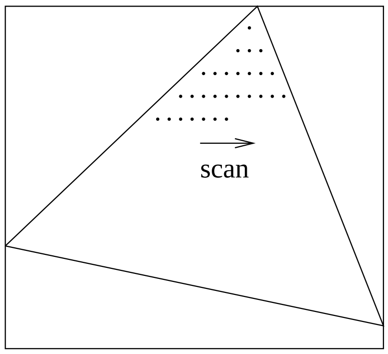

Figure 118: In computing the error of a newly added triangle in the surface approximation algorithm, we can bound the triangle by a box and scan, testing whether or not the pixel lies within the triangle before using it to compute the error.

pixel that we scan, we can check whether or not it falls within the triangle before computing its error. (See Figure 118.) A fancier method would find the edges that border on the left and right and only scan the horizontal lines between them.

2 Intro to VLSI Physical Design Automation

Since the dark ages (i.e., the 1960's), a typical integrated circuit design has grown from a hundred transistors to the millions of transistors required by modern microprocessors. *Very large scale integration* (VLSI) refers to the process of integrating such a large number of transistors for fabrication; *VLSI physical design* is the process of converting a circuit description into a layout. The ability to lay out such enormous circuits has been made possible by the intelligent application of computer algorithms.

Here, we give a brief overview of the overall VLSI physical design process, and show where algorithms play a key role. We will be making a number of references to diagrams shown in [3].[3]

2.1 Overview

The following is a high-level description of the VLSI design process. The focus of our discussions will be on step 5 below.

1. **System specification**: The first step involves the creation of a formal, high-level representation of the system. In this stage, we consider the desired functionality, performance, and physical requirements of the final product.

2. **Functional design**: In this step, we break down the system into functional sub-units, and specify the relationships between them (for example, by generating a timing diagram or showing the desired flow between sub-units).

3. **Logic design**: In this process, we formalize the logical structure of our functional design, typically by generating boolean formulas. An interesting problem is how to minimize such expressions, and neat algorithms like binary decision diagrams have found applications here. However, we will not be discussing these.

4. **Circuit design**: From the logic design, we next need to generate a circuit representation. We have to consider power and speed requirements of our original design, and often perform simulations of electrical behavior in this stage (e.g., through the use of PSpice software).

5. **Physical design**: In this stage (the focus of our discussions), we must actually lay out the circuit designs. We must ensure that our resulting layout conforms to the physical and geometric constraints of our system, which we formulate as *design rules*. These design rules are essentially guidelines based on physical constraints due to limitations in the fabrication process.

6. **Fabrication**: This stage is the actual physical manufacturing step, in which we prepare the wafer and deposit/diffuse various materials onto the wafer.

7. **Packaging, testing, and debugging**: In this stage, we test the actual manufactured and packaged chip.

It is important to note that this entire design process is iterative, i.e., there is a kind of "feedback" effect in which we examine the results, perform any necessary hand-tuning, and re-run the process, continually refining our previous results.

[3] In fact, most of the material here is simply a paraphrase of material that appears in [3].

2.1.1 Physical design

We now turn our attention to the physical design cycle which will be our focus. The following is a brief summary of the high-level process. Note that the input to this process is the circuit design, and the output will be a layout ready for the fabrication stage.[4]

1. **Partitioning**: We must first break down, or *partition*, the input circuit into more manageable blocks (or subcircuits, or modules). In the partitioning process, we must consider the size and number of blocks, as well as the number of interconnections which will obviously affect the routing. The set of interconnections is called a *netlist*. For large designs, the blocks may be hierarchical.

2. **Floorplanning and placement**: In *floorplanning*, we consider such factors as the exact size of the blocks generated in the previous step. This is a computationally difficult step, and often requires a design engineer to do this by hand. During *placement*, we position the blocks exactly as they will appear on the chip. We want to lay out the blocks in such a way so as to minimize the area required to complete all the interconnections. This is generally an iterative process in which we lay out the blocks, and continually refine the placements.

   Furthermore, it is crucial to use a good placement strategy because the placement essentially determines the routing (see below).

3. **Routing**: In the *routing* phase, we lay out the interconnections (i.e., put down wires) specified by the netlist that was generated in the partitioning phase (above). We refer to the space not occupied by blocks as the *routing space*. We partition this space into rectangular regions called *channels* and *switchboxes*. Our router must make the connections using only these two types of regions, and it must do so in such a way so as to minimize wire length.

   The routing process can be divided into two phases. In the first phase, called *global routing*, in which the global router must make a list of channels (i.e., passageways) through which to route the wire. The next phase, called *detailed routing*, completes the actual point-to-point wiring.

   Although the routing problem is computationally hard, many heuristic algorithms have been developed.

4. **Compaction**: In this final phase, we compress the generated layout to reduce the total area. A smaller area would enable us to increase the yield.

2.1.2 Design styles

Although we have taken the entire physical design process and broken it up into smaller subproblems, we have noted that each subproblem is still computationally hard. To make these problems more tractable (and thus reduce the time-to-market and increase the yield), we can further restrict our design methodology into one of the following *design styles*:

[4]Automation of the entire physical design process is akin to the idea of building a "silicon compiler."

1. **Full custom**: We use this term to refer to a design style which places few (if any) constraints on the physical design process. For example, we would place no restrictions on where a block could be placed (e.g., on a grid). Clearly, this is the most difficult design style to automate because of the high degree of freedom. A *full custom* approach is really only appropriate when performance must be high. For instance, this is the method of choice for the layout of modern microprocessors. An example is shown in [3] (Figure 1.3).

2. **Standard cell**: In the *standard cell* design style, we predesign certain functional units as standard cells, or standard blocks, and create a *cell library* of "pre-packaged" blocks that we can then use for placement. Each cell is preanalyzed and pretested. Given that the cells are predesigned, the main challenge is the layout and routing. An example of a standard cell layout is shown in [3] (Figure 1.4). Notice that the blocks are all of a similar shape and size, and that their layout is much more regular than the layout of the full-custom design shown in [3] (Figure 1.3). Observe that in the standard cell design, our only (roughly speaking) degree of freedom is the ability to shift blocks to open up routing channels between placed cells.

3. **Gate arrays**: A *gate array* is like a standard cell except that all the "cells" are the same, and laid out in an array form as shown in [3] (Figure 1.5). The name is indicative of the idea that the cells could be as simple as a single gate (a 3-input NAND gate is shown). Furthermore, the routing channels are basically predetermined. The advantage of the gate array style over the previous two is its simplicity. In particular, it is cheaper and easier to produce a wafer based on this style. Routing is also simplified, since we only have to make connections and not worry about minimizing the area used. An example of a routed gate array for a simple logic circuit is shown in [3] (Figure 1.6).

   However, the cost is the rigidity imposed—we would certainly be forced to produce larger chips and incur a performance penalty.

4. **Field programmable gate arrays**: A *field programmable gate array* (FPGA) is something in-between the gate array and standard cell. An example is shown in [3] (Figure 1.7). Each "cell" of a FPGA (labeled by B_i in the figure) can be viewed as a lookup table that can be programmed. Furthermore, there is no free region over which we do our routing; instead, the wires are fixed, and we connect the cells by burning in *fuses* (shown as squares and circles). The figure shows a sample logic circuit which has been partitioned into four regions. We construct the lookup table for each region, and this is what is programmed into the cells of the FPGA (in this example, the four left-most cells labeled B_i). The connections appropriate fuses are burned in (shown as darkened circles and squares). The partitioning problem for FPGAs is different from that of the other design styles, but the placement and routing problems are similar to those of the gate array.

We summarize the various tradeoffs alluded to above as follows [3] (table 1.2):

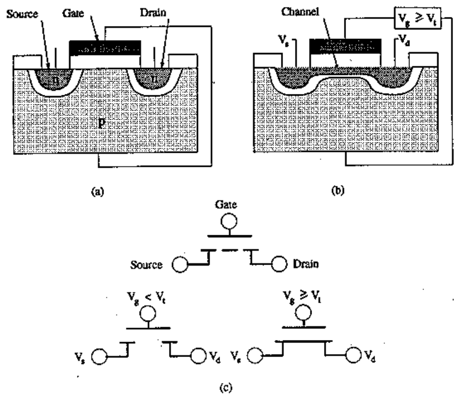

Figure 119: The operation of an nMOS transistor. (a) A vertical cross-section of a transistor based on the $n-p-n$ junction. (b) When a voltage has been applied to the gate, the induced electric field alters the conducting properties of the channel, thereby allowing current to flow. (c) Schematic representations of the process. [Taken from [3] (Figure 2.5).]

	style			
Area	compact	compact to moderate	moderate	large
Performance	high	high to moderate	moderate	low
Fabrication layers	all	all	routing layer	none

In general, within a given column of the table there is a correlation between the rows. Moreover, the columns are roughly ordered in decreasing cost.

2.1.3 Transistor level

When we discuss algorithms in future lectures, it will be sufficient to formulate the routing and placement problem at an abstract level. However, in this section, we will very briefly look at the "small picture" for a little perspective.

Figure 119 shows a diagram of the vertical cross-section of a nMOS transistor, as well as its schematic representations. An nMOS transistor is an example of a more general kind of transistor called a *field-effect transistor*. In such a device, we can control the flow of current between the two terminals (labeled *source* and *drain*) by controlling the voltage applied to the non-conducting terminal called the *gate*. In particular, applying a voltage to the gate sets up an electric field that changes the conducting properties of the region labeled *channel* thereby allowing current to flow from the source to the drain. For a more general introduction to the basic solid-state physics behind the operation of these devices, see [4] and [5].

By combining several transistors, we can construct basic logic gates such as the NAND gate shown in Figure 120. See the figure caption for a rough description of its operation.

Finally, note that in the figure of the NAND gate, there are various rectangular shaded regions which represent different materials. These materials must obey certain rules, namely, they must be of certain dimensions and must be separated by different amounts, as shown in Figure 121. These are the kinds of rules and constraints we will be subject to in our automation algorithms.

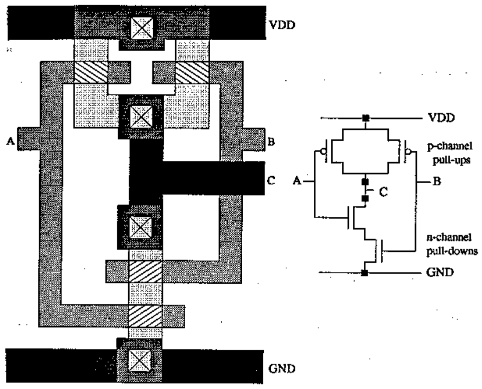

Figure 120: The layout of CMOS NAND gate (*left*) using four transistors, and its schematic representation (*right*). The input to the NAND are labeled A and B, and the output is C. The bottom two transistors are n-channel and open when their gates are high while the top two are p-channel and open when their gates are low. (*Basic operation*) When A and B are both high, the channels of the upper-two transistors are closed and the bottom two are opened, thus enabling a connection from GND to C. In all other cases, the channels of at least one of the lower transistors will be closed which will break the connection between GND and C, whereas when either channel of the upper two transistors are open, C will be pulled up to VDD. [Taken from [3] (Figure 2.7).]

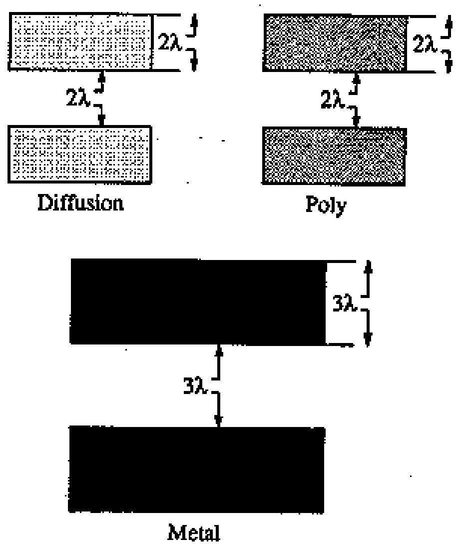

Figure 121: Size and separation rules. The dimensions have been characterized in terms of the design parameter λ. The ".35" of the terms ".35 micron technology," et al., refer to this parameter. [Taken from [3] (Figure 2.10).]

2.2 The role of algorithms

Algorithms have played a crucial role in the explosive growth of VLSI technology. Graph theory has played a particularly important part. Here is a list of some of the many algorithms that have been applied to VLSI physical design automation:

1. Graphs
   - spanning trees
   - shortest path
   - matching
   - Steiner trees
   - maximum independent set
   - maximum clique
2. Computational geometry
   - intersection of lines or rectangles
3. Optimization
   - integer programming

These algorithms have found applications in many of the design areas discussed in the physical design overview, namely

- layout
- partitioning
- placement
- routing
- compaction

Furthermore, these algorithms are applicable in all the different design styles (full-custom, standard cell, etc.). We will be paying particular attention to the routing problem.

Our overall goal will be either to minimize area given physical constraints on our design, or to maximize functionality in a given fixed area. We will have to be concerned with how to:

minimize delays : In particular, we can accomplish by minimizing wirelength and reducing capacitance (by keeping the overall layouts small).

reducing the number of layers Increasing the number of layers required in a design greatly increases fabrication costs.

reducing the number of pins : Of particular concern in multi-chip modules.

All of the major designers and manufacturers of VLSI chips are investing heavily in automation research and techniques. However, hand-tuning still has an important place in the design phases.

2.3 Evolution

We will close by summarizing the history of development tools in physical design automation.

Year	Design Tools
1950-1965	Manual design
1965-1975	Layout editors Automatic routers Efficient partitioning algorithms
1975-1985	Automatic placement tools Well defined phases of design of circuits Significant theoretical development in all phases
1985-present	Performance driven placement and routing tools (in particular, an emphasis on space and speed)
	Parallel algorithms for physical design Significant development in underlying graph theory (i.e., the use of theoretical results) Combinatorial optimization problems for layout

3 Summary

In the first section, we gave some notes about implementation methods for several triangulation algorithms. In particular, we discussed some ideas for data structure representation, with the triangle representation seeming the most useful.

We then gave a brief overview of the entire VLSI design process, and introduced the physical design stage in more detail. We showed that algorithms play an important role, our overall goal being to minimize the layout area (to increase our chip's performance) while subject to various constraints of our physical design (e.g., separation).

References

[1] M. Garland and P. Heckbert. "Fast polygonal approximation of terrains and height fields." CMU Technical Report, CMU-CS-95-181, Carnegie Mellon University, Pittsburgh, PA, September 1995.

[2] J. Ruppert. "A Delaunay refinement algorithm for quality 2-dimensional mesh generation." NASA Technical Report, RNR-94-002, January 1994.

[3] N. Sherwani. *Algorithms for VLSI Physical Design Automation.* Kluwer, 1995.

[4] C. Kittel. *Introduction to Solid-State Physics*. Merriam, (fill in details), 1995.

[5] On-line course notes on "Heterojunction Devices" from the University of Colorado. [http://ece-www.colorado.edu/~bart/6355.htm]