# SHAPE SYNTHESIS FOR ASSEMBLY- CENTRIC DESIGN

by

Steve Robert Burgett

Memorandum No. UCB/ERL M97/10

27 January 1997

# SHAPE SYNTHESIS FOR ASSEMBLY-CENTRIC DESIGN



by

Steve Robert Burgett

Memorandum No. UCB/ERL M97/10

27 January 1997

**ELECTRONICS RESEARCH LABORATORY**

College of Engineering
University of California, Berkeley
94720

# Abstract

Shape Synthesis for Assembly-Centric Design

by

Steve Robert Burgett

**Doctor of Philosophy in Engineering—Electrical Engineering and Computer Sciences**

**University of California at Berkeley**

**S. Shankar Sastry, Chair**

In this dissertation I present a new paradigm for CAD which aims to support the early stages of mechanical design well enough that designers are motivated to use the workstation as a conceptual design tool. This approach is founded on the concept of *Assembly-Centric* Design, wherein the design process focuses on the assembly as a whole rather than on individual parts. The parts themselves are designed largely automatically by the software.

The opportunity for such design automation arises from the fact that many mechanical parts can be defined by two kinds of geometry: features that are critical to a part's function (*application features*), and the material that merely fleshes out the rest of its shape (*bulk shape*). Application features are most often associated with contact surfaces of the part, for example, a bore for a bearing or a mounting surface for a motor. These features are the portals through which the part interacts with other parts in the assembly.

At the assembly level, application features come in groups. For example, a bore and its mating shaft always occur together in the assembly. During the design process the pair may be moved to another location within the assembly, but they are always kept together. Other common feature groups include the nut, bolt, and hole group, and the pair of mating surfaces group. It is these groups that form the high-level entities in terms of which the designer reasons about the design. Since these groups are properties of the assembly rather than individual parts, we term the technique *Assembly-Centric Design.*

There are many software technologies to be developed to support convenient assembly-centric design. This dissertation focuses on just one: the computer generation

of part shapes. A part's geometry is made up of application features and bulk shape. Application features are defined and manipulated in groups at the assembly level. The bulk shape of a part must obey certain constraints, such as noninterference with other parts, minimum allowable thickness of the part, etc., but is otherwise somewhat arbitrary. This presents an opportunity for automation. Using the application features as input, the software can define a shape for the material that holds them together.

The advantages of assembly-centric design are ultimately economic: the amount of input required from the designer is dramatically reduced because application-feature-groups tend to have simple parametric descriptions, and the bulk shapes of parts are synthesized automatically. More complete exploration of the design space is facilitated because feature groups are treated as groups and can be conveniently moved around within the assembly. Such movement implies changes to the bulk shapes of the parts, but these changes are made automatically. Manufacturability of the component parts can be enhanced because the shape-synthesis software has the freedom to tailor part shapes for particular manufacturing processes.

S. Shankar Sastry
Dissertation Committee Chair

To my Dad,

without whose precedent I would never have stayed in school this long.

# Contents

# List of Figures

## Acknowledgements

My years at Berkeley have been both stimulating and enjoyable, filled with interesting research topics and projects. These experiences may be attributed to the advice, guidance, support, and enthusiasm of my advisor, Professor S. Shankar Sastry. To him I am indebted.

I thank Professors Kris Pister and Paul Wright for reviewing this dissertation. I also thank Carlo Sequin, and Paul Wright for serving on my qualifying examination committee, and Professor John Canny for serving as the chair of that committee.

The interaction between the graduate students truly makes the Berkeley experience special. I am grateful for the knowledge gained and shared from these encounters: Matthew Berkemeier, Paul Debevec, Neil Getz, Raja Kadiyalla, Brian Mirtich, Richard Murray, Ed Nicolson, Eric Paulos, Kris Pister, A. K. Pradeep, Adam Schwartz, Dawn Tilbury, Aaron Wallack, Greg Walsh, Joe Weber, Jeff Wendlandt, and to all the rest who have helped along the way.

I especially must thank Roger Bush, whose contribution to this research has been fundamental. The ideas contained herein are as much his as they are mine, forged by untold hours of discussion between us. The synergy between he and I is one of a kind, and I am indeed privileged to have been a part of it.

My work at Berkeley has been generously supported by the U.S. Army Research Office under grants number DAAL-91-G-0191, DAAL03-92-G0124, and DAAH04-94-G-0211; by IBM under a Manufacturing Research Fellowship; and by Scientific Systems Company, Inc., under contract number 1194 (U.S. DOD SBIR contract number N00014-95-C-0382). I thank these agencies for their support and funding.

I thank my family for providing an excellent nurturing environment. A lot of credit must go to my father, mother, and sister for providing the love and backing and to instill in me the importance of higher learning.

Mostly I thank Tracy, Katie, and Kelsey for their love, support and patience which started even before my graduate career; thank you for all the years and all the future.

# Chapter 1

# Assembly-Centric Design

Today's mechanical CAD programs require a user to specify a design using mostly explicit geometry. However, a CAD application that relies on explicit geometric input is not well matched to the design process and thus is usually relegated to the task of design documentation rather than a true aid to designers [?]. In the early stages of design, only a fraction of the geometric elements is known, yet solid modeling software requires models to have physically realizable shapes, even for simple visualization. At any stage of design, routine changes such as resolving interference between two parts may require extensive alteration to the geometric database—a laborious process for the designer.

Not only is it time-consuming to edit explicit geometry, it has been noted that geometry alone is incomplete as a design specification language [?]. After a design is complete, much valuable information is not represented. It is difficult to decide what can be changed, what parts of the design are crucial, and why certain decisions were made in the first place. In fact, a major impetus for *feature recognition* research is the inference of knowledge lost during the design process [?]. A method of specification that facilitates the design process and is richer in information is necessary as a foundation for future mechanical CAD systems. Further, if it is to be useful, this method of specification must have a way of automatically resolving itself into a consistent design that is expressed in explicit geometry.

Approaches to better specification paradigms can be roughly grouped into geometry-based and artificial intelligence (AI) based. The geometry-based approaches combine pieces of geometry into groups and augment them with higher level information. These are generally classed as *Feature Based Design* methods. The term *feature* is subject

to a variety of interpretations. In its broadest definition, "a feature is an element used in generating, analyzing, or evaluating a design."[?] Design-with-features approaches are similar to solid modelers in the level of automation they provide. They still require that parts be explicitly defined by a combination of features—the emphasis is on recording more information than just geometry. In contrast, AI-based paradigms usually seek to provide automation by reasoning in terms of higher-level entities like machine components. They include constraint satisfaction systems and knowledge-based systems. These methods tend to be domain dependent (e.g. for designing helical gears), and do not handle the geometric design well, if at all [?]. For example, an AI system might determine the required size of a bearing, or select an appropriate motor, but rarely is consideration given to designing appropriately shaped parts to hold that bearing or motor in place.

## 1.1 Assembly-Centric Design

A new approach that holds promise is based on the realization that any mechanical part is defined by two kinds of geometry: features that are critical to its function (*application features*), and the material that merely fleshes out the rest of the part (*bulk shape*). Several authors have made this observation [?, ?, ?]. Shapiro and Voelker [?] introduce a new view to interpret mechanical function in terms of energy exchanges and to consider some key portions of geometry as *energy ports*. These are the discrete subsets of a system's physical boundary through which it interacts with its environment. Energy ports on a mechanical part exist at contact surfaces, such as the internal surface of a bearing bore. Their shape is usually dictated by the nature of the contact (e.g. once a specific bearing has been selected, the dimensions of the bore are implied). Thus, the geometries of the energy ports are fully specified, and can be considered to be the *application features* of the part to be designed. In other words, an application feature is a *known geometric solution to a local problem.*

The second important realization is that application features arise in groups. These groups are not properties of any one part, but rather are properties of the assembly. Figure 1.1 shows a typical feature group in cross-section. This is a *hole-bolt-hole* group, which contains three application features: two holes and a bolt. One hole is a feature of something called part $X$, the other is a feature of part $Y$. The bolt is a feature that defines an entire part.

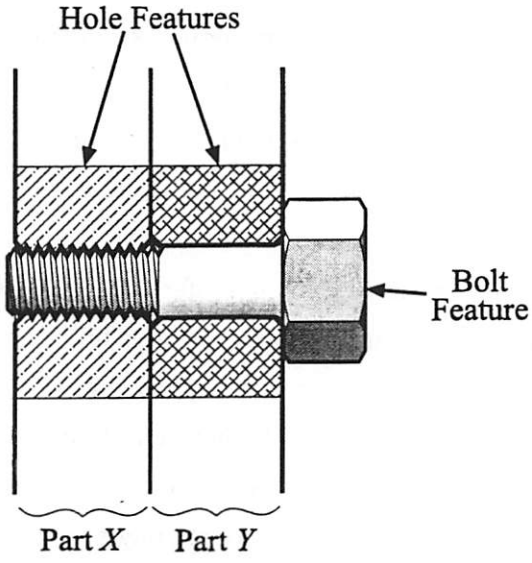



Figure 1.1: A typical feature group, hole-bolt-hole, is shown in cross section. The hole on the left is a feature of part $X$, and the hole on the right is a feature of part $Y$. The bolt is a feature unto itself.

Groups are atomic, parametric, and functional. By atomic we mean that the features belonging to a group are inseparable. For example, you can't have a bolt without two holes. If the designer moves the bolt a centimeter to the left (relative to the rest of the assembly), the holes must move too. Groups are parametric, meaning a group can be fully specified by its type (e.g. it's a hole-bolt-hole group) plus a finite set of scalar parameters such as lengths, diameters, etc. We say that groups are functional because the function of the assembly is fully defined by the set of feature groups. Groups are the entities in terms of which the designer reasons about the design.

Because of this, in a typical design cycle, the geometries of the application features of the parts are known by the designer before that of the bulk shape. A missing link in current approaches to design automation is the automatic synthesis of bulk geometry. Since a part's function is determined by its application features, it is possible to automate the synthesis of the bulk shape, using the application features as a design specification. A few authors have begun to explore *shape synthesis* on these terms. Duffey and Dixon [?] automate the design of cross sections for extruded beams given load and support points and forbidden areas. Graham and Ulrich [?] automate the design of 2-D bending patterns for sheet metal parts using path planning and iterative refinement. Given application

features and paths, Shimada and Gossard [?] generate skinning boundaries by solving for the shape of a deformable curve in a potential energy field.

This dissertation proposes a design environment wherein the designer manipulates assembly-level feature groups, and the bulk shapes of the parts are generated automatically. In the system we envision, user input takes the form of interactively specified application feature groups, design constraints, manufacturing process information, prefabricated vendor parts, and ranges of motion. Higher-level information, such as the kinds associated with design-with-features, is not required by the shape synthesis engine. It requires only that a geometric representation be extractable from each feature, that each feature's membership in a specific part be indicated, and that the relative motions of those parts be available. Part geometry need not be completely specified by the user. Much of the geometry of the design is then synthesized automatically by the program.

In contrast to currently available CAD modelers, this approach will strongly support the early stages of design (conceptual design), where widely varying concepts are explored. There are several advantages of such a system over a conventional solid modeler:

- Far less tedium is imposed on the designer because the need for explicit detailed geometric specification is greatly reduced.
- Rapid design space exploration is facilitated and thus can be much more thorough.
- Manufacturing process knowledge can be applied at the part synthesis stage to yield parts that are cheaper and faster to manufacture.
- Design of the assembly sequence can influence the shape synthesis process and vice versa.
- The design is "live." If a subdesign is dropped into a larger context or assembly, interferences and conflicts can be automatically resolved.
- A measure of design intent is implicitly encoded. Less documentation and explanation is needed to pass a design from one designer to another because the design is represented in a more functional way.

### 1.1.1 Feature Groups

In the system we envision, application feature groups would live in a repository as parametric models, or, they may be attached to geometry provided in on-line catalogs. A group editor would provide the user with the ability to add new group types to the repository. In Figure 1.1, dimensions of the hole diameters, depth, thread pitch, etc., are all parameterized. The group also carries with it the stipulation that one hole must be assigned to one part, while the other hole must be assigned to a different part. The bolt is its own part. To instantiate the group, the designer provides numbers for the parameters and assigns the features to appropriate parts.

Feature groups may also be hierarchical. For example, a bolt pattern could consist of four hole-bolt-hole groups located at the corners of a rectangle. The four groups would use the same values for corresponding parameters, and two additional parameters would specify the dimensions of the rectangle. The hole-bolt-hole groups are *subgroups* of the bolt-pattern group.

Often, vendors offer on-line catalogs containing CAD models of off-the-shelf parts they supply. Ideally, these models will have been built using assembly-centric technology. For example, a gearbox may have a flat bottom and a bolt pattern for mounting. To include the gearbox in an assembly, some other part must provide an appropriate mating surface and four threaded holes in a matching bolt pattern.

When the model of the gearbox is created by the vendor, a bolt-pattern group and a mating-faces group are instantiated to define the geometry of the gearbox. Some of the features in those groups are assigned to the gearbox (i.e. one hole from each hole-bolt-hole subgroup). However, the rest of the features remain unassigned. This is the state in which the model exists in the catalog. When the designer instantiates the gearbox in his assembly, the gearbox model bristles with unassigned features (i.e. holes and mating faces). A new part must be created in the assembly to which these features can be assigned. The shape of this part may then be automatically generated such that it provides the required geometry for mounting the gearbox.

### 1.1.2 A Design Example

In this dissertation, the user interface is not studied in detail. The substantive work to be presented will focus on shape synthesis. However, it is important to have a

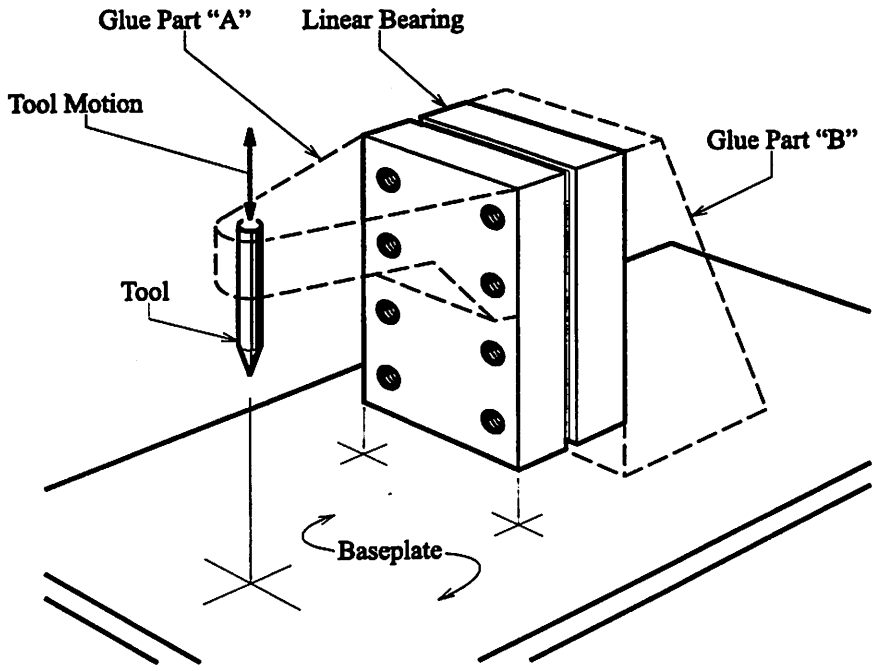



Figure 1.2: A design example.

context to guide our research. We envision a *shape synthesis engine*, which can generate the bulk shape of a part that is defined by application features. One could imagine this being used as an *agent* in a system of collaborating software agents (an AI approach). Alternatively, a feature-based design environment could require the user to manipulate feature groups directly. All reasoning about the design would be provided by the user. Synthesis of bulk shapes is then provided by the shape synthesis engine. We will assume this latter model.

To illustrate the utility of this manifestation of assembly-centric design, we describe a design example: Suppose a design specification requires that a tool have a linear motion along a given axis over a given range (see Figure 1.2). This is the only design requirement; any other parts introduced will exist solely to support this single function.

The designer knows that there must be a linear bearing with its axis parallel to the tool's. Further, she knows that certain connectivity relationships exist: one side of the bearing must be rigidly coupled to the tool, the other side to the machine baseplate. These relationships imply the existence of *glue* parts (shown in dashed lines in the figure), whose sole function is to hold the primary parts fixed in proper orientation to each other.

In the design system we envision, the designer would proceed as follows: Since she knows there must be a linear bearing, she selects one from a CDROM catalog supplied by the bearing vendor. She then interactively positions it in three-dimensional space relative to the baseplate. Since the bearing was modeled with assembly-centric technology, it bristles with unassigned features from the groups it comprises. These features must be assigned to other parts in the assembly.

She creates a new part ($B$ in the figure). At this point it is a null part because it has no features assigned to it, and no geometry. She selects all of the unassigned bolt hole features from one side of the bearing and assigns them to part $B$. Next she instantiates a bolt-pattern feature group. She assigns some of the hole features to part $B$ and the rest to the baseplate. Bulk geometry for part $B$ is then automatically synthesized as shown in Figure 1.2. The shape synthesizer takes care that this geometry does not interfere with the bearing throughout its range of motion. Further, it is manufacturable by the currently selected process, say milling with a three-axis, numerically controlled milling machine. If she decides that the part should be a little more rugged, she increases the minimum allowable thickness for the part and it is automatically resynthesized.

The designer next chooses a tool from the on-line catalog and positions it in space. This tool contains an unassigned hole feature necessary for its mounting. She creates another null part ($A$ in the figure), and assigns to it the hole plus the remaining unassigned bolt-hole features from the bearing. Bulk geometry is automatically synthesized for part $A$ such that it doesn't interfere with other moving parts, obeys the minimum thickness constraint, and is manufacturable. If she decides that the tool must be repositioned, the designer simply drags it to the correct place. As she does so the glue part is resynthesized to accommodate the new position while satisfying all the other design requirements.

With this design stage completed, this subassembly can be imported into a higher-level assembly. As it is positioned in the assembly, there may be several points of interference that must be resolved. Synthesized portions of parts in the assembly and subassembly can be automatically adjusted to eliminate many of these, minimizing the burden on the designer.

A basic tenet of our research is that the CAD system and the user should work in concert, and the link between them should be a highly interactive interface. Essential to this vision is that the system be capable of very rapid synthesis of shapes, and support interactive 3D editing. The application is intended to run on high-performance 3D graphics

workstations. By continuously resynthesizing shapes at or near frame rate, we create the illusion that the bulk shapes of the parts are malleable. For example, when the designer interactively moves a feature, the part will seem to stretch to accommodate. When a new part is added to a crowded assembly, all of the parts give a little where they can to make room.

Further, we strive to minimize the volume of information required from the user, particularly during conceptual design. For example, we do not want to require the designer to specify all of the loads that will be applied to each part. In many cases this would be inappropriate. Designers often don't know what the loads will be beyond the points of application and a general feeling of "large" or "small." Further, it is often desirable to design a part to withstand not just service loads, but general loads from all directions, to make it robust to unmodeled loads, assembly, and accident. This is especially true when designing prototypes, tooling, and other applications where parts are traditionally designed "by eye." Our approach will be to synthesize designs that look reasonable, and present them to the designer for assessment. If the part looks too weak, she may choose a different style, or may issue commands like "fatten it up here," until the part is satisfactory. In other words, the system we envision is not so much one of design automation, but one of designer facilitation.

Since the shape synthesizer will decide on the specific part geometry, manufacturing process knowledge can be applied at the part synthesis stage to yield parts that are cheap and easy to manufacture. If geometric operations that correspond to actual manufacturing operations are used (e.g. mill, turn, drill, etc.), a construction plan is implicitly generated with the part. This simulates the destructive solid geometry approach [?] without requiring the designer to work in terms of these low-level primitives. Global part properties, such as manufacturing process or style of construction, may be changed with relative ease because the shape synthesizer is responsible for the details of the geometry and the manufacturing plan.

### 1.1.3 Observations

Several observations support the viability of this paradigm:

- During conceptual design exploration, only the application features are interesting, e.g. bearing bores, bolt patterns, and load contact points.

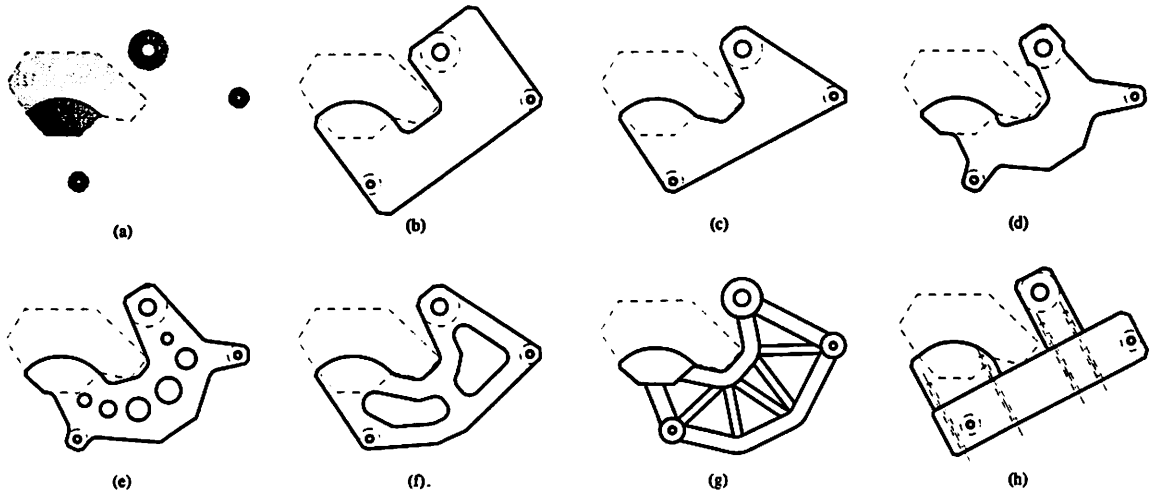



Figure 1.3: A specification and seven conforming designs.

- The bulk shape of a part determines much about the manufacture of the part and vice versa.
- Every part design starts with an initial guess. The refinement of this design depends on the relative importance of manufacturing costs, part performance, etc.

The first observation, as Shapiro and Voelker [?] point out, is that in a typical design cycle the geometries of the application features are known by the designer before that of the bulk shape. In other words, during conceptual design exploration, only certain details of each part are interesting—the *application features*. The rest of the shape of the part, the *bulk shape*, is much more arbitrary. It must obey geometric constraints such as connectivity and noninterference with other parts, but this allows an infinite variety of actual shapes.

Consider the example in Figure 1.3, which shows specification for a 2-D mechanical part and several conforming designs. In this example, a part is needed with several holes and a curved edge (the application features). For the purposes of designing a single part, we may ignore the assembly-level feature groups and focus on just those application features assigned to this part.

Strength requirements dictate a minimum thickness of material around each hole and behind the curved edge (dark shading). The curved edge contacts some object during operation, which gives rise to an *illegal region* that the part must avoid (light shading).

There are infinitely many part designs that satisfy this specification, and seven such solutions are shown. They all share the same application features, yet differ in their bulk shapes. (See also Figure 1.7.)

The second observation is that the bulk shape of a part determines much about the manufacture of the part and vice versa. In many cases, a designer will choose the bulk shape of a part based on knowledge of manufacturing considerations. A shortcoming of using a conventional solid modeler (or even sketching on the back of an envelope) is that since the designer is forced to give *some* shape to each part, she introduces a bias about how that part is likely to be manufactured long before it is desirable to do so. For example, if a part is initially modeled as though it will be milled, it becomes difficult to alter the CAD model to show the part as it would be if stamped and bent from sheet metal.

Another fundamental manufacturing issue is the choice of stock from which the part will be made. There might be an alternative between hogging the part out of a single block versus welding it from bars and plates. Even the simple flat-plate part in Figure 1.3 can be manufactured in many different ways. Design (b) could be made easily on a three-axis milling machine with mostly $x$ and $y$ cuts. If the scale of the problem is larger, design (h) would make more efficient use of stock. Designs (e) and (f) might be appropriate for stamping or if weight is to be minimized. At very large scales, design (g) could be made from bent and welded bars or box-section beam. Ideally, the designer would be able to browse through various manufacturing options at any point in the design process, thus freeing the conceptual phase from such concerns. Conventional CAD systems offer no support for exploring the design variations necessary to accommodate this.

Our final observation is that all part designs require an initial guess. Depending upon the domain, this guess will get refined in different ways. A component in the landing gear of an aircraft, for example, will be subject to iterations of finite element analysis and shape revisions until a satisfactory strength to weight ratio is achieved. In some other domains manufacturing and assembly concerns dominate. In high-volume products, only the most cost-efficient processes are practical, like injection molding and sheet metal stamping. The nature of the chosen process will have a strong influence on the bulk shape of the parts.

On the other hand, a large class of design problems are best satisfied by parts with simple shapes because these are quick to manufacture with processes like milling. Examples include prototypes, custom laboratory equipment, tooling, jigs, fixtures and

even low-volume for-market products. Such parts are traditionally designed by intuition and rules of thumb. Mechanical requirements, like strength and stiffness, are met simply by oversizing the parts by a large margin. The primary consideration for such parts is the ability to quickly design and manufacture them.

In all of these cases the bulk shape of the part must obey geometric constraints. In the latter cases, design criteria such as ease of milling can be translated to geometric constraints as well, such that satisfactory shapes could be generated automatically. In the case of more highly engineered parts, like aircraft parts, the synthesizer can generate a sensible initial guess for input to the refinement process.

## 1.2 Shape Synthesis

**Conjecture:** Shape synthesis is really not that difficult.

Some of the original motivation to do this research was the impression that the designs of simple parts are often "obvious." In other words, perhaps shape synthesis for simple parts can be automated using relatively simple and fast algorithms. To find out if this is so, we embark on this research to find out what it takes to formulate part designs.

We begin by asking the question, what should synthesized part shapes look like? To answer this, we start by categorizing the kinds of requirements that mechanical parts must meet. Though not comprehensive, the following list includes the most common and most important categories:

1. Connectivity: A part must be a single, connected, physical body.
2. Noninterference: The part must not interfere with any other parts in the assembly, their paths of motion, nor with illegal regions arising in any other way.
3. Strength and Stiffness: The part must be strong enough not to fail in service or during assembly, and it must be stiff enough to perform its function correctly.
4. Manufacturability and Assemblability: The part design must be able to be fabricated by cost-effective means, and the assembly itself must be able to be assembled efficiently.
5. Dynamics: A moving part must meet acceptable ranges for mass, moments of inertia, resonant modes, etc., as defined by its function.

6. Electromagnetic characteristics, aerodynamics, hydrodynamics, aesthetic styling, thermal expansion, etc.

This list is shown in an approximate order. Those requirements near the top must be met by *all* parts, regardless of the application. As we move down the list, the requirements become progressively more specialized, and may or may not be important for a given part.

### 1.2.1 Division of Labor

We have chosen to divide the synthesis problem into two steps. In the first step we will generate a *skeleton* which abstracts the structure of the part. That is followed by a *material synthesis* step which fleshes out the actual shape of the part using the skeleton as a guide. This is identical to the procedure of Shimada and Gossard [?], who apply the term skeleton to the energy paths of Shapiro and Voelker [?]. We also note that the skeleton is analogous to the STICKS diagram [?] used in CAD for integrated circuit design.

There are alternatives to this two step approach. For example, Duffey and Dixon [?] use recursive application of topological operators to generate designs for extrusions. In figures 1.3 and 1.7, there are examples which could be synthesized in a single step. Design 1.3(b) is a modification of a rectilinear bounding box. Designs 1.3(c) and 1.7(b) are modifications of the convex hull of the features. In these cases the modification is essentially the removal of illegal regions.

In more complex cases however, these simple design styles will need some help. For example, if an illegal region completely divides the bounding box or convex hull, it becomes unclear where material must be added to make a connected part. Further, the more refined design styles are much easier to generate if we first abstract their structure with a skeleton. Designs 1.3(d), (e), (h), and 1.7(d), (g), and (h) are based on tree-type skeletons. Designs 1.3(f), (g) and 1.7(b), (e), and (f) are based on triangulations.

The skeleton also has the advantage of being an intuitive intermediate abstraction. If the designer feels the need to make minor modifications to the synthesized skeleton, it would be natural to add or delete paths to change its overall character—a process much faster than editing explicit geometry. In this way, the skeleton can be seen as a tool to manage the vast complexity of the design space. It is a representation of an infinite family of parts with the same topology, but different geometry.

Skeletons (which have zero width), will be synthesized so that they can be widened by some amount without interfering with illegal regions. This amount will be dictated by the current design rules. The first action of the material synthesizer is to perform this widening, yielding a *thick skeleton.* This thick skeleton is essentially the minimally sufficient part, satisfying the required connectivity and providing at least the minimum allowable material thickness everywhere. The remainder of material synthesis adds material to the thick skeleton to improve manufacturability, reworkability, and assemblability. Figure 1.7(a) and (c) depicts two styles of thick skeleton for the same specification.

A further advantage of separating the synthesis process into two steps is that it allows us to mix and match skeleton and material synthesis algorithms and thus multiply the number of design styles we are capable of generating. Some combinations may not be as useful as others, of course, but we should still realize a rich selection.

## 1.3 Program Structure

For completeness we outline the software architecture of the complete assembly-centric design environment. Subsequent chapters will be concerned only with those aspects relevant to shape synthesis. Figure 1.4 depicts the program flow schematically. A strength of this paradigm is that it separates into fairly independent components, and it allows a single-pass program flow (complications are discussed in Section 1.3.6). In the diagram all information flows down. The user inputs a functional description of the assembly by instantiating feature groups. These are groups of application features, which are assigned to various parts in the assembly. To instantiate feature groups, the designer specifies their relative positions in space, and provides values for the parameters specific to each group type.

Much of the kinematic behavior of the assembly will be implicit from the instantiation and placement of the groups and assignment of their features to specific parts. For example, a bore-shaft-bearing-bore group implicitly defines a revolute joint. When one bore is assigned to (say) part $A$ and the other bore is assigned to part $B$, the two parts are constrained to have one revolute degree of freedom between them.

The designer must also be able to stipulate arbitrary motion constraints when the implied constraints do not sufficiently define the assembly's function. There may also be

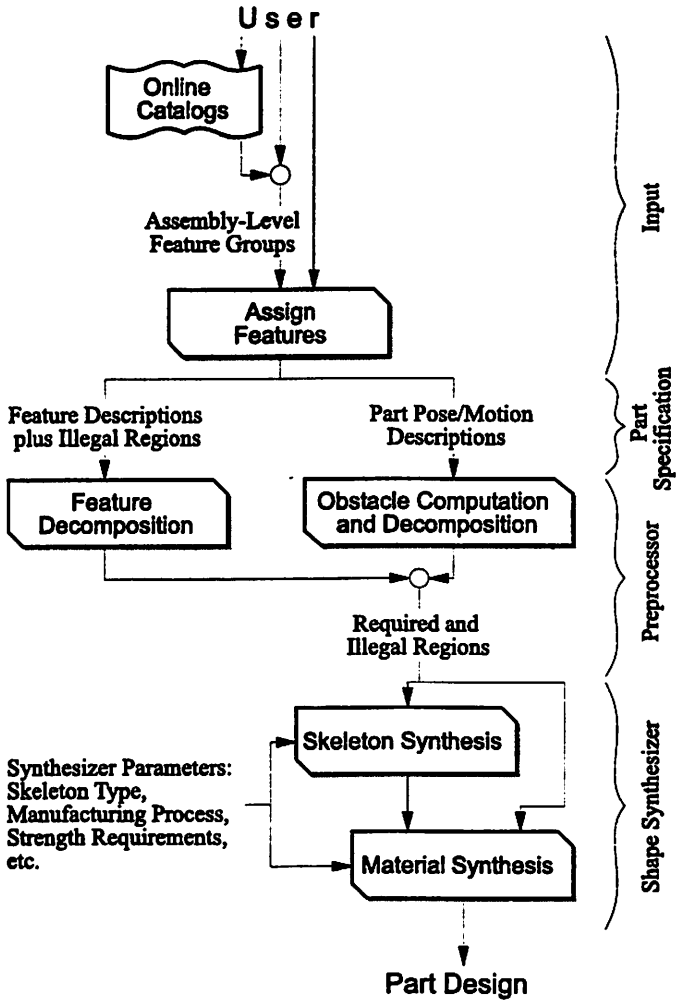
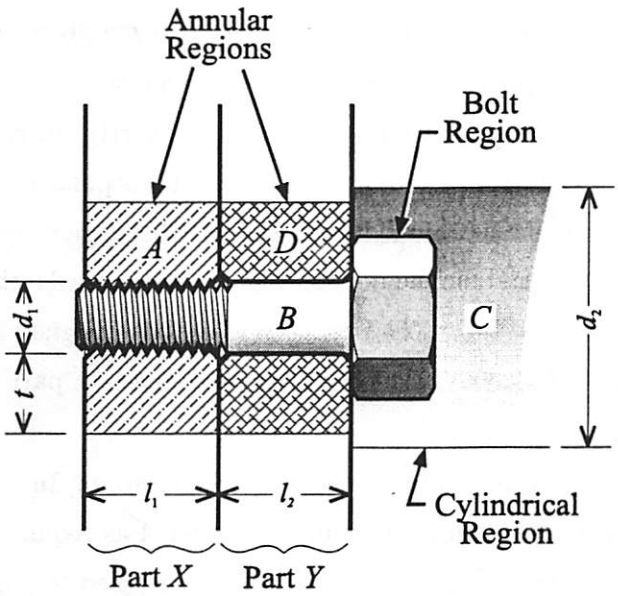



Figure 1.4: Program Flow.

stationary or moving illegal regions that the designer wishes to reserve so that no material will be synthesized in them. This information is preprocessed to yield a set of required and illegal regions. The set is then passed to the shape synthesizer, which consists of the skeleton and material synthesizers. In the following sections, we look at each of these in more detail.

### 1.3.1 Input

As stated above, the input to our hypothetical system includes application feature groups, external part positions and motions, and reserved illegal regions. In an ideal




| Feature | Region | | | |
|---|---|---|---|---|
| | *A* | *B* | *C* | *D* |
| left hole | + | − | − | |
| bolt | | + | | |
| right hole | | − | − | + |

Figure 1.5: A feature groups is composed of required and illegal regions. Features are lists of regions with polarities.

implementation the user would select feature groups from a repository. Off-the-shelf parts would be modeled using assembly-centric technology and would be available from an online catalog. The individual features in each group must then be assigned to specific parts, thus defining the specification for each part.

Figure 1.5 shows the *hole-bolt-hole* feature group, which contains three features. To instantiate this group, the user supplies values for the various parameters ($t, d_1, d_2, l_1, l_2$, thread pitch, etc.), and assign each feature to a different part. In Figure 1.4, the information yielded by the input process includes application feature group instantiations, their relative positions, general illegal regions, and motion descriptions.

### 1.3.2 Feature Group Decomposition

In practice, application feature groups or even individual features are too high-level to use as input to a shape synthesizer. The shape synthesizer developed in this dissertation will operate on more primitive input: systems of *required regions* and *illegal regions*. (These may also be thought of as positive and negative volumes of material.) Figure 1.5 shows a number of regions that make up the three application features of the

hole-bolt-hole group. The true nature of an application feature is that it is simply a list of some of the regions from the group, plus a polarity for each region on the list.

For example, in Figure 1.5 there is a region (volume of material) exactly the size and shape of the bolt. The bolt feature lists this as its only region, whose polarity is positive: the region is required for the bolt. Each hole feature also lists this region, but with a negative polarity: the bolt region is illegal for the hole features. In other words, the part ($A$) must have a hole the right size for the bolt[1]. The bolt feature will be assigned to the bolt part, and it will be the part's only feature. That being the case, the bolt part is fully defined and requires no shape synthesis.

The table in Figure 1.5 depicts the features in the hole-bolt-hole group. In addition to listing region $B$, the *left hole* feature also lists the annular region $A$ as required, and the annular region $D$ as illegal. In Figure 1.5, the *left hole* feature is assigned to part $X$; the *right hole* feature is assigned to part $Y$.

The feature group in Figure 1.5 also contains a semi-infinite cylindrical region ($C$) that reserves a path for the bolt to be inserted during assembly. The hole features both list this region with a negative polarity: the region is illegal for parts $X$ and $Y$. In addition, this illegal region must be propagated to all other parts in the assembly, or at least to those whose position in the assembly sequence is earlier than that of the bolt. This assures that the bolt will indeed be able to be inserted when the parts are assembled.[2]

In Figure 1.4 the output from the feature group decomposition step is a system of required and illegal regions for each part. This is the form of input suitable for the shape synthesizer.

### 1.3.3 Obstacle Computation

Let us use the term *target part* to refer to the one currently being synthesized. As described above, illegal regions are generated by feature group decomposition, and may also be input explicitly by the user. In addition, illegal regions arise because, from the point of view of the target part, all other parts define illegal regions. By extension, if a

[1]This is somewhat of a simplification. In practice there would probably be two bolt-shaped regions, one slightly larger than the other, but occupying roughly the same position in space. The inner region would be the required region for the bolt feature. The outer one would be the illegal region for the hole features. The difference in their sizes would allow clearance between the bolt and the holes.

[2]From this it becomes clear that assembly-centric design should also involve planning the order in which parts are assembled. See Chapter 5 for some thoughts on further research in this area.

part moves relative to the target part, the entire space it sweeps out, as viewed from the coordinate frame of the target part, is an illegal region. Note that our whole approach of starting with illegal regions and synthesizing noninterfering parts contrasts with the method supported by traditional CAD systems. There the designer models all of the parts, then has the software check for interference. If any is found, she edits the offending geometry.

In our proposed system, all applicable illegal regions must be computed so that they can be used as input to the shape synthesizer. In the case of parts that are stationary relative to the target part, this is trivial. For parts that move relative to the target part, we must do some work. Though it may at first appear difficult to synthesize shapes in the presence of moving obstacles, it turns out that the problem of synthesis is separable from the motion if we make one assumption: we must have full descriptions of the relative motions of all parts in the assembly. As noted in Section 1.3, this will be easy for the designer to input, in many cases, because degrees of freedom are encoded in feature groups. In addition, catalog models, such as the bearing in Section 1.1.2, would encode degrees of freedom that they provide.

With this information in hand, we can compute *static* illegal regions for all parts in the assembly. This is important: **shape synthesis for the target part can always be done in a static environment.** The illegal regions that populate this environment are the swept volumes of the other parts in the assembly. Each part is swept along its degrees of freedom, as viewed from the frame of reference of the target part. If a part has multiple degrees of freedom relative to the target part, it must be swept recursively, starting with the axis most distal from the target part along the kinematic chain. The process is similar to the computation of the work envelope of a robot [?, ?].

An example is shown in Figure 1.6. At (a) the target part, a clamp, is shown as a set of desired features. It is to have one revolute degree of freedom relative to the base plate, with joint limits at +45 and −30 degrees. At the left is a stage which has a prismatic degree of freedom relative to the base. The stage and the clamp thus have two degrees of freedom relative to each other. To compute all the illegal regions that apply to the clamp, we first attach ourselves to its frame of reference. Starting with the part farthest down the kinematic chain (the stage), we sweep along its axis through its complete range of motion. This gives rise to an intermediate illegal region, shown (shaded) at (b). Proceeding along the kinematic chain toward the target part, we sweep these intermediate

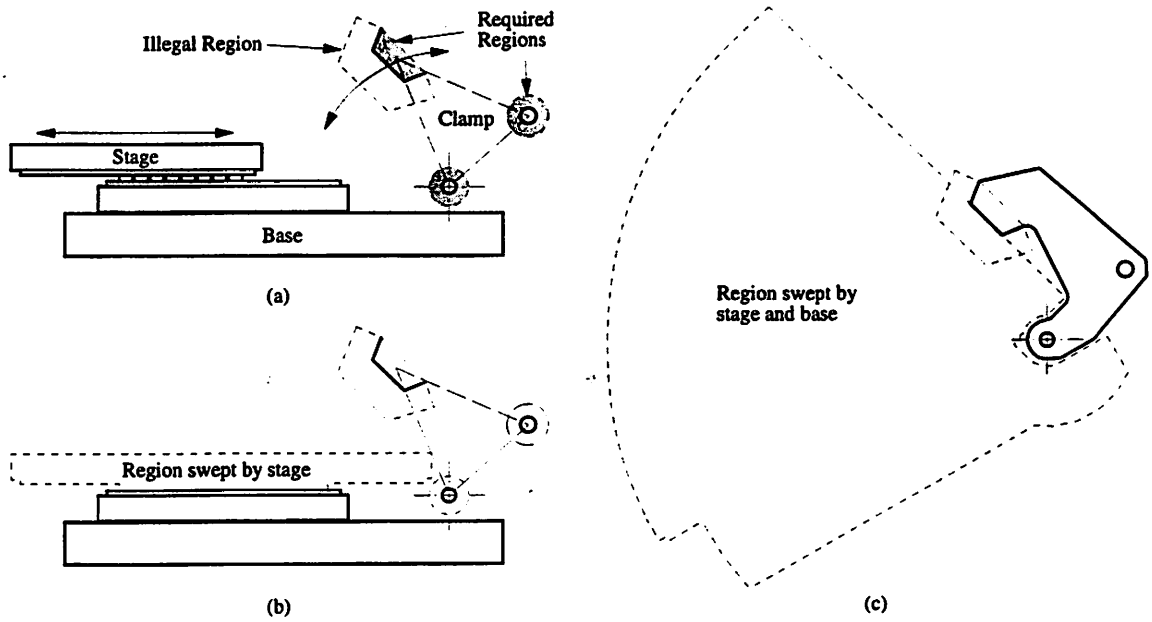



Figure 1.6: Generating illegal regions by sweeping.

regions, plus other parts, along their degrees of freedom. The final illegal region applicable to this target part is shown at (c), along with a possible design.

In this example, we have assumed that the individual degrees of freedom are fully independent. This is the most conservative assumption, yielding the largest illegal regions, but it may not always be appropriate. There could be mechanical stops, or the actuation could be under software control, such that, say, the clamp only moves when the stage is at one specific position. To handle this we need only modify the sweeping process slightly. Rather than sweeping the intermediate region recursively, the stage would be swept along both axes separately, and the union of the two resulting regions would be used as the illegal region.

Any kind of interdependence between part motions can be handled similarly, as long as we are provided with an accurate description of the rules governing these dependencies. Note that if the dependencies are imposed by software control, it is worthwhile to discard them wherever possible and use the most conservative illegal regions. This guarantees that no software error can ever cause a physical machine crash. We also note that linkages do not represent dependent degrees of freedom between their links, just complex degrees of freedom. Computation of the swept areas of planar links has been studied by

Ling and Chase [?].

Readers familiar with configuration space approaches used in robotics literature for path planning may worry about the "curse of dimensionality," which makes problems intractable when they must be solved in high-dimensional spaces. We point out that there is a subtle difference between the robotics problem and the shape synthesis problem. This difference is crucial: in robotics, we are given the shapes of all the parts and the motion is unknown. In shape synthesis, we assume that all motions are known and we must find the shapes of the parts. The dimension of the space in which the shape synthesis problem must be solved is the same as that of the physical space in which the part will live (2 or 3).[3]

### 1.3.4 Skeleton Synthesis

Once the user input has been preprocessed to yield primitive input in the form of required and illegal regions, shape synthesis can begin. As described in Section 1.2.1, we have chosen to divide this into two steps: skeleton synthesis and material synthesis. This section outlines skeleton synthesis, and the following section outlines material synthesis.

We reiterate that shape synthesis is the primary focus of this dissertation. This description of the design environment is purely to provide a concrete context in which to frame the shape synthesis problem. In the remaining chapters of this dissertation, we will place proportionally more weight on skeleton synthesis, but that is only because there is not time to study all these areas in detail.

We represent a skeleton as a graph plus a Euclidean embedding (sometimes called a *diagram*), where the embedding may have non-straight edges. Graphs include several specializations, such as triangulations and spanning trees, and there are two broad categories of approach: computational geometry and numerical optimization. Of interest to us is a class of definitions that require the graph to meet some cost based on geometric characteristics of the edge set, such as length of edges or stiffness of node-to-node paths. Many such graphs are treated in the literature, including the Euclidean minimum spanning tree, Steiner minimum spanning tree [?], $t$-spanners [?, ?], and Delaunay triangulations [?]. The differences between these are the nature of the cost function that the graph must satisfy, and whether the algorithm can add nodes. The investigation of skeleton-generating

[3]Exceptions to this may arise if we wish to parameterize some aspect of a part's shape (see Section 5.1.2), but they do *not* arise from degrees of freedom of moving parts.

algorithms is the topic of Chapter 2.

Regardless of the algorithm we use to create the skeleton, we must ensure that it can be widened appropriately to create the thick skeleton without interfering with any illegal regions. In its simplest form, the problem is analogous to finding a path for a mobile robot through a roomful of obstacles, and we can borrow the algorithms used to solve those problems. This technique was investigated by Graham and Ulrich [?]. Most techniques account for the size of the robot by transforming the problem to *configuration space* (CSPACE) [?, ?]. In the case of a circular robot moving through a two-dimensional field of obstacles, CSPACE is parameterized by the coordinates of the center of the robot. For skeleton synthesis, CSPACE is parameterized by the coordinates of a point on the skeleton. *Obstacles* in CSPACE are sets that must not contain any point on the skeleton, lest the thick skeleton interfere with an illegal region in physical space. If the thick skeleton is to be of uniform width, CSPACE is computed simply by growing the illegal regions by one-half the width of the thick skeleton. The skeleton synthesis algorithm can then operate in this transformed space to plan the skeleton, and we will be assured of being able to grow the thick skeleton without interference. In fact, clearance can be added by growing the obstacles an extra amount.

If the thick skeleton is to have varying widths, e.g. tapered struts, CSPACE must acquire an extra dimension to account for the half-width of the skeleton. For a 2-D design problem, CSPACE becomes three-dimensional and obstacles are generalized frustums whose tops are the physical illegal regions, and whose sides have a slope of unity. See Chapter 5 for a discussion of this idea.

In all of the skeletonizing algorithms we will consider, there are two general approaches to avoiding illegal regions. The first implementation generates a connectivity graph without regard for the illegal regions, then uses path planning to route the edges. The second implementation directly generates a skeleton in CSPACE that respects the illegal regions. In most cases this produces a superior skeleton, but many of the algorithms studied in the literature have not been extended to handle obstacles. Chapter 4 discusses the former technique. Chapter 5 suggests some ways to implement the latter.

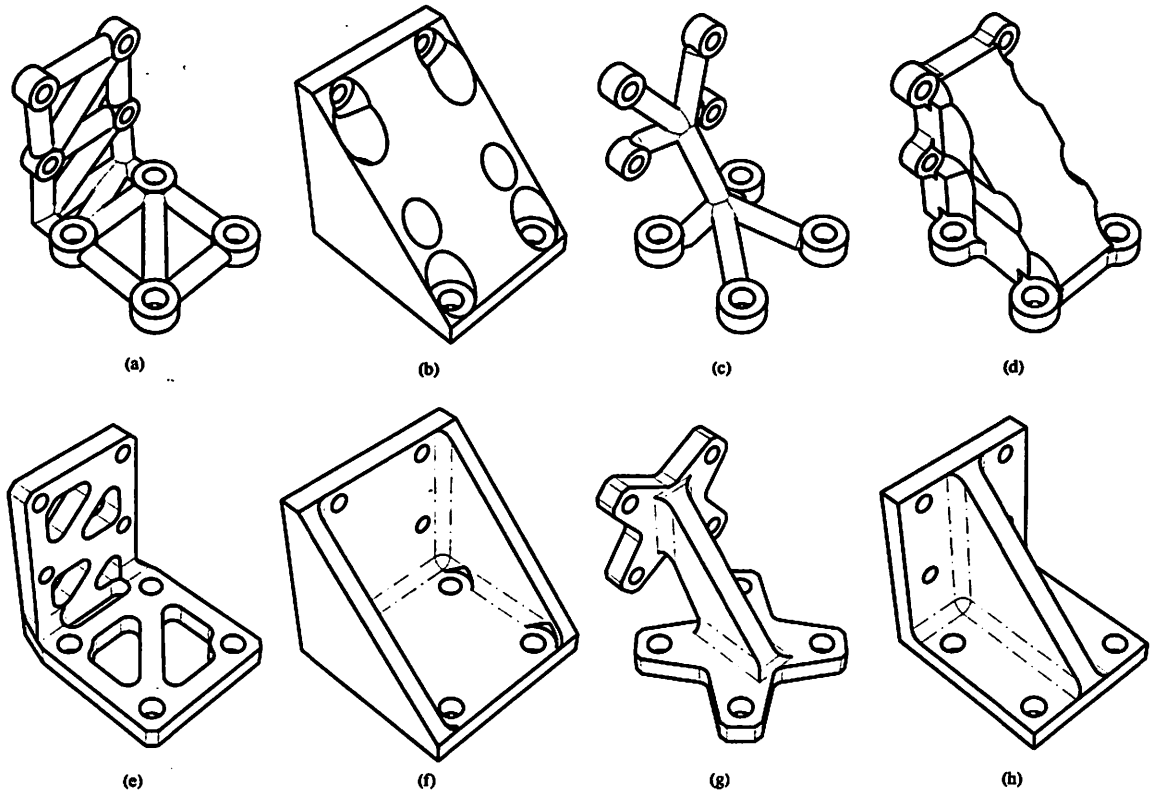



Figure 1.7: Two thick skeletons (A & C) and six designs, all satisfying the same specification (not shown).

### 1.3.5 Material Synthesis

The material synthesizer has the responsibility of generating the final shape of the part. As it does so, it follows design rules relating to minimum material width, manufacturing methods, assemblability, etc. The material synthesizer must guarantee that the finished shape completely contains the skeleton, and that no portion violates any design rules or illegal regions. The output of the material synthesizer is the actual part design. Shimada's and Gossard's [?] elegant method of generating skinning boundaries by placing a deformable curve in a potential field represents one possible material synthesis style. Our goal is to be able to generate a wide selection of part styles, some similar to those shown in figures 1.3 and 1.7.[4]

The simplest synthesis technique is to approximate the thick skeleton using the

[4]It is worth noting that the parts illustrated in fig 1.7 took over 30 hours to model interactively in Pro/ENGINEER [?]. This highlights the difficulty of using solid modelers to explore design space.

selected manufacturing process. For example, if the selected process is milling, material is added to the thick skeleton until the part is composed of flat faces and fillets. Examples are seen in Figure 1.7 (e), (g) and 4.1. An interesting problem to be solved then is how to synthesize skeletons that ensure sufficient clearance for fillets to be added without creating interferences.

Perhaps the next simplest technique is to conceptually wrap a skin around the thick skeleton and pull it tight, letting it drape around the illegal regions. Figure 1.3(c) shows one such design in two dimensions. This can be thought of as a modification of the convex hull of the application features.

More sophisticated material synthesis will add more knowledge about the selected manufacturing process. We envision a planner that reasons in terms of subtractive operations that correspond to milling, turning, drilling, etc., and additive operations that correspond to bolting and welding. This knowledge will be used to add material strategically to yield an easily-manufacturable part. This raises the specter of the well-known difficulties of automatically generating manufacturing plans for a given design. In this case the problem is easier because the synthesizer has the freedom to choose the shape of the part.

In most cases it will be desirable for the material synthesizer to generate axis-aligned geometry wherever possible. This will yield parts that are easier to machine, inspect, and assemble than those with freeform geometry. Figure 1.3(h) displays an example with almost exclusively rectilinear geometry, typical of built-up parts in a prototype. In Figure 1.7, all six designs exhibit geometry that is aligned to one or more principal axes. Solutions (e) through (h) have all geometry restricted to axis-aligned slabs.

In the case of prototyping, evolution of the design often requires part designs to be modified. When possible, it is often faster and more economical to re-machine the same physical part than to make a new part from scratch. It may be desirable for the synthesizer to add excess material judiciously to improve reworkability of the parts.

For this dissertation only the simplest material synthesis techniques were implemented: approximating the thick skeleton. One variation is to construct uniform width spars along each edge of the skeleton. A slightly more advanced technique is to use spars of nonuniform width. These implementations are presented in Chapter 2. See Chapter 5 for further meditations on material synthesis.

### 1.3.6 Complications

There are several issues which complicate using the single-pass approach we have outlined. The most important complication is the simultaneous design of multiple parts. In this single-pass model, a problem arises when we must synthesize designs for all the parts in an assembly: illegal regions cannot be computed for parts whose shape is not yet known. Our first approach to solving this problem is to use prioritization and iteration. The application features of a part should provide a good placeholder for the space that it will ultimately occupy. The designer first inputs all feature groups, assigns features, then to each part she assigns a priority. The synthesizer begins with the highest-priority part, using the application features of all other parts as illegal regions. One by one, each of the parts are synthesized in order of descending priority.

Clearly, prioritization will not always work. It may happen that the $i^{th}$ part cannot be synthesized at all because of the way a higher-priority part was synthesized. Sometimes a reordering of the part priorities will solve this, but not always. In some cases, each part must give a little so that they can coexist. Another approach is to try to carve up space and apportion a region to each part before any synthesis begins. A Voronoi partitioning based on application features is one possibility. In this scheme space is divided into cells, each belonging to a particular part. The shape of each cell is determined so that at any point inside the cell, the nearest application feature is one belonging to the same part as the cell. In other words, the cell walls appear "halfway" between the nearest features of adjacent parts. This may be modified to a *weighted* Voronoi partition if it is necessary to give some parts proportionally more space.

A further complication is that the two stages of shape synthesis may have to become more tightly integrated than we have described. For example, the material synthesizer may be able to recognize potential improvements to the design that require modification of the skeleton it was passed. For instance, it might be that a simpler machining operation would be usable if some portion of the skeleton were moved a bit. In three dimensions, there are many ways to thicken the skeleton. Iteration between the skeleton and material synthesizers is a possible approach to try first, and can be viewed as a form of optimization.

Finally, it remains to be seen how natural it will be for designers to input designs as systems of feature groups. For small assemblies it seems straightforward, but it may

require considerable imagination on the part of the designer for complex systems. We are hopeful that rapid shape synthesis will come to the rescue. If a designer is able to see representative solid bodies for the parts, it should greatly facilitate her understanding of the design she is constructing.

## 1.4 Conclusion

We have presented a blueprint for what we believe will be a genuinely useful new kind of CAD system. Such a system will have clear economic advantages. Assembly-centric design is analogous to standard-cell based CAD for integrated circuit design. Feature groups are analogs of the standard cells, and shape synthesis is analogous to automatic placement and routing of interconnects. It would be inconceivable today to attempt a VLSI design without such tools. We imagine that assembly-centric CAD could be just as indispensable in the future. Minimizing the stream of input required from the designer will not only speed up the design cycle, it will also facilitate vastly more thorough exploration of the design space than is possible with current CAD systems.

In this design paradigm, conceptual design is more naturally supported because arbitrary geometric details are not hard coded into the design. A measure of functional intent is encoded by modeling only application features and augmenting them with relationship information. This functional level requires less information to unambiguously specify than does explicit geometric construction, and is less subject to change over the life of the design. The representation of the design can be thought of as "live," i.e. that the design represents an infinite family of designs that satisfy the design criteria. Further, since it is the computer that designs the specific part geometry, manufacturing process knowledge can be applied at the part synthesis stage to yield parts that are cheap and easy to manufacture. Global part properties, such as manufacturing process or style of construction, may be changed with relative ease at any time during the design cycle.

As far as this dissertation is concerned, the salient aspect of assembly-centric design is that the shape synthesis step is separable from most other concerns. First, it can be decoupled from the choice of data structure used to specify application features. We need only that required and illegal regions be extractable. Shape synthesis is also separable from issues of motion if we assume that complete descriptions of assembly kinematics are available. The important result is that shape synthesis of an individual part can proceed

in a static environment. Thus we can imagine shape synthesis fitting into a variety of software paradigms. It could provide support to an interactive design system wherein the user manipulates assembly-level feature groups (the paradigm sketched in this chapter). In a more automated scheme, a system of collaborating software agents could reason in terms of feature groups, with a shape synthesis engine providing background services.

We conjecture that shape synthesis is not difficult for simple parts, and the remainder of this dissertation will be spent showing that this is true. Chapters 2 and 3 discuss the generation and optimization of skeletons. Chapter 4 examines issues arising from obstacles. Chapter 5 is devoted entirely to research directions and future work. Finally, the appendix provides user documentation for the software.

# Chapter 2

# Part Skeletons for Shape Synthesis

In Chapter 1 we conjecture that reasonable part shapes can be synthesized by fairly simple and cheap algorithms. Since we have chosen to divide the process into skeleton synthesis and material synthesis, one purpose of this research is to examine a wide range of skeleton-generating algorithms and assess their suitability.

As stated in Chapter 1, the skeleton abstracts the structure of the part. It is a connected set of zero-width paths that describe where material will go to connect the application features. The skeleton is a *graph*, in that it represents information about the connectivity of the application features. In graph-theoretic terminology, the application features are *nodes*. Pairs of nodes are connected by *edges*; each edge connects two nodes. The skeleton also has an *embedding*, which gives it a concrete existence in three-dimensional space. This embedding describes the path through space that each edge takes to connect its two nodes.

There are a vast number of ways to compute skeletons. Many of these come from graph theory and computational geometry. In this chapter, we present a sampling of skeleton-generating algorithms, all of which have been implemented in one or more software prototypes.[1] The particular algorithms were selected through a process of sequential discovery: at first, the simplest were implemented and used to generate a large number of skeletons. Typically, there would be obvious flaws, so new algorithms were sought or devised to correct these flaws. The best algorithms that are currently implemented in the software are a result of several iterations of this process.

When synthesizing a skeleton, we must consider the list of requirements that the

[1] See Appendix A for the user manual for the `graphlab` program.

part must meet. In Section 1.2 we put forth the following list of categories:

1. Connectivity.
2. Noninterference.
3. Strength and Stiffness.
4. Manufacturability and Assemblability.
5. Dynamics.
6. Electromagnetic characteristics, aerodynamics, hydrodynamics, aesthetic styling, thermal expansion, etc.

Accordingly, in our search for algorithms, we begin by ensuring that the first requirement, connectivity, is always satisfied. This is not particularly difficult—any graph that spans the input set guarantees a connected part. The second requirement, however, is a large and challenging issue unto itself. It will be saved for Chapter 4. In the present chapter we concentrate on formulating parts in uncluttered space.

The third requirement, Strength and Stiffness, is one of the gray-area requirements. Its importance and satisfaction depend on the application. However, we believe that even in the absence of any knowledge of the application, some parts are clearly superior to others. For example, two designs for the same part may use the same amount of material, but one is much stiffer. For this research, we assume that stiffer design is always better.[2] While it is true that the computation of part stiffness requires application knowledge, it is possible to qualitatively assess that some designs will almost always be stiffer than others, even in the absence of application information. We will seek to generate skeletons that have high likelihood of being close to the stiffest possible.

Because there is only so much that can be done in the scope of this dissertation, we will neglect the remaining items in the list. Thus, in this chapter and the next, we seek part designs that are connected and stiff.

## 2.1 A Note About The Prototype Software

Much of the research effort for this dissertation has been spent on the implementation of two prototype computer programs. The first, called `racerx`, is a testbed

[2]Obviously some parts are *required* to be compliant. For now, such design is outside the scope of our research.

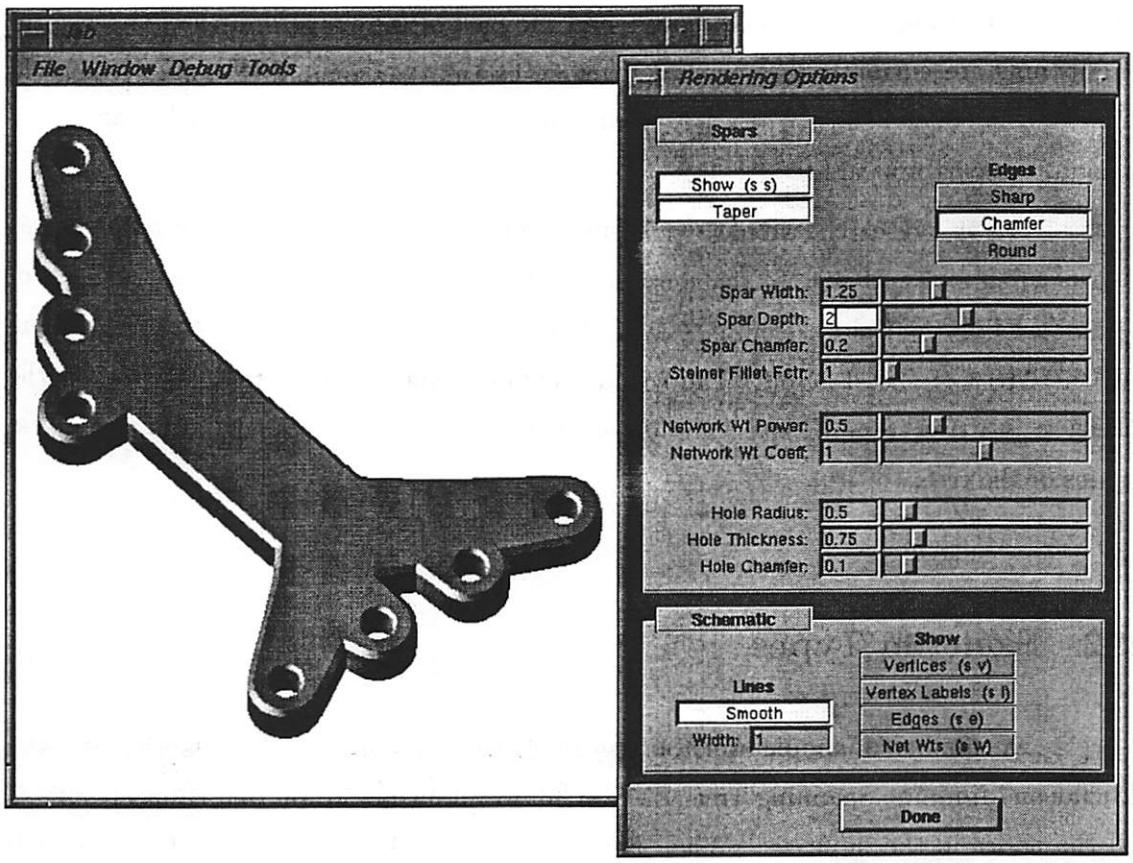



Figure 2.1: A part being designed in the `graphlab` program.

for obstacle avoidance using path planning. The second, called `graphlab`, is a testbed for skeleton-generating algorithms. Both are interactive programs designed to simulate some of the features of the system proposed in Chapter 1. A screen shot of the `graphlab` program is shown in Figure 2.1.

Neither of these programs attempts to support assembly-centric design using feature groups. Their purpose is to provide an environment for studying the shape synthesis problem in two dimensions. Furthermore, they only provide for the design of a single part. As such, the user inputs application features and illegal regions directly, and only one feature type is supported: the hole.

The input features and regions, as well as the synthesized part, are all rendered in three dimensions. A variety of parameters can be modified interactively, including hole diameters, minimum material thickness, and so-on. The hole features and illegal

regions can be dragged with the mouse, and the part is automatically resynthesized. The algorithms are currently fast enough to resynthesize a reasonably complex part several times per second. Thus, when the user interactively drags a feature or edits an illegal region, the part appears to stretch.

In this research, we have given only limited treatment to the material synthesis aspect of shape synthesis. In the software, two kinds of material synthesis are employed, both based on the thick skeleton. The simplest constructs spars of a uniform width along all the edges of the skeleton. The second technique embellishes this slightly when the skeleton is a tree: tapered spars are are constructed, the narrowest being at the terminal nodes of the tree.

## 2.2 Skeleton Types

A number of simple skeletons can be generated using familiar algorithms. The Euclidean Minimum Spanning Tree (EMST), for example, is a graph that connects (spans) all the nodes in the input set. When used as a skeleton, the result is a connected part. However, such skeletons are unlikely to be stiff except in special circumstances. We have found that some of best skeletons are generated by a two-step process: first choose a topology, then optimize the embedding, possibly altering the topology in the process.

In this chapter, we present algorithms that generate a topology. Some of these may be suitable for skeletons as is. Most, however, are mainly useful as starting points for the optimizer, which is covered in Chapter 3. Several topology-generating algorithms are examined, including the Euclidean Minimum Spanning Tree, Centroid Star Tree, the Principal Axis Tree, the Hierarchical Nearest-Neighbor Tree, two kinds of $t$-Spanners, and the Delaunay Triangulation.

Much of the work presented here focuses on trees. Many of the algorithms examined need to be fully understood in the context of trees before they are generalized to graphs. In addition, for many mechanical parts, a tree-type skeleton is appropriate. Note that for such parts, the material synthesizer has the discretion to construct the spars with voids in them. This may include bars with drilled holes, or each spar may be a truss, for example. We will still consider the skeleton of the part to be tree-based.

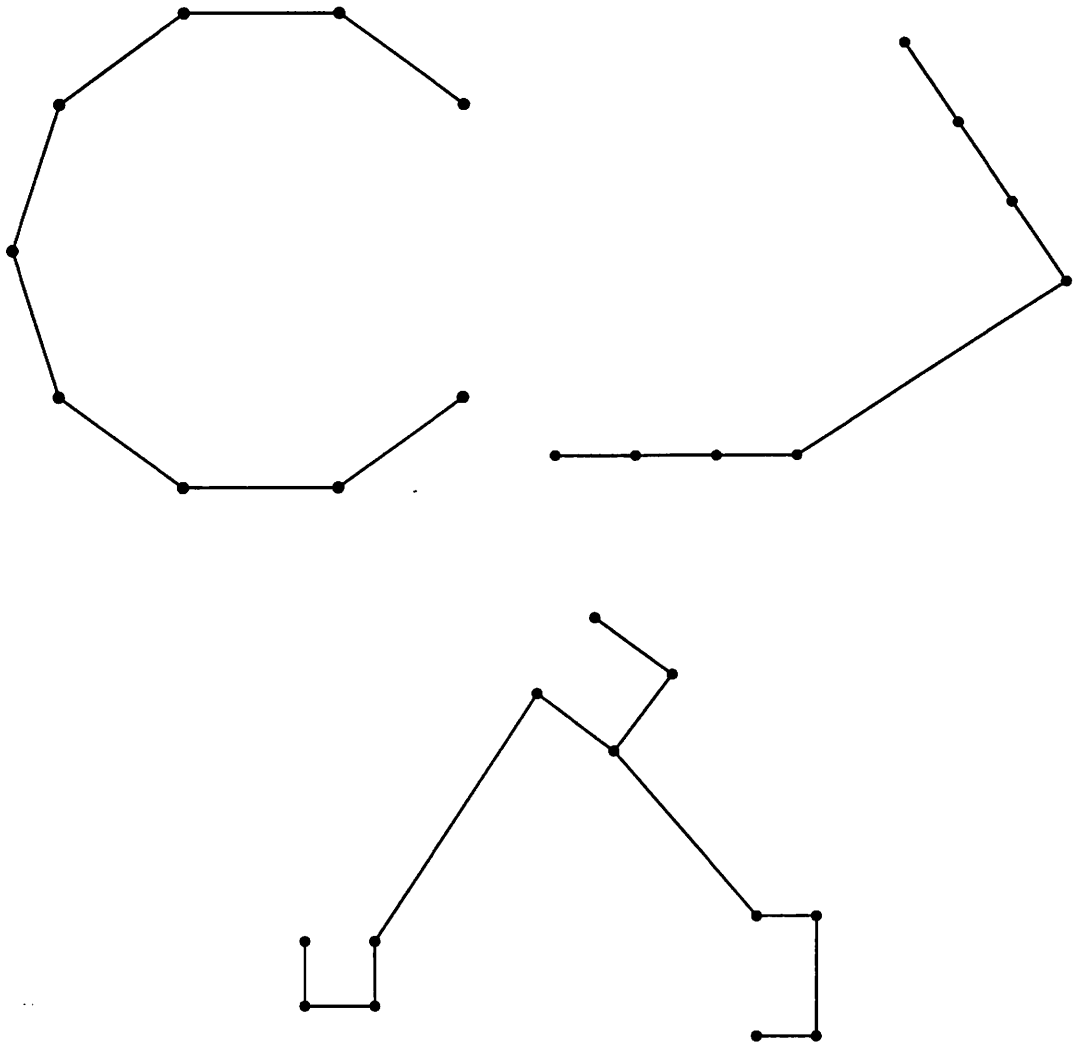


Figure 2.2: Euclidean Minimum Spanning Trees for three example point sets.

### 2.2.1 Euclidean Minimum Spanning Tree

The first spanning tree we implemented is the Euclidean Minimum Spanning Tree. For a set of points, $V$, a minimum spanning tree is defined to be a spanning tree whose weight (sum of weights of all edges in the tree) is less than or equal to the weights of all other possible spanning trees over the same set $V$. If we define the weight of an edge to be its Euclidean length, then the tree is said to be a *Euclidean Minimum Spanning Tree* of $V$. This requires that $V$ be embedded in $\mathbf{R}^d$, i.e., every vertex $v \in V$ has $d$ coordinates that define its position in $d$-space. The `graphlab` software employs a simple implementation

Figure 2.3: Parts generated from the skeletons in Figure 2.2.

of Kruskal's algorithm [?] to generate the Euclidean Minimum Spanning Tree of the set of nodes input by the user. Figure 2.2 shows the EMST's for three input sets. Figure 2.3 shows parts generated from those skeletons. These same three point sets will be used for all examples in this chapter and the next.

It can be seen that the minimum spanning tree skeleton may not be the most desirable way to design a part. This is because all emphasis has been placed on conserving material, and no emphasis has been placed on designing a part that is stiff. Further, the algorithm only considers edges whose endpoints are in $V$. For the purposes of synthesizing part skeletons, it is perfectly acceptable for generated edges to have junctions that are not members of $V$. Thus we should consider algorithms that add nodes not in $V$ when it

serves to improve the skeleton.

### 2.2.2 Principal Axis Tree

We have implemented several algorithms that may add nodes to the graph. The first such we call a *principal axis tree.* It is a simple spanning tree whose embedding is based on the inertia tensor of the node set. It has the characteristic that all its edges are parallel or perpendicular to each other. Nodes are added by the algorithm to make this possible.

For the remainder of this chapter, we will use the term *Steiner node* to distinguish a node not in $V$ that is added to $G$ by the algorithm. When necessary, we will refer to non-Steiner nodes as *V-nodes* to emphasize that they belong to the original set $V$.

Conceptually, the inertia tensor of the node set is computed, and a diagonalizing transform is found. This transform moves the set $V$ so that its centroid is at the origin, and rotates the set so that its principal axis of inertia lies on the $x$-axis. Each node in $V$ is projected to the transformed $x$-axis and a Steiner node is added at that location. An edge is constructed to connect each $V$-node to its Steiner node, and edges are added along the $x$ axis to connect all the Steiner nodes together. Figure 2.4 shows the principal-axis trees for the example point sets. Figure 2.5 shows parts generated from those skeletons.

To find the diagonalizing transform in $\mathbf{R}^2$, we need only compute the centroid of $V$ and the angle of the rotation. The centroid is computed as

$$\mathbf{v}_{\text{cen}} = \frac{1}{|V|} \sum_{v \in V} \mathbf{v} \tag{2.1}$$

where $\mathbf{v}$ is the vector $[x_v, y_v]^T$, the coordinates of the point $v$ in $\mathbf{R}^2$. The angle is computed as

$$\theta = -\frac{\text{atan2}\left( \sum_{v \in V} \overline{x}_v \, \overline{y}_v \, , \, \sum_{v \in V} \overline{x}_v^2 - \sum_{v \in V} \overline{y}_v^2 \right)}{2} \tag{2.2}$$

where $\overline{x}_v$ and $\overline{y}_v$ are the coordinates of each point, $v$, measured from the centroid of the set, $v_{\text{cen}}$.

This type of skeleton could be a starting point in the synthesis of parts with mostly rectilinear geometry. However, this simplistic algorithm optimizes neither material use nor stiffness, and would probably not be useful without some enhancements. We

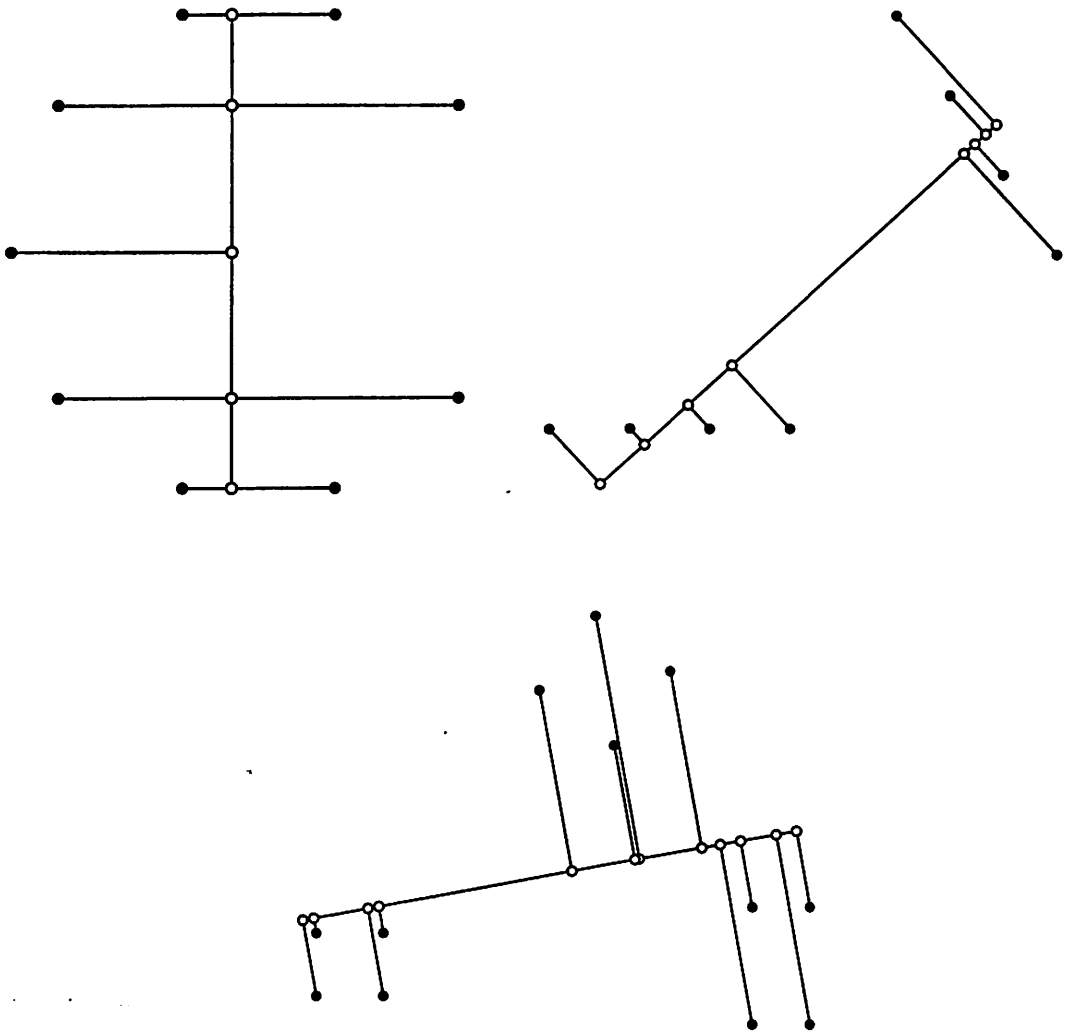


Figure 2.4: Principal Axis Trees for the example point sets. The white nodes are Steiner nodes.

imagine that optimization schemes could be devised that would improve upon the output of this algorithm. For example, a material optimizer could identify peripheral spars that are near each other and coalesce them.

We assume that the motivation for using a skeleton of this type is that rectilinear geometry is desired. Optimization in this context is markedly different from that which will be presented in the Chapter 3. When using this tree as an initial graph, that optimizer often produces very bad geometry, with many undesirable zigzags in the skeleton. Preserving rectilinearity represents an additional constraint that the optimizer must observe. We feel that the optimization process must first be understood without such additional

Figure 2.5: Parts generated from the skeletons in Figure 2.4.

constraints, so no additional study of the principal axis approach has been conducted.

### 2.2.3 Centroid Star Tree

The first two tree skeletons, above, are both suboptimal in terms of material use and stiffness. This leads us to investigate numerical optimization approaches for generating skeletons, which will be presented in Chapter 3. These approaches require an initial graph, which will then be modified to improve its stiffness, material usage, or both.

As will be seen in that section, the optimizer often drastically modifies the initial graph. Therefore, we should not limit our consideration of initial-graph algorithms to just

those we expect to produce good skeletons. It will be seen that even very crude initial graphs can be transformed into a reasonable skeleton by the optimizer. We may wish to use as an initial graph something that is cheap to compute. A feature of the optimizer presented in Chapter 3 is that it will not transform an unconnected graph into a connected one. Thus, when we choose an initial graph, we must at least guarantee that it is connected and that it spans $V$.

One very simple spanning graph is the *star* graph. Such a graph can be easily constructed by choosing (or adding) a node and constructing $n-1$ (or $n$) edges to connect that node to all the other nodes in $V$. We will find it expedient to insert a Steiner node, $v_{\text{cen}}$, at the centroid of the set $V$, as computed in Equation 2.1. We will call this a *Centroid Star Tree.* Figure 2.6 shows the Centroid Star Trees for the example point sets.

In many cases the optimizer transforms the Centroid Star Tree into a reasonable skeleton. However, its worst-case performance is poor, and it requires more effort than necessary on the part of the optimizer. This leads us to devise the Hierarchical Nearest Neighbor Trees.

### 2.2.4 Hierarchical Nearest Neighbor Trees

The optimizer in Chapter 3 uses graph weight and compliance as cost criteria. Our experiences using the optimizer with the above algorithms have lead us to an insight: the best topologies have edges between nodes that are near each other. Of course, this should have been obvious in the first place. It leads us to devise what we call the *Hierarchical Nearest Neighbor Trees.*

In these graphs, each node in $V$ is paired with one nearby. Each pair-group then gets a Steiner node added midway between the pair. Two edges are constructed from the Steiner node to connect it to its two $V$-nodes. Call the set of Steiner nodes $S_1$, and repeat the process on $S_1$, pairing nearby Steiner nodes. This yields a second set of Steiner nodes, $S_2$. It is easy to see that each successive set of Steiner nodes is half the size of its predecessor (ignoring, for now, that some of these will have odd numbers of nodes). The process is repeated until only one pair of Steiner nodes remains. An edge is constructed to connect these two, yielding a balanced binary tree that spans $V$. Figure 2.7 shows the Hierarchical Nearest Neighbor Trees for the example point sets.

The formal algorithm is shown in Algorithm 2.1. Note that just before line 14

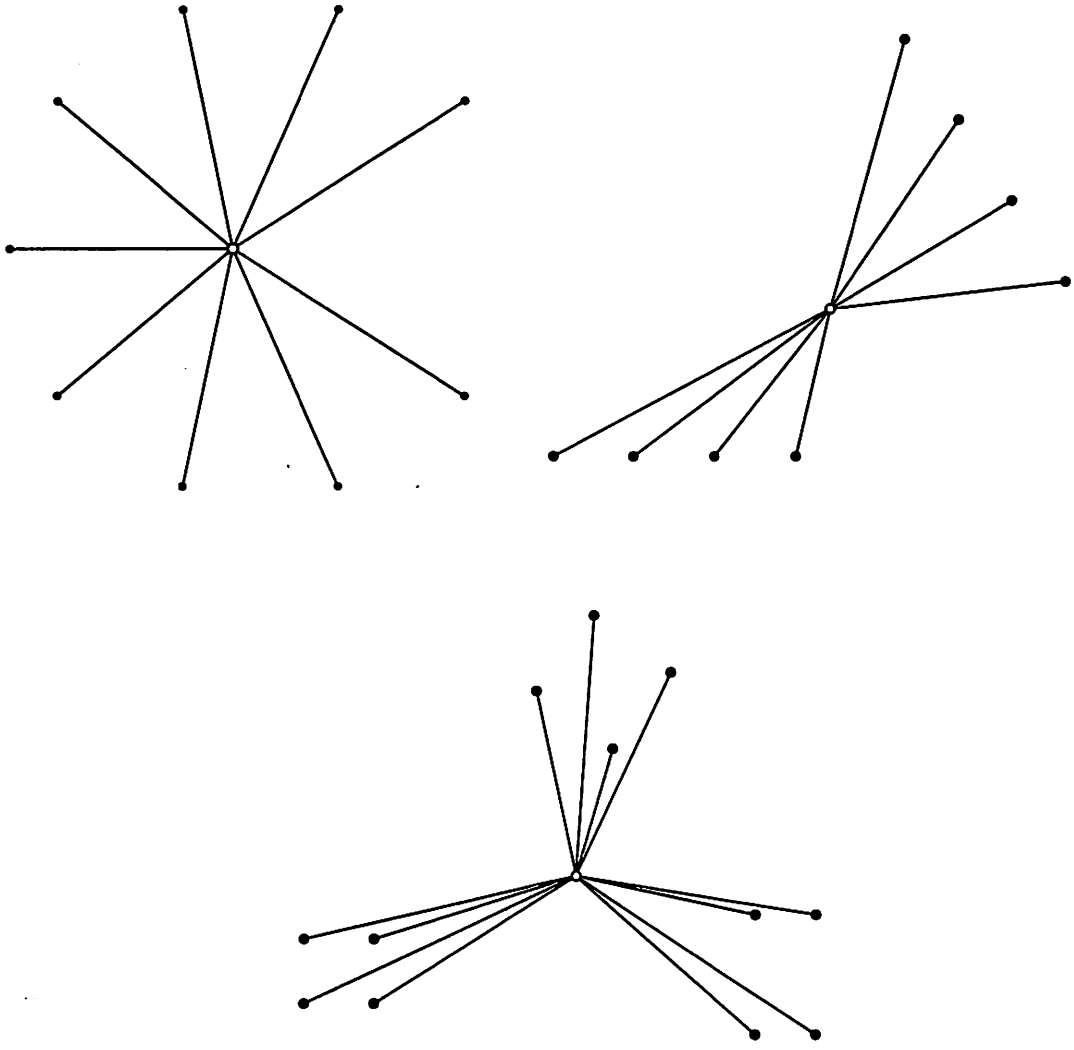


Figure 2.6: Centroid Star Trees for the example point sets.

is executed, the set $S_i$ is either empty, or contains only one node. In the latter case, $S_i$ began the inner loop with an odd number of nodes. In line 18, we are guaranteed that $|S_{i-1}| = 2$. This can be seen from the fact that

$$|S_{i+1}| = \left\lceil \frac{|S_i|}{2} \right\rceil, \tag{2.3}$$

Hence, after line 14 is executed, $|S_i|$ will never equal 1, and the test in line 5 will always terminate the loop with $|S_i| = 2$.

The complexity analysis is as follows. From computational geometry, we know

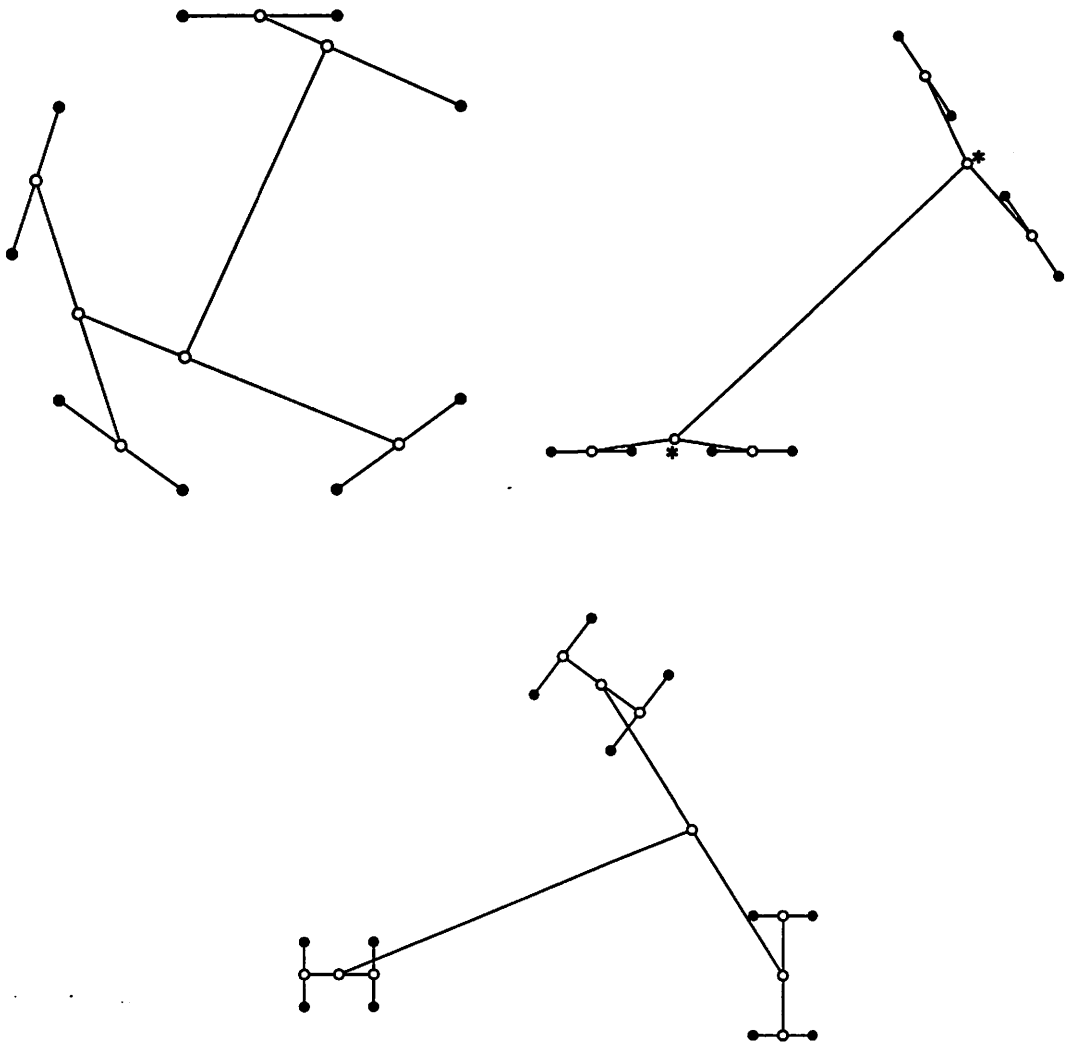


Figure 2.7: Hierarchical Nearest Neighbor Trees for the example point sets. The Steiner Nodes marked with a "*" have been moved slightly for clarity. The actual position of each is at the asterisk.

that the best time complexity of line 7 is

$$T_7 = O(\log n_i), \tag{2.4}$$

where $n_i = |S_i|$. To support this requires that line 11 also have time complexity

$$T_{11} = O(\log n_i), \tag{2.5}$$

because we must remove two items from a heap. Lines 8, 9, 10, and 12 each execute in constant time. Thus lines 7 and 11 dominate the inner loop.

**Algorithm 2.1** HNN1($V$)

*Takes a set $V$ and computes a Hierarchical Nearest Neighbor Tree, $G$.*

1: $G \Leftarrow (V, \{\})$.
2: $S_0 \Leftarrow V$.
3: $S_1 \Leftarrow \{\}$.
4: $i \Leftarrow 0$.
5: **while** $|S_i| > 2$ **do**
6: **while** $|S_i| > 1$ **do**
7: Find the closest pair $p = \{u, v\}$, $u, v \in S_i$.
8: Create Steiner node, $s$, midway between $u$ and $v$.
9: Add $s$ to $G$.
10: Add edges $e_1 = [s, u]$ and $e_2 = [s, v]$ to $G$.
11: $S_i \Leftarrow S_i \setminus \{u, v\}$.
12: $S_{i+1} \Leftarrow S_{i+1} \cup \{s\}$.
13: **end while**
14: $S_i \Leftarrow S_i \cup S_{i+1}$.
15: $S_{i+1} \Leftarrow \{\}$.
16: $i \Leftarrow i + 1$.
17: **end while**
18: $S_{i-1}$ contains two vertices, call them $u$ and $v$.
19: Add edge $e = [u, v]$ to $G$
20: **return** $G$.

Because line 11 removes two elements from $S_i$, the inner loop iterates $\lceil \frac{n_i}{2} \rceil$ times for one iteration of the outer loop. Thus the complexity of the inner loop for one iteration of the outer loop is $O(n_i \log n_i)$.

Line 14 requires $O(n_i \log n_i)$ time because a data structure must be built to support the query in line 7. Lines 15 and 16 each take constant time, so one iteration of the outer loop requires $O(n_i \log n_i)$ time.

On the first iteration of the outer loop, $n_i = |V|$. On each subsequent iteration $n_i$ is halved, so the total time complexity is

$$T_{\text{HNN1}} = \sum_{i=0 \to r} \frac{n}{2^i} \log\left(\frac{n}{2^i}\right), \tag{2.6}$$

**Algorithm 2.2** HNN2($V$)

*Takes a set $V$ and computes a Hierarchical Nearest Neighbor Tree, $G$.*

1: $G \Leftarrow (V, \{\})$.
2: $S_0 \Leftarrow V$.
3: $i \Leftarrow 0$.
4: **while** $|S_0| > 2$ **do**
5: Find the closest pair $p = \{u, v\}, u, v \in S_0$.
6: Create Steiner node, $s$, midway between $u$ and $v$.
7: Add $s$ to $G$.
8: Add edges $e_1 = [s, u]$ and $e_2 = [s, v]$ to $G$.
9: $S_0 \Leftarrow S_0 \setminus \{u, v\}$.
10: $S_0 \Leftarrow S_0 \cup \{s\}$.
11: **end while**
12: $S_{i-1}$ contains two vertices, call them $u$ and $v$.
13: Add edge $e = [u, v]$ to $G$
14: **return** $G$.

where $n = |V|$ and $r = \log n$. This can be rearranged as

$$T_{\text{HNN1}} = n \log n \sum_{i=0 \to r} \frac{1}{2^i} - \sum_{i=0 \to r} \frac{i}{2^i}, \tag{2.7}$$

so we conclude that

$$T_{\text{HNN1}} = O(n \log n). \tag{2.8}$$

This is certainly good news. The algorithm is as efficient as we could hope for, and reasonably simple.[3] The only hidden complexity in this algorithm is that it requires an efficient computation of the Delaunay Triangulation to support the query in line 7. We will see below that we need the Delaunay Triangulation to support other algorithms, as well as for its own sake as a skeleton algorithm.

A variation of this algorithm is possible, which we will call *HNN2*. It is shown in Algorithm 2.2. The difference is that the Steiner nodes are added to the working set $S_0$ when they are created, rather than being saved in a separate set until $S_0$ is exhausted, as HNN1 does. The complexity analysis is similar, and gives $T_{\text{HNN2}} = O(n \log n)$

[3]The current implementation is actually simpler, but less efficient. Its complexity is $O(n^2)$

We will see in Chapter 3 that the Hierarchical Nearest Neighbor Trees gives reasonably consistent results when used as initial graphs for the optimizer. However, there does not seem to be a clear choice between HNN1 and HNN2.

### 2.2.5 $t$-Spanners

In addition to the tree-type skeletons above, we have also investigated several algorithms that yield general graphs. These include the $t$-Spanners and the Delaunay Triangulation. In some cases we will find that these graphs provide a stiffer skeleton than do the trees presented above.

These graphs can be used as initial graphs for the optimizer. However, we will see that the optimizer we have implemented is really trying to drive the graph to be a tree. If it terminates before doing so, it is only because it has gotten stuck in a local minimum. If we desire to optimize general graphs, we will probably need to devise a more specific optimizer.

For designing mechanical parts, a promising class of graphs is the *t-Spanners*. Given a set of points $V$ in $d$ dimensional space, a graph $G = (V, E)$ is said to be a *Euclidean t-Spanner* if for every $u, v \in V$ the distance from $u$ to $v$ in $G$ is at most $t$ times longer than the Euclidean distance between those points.

Numerous constructions for such graphs are given in the literature. In our software we have implemented two of them (specialized to $\mathbf{R}^2$). The first is based on a modification of Kruskal's algorithm for finding the Euclidean minimum spanning tree. It generates a graph with $O\left(n^{1+\frac{2}{t-1}}\right)$ edges and weight that is $O(\log n)$ times the weight of the minimum spanning tree; where $n = |V|$. The algorithm takes $O(n^3 \log n)$ time (though in our current implementation it takes $O(n^4)$ time.[4]) The second is based on an algorithm for the all-nearest-neighbors problem. It generates a graph with $O\left(\frac{n}{(t-1)^{2d}}\right)$ edges in $O\left(n \log n \frac{n}{(t-1)^{2d}}\right)$ time.

The formal definition of a spanner is as follows. Let $V$ be a set of points in $\mathbf{R}^d$ and $G = (V, E)$ is a connected graph that spans $V$ and has Euclidean edge weights. $G$ is

[4]It is not yet clear whether high-order complexities such as these would pose a problem in practice. We have observed that many real-world parts have only a small number of application features (from which the nodes are derived). Alternatively, we could modify the algorithm to compute a graph-spanner of the Delaunay Triangulation.

said to be a *Euclidean t-Spanner* of $V$ if

$$\max_{u,v \in V} \frac{d_G(u,v)}{d(u,v)} \leq t, \tag{2.9}$$

where $d_G(u,v)$ is the length of a shortest path from $u$ to $v$ in $G$ and $d(u,v)$ is the Euclidean distance from $u$ to $v$ in $\mathbf{R}^d$. The value of $t$ is called the *stretch factor* of $G$. Note that the Euclidean $t$-spanner is a special case of a *graph spanner*. Peleg and Shaffer [?] define a $t$-spanner for a graph $G' = (V, E')$ to be a subgraph $G = (V, E)$ such that

$$\max_{u,v \in V} \frac{d_G(u,v)}{d_{G'}(u,v)} \leq t. \tag{2.10}$$

A Euclidean spanner can be thought of as a graph spanner of the complete Euclidean graph of $V$, $K_V$. The two algorithms that we have implemented in the software differ from each other in that the first generates spanners of arbitrary graphs, while the second applies only to Euclidean spanners. Both algorithms can generate graphs in arbitrary dimensions, but we have implemented them only for $\mathbf{R}^2$. Furthermore, we have applied the first algorithm only to generating Euclidean spanners, that is, the algorithm is passed $K_V$ and asked to compute a spanner of that.

Spanners have been studied extensively in the literature. Most authors present algorithms for spanner construction and then derive bounds on various resulting measures such as the size of the graph, its weight, maximum vertex degree, and cost of construction. Spanners for arbitrary positive-edge-weighted graphs were considered by Althöfer et al. [?, ?], and it was shown that a Greedy algorithm (one of the two we implemented) constructs $t$-spanners with size $O\left(n^{1+\frac{2}{t-1}}\right)$, weight less than $\left(1+\frac{n}{t-1}\right) \cdot \texttt{weight}(\text{MST})$ and runs in time $O(n^3 \log n)$; $n = |V|$. Chandra et al. [?] improve the weight bound to $O(\log n) \cdot \texttt{weight}(\text{MST})$ for the same algorithm using an improved analysis. The constant implicit in the big $O$ depends on $d$ and $t$. They go on to present an algorithm which constructs in $O(n \log n)$ time a $t$-spanner with $O(n)$ edges and weight $O(\log n) \cdot \texttt{weight}(\text{MST})$.

Other authors present algorithms with similar bounds on size, weight, and running time. Some consider only a specific value of $t$ [?], or a limited range of values [?, ?, ?], while others consider arbitrary values of $t$ greater than some minimum, usually 1 [?, ?, ?, ?, ?]. Salowe [?] presents an algorithm (the second we implemented) which generates a graph with $O\left(\frac{n}{(t-1)^{2d}}\right)$ edges in $O\left(n \log n \frac{n}{(t-1)^{2d}}\right)$ time. Another approach has been to consider specific classes of graphs, show that they are spanners, and determine (or bound) their stretch factor [?, ?]. Several authors have done this for the Delaunay triangulation,

**Algorithm 2.3** GREEDY$(G' = (V, E'), t)$

*Takes a graph $G'$ and computes a $t$-spanner, $G$.*

1: Sort $E'$ by non-decreasing weight.
2: $G \Leftarrow (V, \{\})$.
3: **for each** edge $e = [u, v]$ in $E'$ **do**
4: Compute $P(u, v)$, the shortest path from $u$ to $v$ in $G$.
5: **if** $t \cdot \texttt{weight}(e) < \texttt{weight}(P(u, v))$ **then**
6: Add $e$ to $G$.
7: **end if**
8: **end for**
9: **return** $G$.

with the best stretch factor of $\approx 2.42$ reported in [?]. Some of these authors are additionally interested in finding spanners with bounded vertex degree, with bounds as low as 4 (reported in [?]) and 3 (reported in [?]).

### 2.2.6 The Greedy $t$-Spanner

The first $t$-spanner algorithm we implemented is the Greedy algorithm of Althöfer et al. [?]. It is a modification of Kruskal's algorithm for finding the minimum spanning tree. The second is the algorithm of Salowe [?] and is discussed in Section 2.2.7. Both are incorporated into the `graphlab` software.

The Greedy algorithm is elegant in its simplicity, if not in its efficiency. It is shown in Algorithm 2.3. Implementing this algorithm is straightforward since we already have an implementation of Kruskal's algorithm. Support for it requires an implementation of Dijkstra's algorithm for shortest path (line 5), which we will also use in Chapter 4 to avoid obstacles. Figure 2.8 shows Greedy $t$-Spanners for the example point sets. The stretch factor, $t$, is 1.2 in the left column, 1.5 in the right. Figure 2.9 shows an example of parts generated from the skeletons in the right column of Figure 2.8.

We tested the algorithm on a large number of interactively constructed point sets with $n$ as large as 50. The sizes and weights of the resulting graphs appear to follow the bounds presented in [?] and [?]. The required CPU time is more like $O(n^4)$ due to our crude implementation of Dijkstra's algorithm which requires $O(n^2)$ time to compute a shortest path. Precise benchmark tests were not conducted.

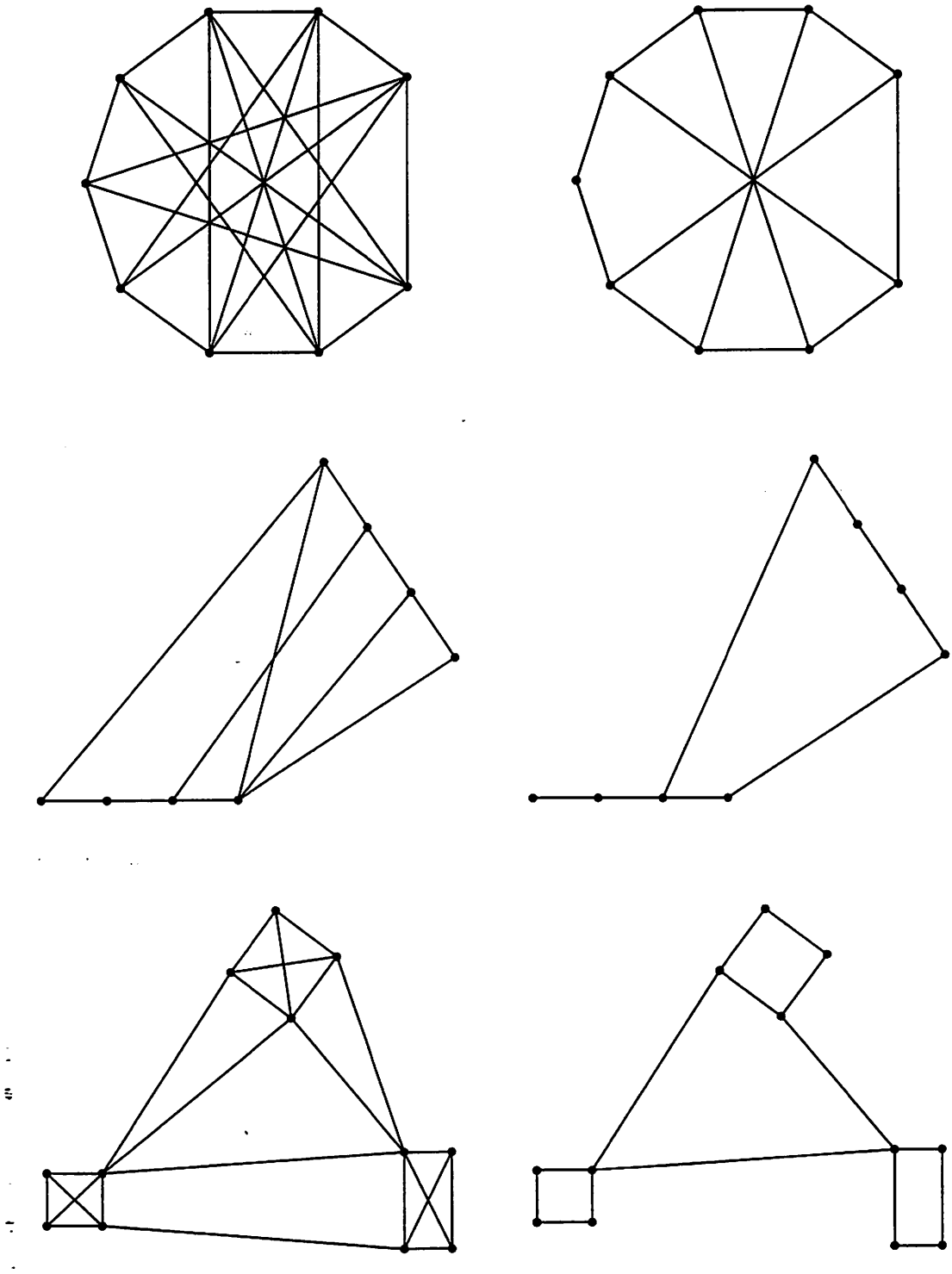


Figure 2.8: Greedy $t$-Spanners for the example point sets. In the left column $t = 1.2$, in the right column $t = 1.5$.

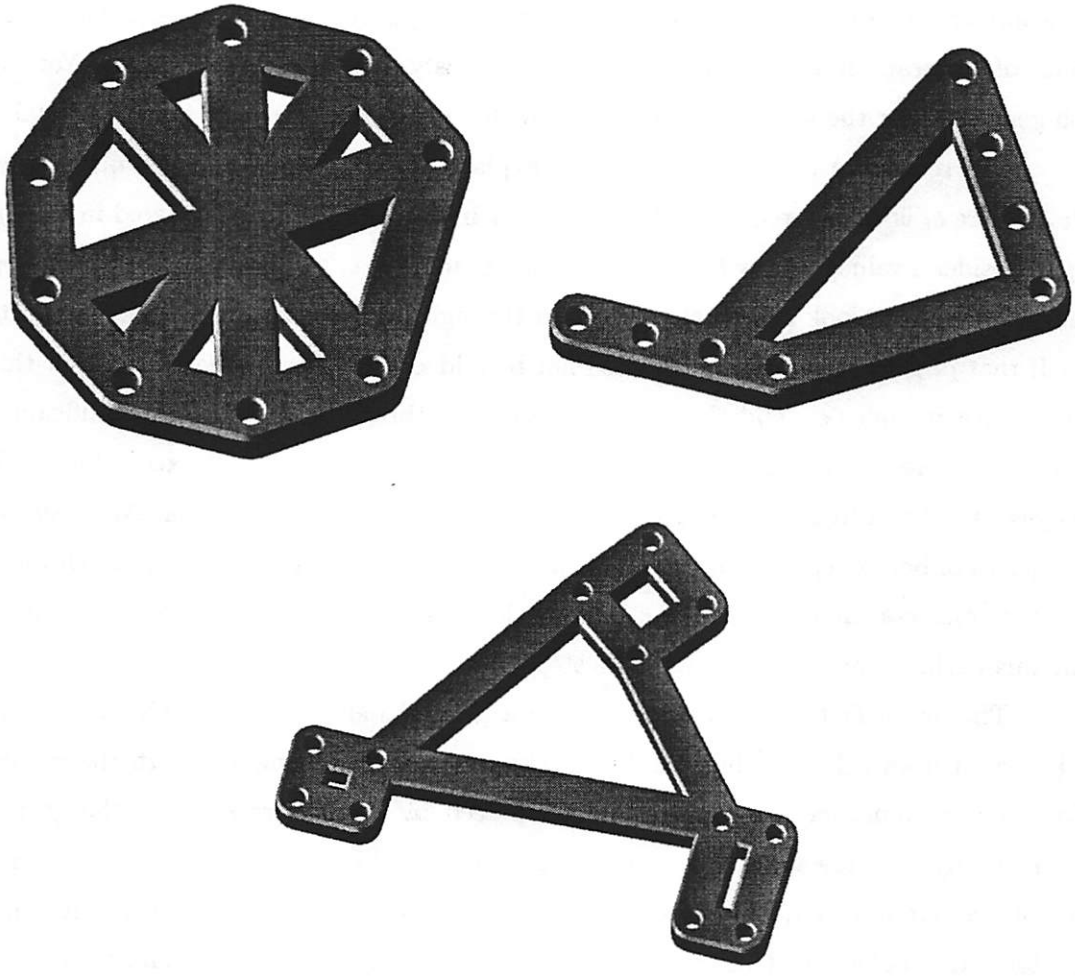


Figure 2.9: Parts generated from the skeletons in the right column of Figure 2.8.

### Discussion

We have observed some interesting characteristics of the generated spanners. In general, $G$ is non-planar, and it is possible to contrive point sets that will require non-planar spanners for arbitrarily large values of $t$. Random point sets, however, almost always seem to have planar spanners for $t$ as low as 1.5. It is also interesting to note (and easy to see by inspection) that this algorithm produces the complete graph at $t = 1$ and the minimum spanning tree at $t = \infty$. Note for both the example point sets, a stretch factor of 5.0 is sufficient to admit the EMST result.

The algorithm exhibits a seemingly curious behavior when the value of $t$ is varied interactively. Sometimes, as $t$ increases above a particular threshold value, a single edge

will disappear, to be replaced somewhere else in the graph by a single, longer, edge. Both versions of the graph have the same size and both satisfy the larger stretch factor. Yet the graph generated for the larger value of $t$ has greater weight. Clearly this is not optimal.

The reason for this behavior is easily explained. Call the two edges in question $e_i$ and $e_j$, where $e_i$ is considered for addition to $G_{i-1}$ in step $i$ and $e_j$ is considered in step $j$, $i < j$. Consider a value of $t$, say $t_1$, just small enough to force $e_i$'s addition in step $i$. At step $j$, the algorithm will look for the shortest path through $G_{j-1}$ that connects the endpoints of $e_j$. If that path is short enough, $e_j$ need not be added. Suppose, though, that that this shortest path includes $e_i$, and that the shortest path through $G_{j-1}\backslash\{e_i\}$ is significantly longer. Now increase $t$ is slightly and re-run the algorithm. Assume that except for $e_i$, all the edges considered from $e_0$ through $e_{j-1}$ are not sensitive to this particular $\Delta t$. Because $t$ is larger than before, $e_i$ will not get added at step $i$. At step $j$ the shortest path through $G_{j-1}$ that connects the endpoints of $e_j$ is now the longer one. If $\Delta t$ was not sufficient to admit this path, $e_j$ must be added at this step.

This is, in fact, a specific example of a general behavior of this algorithm. As $t$ is decreased from a large value, we do not simply see edges being added to the graph. Multiple edges sometimes disappear, to be replaced by others elsewhere in the graph. Experimentally, we were able to construct graphs that would exhibit this behavior around values of $t$ as large as 4.4, but it became increasingly difficult to construct graphs that would force this behavior at larger values of $t$. For random graphs, this behavior ceased above values of $t$ greater than about 2.

It would be interesting to try to improve this algorithm so that the lower-weight solution would always be generated. Consider the following modification: At step $i$, edge $e_i$ is always added, but it is marked as either *required* or *not-required*. At a later step, $j$, when a shortest path is searched for, three possible cases exist:

1. A shortest path search is performed in which only required edges in $G_{j-1}$ are considered. If a short enough path connecting the endpoints of $e_j$ is found, the edge is added to the graph and marked as not-required.

2. If no such path is found, a second search is performed in which both required and not-required edges in $G_{j-1}$ are considered. If a short enough path is found this time, the edges along the path are re-marked as required. Then $e_j$ is added and marked as not-required.

3. Finally, if no sufficiently sort path is found in either search, $e_j$ is added and marked as required. After the algorithm is run, a post-processing step removes all edges from $G$ that are still marked as not-required.

This modification has an obvious fault: If $e_j$ is added to $G_{j-1}$ under case 2, some edges in $G_{j-1}$ will have their marks changed from not-required to required. Suppose that in some later step, $k$, it becomes necessary to change $e_j$'s mark from not-required to required. The edges whose marks were changed in step $j$ are no longer required because $e_j$ now is. The resulting graph would clearly have larger weight and size than necessary. A solution to this problem is to attach to $e_j$ a list of pointers to the edges in $G_{j-1}$ whose marks were changed in order that $e_j$ could be added as not-required. At step $k$, if $e_j$'s mark must be changed to required, the edges pointed to by $e_j$'s list get their marks set back to not-required.

So far this looks like a promising modification, but we must still make one addition. In step $k$, we can not simply change the marks of the edges on $e_j$'s list back to not-required, because they might also be required by edges considered at steps between $j$ and $k$. What we must do is use a *requirement-counter* rather than a marker. At step $j$ (if $e_j$ is added under case 2) we increment the requirement-counters of *all* the edges on the path connecting $e_j$'s endpoints, and we place pointers to all of those edges on $e_j$'s list. At step $k$, if $e_k$ is added under case 2, we increment $e_j$'s counter, and decrement the counters of all edges on $e_j$'s list. Furthermore, we must look at the pointer lists of these edges, *increment* the counters of all edges pointed to, and so on. In general, every edge in $G_i$ at some step $i$ forms the root of a subtree. Whenever an edge's requirement-counter is modified, all of the edges in its subtree must have their counters modified as well. If counters are incremented at level $\ell$ in the subtree, then at level $\ell + 1$ the counters are decremented and vice versa.

Clearly there are some details of this modified algorithm that would have to be worked out. It should be straightforward to show that the size and weight of the generated graphs are no larger than for graphs generated by the Greedy algorithm. The time complexity of the modified algorithm is obviously greater, except for point sets where the algorithm does not add any edges under case 2. However when edges are added under case 2, the cost of that step of the algorithm becomes proportional to size of the subtree of edges rooted at the new edge. Clearly this can not be greater than $n^2$, so the complexity

of the modified algorithm is $\Omega(n^6 \log n)$. We conjecture that the expected time cost might be smaller because we can't see any reason that the subtree of every edge added would necessarily include all previously added edges in $G$. It should be interesting to implement this algorithm and observe the kind of graphs it generates.

### 2.2.7 Salowe's $t$-Spanner

The second algorithm we have implemented is due to Salowe [?]. It is a modification of a solution to the all-nearest-neighbors problem [?], and is much more efficient than the Greedy Algorithm. It constructs Euclidean spanners in arbitrary dimensions, but for simplicity it is presented here for two dimensions. First define the following concepts:

- A *box*, $b$, is a square region of $\mathbf{R}^2$, with the following attributes:
  1. contents($b$) is a set of points in $\mathbf{R}^2$. The box $b$ maintains the property that it always has the smallest possible side length to contain contents($b$) while remaining square.
  2. diameter($b$) is the diagonal measure of $b$. Note that if $|\text{contents}(b)| = 1$ then diameter($b$) $= 0$.
  3. neighbors($b$) is a list of boxes such that the distance from $b$ to one of its neighbors is less than diameter($b$) AND neighbors($b$) is maintained to be a subset of the *active list* (see below).
  4. attractors($b$) is a list of boxes such that if $b$ is a neighbor of $b'$, $b'$ is an attractor of $b$ AND attractors($b$) is maintained to be a subset of the *active list.*
- A *box tree*, $T$, is a tree of boxes in which each box has from zero to four descendants. Each box $b$ in $\mathbf{R}^2$ is a subset of one of the quadrants of its parent, $\hat{b}$, and contents($b$) $= b \cap \text{contents}(\hat{b})$

The algorithm is shown in Algorithm 2.4. The subroutine SUBDIVIDE($b$) takes a box $b$ and subdivides its contents by passing orthogonal lines through its center to form four quadrants. Children of $b$ are created corresponding to each quadrant, and contents($b$) are doled out to the children accordingly. Nonempty children are then added to $T$, $b$ is removed from the *active list* and its children in $T$ are added to the *active list.* In the

**Algorithm 2.4** SALOWE$(V, m)$

*Takes a set $V$ of points in $\mathbf{R}^2$ and an integer $m$ greater or equal to 3, and returns $G$, a Euclidean $t$-Spanner of $V$, where $t = \frac{1}{2^{(m-2)}-1}$*

COMMON VARIABLES:

- *active list*: The list of boxes that are the current leaves of $T$ as $T$ is being built.
- *deleted list*: A list of pairs of box that are to be connected by some edges. It is built up by the SUBDIVIDE subroutine.

1: *active list* $\Leftarrow \{\}$.
2: *deleted list* $\Leftarrow \{\}$.
3: $G \Leftarrow (V, \{\})$
4: contents$(b_0) \Leftarrow V$.
5: $T \Leftarrow b_0$
6: **call** SUBDIVIDE$(b_0)$.
7: **while** there is a box on the *active list* with nonzero diameter **do**
8: $\quad \hat{b} \Leftarrow$ the largest box on the *active list.*
9: $\quad$ **call** SUBDIVIDE$(\hat{b})$.
10: **end while**
11: **for each** pair of boxes $(b_1, b_2)$ on the *deleted list* **do**
12: $\quad$ **call** CONNECT-SUBTREES$(b_1, b_2, m)$.
13: **end for**
14: **return** $G$.

process, neighbors() and attractors() must be updated for $b$ and its children. A by-product of this is that certain pairs of boxes get added to the *deleted list.*

The subroutine CONNECT-SUBTREES$(b_1, b_2, m)$ takes a pair of boxes, $b_1$ and $b_2$ and considers them as the roots of subtrees in $T$. Its job is to join these subtrees by adding edges to $G$. To do this it descends both subtrees as far as it can to a depth at most $m$ and connects pairwise all of the leaves it finds in one to all of the leaves in the other. Connecting a pair of boxes with an edge means choosing an arbitrary point from each box and constructing an edge between those two points.

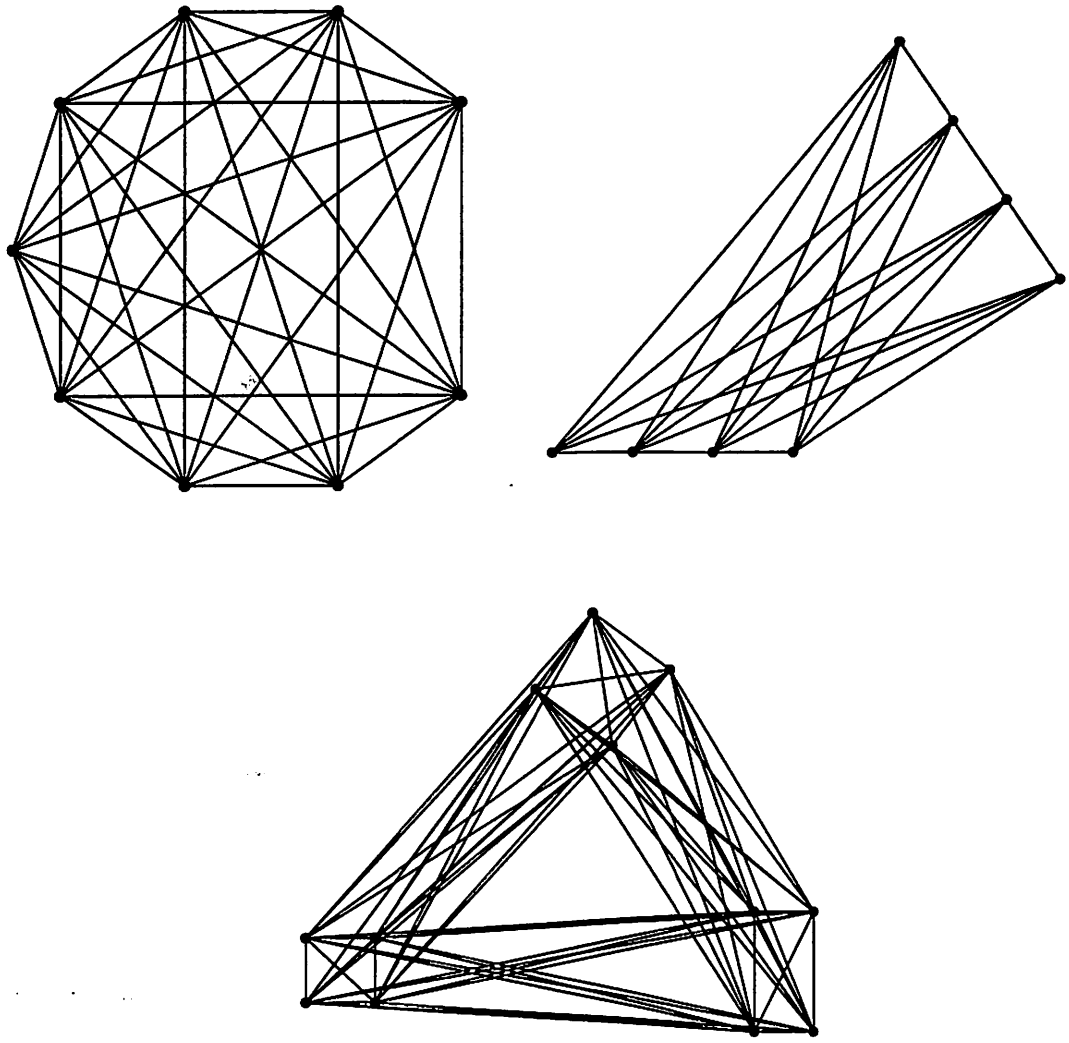


Figure 2.10: Salowe's $t$-Spanners for the example point sets.

The results of running Salowe's algorithm on the example point sets is shown in Figure 2.10. For small point sets like these, this algorithm constructs the complete graph, or nearly so.

### Discussion

This algorithm is considerably more difficult to implement than the Greedy algorithm, in spite of the fact that many of the data structures and support routines can be reused. One of the hardest problems we had was satisfying ourselves that the code to maintain neighbors and attractors was correct. For debugging, we draw the tree of boxes

on the graphics screen and show lines connecting each box to its neighbors. Using a debugger we can step through the code a line at a time and watch the relationships change. Using a ruler to measure distances on the screen we can be reasonably confident that the code is doing the right thing.

Another problem that plagued us for quite a while was that the paper is not explicit that when a box is taken off the active list its neighbors must be made to take it off their attractor lists. If this is not done there will be problems later on when one of these neighbors is subdivided. The SUBDIVIDE subroutine tries to do the proper maintenance on neighbor and attractor lists. Part of this involves visiting all of attractors of the box being subdivided and updating their neighbor lists.

In our implementation, we allow the parameter $m$ to take on any positive integer value, even though the size bound and stretch factor given in Salowe's paper only hold for $m \geq 3$. From the algorithm it is easy to see that the size of $G$ is $O\left(2^{4m}n\right)$, so that for $m = 3$ we do not expect $G$ to be any sparser than the complete graph until $n$ is in the range of 4096. In our implementation such large graphs turn the the entire screen to a single solid color. For shape generation, we are primarily interested in smaller problem sizes and very sparse solutions. When $m$ is allowed to go to zero the algorithm still generates some interesting graphs which appear (though we haven't proved this) to have spanner properties.

One observation we have made is that the computed graphs seem to have larger size and weight than they really need. This is partly due to the fact that the CONNECT-SUBTREES subroutine chooses arbitrary points to construct edges between. The weight of the graph could be reduced by simply choosing the closest pair of points from indicated boxes. Further, the size of the graph could be reduced by sorting the *deleted list* so that the lowest-level boxes are connected first. Constructing an edge between a box $b_1$ and a box $b_2$ has the side effect of also connecting pairs of all the boxes on paths from $b_1$ and $b_2$ to their least common ancestor. If these pairs are marked as already connected, it will obviate later connections as the rest of the *deleted list* is processed.

We have implemented the first of these improvements (choose the closest pair of points)—with disappointing results. At least subjectively, the generated graphs are no lighter than before. The real problem is that this algorithm generates graphs that are much too highly connected for our purposes.

It is difficult to compare these two algorithms as we have implemented them,

because the Greedy algorithm is only suitable for small problems and Salowe's algorithm is more appropriate for large problems. For our purposes, the Greedy algorithm produces graphs with better characteristics because we require sparse solutions for problems with small (e.g. $n$ less than 100) point sets. For these reasons, no further examination of Salowe's algorithm has been conducted.

### 2.2.8 Other Spanner Algorithms

In addition to the improvements mentioned above for the two implementations, we should continue to investigate other algorithms, particularly those that give small size bounds and fast execution times. The algorithm of Ruppert and Seidel [?] is one such algorithm. For any integer $k > 6$ the algorithm produces a graph with at most $kn$ edges and runs in $O(n \log n)$ time. It also appears to be reasonably easy to implement.

Also, we must keep in mind that we are primarily interested in producing interactive programs. As such it is not so important that the algorithms be fast when constructing a spanner from scratch, but they must be able to support fast *editing* operations. These operations include inserting, deleting, and, most importantly, moving the points in $V$. Also, the stretch factor parameter should be able to be varied interactively.

We estimate that a large amount of bookkeeping would be required to support fast point editing in the Greedy algorithm. In the frameworks of both Salowe's algorithm and Ruppert and Seidel's algorithm it appears that a data structure could be maintained that would support point editing in $\log n$ time. In the case of Salowe's algorithm this would be the box tree, and in Ruppert and Seidel's it would be an augmentation of the graph itself.

### 2.2.9 Delaunay Triangulation

For a very stiff part, a highly connected skeleton may be used. The most highly connected planar graphs are the triangulations. One triangulation that we think holds promise for the generation of part designs is the Delaunay Triangulation [?]. Figure 2.11 shows the Delaunay Triangulations for the example point sets. The parts generated from those skeletons are shown in Figure 2.12.

Given $n$ two-dimensional (2-D) points, the *Delaunay Triangulation* problem is to connect them into non-overlapping triangles which fill the convex hull of the points

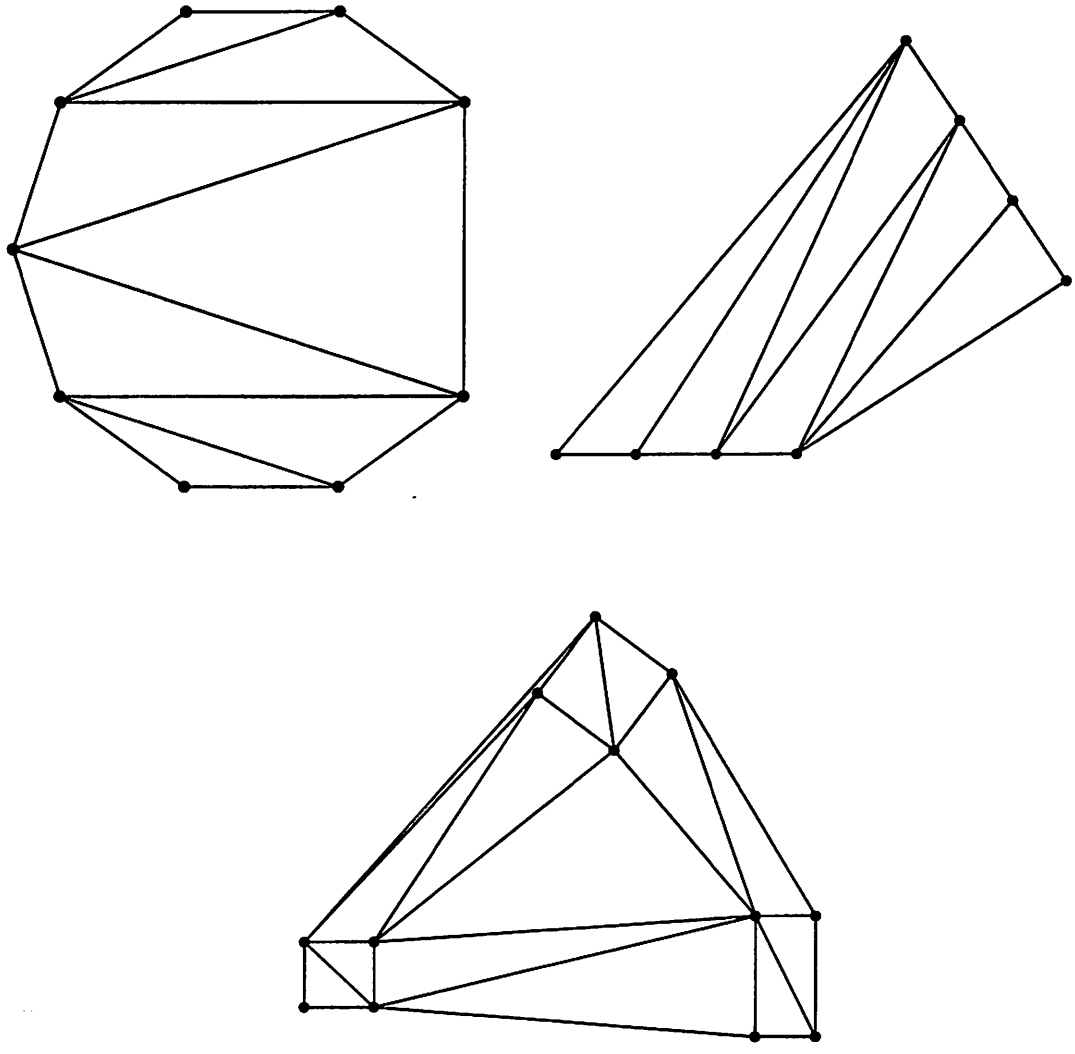


Figure 2.11: Delaunay Triangulations for the example point sets.

such that the circle criterion is satisfied, i.e., the circumcircle of the three vertices of any triangle of the triangulation contains none of the given $n$ points in its interior. A Delaunay Triangulation is unique if no four of the $n$ given points are co-circular [?]. It is canonical with respect to rigid body transformations, i.e., we can apply any rigid body transformation to the set of points and we will still get the "same" Delaunay Triangulation, appropriately transformed. The same is true of scaling transformations. The circle criterion discourages the triangulation from having long skinny triangles, wherever there is a choice. This is what makes the Delaunay Triangulation likely to generate stiff skeletons.

There are a variety of efficient algorithms for computing the Delaunay Triangu-

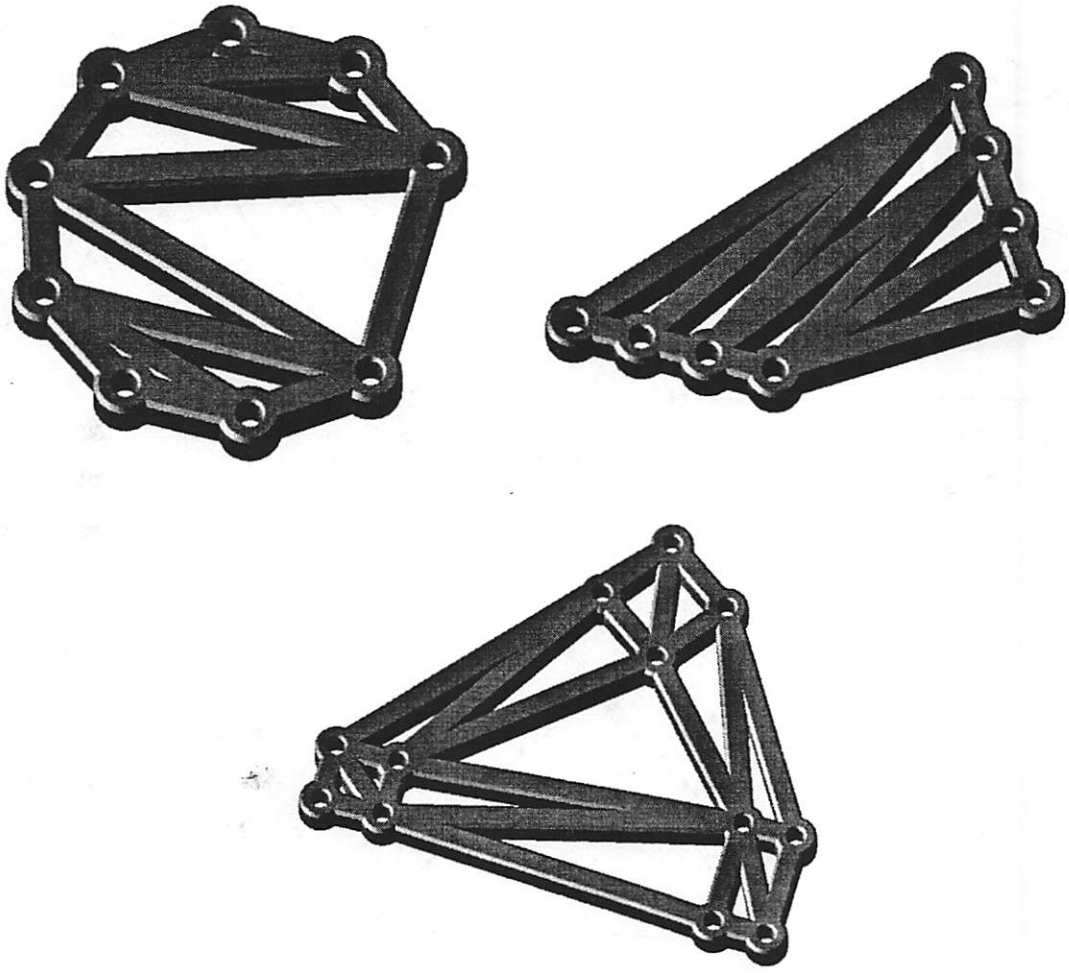


Figure 2.12: Parts generated from the skeletons in Figure 2.11.

lation, most of which run in $O(n \log n)$ expected time, including the one implemented for this research [?].

## 2.3 Conclusion

We are well on our way to confirming the conjecture of Chapter 1. In this chapter we have seen that reasonable skeletons can be generated by relatively simple and fast algorithms. Of those presented in this chapter, the Greedy $t$-Spanner and the Delaunay Triangulation will be directly useful as part skeletons. The Hierarchical Nearest Neighbor Graph also looks promising, but will benefit from some optimization of the embedding

of its Steiner nodes. The Euclidean Minimum Spanning Tree may be useful for some applications.

Some of the graphs we have studied are less promising for use as skeletons. By itself, the Centroid Star Tree is unlikely to produce reasonable parts. And in Chapter 3 we will see that as an initial graph for the optimizer, the Hierarchical Nearest Neighbor Graph is almost always better. The $t$-spanner generated by Salowe's algorithm does not appear to be useful to us at all because it is too highly connected except with very large node sets.

The Principal Axis Tree is probably too simplistic for applications that demand rectilinear parts, though it may be useful as an initial graph for optimization. The optimizer presented in the next chapter does not preserve rectilinearity, however. It will be worthwhile to investigate other rectilinear algorithms, and to devise optimizers that preserve rectilinearity.

Of the algorithms we have studied during the course of this research, the only ones that add Steiner nodes are tree algorithms. It seems reasonable that good skeletons could have general graph topologies that include Steiner nodes. It will be worthwhile to investigate general graph generating algorithms that add Steiner nodes.

It may also be worthwhile to use the Greedy $t$-Spanner algorithm to decimate a Delaunay Triangulation. Recall that the Greedy algorithm computes graph spanners, i.e., it must be given an initial graph of which it then computes a spanner. By adjusting the stretch factor, we can select from a spectrum of graphs that range from the Delaunay Triangulation ($t = 1$) to the EMST ($t = \infty$). This will also be more efficient than starting the Greedy algorithm with the complete graph, because the Delaunay Triangulation has only $O(n)$ edges, whereas the complete graph has $O(n^2)$ edges.

# Chapter 3

# Optimization of Skeletons

As noted in Chapter 2, many of our graph-generating algorithms need some help to create skeletons worthy of mechanical parts. In this section, we present one kind of help: optimization of material use and part stiffness.

Early in this research we took note of the Steiner Minimum Tree (SMT), thinking that it might yield good skeletons. The SMT Problem has received extensive study, dating back to the $17^{th}$ century [?]. It was first studied in the plane using the Euclidean distance metric: Given a set of points, $V$, in $\mathbf{R}^2$, the Euclidean Steiner Minimum Tree problem is to find the lowest-weight graph, $G$, that spans $V$ [?]. The nodes of $G$ are $\{V, S\}$, where $S$ is the set of Steiner nodes. There is no other graph in the plane that spans $V$ and has a lower weight than the Steiner Minimum Tree.

Unfortunately, computation of the Steiner Minimum Tree is NP-complete. An early technique for solving the Steiner Minimum Tree problem was mechanically based. It used nails in a board to represent the set of nodes $V$, and soap films to find the edges and Steiner nodes. Hundreds of papers have been written on the subject (and related Steiner Tree problems). Hwang and Richards [?] present a survey of 310 papers through 1989.

Since computing the Steiner Minimum Tree is NP-complete, most of the literature concerns algorithms for approximating it. This is also the approach we have taken. The literature contains a variety of heuristics, some computational, and others with more of an optimization flavor. We will ultimately want to define the weight (or cost) of generated skeletons in more sophisticated ways than just the sum of the Euclidean edge weights, so we choose to focus our attention on optimization-based schemes.

There really are two problems to be solved when seeking an optimal graph to

span a set of nodes. A topology must be chosen, and for that topology, an embedding in $\mathbf{R}^2$ must be chosen. The latter can be addressed using continuous optimization techniques, however the former cannot be. Solutions studied in the literature employ both simulated annealing [?, ?], and genetic algorithms [?].

Since we are striving for interactive speeds, we have implemented our own technique for addressing the discrete optimization problem. It is based on the observation that two topologies can be considered *adjacent* when their embeddings are degenerate. For example, when an edge has length zero (its end vertices are coincident), the graph appears the same as if the edge were removed and the two vertices were instead just one vertex. We say that these two topologies are *adjacent at this embedding.* In section 3.2.2 we describe our algorithm for exploiting this observation.

## 3.1 Local Continuous Optimization

In all our approaches, we will employ a continuous optimizer to find the best embedding for a chosen topology. We have implemented two continuous optimizers in the `graphlab` software: a simple relaxation method, and a robust and efficient conjugate gradient optimizer. The latter converges rapidly for all test inputs that we have tried. We thus feel confident that at least this portion of the global optimization problem is easily handled.

In both optimizers, we assume that we are not allowed to move the $V$-nodes, since they arise from application features input by the user. The optimization process will move only Steiner Nodes, possibly adding and deleting some. The optimization process may also add and delete edges.

### 3.1.1 Relaxation

The first continuous optimizer we have implemented is a simple relaxation technique. This technique treats graph edges as applying a fictitious force on the nodes. It is easy to show that when the objective function is the Euclidean graph weight, the appropriate edge force is a constant (independent of edge length) tension. For the relaxation process, a constant gain must be chosen that maps these fictitious forces to displacements. For each Steiner vertex in the graph the applied edge forces are summed. The resulting vector is multiplied by the relaxation gain and the node is moved by that amount.

**Algorithm 3.1** RELAX-GRAPH($G = (V, E), f(\cdot), k$)

*Takes a graph $G$ and finds an embedding in $\mathbf{R}^2$ that minimizes $f(G)$. The gain, $k$, controls the rate of convergence and the stability.*

```
1: repeat
2:    for each Steiner vertex v ∈ V do
3:       Let w_v be a vector stored with v.
4:       w_v ⇐ the vector sum of fictitious forces applied to v by D(f(G)).
5:    end for
6:    for each Steiner vertex v ∈ V do
7:       Move v to new location v + k · w_v.
8:    end for
9: until forever
```

The algorithm is shown in Algorithm 3.1. Note that no termination condition is indicated. In our implementation, the Relaxation Optimizer is allowed to run during idle CPU cycles. The rate of convergence is very slow, and gives rise to a rather interesting "rubber band" animation as nodes are interactively dragged and others respond to the new forces. The gain $k$ can be specified in the user interface. Values above 0.8 tend to introduce instability, and values above 1.0 cause the graph to explode immediately. Since this optimizer was really just a quick-and-dirty implementation to get something running, no further development of it was done.

### 3.1.2 Conjugate Gradient Optimization

The Relaxation Optimizer made it clear to us that optimization was a viable technique for generating skeletons. We therefore spent our effort on implementing a robust, efficient continuous optimizer that could be generally applied to improve the embedding of a given graph.

The core optimizer was taken from Press et al. [?]. It is a Polak-Ribiere conjugate gradient optimizer that works in arbitrary dimension, either with or without gradient information. The objective function we implemented is described below. It can compute gradient information, and a user-interface option selects whether the gradient is used by the optimizer. As one would expect, convergence is much faster when gradients are used.

This optimizer is not global. In general, techniques which seek global optima are

dramatically more expensive than those that operate locally. After having experimented with this optimizer, our subjective assessment is that the local optima that it finds are not terribly different from the global ones. Ultimately it may be desirable to provide a fast local optimizer for interactive use, and a global optimizer for offline use. The latter could be based on Adaptive Partitioned Random Search [?], Simulated Annealing, or Genetic Algorithms.

### Objective Functions

The first objective function we implemented is a simple generalization of the Euclidean weight. $f_1'(G = (V, E))$ is defined to be the sum of the individual costs of the edges in $E$, where these costs are computed as a quadratic polynomial in the Euclidean length of each edge $e$:

$$f_1'(G) = \sum_{e \in E} a\|e\|^2 + b\|e\| + c\|e\|^{\frac{1}{2}} \tag{3.1}$$

where $a$, $b$ and $c$ are constant coefficients.

The term $c\|e\|^{\frac{1}{2}}$ is worthy of comment: it is included for the purpose of smoothing out $f_1'(G)$ when the length of an edge becomes zero. Without this, $f_1'(G)$ is undefined at such points, and the CG optimizer gets stuck. A graphical interpretation of this term is to imagine that, though we are solving the problem in $\mathbf{R}^2$, the two endpoints of an edge always have $z$-coordinates that differ by some constant, i.e., one end is lifted out of the plane slightly. Thus, the length of an edge can never be zero. It is interesting to note that this discourages the actual ($\mathbf{R}^2$) length from ever being zero. The optimizer has little motivation to drive the in-plane length of the edge shorter than the fictitious $z$-offset. If $c$ is kept small, the resulting graphs are only slightly different than if $c$ were zero.

The quadratic term, $a\|e\|^2$, has a similar effect, but for the opposite reason: it penalizes long edges, so the optimizer spends less effort trying to shorten already-short edges. Thus the presence of the quadratic term also tends to smooth our $f_1'(G)$ so that the CG optimizer is less likely to get stuck.

It turns out, however, that when we apply our global optimization techniques in Section 3.2.2, we will actually *depend* on edges being driven to zero length. Further, except for their ability to smooth $f_1'(G)$, these two terms in Equation 3.1 are not very

useful for part skeletons. So, in practice we use only the linear term:

$$f_1(G) = \sum_{e \in E} \|e\| \tag{3.2}$$

### Network Weight

The second objective function we implemented is an augmentation of the first. It occurred to us that it may be desirable to penalize some edges differently than others depending on the role they play in the graph. In particular, some edges are elements of many node-to-node paths, while others may be contained in only a few paths. For a tree, it is easy to count the number of paths that contain a given edge by cutting the tree at that edge. The number of paths is the product of the numbers of $V$-nodes in each of the two resulting subgraphs.[1] We call this the *network weight* of the edge $e$, and define an objective function

$$f_2(G) = \sum_{e \in E} n_e \|e\| \tag{3.3}$$

where $n_e$ is the network weight of $e$. This objective function penalizes edges that participate in many paths more heavily than those that participate in just a few.

The results of this experiment are inconclusive. For some inputs, the resulting graphs appear to be stiffer than when network weights are turned off, but for other inputs the opposite is true. In fact, we can see a case for inverting the network term so as to favor edges that participate in many paths, then make spars in the part corresponding to those proportionally heavier. The second material synthesis technique implemented in the `graphlab` software uses this approach. For each edge of the skeleton, a spar is constructed with width proportional to its network weight. Figure 3.1 shows the results of running the optimizer on the example point sets. In the left column, the objective function is as in Equation 3.2, the total weight of the graph. The skeletons in the right column are optimized for the objective in Equation 3.3.

### Stiffness Criterion

Ultimately, the introduction of network weights is really just a kludge to try to make the optimizer compute graphs that will yield stiff parts. A more reasonable approach

[1] This only works for trees, not for general graphs. A more elaborate definition is necessary if we want to use network weights with all of our skeleton generation algorithms.

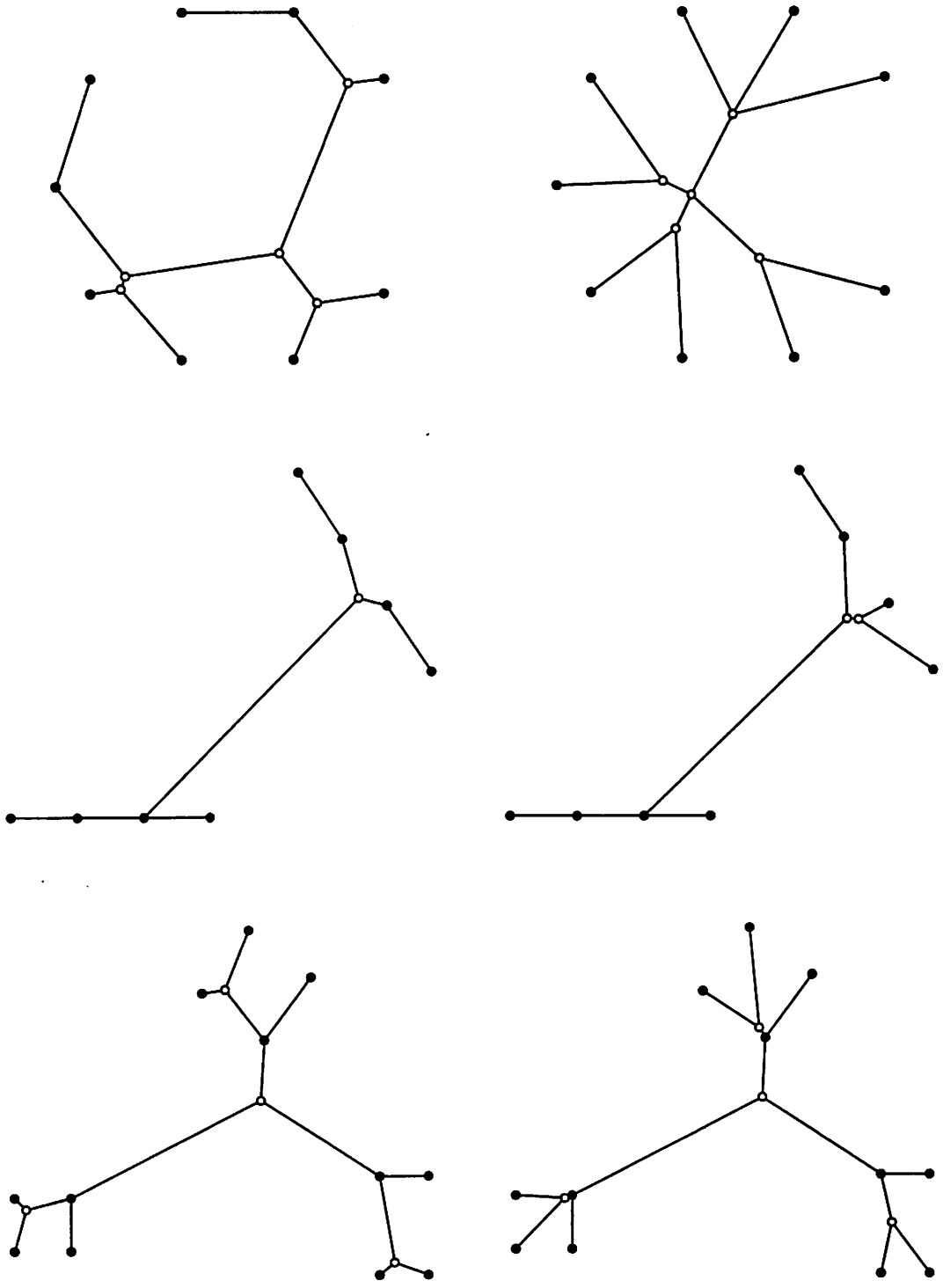


Figure 3.1: Skeletons optimized for the functions $f_1(G)$ (left column) and $f_2(G)$ (right column).

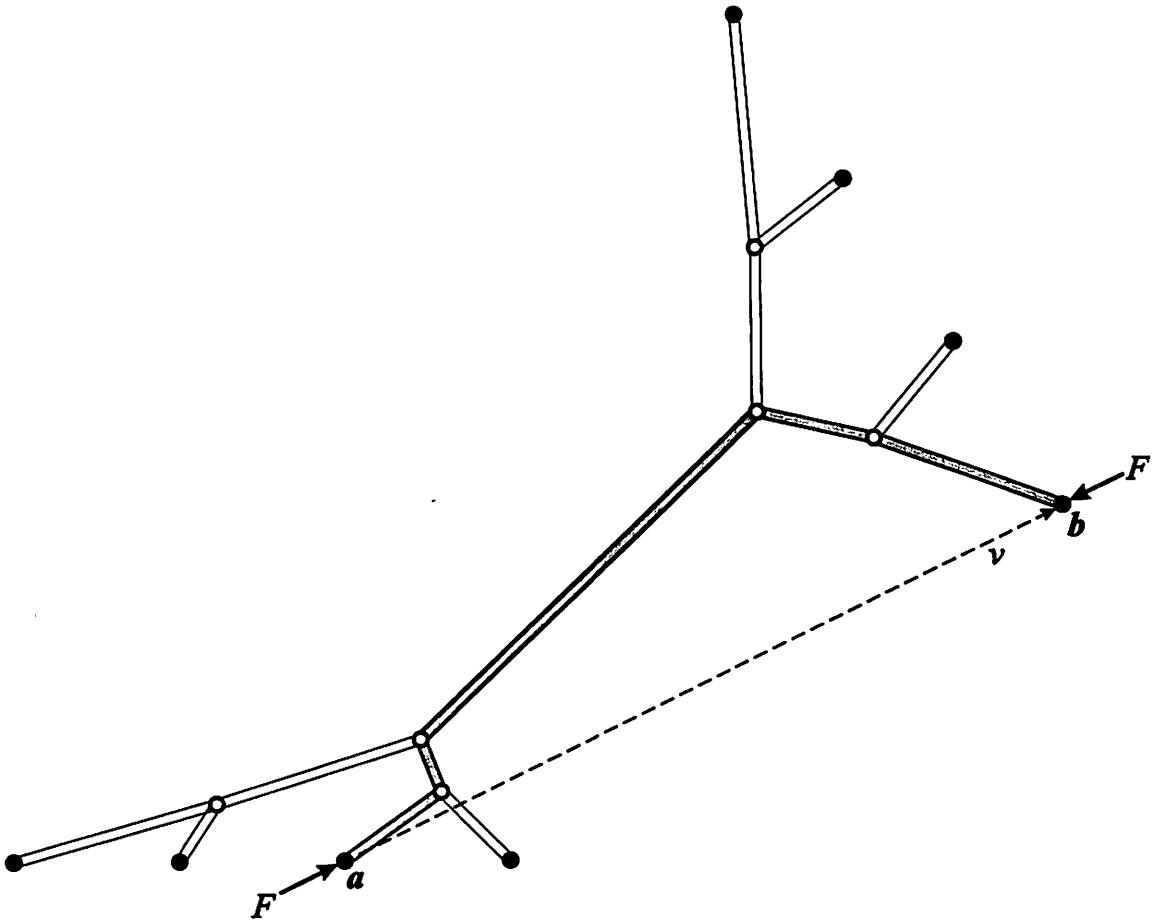



Figure 3.2: Beam elements along the path from node $a$ to node $b$.

is to construct an objective function that directly measures the stiffness of the part. What is required is to come up with some scalar-valued function that characterizes the stiffness of a given skeleton. In stress analysis we would simply apply the loads and compute the deflections. But remember that a basic goal of our research is to obviate this kind of input from the designer during the early stages. Thus we must contrive some way to estimate the goodness of a skeleton without knowing the loads.

We will assume that loads are only applied at the features. An example metric would be to compute a generalized compliance matrix and find its largest singular value (or sum of them). In $\mathbf{R}^2$ this matrix has dimension $3n \times 3n$, where $n = |V|$ is the number of nodes arising from application features. Computing this requires finite element analysis, so obviously it is going to be more expensive than the simpler objective functions

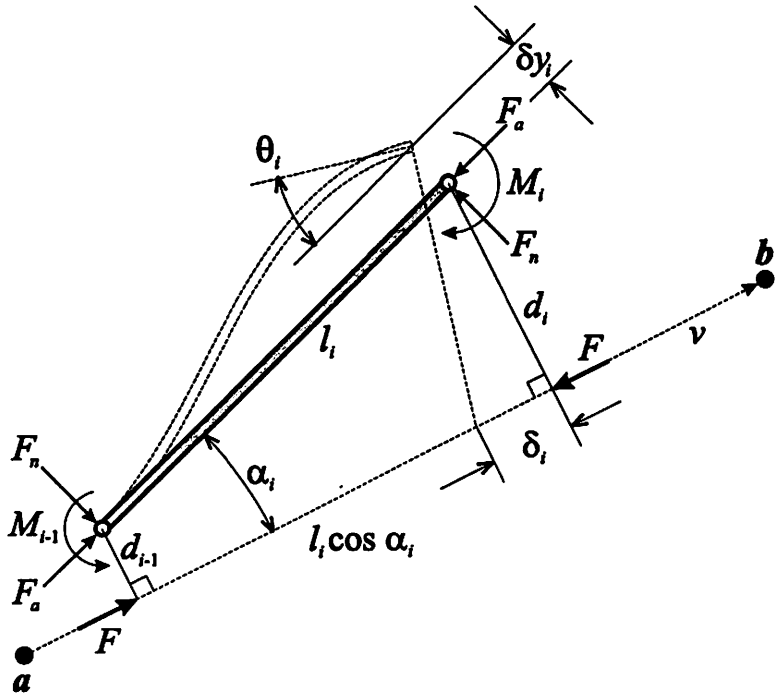



Figure 3.3: Free-body diagram of the $i^{th}$ edge along the path from node a to node b.

in Equations 3.2 and 3.3.

We have implemented a somewhat simpler measure of compliance for tree skeletons wherein the $\binom{n}{2}$ scalar node-to-node compliances are computed and summed. Figure 3.2 shows the skeleton for a hypothetical part. The black nodes are the $V$-nodes; the white are the Steiner nodes. We imagine applying equal and opposite forces, $F$, at two nodes $\boldsymbol{a}$ and $\boldsymbol{b}$, and computing the resulting deflection. The compliance contributed by this node pair is the ratio of the deflection to the applied force. This calculation is repeated for all $\binom{n}{2}$ pairs of $V$-nodes in the skeleton and the results are summed. We call this the generalized compliance of the skeleton.

While considering the pair $\boldsymbol{ab}$, we assume that no other loads are applied to the part. For static equilibrium, the force $F$ in Figure 3.2 must be collinear with the vector $v_{ab}$. Define the deflection to be the change in length of $v_{ab}$. To compute the compliance of this path through the skeleton, we must find the compliance contributed by each edge element along it. For each such element, we consider a free-body diagram, as shown in Figure 3.3.

Neglecting axial deflection of the element,

$$\delta_i = \delta y_i \sin\alpha_i + d_i\theta_i. \tag{3.4}$$

From simple beam equations we know that

$$\delta y_i = \frac{F_n l_i^3}{3EI} - \frac{M_i l_i^2}{2EI}, \tag{3.5}$$

and

$$\theta_i = -\frac{F_n l_i^2}{2EI} + \frac{M_i l_i}{EI}, \tag{3.6}$$

were $E$ is the Young's modulus of the material and $I$ is the beam's cross-sectional moment of inertia.

Substitute $F_n = F\sin\alpha_i$, $M_i = Fd_i$, combine and rearrange to get

$$\frac{EI\delta_i}{F} = \frac{1}{3}\sin^2\alpha_i\, l_i^3 + \sin\alpha_i\, d_i l_i^2 + d_i^2 l_i. \tag{3.7}$$

Call the term on the right hand side the normalized compliance for this element. Next apply the identity $l_i \sin\alpha_i = d_i - d_{i-1}$ and rearrange to get

$$\frac{EI\delta_i}{F} = \frac{1}{3}\left(d_i^2 + d_i d_{i-1} + d_{i-1}^2\right) l_i. \tag{3.8}$$

This is the normalized compliance contributed by the $i^{th}$ edge element along the path $\boldsymbol{ab}$.[2]

To correctly traverse the tree skeleton and compute all the necessary edge compliances we must do some bookkeeping. The total compliance of the graph is

$$f_3'(G) = \sum_{v\in V} \textsc{Subtree-Compliance}(G, v, \textbf{null}). \tag{3.9}$$

The SUBTREE-COMPLIANCE() algorithm is shown in Algorithm 3.2. The function PATH-COMPLIANCE($L$) evaluates Equation 3.8 for each edge in the list $L$ and returns the sum. Note that, as shown, Equation 3.9 and Algorithm 3.2 would actually compute every path compliance twice, once coming and once going. The actual implementation adds some bookkeeping to make sure that each gets computed only once.

[2]Interestingly, this also happens to be the second moment of inertia of the $i^{th}$ edge element measured relative to the axis $\boldsymbol{ab}$:

$$I_{ab} = \int_0^l [d(x)]^2\, dx$$

**Algorithm 3.2** SUBTREE-COMPLIANCE$(G = (V, E), v, L)$

*Takes a vertex $v$ in graph $G$ and computes the compliance of all paths in the subtree rooted at $v$. $L$ is the ordered set of edges leading up from $v$.*

```
1: local comp ⇐ 0.
2: for each edge e incident to v do
3:    Let u be the other endpoint of e.
4:    if u is terminal in G then
5:       comp ⇐ comp + PATH-COMPLIANCE({L, e}).
6:    else
7:       comp ⇐ comp + SUBTREE-COMPLIANCE(G, u, {L, e}).
8:    end if
9: end for
10: return comp.
```

**Multicriterion Optimization** Ultimately we will want to optimize for multiple criteria; Equations 3.2 and 3.9 are two perfect examples: We want to trade off material use and stiffness. But these two metrics exhibit a classic problem in multicriterion optimization. Their measurements are not in the same units, and they do not scale the same when the size of the input set changes. Equation 3.2 computes a cost in units of length, while Equation 3.9 returns units of length cubed. If the scale of the input set is doubled (i.e. multiply every coordinate of every node in $V$ by 2), the cost computed by Equation 3.2 doubles, but that computed by Equation 3.9 is multiplied by eight. We certainly don't want a change of units from inches to centimeters to suddenly shift nearly all the emphasis to stiffness. To offset this we will normalize the stiffness objective by the scale of the input set squared. For that measurement we will need some length that is representative of the scale of the input set. We will choose the graph's *diameter*, the maximum distance between any two nodes in $V$.

A second difference between these two cost functions appears when a new $V$-node is added to the input set. The skeleton generator must add at least one edge to connect this to the rest of $G$, so the sum-of-lengths cost in Equation 3.2 increases by the length of this edge. The compliance measured by Equation 3.9, on the other hand, varies proportionally to $n(n-1)$, which again shifts the emphasis to stiffness. To offset this we

will normalize by $n-1$. The normalized measure of compliance is thus

$$f_3(G) = \frac{f_3'(G)}{\phi(G)^2(n-1)}, \tag{3.10}$$

where $n = |V|$, and $\phi(G)$ is the diameter of $G$.

With this modification, the objectives for material use and part compliance can be linearly mixed with reasonably consistent results:

$$f(G) = (1-\lambda)f_1(G) + \lambda f_3(G) \tag{3.11}$$

Figure 3.4 shows the results of running the optimizer on the example point sets using Equation 3.11. In the left column, $\lambda = 0.5$; in the right column $\lambda = 1.0$. Figure 3.5 shows parts generated from those skeletons.

## 3.2 Global Optimization

The other half of the optimization problem is to find a topology. This is a discrete optimization problem, though we have found that local minima can be found using a sequence of continuous optimizations, with topology switching operations in between.

### 3.2.1 Collapsing Optimizer

The first discrete problem is somewhat of a gray area. A problem occurs with many continuous optimizers, including the conjugate gradient optimizer that we use: they get stuck if the gradient of the objective function becomes undefined. With many objective functions, this happens whenever an edge is driven to zero length.

One technique we have considered is to optimize once with nonzero $a$ and $c$ in equation 3.1, then smoothly decrease them on repeated optimization runs until they are negligible. This will allow the optimizer to reach a solution for the Euclidean weight problem without getting stuck on zero-length edges. We observe, however, that this may not be what is desired. A graph with many zero-length edges is really trying to be a different topology.

The approach we implement is to abort the CG optimizer whenever an edge's length is driven to zero. We then make the assumption that in the optimal solution this edge's length will still be zero. In other words, the optimal solution is actually on the topology that doesn't include this edge. We remove the null edge and merge its

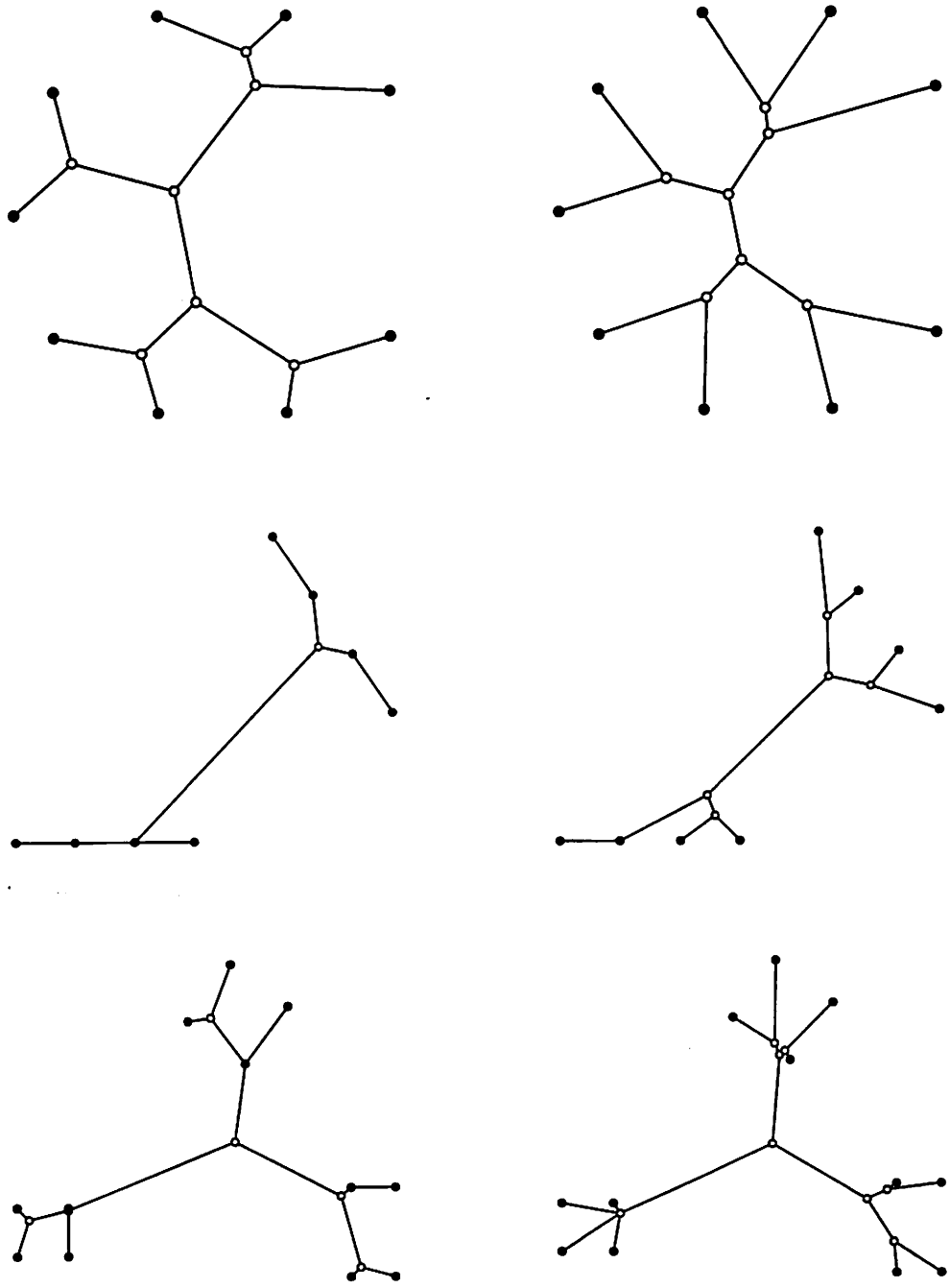


Figure 3.4: Skeletons optimized for the function $f(G)$ with $\lambda = 0.5$ (left column) and $\lambda = 1.0$ (right column). The Hierarchical Nearest Neighbor Tree was used as an initial graph.

Figure 3.5: Parts generated from the skeletons in Figure 3.4.

**Algorithm 3.3** COLLAPSING-OPTIMIZER($G = (V, E), f(\cdot)$)

*Takes a graph $G$ embedded in $\mathbf{R}^2$ and moves its Steiner vertices to a local minimum of the objective function $f(G)$.*

```
1: repeat
2:   success?⇐CONJUGATE-GRADIENT-OPTIMIZER(G, f).
3:   if success? and ∄ e ∈ E such that length(E) = 0 then
4:     return G.
5:   else
6:     for each edge e = [v1, v2] in E such that length(e) = 0 do
7:       if neither v1 nor v2 is a Steiner vertex then
8:         continue
9:       else if v1 is a Steiner vertex then
10:        Let v be v1 and let u be v2.
11:      else
12:        Let v be v2 and let u be v1.
13:      end if
14:      Let E' be the set of edges incident to v, e ∉ E'.
15:      Reassign the edges in E' to u.
16:      Delete e and v from G.
17:    end for
18:  end if
19: until forever
```

two vertices. Edges incident to the original two are now incident to the resulting single vertex. The CG optimizer is then restarted. This process continues in a loop until the CG optimizer converges without getting stuck. The algorithm is shown in Algorithm 3.3.

### 3.2.2 Topology Switching Optimizer

The above technique can be viewed as an augmented continuous optimizer that is able to get itself over bumps in the gradient by "fixing" the topology as it goes along. The result is that the graph is driven to an embedding where the objective is at local minimum.

It is often possible at this point to choose a new topology for the graph whose

**Algorithm 3.4** SWITCH-TOPOLOGY($G = (V, E), f(\cdot)$)

*Takes a graph $G$ embedded in $\mathbf{R}^2$ and finds a topology $G'$ that may be driven to a lower cost, $f(G')$, by the continuous optimizer.*

```
1: while ∃ a Steiner vertex v ∈ V such that degree(v) ≠ 3 do
2:   call COLLAPSING-OPTIMIZER(G).
3:   for each Steiner vertex v ∈ V such that degree(v) = 2 do
4:     Let e1 = [v, v1], e2 = [v, v2] be the edges incident to v.
5:     Delete v, e1, and e2 from G.
6:     Create new edge e = [v1, v2] in G.
7:   end for
8:   call COLLAPSING-OPTIMIZER(G).
9:   for each Steiner vertex v ∈ V such that degree(v) > 3 do
10:    call SPLIT-VERTEX(v).
11:  end for
12: end while
13: return G.
```

objective can be further lowered by the above methods. The technique is based on the observation that two topologies can be considered *adjacent* when their embeddings are degenerate. The simplest kind of degeneracy is a vertex of degree two (two incident edges). This is really just one edge with a bend in it (at the vertex). In the absence of obstacles, any objective function that seeks to minimize material use or maximize stiffness will never choose a bent edge over a straight edge. In other words, the continuous optimizer will always drive degree-two vertices to have their edges incident at 180 degree angles. Such vertices can always be removed and their edges replaced with one.[3]

The main opportunity for re-wiring the graph occurs at vertices with large degree (many incident edges). Such a vertex can be viewed as a kind of degeneracy: it looks just like two coincident vertices with a zero-length edge between them. Under certain conditions, we can replace the vertex with exactly that configuration, dividing the incident

[3]Note however that this does have implications when the objective function includes terms that are nonlinear in the lengths of the individual edges (e.g., sum of squares). Consider the design of a six-inch long straight bar. The simplest graph is two vertices with one six-inch edge between them. Equivalently, this design could be represented by a graph with seven vertices and six one-inch edges all in a straight row. A sum-of-squares objective function would assign a lower cost to the later representation even though the resulting part is clearly equivalent. We must keep this in mind when we design objective functions.

edges among the two new vertices. The continuous optimizer can then drive these vertices apart in a way that lowers the overall cost of the graph. The algorithm is shown in Algorithm 3.4. The function SPLIT-VERTEX($v$) is discussed in Section 3.2.3, below.

This algorithm depends on topologies being adjacent. In other words, the COLLAPSING-OPTIMIZER must drive edges to zero length or very few topologies will be explored. Because of this, the type of objective function we choose is limited. In particular, the square-root term in Equation 3.1 prevents edges from ever being driven to zero length. The quadratic term is not quite as severe, but it does discourage edges from getting too short. For this reason we use the cost function in Equation 3.2 in preference to Equation 3.1. Figures 3.6, 3.7 and 3.8 show the results of running the SWITCH-TOPOLOGY optimizer on each of the example point sets with a variety of initial graphs. The objective is as in Equation 3.2.

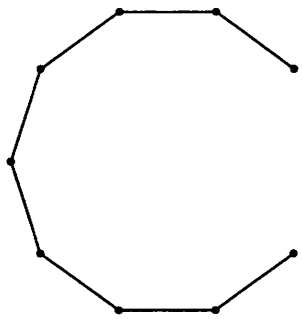
### 3.2.3 Vertex Splitting

One of the more involved aspects of the above algorithm is the SPLIT-VERTEX($v$) function. In this routine we are asked to split a vertex to create a topology that can be driven to a lower cost than the present cost. The result is to be two vertices, and the set of incident edges is divided between the two. A new edge is created to join the two vertices. The job of this function is to decide how the original edge set should be divided.

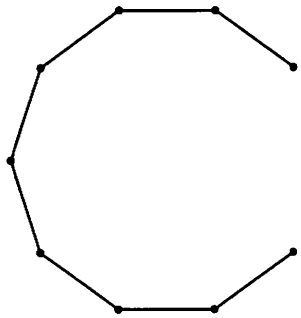
The idea is to divide the edge set in such a way as to maximize the gradient of the objective in the resulting topology. Ideally this is done by examining partial derivatives of the contribution that each incident edge makes to the objective. This we implemented, but first we implemented a simpler heuristic.

#### Largest and second largest angle

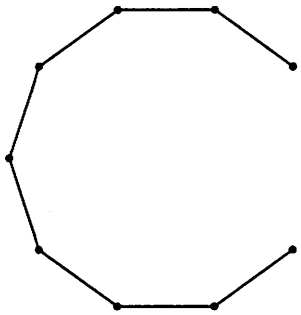
The heuristic is to look at the angles of the incident edges and find the two edges that form the largest angle. The set is then divided by a cut that falls within this angle such that each set includes edges that fall within a 180-degree segment. This still leaves some ambiguity, so we choose to look for the second-largest angle and use that for the opposite side of the cut. If the graph was previously optimized then this angle will usually fall roughly opposite the largest angle. None of this is very scientific, but it works reasonably well in practice. This is because when using a Euclidean objective and the



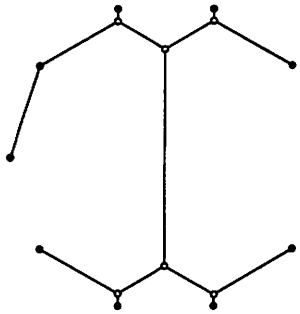
EMST



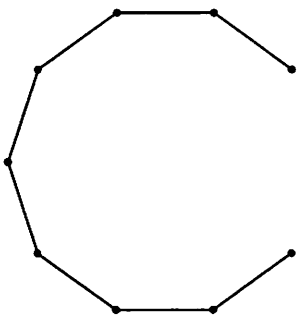
Principal Axis



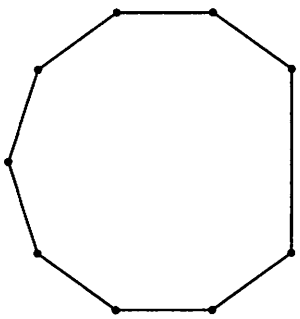
Centroid Star Tree



Hierarchical Nearest Neighbor

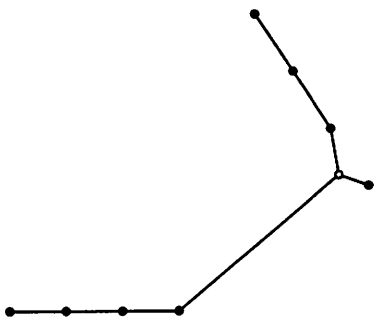


Greedy $t$-Spanner

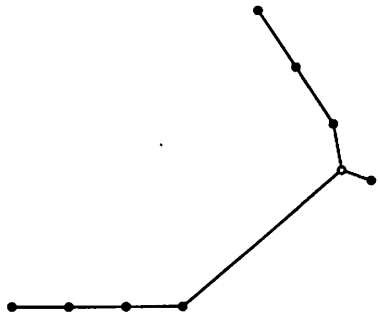


Delaunay Triangulation

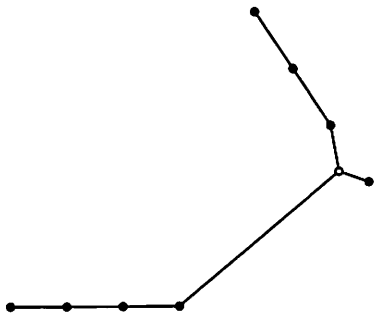
Figure 3.6: Skeletons optimized by SWITCH-TOPOLOGY() for the function $f_1(G)$ and a variety of initial graphs.



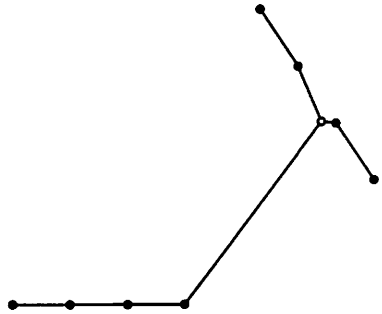
EMST



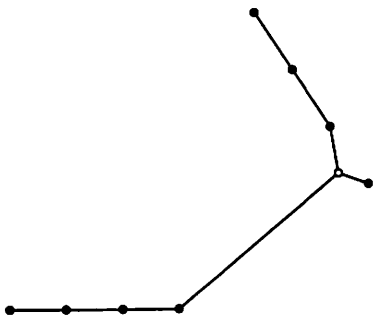
Principal Axis



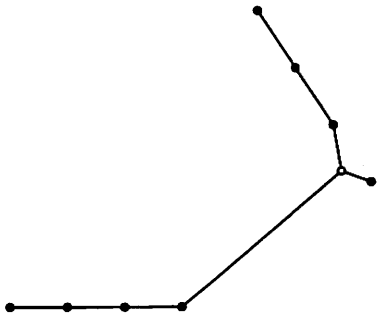
Centroid Star Tree



Hierarchical Nearest Neighbor

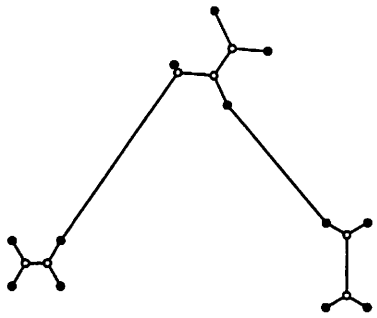


Greedy $t$-Spanner

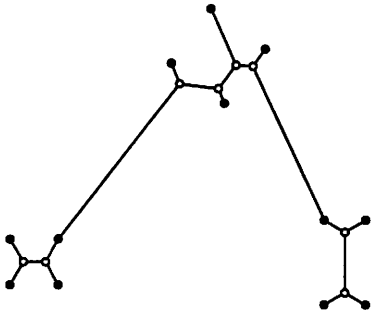


Delaunay Triangulation

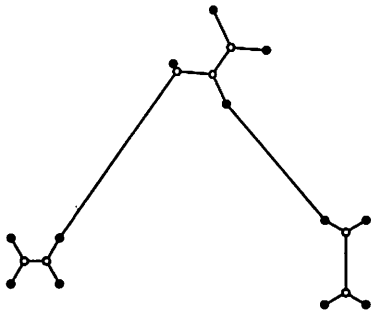
Figure 3.7: Skeletons optimized by SWITCH-TOPOLOGY() for the function $f_1(G)$ and a variety of initial graphs.



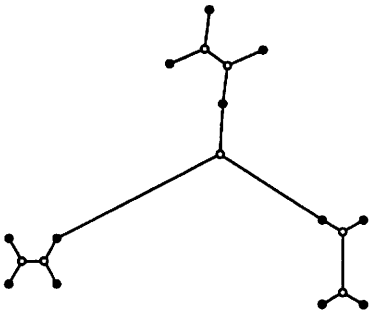
EMST



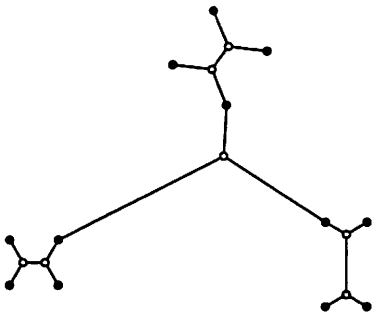
Principal Axis



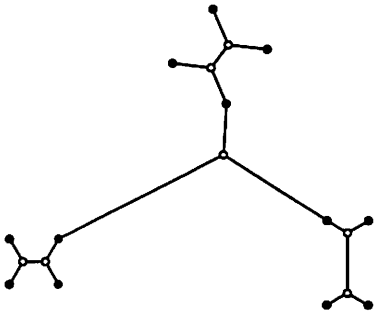
Centroid Star Tree



Hierarchical Nearest Neighbor



Greedy $t$-Spanner



Delaunay Triangulation

Figure 3.8: Skeletons optimized by SWITCH-TOPOLOGY() for the function $f_1(G)$ and a variety of initial graphs.

graph has been locally optimized, for many nodes this division does indeed maximize the resulting gradient.

At this point we should explain that it is not productive to divide a vertex such that one of the resulting edge sets has only one edge. When divided, the vertex will be replaced by two, and a new edge will be created between them. If one of the two vertices is assigned only one of the original edges, then it will end up with degree equal to two. As such this vertex will be removed in a subsequent iteration of SWITCH-TOPOLOGY(). A corollary to this is that we cannot split vertices of degree less than four.

### Principal axes of stress

The simple heuristic served us well for some time, but it became apparent that it was not correct in all cases. We thus decided to implement a splitting algorithm that was based on actual gradient information. Subject to a few assumptions, it can be shown that this technique always computes the split that maximizes the resulting gradient of the objective. The method is based on the concept of the "internal stress" in the vertex.

Consider the gradient vector of the objective function. It is a $2n$ vector of partial derivatives. Each vertex contributes two elements to this vector (for problems in $\mathbf{R}^2$). One element describes the rate of change of the objective function as the vertex moves in the $x$ direction (the partial with respect to $x$). The other element does the same for movement in the $y$ direction.

For the objective functions described by Equations 3.2 and 3.10, the partial derivatives at a vertex can be decomposed as the sum of contributions from each of the edges attached to that vertex. Thus each edge can be viewed as applying a force (vector) to the vertex. In the case of Equation 3.2, this force will be tensile and aligned with the edge. For the objective defined by Equation 3.10 the force may be in any direction. If the graph has been optimized prior to calling this splitting routine, then the gradient of the objective function should be zero. In other words, the forces applied to a vertex should sum to zero—the graph is in static equilibrium.

From stress analysis we know that the internal stress of an infinitesimal element can be measured relative to any coordinate direction, and that there is a direction that maximizes the value of tensile stress. We seek to split the vertex perpendicular to that direction. Note that this is a fictitious stress, not a physical one.

The method we use is to imagine a line through the vertex that divides the edges into two sets: a *cut*. A force vector is computed as the sum of the partial derivative vectors of all the edges in one set. (If we know that the vertex is in static equilibrium, we can assume that the force produced by the set on the other side of the cut must be equal and opposite. It turns out not to be practical to make this assumption, however, so we compute the force for each set.)

The cutting line may be oriented at any angle between zero and $2\pi$. We are looking for the cut that produces the force with the largest magnitude. Thus we imagine rotating the line through this range and examining all the possible divisions into two sets. During this rotation of the line, we compute the force vector for each orientation of the cut. To facilitate this, we establish a queue to contain those edges in one of the sets. As the line rotates, an edge not currently in the queue is added at the front when the line crosses over its partial derivative vector. Likewise, when the line crosses the vector corresponding to an edge currently in the queue it is removed from the end. Each such crossing is deemed an *event*.

We must remember the above restriction that the two resulting edge sets must both have at least two edges. Thus we seek the cut angle that produces the largest force subject to this restriction. To support this, we allow the head and tail of the queue to lag or precede the cutting line by one edge when necessary to ensure that enough edges are in each set at all times.

### Limitations

The current implementation of the `graphlab` software does not include gradient computations for the objective of Equation 3.9. This information is necessary for the SPLIT-VERTEX() function, so we cannot yet use the SWITCH-TOPOLOGY() (Algorithm 3.4) optimizer with that objective or the one in Equation 3.11.

We can, however, use the COLLAPSING-OPTIMIZER() algorithm, followed by a cleanup step to remove any degree-2 Steiner nodes. This is equivalent to executing lines 2 through 8 of Algorithm 3.4. We must choose an initial graph that has plenty of Steiner nodes, because this approach can only remove them, not add them. The skeletons in Figure 3.4 are optimized this way, using the Hierarchical Nearest Neighbor Tree as an initial graph. The objective is Equation 3.11 with $\lambda = 0.5$ for the left column and $\lambda = 1.0$

for the right column. In that figure we see that all the Steiner nodes have degree three. If we were trying to optimize this with Algorithm 3.4, lines 9 through 11 would not execute because of this, and the algorithm would terminate. Thus, for the three example problems, the Hierarchical Nearest Neighbor Tree provides an initial graph that does not require the services of the SWITCH-TOPOLOGY() optimizer. We have observed that this is not always true for arbitrary problem sets, but it does happen frequently.

Figure 3.9 shows the results of applying the same technique to Principal Axis Trees. In that figure we see a number of higher-degree Steiner nodes, indicating that these graphs could probably be further optimized if we were able to apply the SPLIT-VERTEX() function. The other four initial-graph algorithms do not generate enough Steiner nodes, so we cannot use those at all as initial graphs until gradient computations are implemented for Equation 3.9.

## 3.3 Conclusion

The research presented in this chapter has demonstrated that numerical optimization is a viable approach to generating part skeletons. Of the skeleton algorithms presented in Chapter 2, those that add Steiner nodes can all benefit from an application of the optimizer to improve the position of those nodes.

The SWITCH-TOPOLOGY optimizer makes fairly drastic changes to the topology of the skeleton such that, for many problem sets, the selection of initial graph has little influence on the final skeleton. This choice does, however, affect the amount of work that the SWITCH-TOPOLOGY optimizer must do. The Hierarchical Nearest Neighbor Tree seems to be the best all-around choice of initial graph. It consistently provides an initial graph that requires little effort on the part of the SWITCH-TOPOLOGY optimizer. In cases where the resulting graph differs markedly from what would result from other initial graphs, the skeleton optimized from the the Hierarchical Nearest Neighbor Tree usually appears (qualitatively) to be the most reasonable. For cost functions where we can not yet use the SPLIT-VERTEX step of the SWITCH-TOPOLOGY optimizer, we still find that high-quality skeletons can be produced from the Hierarchical Nearest Neighbor Tree.

It seems clear that a cleanup step in the material synthesizer will be an important addition. Looking at Figure 3.5 we see numerous bumps and gouges that are artifacts from the skeleton algorithms. If the part is to be cut on a milling machine, fillets must be added.

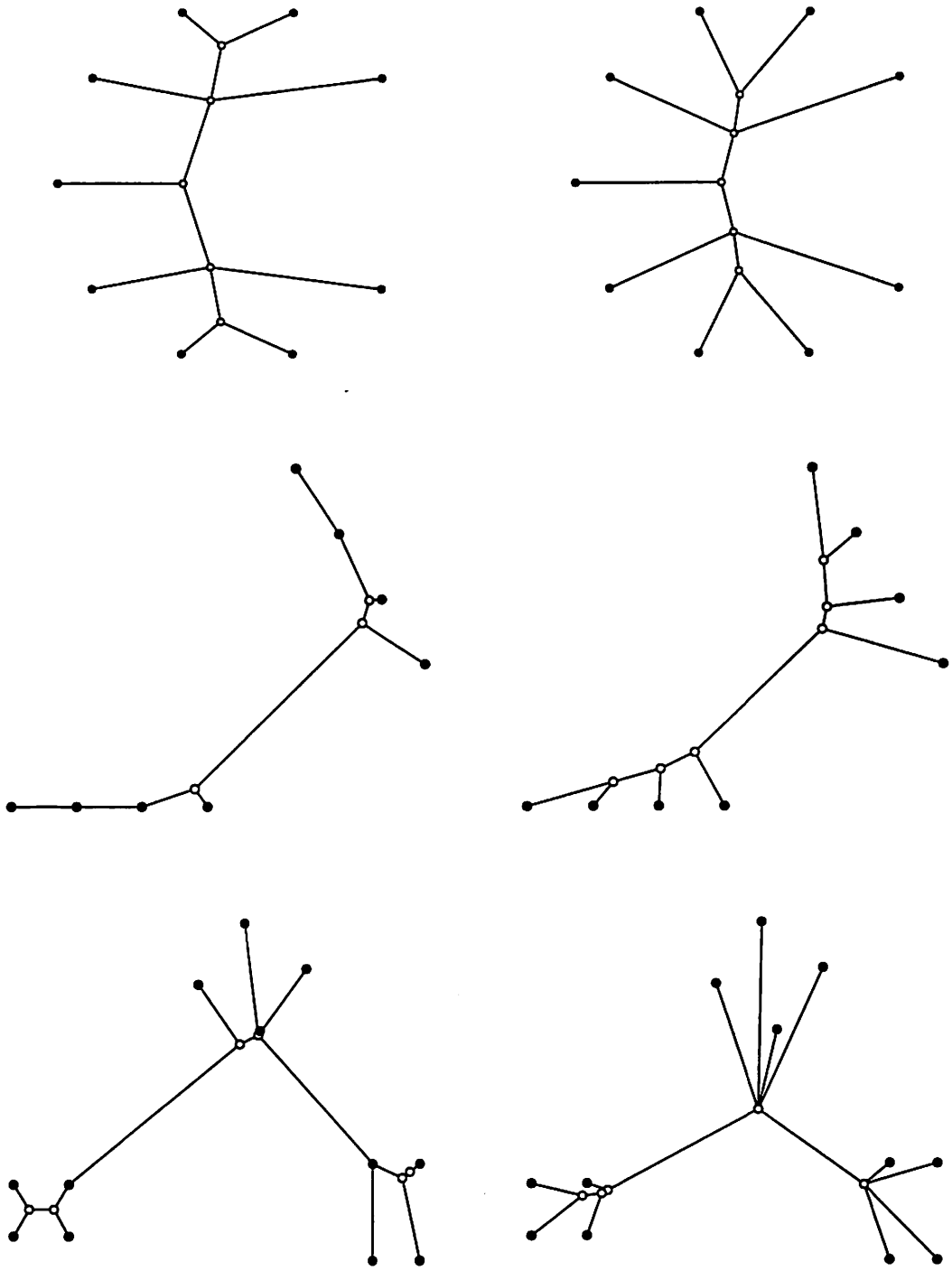


Figure 3.9: The same optimization as in Figure 3.4, but using the Principal Axis Tree as the initial graph.

This will naturally have the effect of smoothing some of the deeper gouges.

It also appears that some sort of clustering should be done. For example, the third point set in all the examples is composed of three sets of holes, each with four holes arranged in a rectangle. A good shape for this part might be similar to that shown at bottom-left of Figure 3.5, but with solid rectangles of material around each group of four holes. One approach might be to do this in the material synthesizer using an alpha-hull. The computation is nearly identical to finding the fillets that would be produced by a milling cutter. The larger the diameter of cutter, the more concavities that are removed. The actual alpha hull computation doesn't produce the scallops that would have been caused by the imaginary cutter. Instead, straight edges and flat surfaces cover over the former concavities.

More sophisticated objective functions may be less likely to drive edge lengths to zero. It may be useful to generalize the Collapsing-Optimizer and the Split-Vertex functions to switch between topologies that are *nearly adjacent.* In other words, instead of only collapsing zero-length edges, some criterion would select edges that are *short enough* and evaluate whether there is a better way to reapportion the edges incident to its two end nodes.

# Chapter 4

# Avoiding Obstacles

It is essential that synthesized part shapes not interfere with other parts in the assembly. One approach to ensuring this is to treat shape synthesis as a path-planning problem and borrow techniques from the field of robotics. As stated in Chapter 1, we will preprocess the information about other parts to yield a set of illegal regions, or obstacles, that pertain to the part being synthesized.

Most robotics path-planning techniques account for the size of the robot by transforming the problem to *configuration space* (CSPACE) [?, ?]. In the case of a circular robot moving through a two-dimensional field of obstacles, CSPACE is parameterized by the coordinates of the center of the robot. For skeleton synthesis, CSPACE is parameterized by the coordinates of a point on the skeleton. *Obstacles* in CSPACE are sets that must not contain any point on the skeleton, lest the thick skeleton interfere with an illegal region in physical space. If the thick skeleton is to be of uniform width, CSPACE is computed simply by growing the illegal regions by one-half the width of the thick skeleton, plus any required clearance distance. The skeleton synthesis algorithm can then operate in this transformed space to plan the skeleton, and we will be assured of being able to grow the thick skeleton without interference.

We have implemented this approach in the `racerx` program. Figure 4.1 shows a part being designed in the presence of four polygonal obstacles. In the program, features (holes) and illegal regions (polygons) are positioned with the mouse. Both can be modified interactively by dragging. A thick skeleton is generated that connects the features and avoids the illegal regions. The resulting design is rendered by extruding the part into the third dimension, and resembles a physical part that could be made by bending and welding

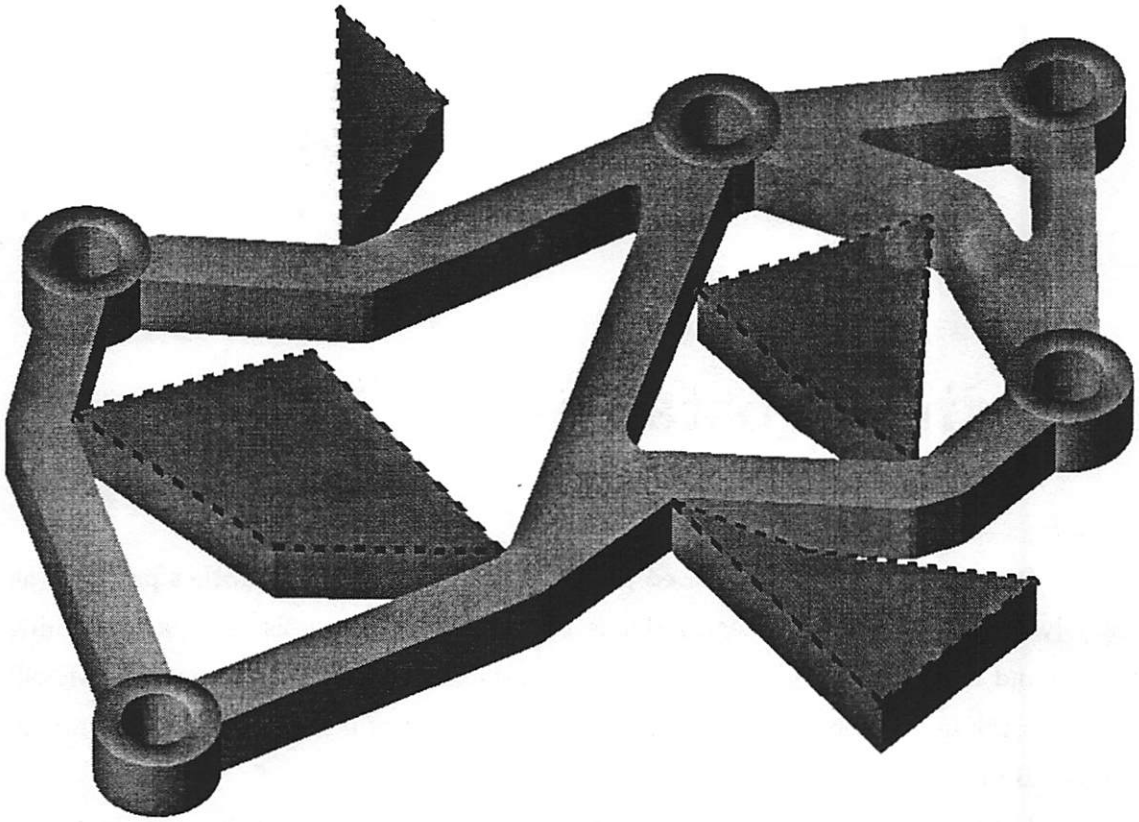


Figure 4.1: A 2-D part synthesized by the `racerx` software.

bars, or by milling.

In addition to specifying local features (application features) the `racerx` program supports global attributes for parts. Two such part attributes are spar width and illegal region clearance. The user can modify these attributes for the part via an on-screen slider, and see the part dynamically resynthesize. The algorithms are currently fast enough to resynthesize a reasonably complex part several times per second, thus when the user interactively drags a feature or edits an illegal region, the part appears to stretch.

This program implements only one skeleton algorithm, the Delaunay Triangulation, which is computed without regard for illegal regions. Then, for each edge of the triangulation, a path is planned through the field of obstacles. This is done by first computing CSPACE (illegal regions are grown by half the spar width plus the clearance), and its visibility graph [?]. Dijkstra's algorithm [?] is then used to find the shortest path for each connectivity edge.

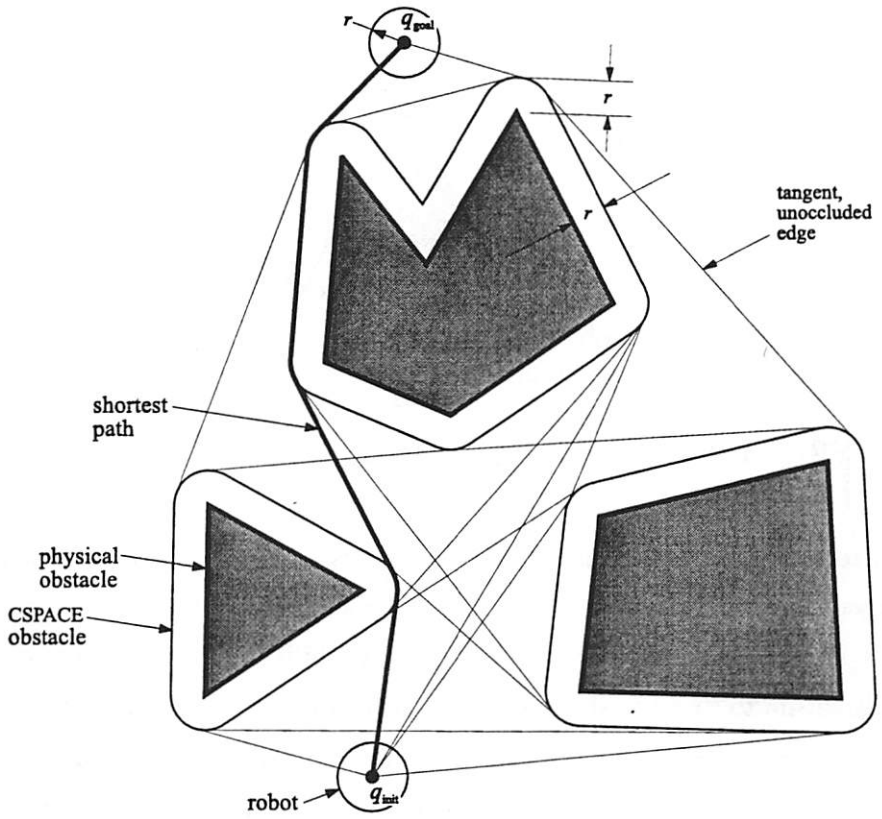



Figure 4.2: CSPACE and the Reduced Visibility Graph for a circular robot.

## 4.1 Configuration Space

The conventional definition of CSPACE for a mobile robot (with no rotation allowed) is the Minkowski sum of the environment with the robot. For a circular robot, the CSPACE obstacles are made up of straight edges and arcs of radius $r$, the radius of the robot. This can be computed by offsetting the edges of the obstacles by $r$ and constructing an arc of radius $r$ at each vertex.

Figure 4.2 shows an example path-planning problem. There are three polygonal obstacles (in physical space), and the robot is to be moved from the start location, $q_{\text{init}}$ to the finish, $q_{\text{goal}}$, via the shortest possible path. The CSPACE obstacles are computed by enlarging the physical obstacles by $r$ on all sides.

To find the shortest path from $q_{\text{init}}$ to $q_{\text{goal}}$, we begin by computing the *reduced visibility graph* [?]. The edges of this graph include (most of) the straight edges and arcs that outline the CSPACE obstacles. In addition, this graph contains all line segments that

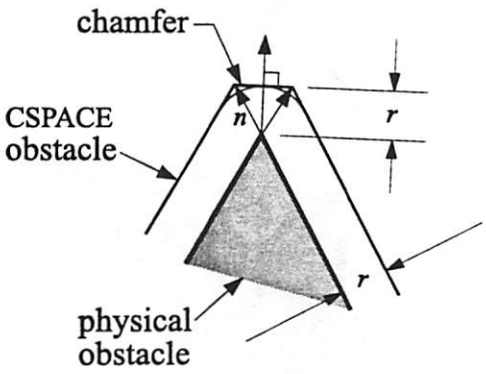



Figure 4.3: Chamfered approximation to a CSPACE vertex arc.

are tangent to two arcs, and do not intersect any CSPACE obstacles. Similarly included are all line segments that are tangent to one arc, incident to either $q_{\text{init}}$ or $q_{\text{goal}}$, and do not intersect any CSPACE obstacles. The reduced visibility graph is then searched using Dijkstra's algorithm to find the shortest path between two points in a weighted graph.

This technique also solves the problem of routing an edge of a part skeleton. The points $q_{\text{init}}$ and $q_{\text{goal}}$ represent the locations of two application features that are to be connected with a spar of solid material. The width of the spar is to be $2r$, and the shape of the resulting spar is identical to the path that would be swept out by the robot. To synthesize a part as in Figure 4.1, CSPACE and its VGRAPH are computed once. For each edge of the Delaunay Triangulation, one endpoint is arbitrarily deemed $q_{\text{init}}$, the other is deemed $q_{\text{goal}}$, and the shortest path through the VGRAPH is found. This is repeated for all edges of the Delaunay Triangulation, and the union of all these paths comprises the thick skeleton.

### 4.1.1 Chamfered CSPACE

In the `racerx` program, we have implemented an approximation of the above technique. When computing the configuration space obstacles, a conservative approximation is used so that CSPACE remains polygonal. For each vertex of a physical obstacle, the `racerx` program computes a *chamfer* of width $r$ rather than an arc of radius $r$. Figure 4.3 illustrates a chamfer for one vertex. It is computed so that the CSPACE obstacle has one edge for each edge of the physical obstacle, plus one additional edge for each vertex of the physical obstacle. The chamfer edge is perpendicular to the line that bisects the physical

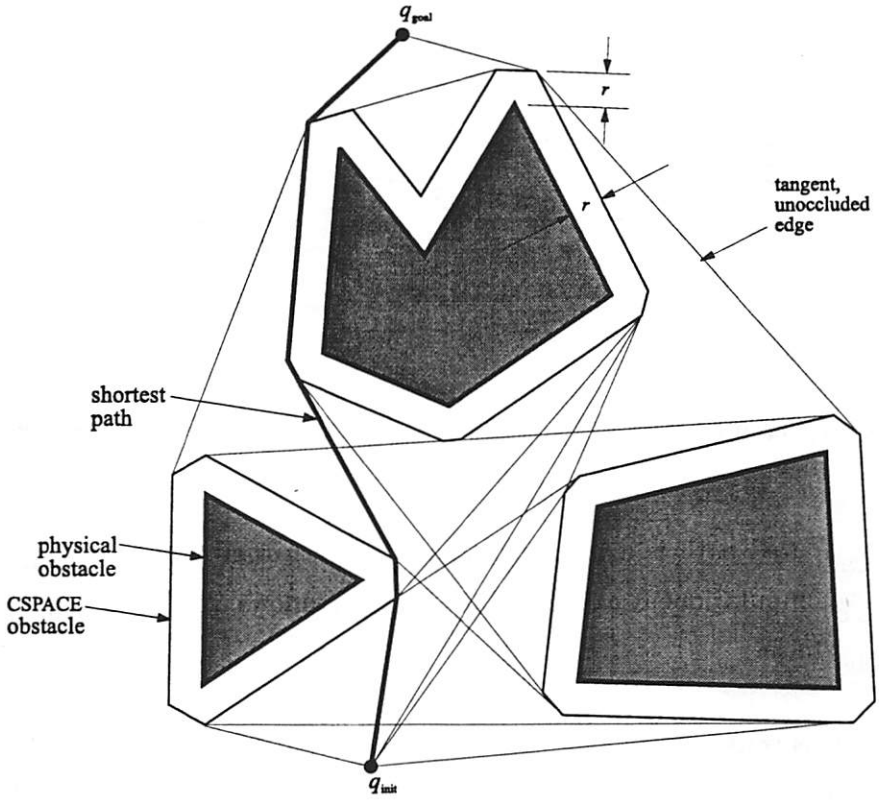



Figure 4.4: Chamfered CSPACE and Reduced Visibility Graph.

vertex, and lies at a distance $r$ from the vertex.

Figure 4.4 shows the chamfered CSPACE for the same path planning problem as in Figure 4.2. Since the chamfer circumscribes the true CSPACE arc at a vertex, this technique is conservative. Further, this technique has some advantages over the true CSPACE computation:

- The resulting configuration space is simpler because it is polygonal. Eliminating arcs from the visibility graph simplifies a number of computations.

- The visibility graph has fewer vertices than when CSPACE obstacles include arcs in their boundaries. When several VGRAPH edges are tangent to the same arc, their points of tangency will all be different (except in the case of collinear edges). Each point of tangency results in a vertex in the visibility graph.

There are also some slight disadvantages to this technique:

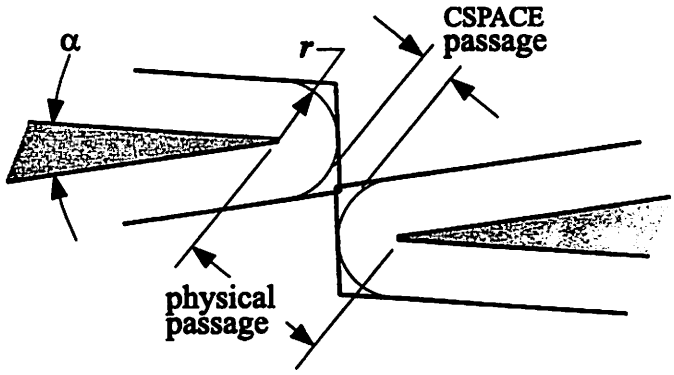



Figure 4.5: Chamfered CSPACE may in some cases erroneously show a passage to be too narrow.

- The computed path is a bit longer than the path that would be found by a true CSPACE computation. It also has kinks where it follows a chamfer around the vertex of an obstacle.

- Because it is conservative, the technique is not complete: there are cases where the shortest-path solution will be erroneously discarded. A longer path may be found instead, or the algorithm may decide that no solution exists. Figure 4.5 shows that the worst case error is $r(1-\sqrt{2})$, i.e. that as $\alpha$ goes to zero, a passage of width $2r\sqrt{2}$ may exist but be considered too narrow to admit a path.

At this point we have no way to judge whether these disadvantages would be important in any given application. If they are, one solution is to increase the number of chamfer edges at vertices with small $\alpha$. In this case, we may also wish to eliminate the chamfer edge at vertices for which $\alpha$ is large.

## 4.2 The Parametric Visibility Graph

When designing real parts, it is likely that we will want spars of differing widths. This arises when each spar in the part must have its own width, and when the user interactively varies the width of some spars. In the latter case, a large number of spar widths must be computed and displayed, because the part is synthesized several times a second. In path planning, this would be analogous to solving the problem for different sized robots.

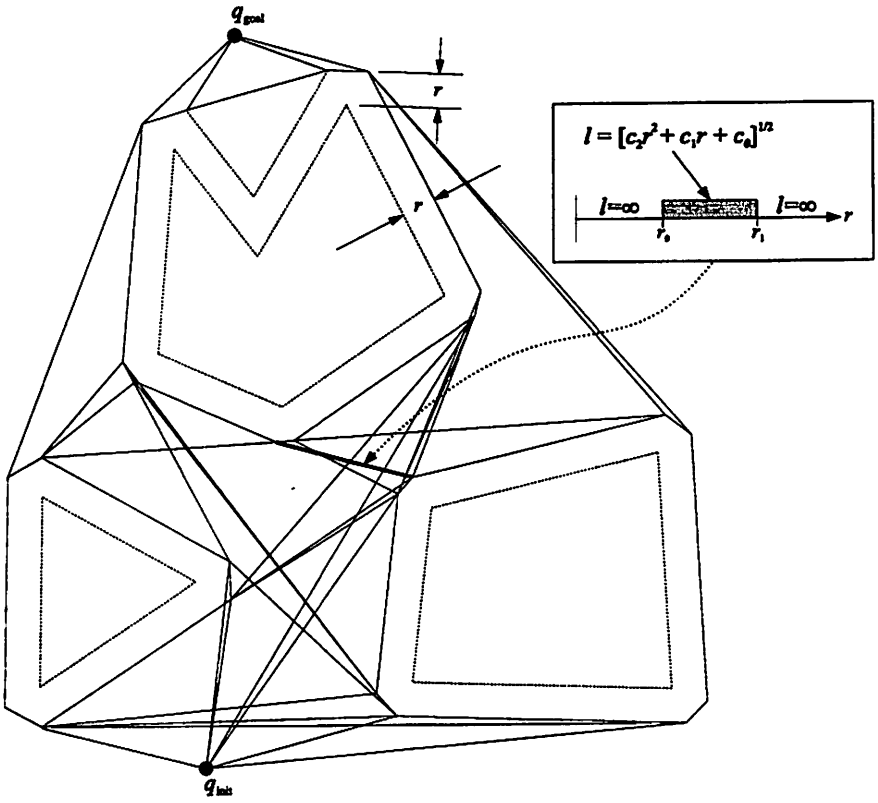



Figure 4.6: The Parametric Visibility Graph.

The direct way to do this is to recompute CSPACE and its VGRAPH for each new value of $r$. But this potentially requires a great deal of computation because of all the different values of $r$ that may be required. The interactive speeds we are trying to achieve may be compromised. We have developed a data structure which represents all possible configuration spaces and their visibility graphs as a function of a single parameter, $r$. We call this the *Parametric Visibility Graph* (PVGRAPH).[1] It is based on the chamfered approximation of CSPACE.

Figure 4.6 illustrates the PVGRAPH. The data structure is a graph, with a vertex corresponding to each vertex of the chamfered CSPACE obstacles, and a vertex corresponding to each application feature. The set of edges of this graph include each edge that could possibly belong to the visibility graph for some positive value of $r$. To each such edge in the PVGRAPH data structure is attached information about that edge, parameterized by

[1]This has not been implemented in the racerx software.

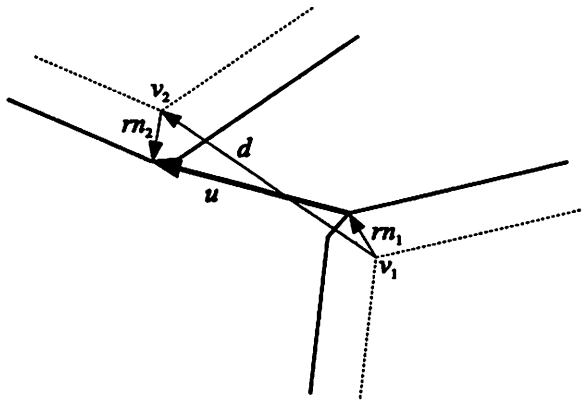



Figure 4.7: Computing the length of a visibility edge in the Parametric Visibility Graph.

$r$. As shown in Figure 4.6, this includes the length of the edge as a function of $r$, and the ranges of $r$ for which this edge actually appears in the reduced visibility graph.

The computation of this data is straightforward when the chamfered CSPACE approximation is used. Figure 4.7 depicts a typical visibility edge. The directed edge, $u$, is the vector sum

$$u = d + r(n_2 - n_1), \tag{4.1}$$

where $n_1$ and $n_2$ are vectors that define the locations of the chamfer vertices. They range in length from unity to $\sqrt{2}$. Define

$$\delta = n_2 - n_1, \tag{4.2}$$

and substitute into Equation 4.1 to get

$$u = d + r\delta \tag{4.3}$$

When one of the vertices corresponds to an application feature, its position in CSPACE is fixed, independent of $r$, so its $n$ vector is zero. In such cases, $\delta$ is either $n_2$ or $-n_1$ as appropriate. When both vertices correspond to application features, $\delta = 0$ and $u$ is fixed, independent of $r$.

### 4.2.1 Length of an Edge

The length of $u$ is defined by

$$\| u \| = \left[c_2 r^2 + c_1 r + c_0\right]^{\frac{1}{2}}, \tag{4.4}$$

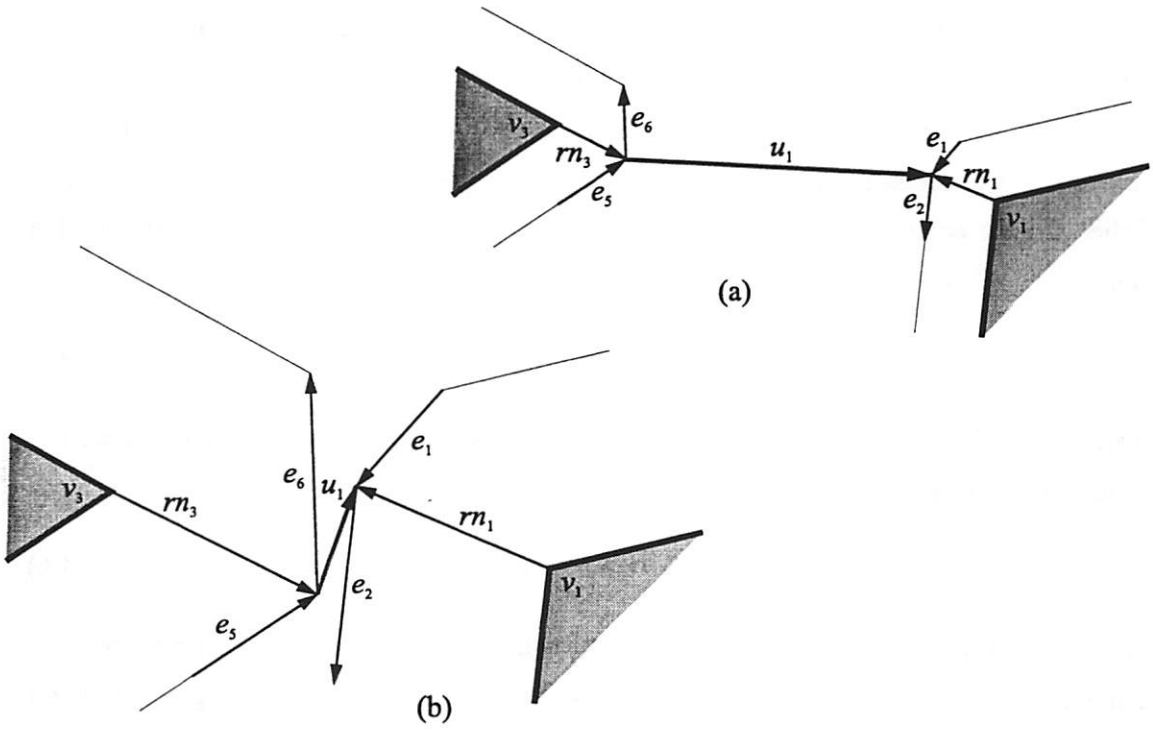



Figure 4.8: Tangency of a Visibility Edge

where

$$\begin{aligned} c_2 &= \delta^T \delta \\ c_1 &= 2d^T \delta \\ c_0 &= d^T d \end{aligned}$$

### 4.2.2 Tangency of an Edge

An edge between two vertices in CSPACE will exist in the reduced visibility graph when it is tangent at both ends and is not occluded by any CSPACE obstacle along its length. For each edge, there are certain values of $r$ at which an end changes from being tangent to being not tangent, or vice versa, as $r$ increases. At other values of $r$ the edge may change from being not occluded to being occluded, or vice versa, as $r$ increases. We term these *tangency events* and *occlusion events*, respectively; *visibility events* collectively.

Figure 4.8 illustrates the tangency of an edge $u_1$ for two different values of $r$. For small $r$ neither end of $u_1$ is tangent to its CSPACE obstacle (a). For larger $r$, both ends

are tangent (b). At even larger values of $r$, the edge will no longer appear in the VGRAPH because the two CSPACE obstacles will overlap.

For the head of $u_1$ in the figure, tangency occurs when $e_1$ and $e_2$ are both on the same side of $u_1$, unless the vertex is concave. The vector $e_1$ is a unit vector in the direction of the chamfer edge, $e_2$ is a unit vector in the direction of the edge of the obstacle. For tangency

$$(e_1 \times u_1)(u_1 \times e_2) > 0 \tag{4.5}$$

where the "$\times$" represents the scalar-valued two-dimensional cross product. Tangency events occur when an edge $e_i$ becomes collinear with the visibility edge $u_1$:

$$r_i = \frac{e_i \times d_i}{e_i \times \delta_i} \tag{4.6}$$

For each edge between CSPACE obstacle vertices there are four such events (two events at each end of the visibility edge).[2] The edge appears in the reduced visibility graph only for those values of $r$ where it is tangent at both ends. In particular, we are only interested in those ranges where $r$ is positive.

If either endpoint of a visibility edge occurs at a concave vertex (e.g. $e_1 \times e_2 < 0$), the edge is never tangent and will never appear in the reduced visibility graph. When one endpoint of a visibility edge corresponds to an application feature (e.g. $q_{\text{init}}$ or $q_{\text{goal}}$), tangency at that endpoint is not an issue. Such an edge will have only two tangency events, and those will determine the tangency of the edge in the PVGRAPH. When both endpoints correspond to application features, then an edge has no tangency events. Its appearance in the reduced visibility graph is governed only by occlusion.

### 4.2.3 Occlusion of an Edge

An edge cannot appear in the reduced visibility graph if it intersects a CSPACE obstacle along its length. For the parametric visibility graph, we need to determine the values of $r$ for which an edge becomes occluded by an obstacle. Figure 4.9 depicts the computation of occlusion events. In the general case, three CSPACE vertices are involved, and form a triangle. Occlusion events occur when this triangle collapses. As $r$ is increased

[2]Note also that in general, because of the chamfer, there are two CSPACE vertices corresponding to one vertex on the physical obstacle. This gives rise to a total of four visibility edges between the two obstacles, each of which has four tangency events.

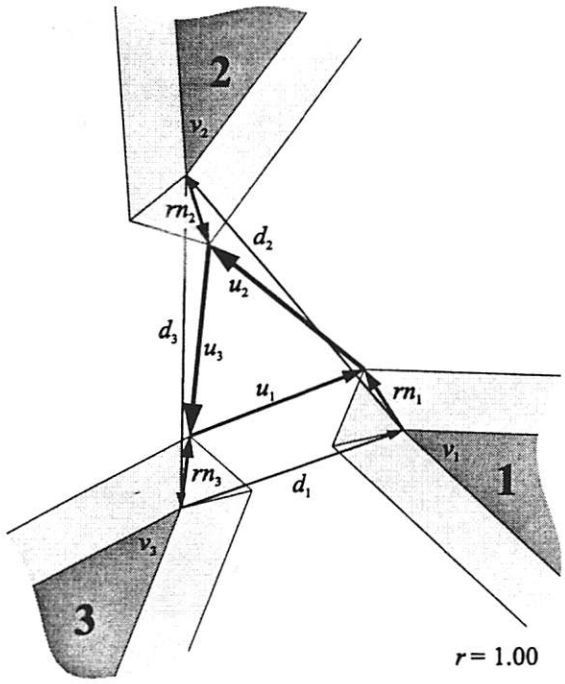



Figure 4.9: Occlusion Triangle for Three Visibility Edges

through the event, one of the edges' state changes from unoccluded to occluded, or vice versa. At the occlusion event, the edges $u_1$, $u_2$, and $u_3$ become collinear, so we can express the event as a cross product:

$$u_1 \times u_2 = 0, \tag{4.7}$$

which can also be written as

$$(d_1 + r\delta_1) \times (d_2 + r\delta_2) = 0 \tag{4.8}$$

where $\delta_1 = n_1 - n_3$ and $\delta_2 = n_2 - n_1$. The values of $r$ for which this happens are the roots of the quadratic

$$c_2 r^2 + c_1 r + c_0 = 0 \tag{4.9}$$

where

$$\begin{aligned} c_2 &= \delta_1 \times \delta_2 \\ c_1 &= d_1 \times \delta_2 + d_2 \times \delta_1 \\ c_0 &= d_1 \times d_2 \end{aligned}$$

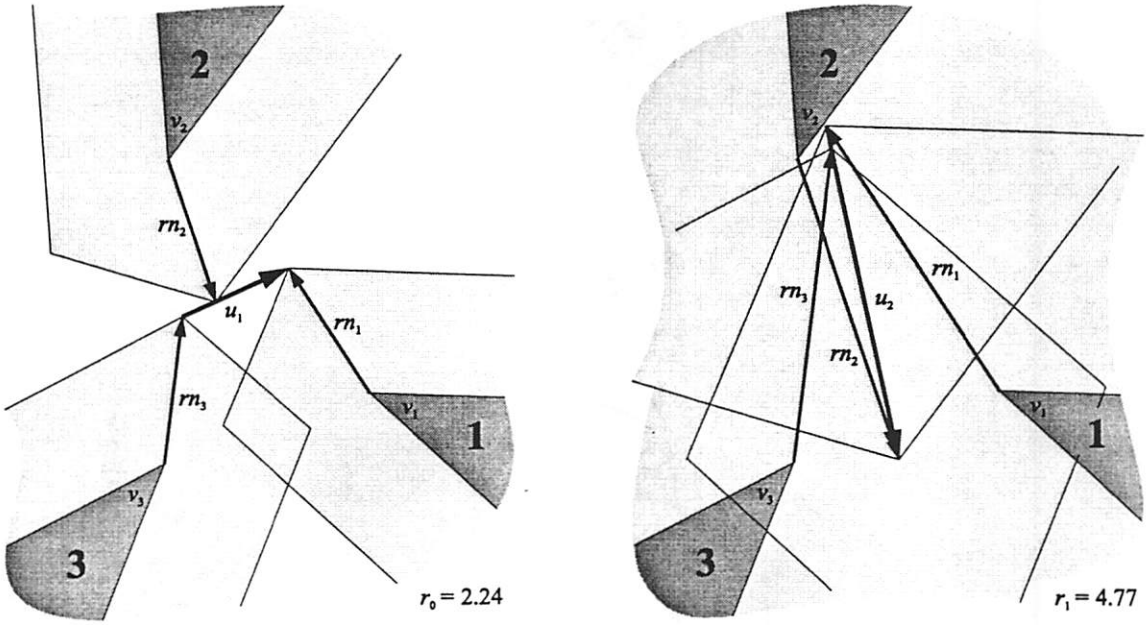



Figure 4.10: Occlusion Events for the Triangle of Figure 4.9.

Thus, for each such triangle there are two occlusion events, $r_0$, and $r_1$. We are only interested in those that are positive and real. Figure 4.10 depicts the two occlusion events for the triangle of Figure 4.9.

When an occlusion event occurs, the three CSPACE vertices are collinear. The one that is in the middle is the one doing the occluding. Which vertex this is can be determined by inner products:

$$u_1^T u_2 > 0 \quad \Rightarrow \quad u_3 \text{ occluded by obstacle 1} \tag{4.10}$$

$$u_2^T u_3 > 0 \quad \Rightarrow \quad u_1 \text{ occluded by obstacle 2} \tag{4.11}$$

$$u_3^T u_1 > 0 \quad \Rightarrow \quad u_2 \text{ occluded by obstacle 3} \tag{4.12}$$

Clearly, only one of the above inequalities can be true when $r = r_0$ or $r = r_1$. This test must be performed for each of these two values to determine which edge is occluded at each event. For a particular event, if the occluded edge belongs to the CSPACE obstacle, then we must also consider this an occlusion event for all the visibility edges incident to the occluding vertex.

It is tempting to assume that as $r$ increases through $r_0$ (or $r_1$), the edge becomes occluded. However, because we use the chamfered representation of CSPACE, there are

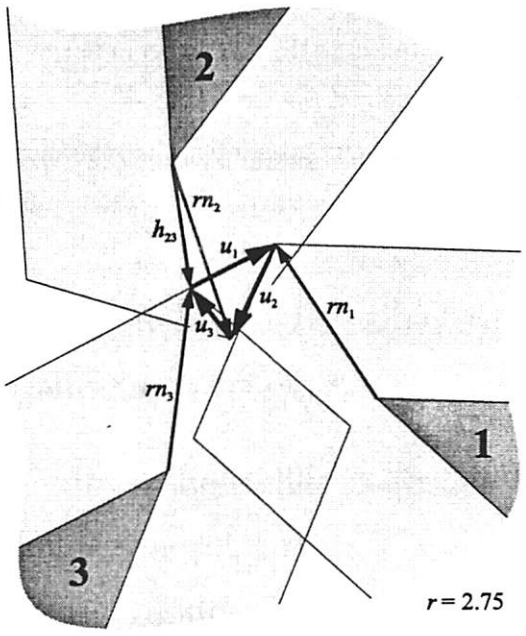



Figure 4.11: Testing the Sign of an Occlusion Event.

cases when an edge can become *unoccluded* as $r$ increases through this value. Thus, for each occlusion event, e.g. $r_0$, we must calculate whether the edge is occluded for $r > r_0$ or for $r < r_0$. To this end we must inspect another cross product.

Figure 4.11 shows the relevant vectors for the $r_0$ case of Figure 4.10. In the figure, $r > r_0$. It is clear that the edge $u_1$ is occluded for such values and not occluded for $r < r_0$. For $r > r_0$, the vector[3] $rn_2$ intersects $u_1$. In general, we will check for intersection of the directed edge, $u_i$, with the vector $rn_j$ to test when the edge $u_i$ is occluded by obstacle $j$. These vectors intersect when the vectors $-u_3$ and $-h_{23}$ are on opposite sides of $u_1$:

$$(u_1 \times u_3)(h_{23} \times u_1) > 0 \Rightarrow u_1 \text{ is occluded for } r > r_0. \tag{4.13}$$

We use this formulation for this example because Equation 4.11 is true. Without loss of generality, analogous formulations for the other two cases can be found by renumbering the vertices.

[3]In the interest of readability, we abuse notation here and refer to the edge from $v_i$ to $v_i + rn_i$ as the *rooted vector* $rn_i$, or simply the *vector* $rn_i$.

We know that at $r_0$ (and $r_1$), the cross product $u_1 \times u_3$ is zero. Thus we must differentiate Equation 4.13 with respect to $r$, and evaluate the derivative at $r_0$ (or $r_1$). If the derivative is positive then we know that $u_1$ is occluded by obstacle 2 for $r > r_0$. If it is negative then $u_1$ is occluded for $r < r_0$.

Since we must account for the possibility that the derivative is zero at $r_0$ (or $r_1$), we must be prepared to evaluate higher derivatives in some cases. The first four derivatives are as follows:

$$\begin{aligned}\frac{d}{dr}[(u_1 \times u_3)(h_{23} \times u_1)] = \quad & (u_1 \times u_3)(h_{23} \times \delta_1 + n_3 \times u_1) \\ + \quad & (\delta_1 \times u_3 + u_1 \times \delta_3)(h_{23} \times u_1)\end{aligned} \tag{4.14}$$

$$\begin{aligned}\frac{d^2}{dr^2}[(u_1 \times u_3)(h_{23} \times u_1)] = \quad & 2(u_1 \times u_3)(n_3 \times \delta_1) \\ & + 2(\delta_1 \times u_3 + u_1 \times \delta_3)(h_{23} \times \delta_1 + n_3 \times u_1) \\ & + 2(\delta_1 \times \delta_3)(h_{23} \times u_1)\end{aligned} \tag{4.15}$$

$$\begin{aligned}\frac{d^3}{dr^3}[(u_1 \times u_3)(h_{23} \times u_1)] = \quad & 6(\delta_1 \times u_3 + u_1 \times \delta_3)(n_3 \times \delta_1) \\ & + 6(\delta_1 \times \delta_3)(h_{23} \times \delta_1 + n_3 \times u_1)\end{aligned} \tag{4.16}$$

$$\frac{d^4}{dr^4}[(u_1 \times u_3)(h_{23} \times u_1)] = 24(\delta_1 \times \delta_3)(n_3 \times \delta_1) \tag{4.17}$$

Higher-order derivatives of this expression are zero. Since we will evaluate these derivatives only at $r_0$ or $r_1$, we can simplify the above by noting that $(u_1 \times u_3) = 0$ whenever $r$ equals either of those two values:

$$\frac{d}{dr}[(u_1 \times u_3)(h_{23} \times u_1)] = \quad (\delta_1 \times u_3 + u_1 \times \delta_3)(h_{23} \times u_1) \tag{4.18}$$

$$\begin{aligned}\frac{d^2}{dr^2}[(u_1 \times u_3)(h_{23} \times u_1)] = \quad & 2(\delta_1 \times u_3 + u_1 \times \delta_3)(h_{23} \times \delta_1 + n_3 \times u_1) \\ & + 2(\delta_1 \times \delta_3)(h_{23} \times u_1)\end{aligned} \tag{4.19}$$

$$\begin{aligned}\frac{d^3}{dr^3}[(u_1 \times u_3)(h_{23} \times u_1)] = \quad & 6(\delta_1 \times u_3 + u_1 \times \delta_3)(n_3 \times \delta_1) \\ & + 6(\delta_1 \times \delta_3)(h_{23} \times \delta_1 + n_3 \times u_1)\end{aligned} \tag{4.20}$$

$$\frac{d^4}{dr^4}[(u_1 \times u_3)(h_{23} \times u_1)] = 24(\delta_1 \times \delta_3)(n_3 \times \delta_1) \tag{4.21}$$

Computation of these derivatives requires knowing derivatives of the terms $u_1 \times u_3$ and $h_{23} \times u_1$, which we note here for reference:

$$\frac{d}{dr}(u_1 \times u_3) = (\delta_1 \times u_3 + u_1 \times \delta_3) \tag{4.22}$$

$$\frac{d^2}{dr^2}(u_1 \times u_3) = 2(\delta_1 \times \delta_3) \tag{4.23}$$

$$\frac{d^3}{dr^3}(u_1 \times u_3) = 0 \tag{4.24}$$

$$\frac{d}{dr}(h_{23} \times u_1) = (h_{23} \times \delta_1 + n_3 \times u_1) \tag{4.25}$$

$$\frac{d^2}{dr^2}(h_{23} \times u_1) = 2(n_3 \times \delta_1) \tag{4.26}$$

$$\frac{d^3}{dr^3}(h_{23} \times u_1) = 0 \tag{4.27}$$

A further point can be made about occlusion events. In the left half of Figure 4.10, we see the occlusion event corresponding to $r_0$. Note that the edge $u_1$ is tangent to all three obstacles at this value of $r$. Since this is an occlusion event, $u_1$, $u_2$, and $u_3$ are collinear and thus all are tangent to all three obstacles. If $u_1$ were not tangent to obstacle 1 or not tangent to obstacle 3, then this would not be an interesting occlusion event for this edge, because we would already know that $u_1$ does not appear in the reduced visibility graph at this value of $r$. If $u_1$ were not tangent to obstacle 2, then, by the definition of tangency, $u_1$ would already intersect one of the edges of obstacle 2, since it presently intersects a vertex of obstacle 2. Again this would not be an interesting occlusion event.

Thus, when enumerating occlusion events for the scene, it is safe (with one exception) to omit visibility edges that are never tangent for $r > 0$. It is also safe to skip characterization of an occlusion event unless all three edges are tangent to all three obstacles at that value of $r$.

The exception to this tangency rule applies to edges of CSPACE obstacles themselves. As noted above, when a CSPACE edge crosses a vertex, all visibility edges incident to that vertex become occluded. This is true even if the CSPACE edge is not tangent to its obstacle at both endpoints (i.e., one or both of those vertices are concave). Thus, we can reduce the computational burden of finding occlusion events by skiping non-tangent visi-

**Algorithm 4.1** PVGRAPH($V_A, O = (V_O, E_O)$)

*Takes a set of application-feature vertices $V_A$, a set of parametric polygonal obstacles $O = (V_O, E_O)$, and computes the Parametric Visibility Graph $P = (V_P, E_P)$.*

1: $V_P \Leftarrow V_A \bigcup V_O$.
2: $E_P \Leftarrow \binom{V_P}{2}$. {*All edges in the complete graph.*}
3: **for each** edge $u$ in $E_P$ **do**
4: $T_u \Leftarrow u$'s tangency as a function of $r$ (Equation 4.6).
5: $R_u \Leftarrow T_u$.
6: **if** $T_u = \emptyset$ AND $u \notin E_O$ **then**
7: $E_P \Leftarrow E_P \setminus u$
8: **end if**
9: **end for**
10: **for each** triangle $\tau = (u_1, u_2, u_3)$ in $\binom{E_P}{3}$ **do**
11: Compute $r_0$ and $r_1$, the occlusion events for $\tau$ (Equation 4.9).
12: **for each** $r_i \in \{r_0, r_1\}$ **do**
13: **if** $r_i \in T_{u_1} \bigcap T_{u_2} \bigcap T_{u_3}$ **then**
14: Characterize this event per Equations 4.10–4.12 and 4.18–4.21.
15: Modify $R_{u_1}, R_{u_2}, R_{u_3}$ as appropriate.
16: **end if**
17: **end for**
18: **end for**
19: **for each** edge $u$ in $E_P$ **do**
20: **if** $R_u = \emptyset$ **then**
21: $E_P \Leftarrow E_P \setminus u$
22: **else**
23: Compute `length`($u$) (Equation 4.4).
24: **end if**
25: **end for**
26: **return** $P \Leftarrow (V_P, E_P)$.

bility edges, but we must consider all edges that belong to the boundaries of the CSPACE obstacles regardless of their tangency.

The computation of the PVGRAPH is shown in Algorithm 4.1. Note that vertices

in the input set $V_O$ are parameterized by $r$. In general, they are equal to $v+rn$, where $v$ is a vertex of the physical obstacle, and $n$ is a vector. The Parametric Visibility Graph is a data structure consisting of vertices and edges. To each edge $u$ is attached two additional pieces of information. The first is the length of $u$ as a function of $r$ (Equation 4.4). The second is a set $R_u \subset \mathbf{R}$ such that $u$ appears in the reduced visibility graph for $r \in R_u$. The set $R_u$ can be represented as a list of real numbers denoting the endpoints of ranges that comprise the set.

In line 4, tangency events are computed for the edge $u$. If both endpoints of $u$ are in $V_O$, four values of $r$ result. If one endpoint is in $V_A$, two values of $r$ result. If both endpoints are in $V_A$ then tangency is irrelevant for $u$ ($T_u$ is assigned all of $\mathbf{R}$). Each of these values of $r$ can be considered to divide the reals into two semi-infinite sets: those values for which $u$ is not tangent, and those for which $u$ *may* be tangent. We seek the intersection of these two or four sets. The result is a range of values in $\mathbf{R}$ for which $u$ is tangent at *both* ends. Because this is an intersection of semi-infinite sets, the range cannot be disjoint. Thus it is representable by two numbers. We assign $T_u$ to be this range. (In some cases the resulting intersection is semi-infinite. We represent this by setting one of the numbers to -1.)

In line 14 we characterize an occlusion event. First, we determine which of $u_1, u_2$, or $u_3$ this event applies to. We then find a semi-infinite set $R$ for which that edge (call it $u$) *may* be unoccluded. That set is intersected with the appropriate $R_u$ and we replace $R_u$ with $R_u \bigcap R$ in line 15.

Finally, in line 23, for those edges having nonempty $R_u$, we compute $c_2, c_1$ and $c_0$, the coefficients of the length function in Equation 4.4. These three numbers are also attached to the edge $u$ so that its length can be calculated for arbitrary values of $r$.

The PVGRAPH can be searched in the same way as a static graph. A value for $r$ is chosen before the search, and remains in effect throughout. During the search, when information about an edge is required, i.e. its existence or length, the data structure is evaluated for the current value of $r$ and the required information is returned.

## 4.3 Conclusion

It is clear that shape synthesis must observe the presence of other parts so as not to create interference. This problem shares a great deal with the problem of path

planning in robotics, so many techniques can be adapted from that discipline. In this research we have chosen to examine visibility graph techniques, and have shown that they are useful in synthesizing noninterfering parts. Other techniques, such as potential fields, and roadmaps, may also be useful.

A significant difference between shape synthesis and robotic path planning is that, while the shape synthesis problem may have fewer degrees of freedom, it has an additional variable representing the width of the path. The Parametric Visibility graph may be useful for finding paths of varying widths through a field of obstacles. It is little more difficult to compute than the static visibility graph, and it holds for all positive values of width.

The PVGRAPH, however, does have an important shortcoming. It can only plan paths that have uniform width along their entire length. In general, it will be useful to find paths with nonuniform with, such as tapered paths. In Chapter 5, we will suggest some ideas for for solving this problem.

# Chapter 5

# Research Directions

This dissertation has presented a vision for a new paradigm for mechanical design. It has described in broad strokes the various elements required to realize this vision. Because of this breadth, only a few topics have been treated in detail. This chapter presents a summary of topics that need further study.

## 5.1 Skeleton Generation in the Presence of Obstacles

Of primary importance to our efforts at shape synthesis is the ability of algorithms to construct skeletons in the presence of obstacles. None of the algorithms presented in Chapter 2 address this, so it will be important to determine which methods can be augmented to handle obstacles and how it should be done.

One approach that works with any skeleton that doesn't contain Steiner nodes is to handle obstacles after the fact. This is the only approach that we have implemented in the `racerx` software. The technique is to first ignore obstacles and construct a graph using one of the above (non Steiner) methods like EMST, Spanner, or Delaunay. Following that, the configuration space (CSPACE) of the scene and its visibility graph are computed. Then, for each edge in the graph, we find the shortest path through the visibility graph and add that path to the skeleton.

We have observed (and it is easy to show) that this technique is far from ideal. It sometimes produces absurd looking parts, because the initial graph specifies connectivities that require the path planner to find long, circuitous paths. It is clear that it will be desirable to incorporate obstacle avoidance into the graph construction algorithms themselves

if we are to generate sensible part designs.

### 5.1.1 Graph Generation Algorithms

Of the algorithms studied in Chapter 2, we believe that some can be modified to operate in the presence of obstacles. These include the EMST, the Greedy $t$-Spanner, and the Delaunay Triangulation.

**The Euclidean Minimum Spanning Tree.** It may be possible to redefine the Euclidean Minimum Spanning Tree to account for the presence of obstacles. The EMST can be thought of as a specialization of the Minimum Spanning Tree, which in general is a subgraph $G' = (V, E')$ of some graph $G = (V, E)$. $E'$ is chosen so that $V$ is spanned and the weight of $G'$ is minimal. We can say that the EMST is the MST of the Euclidean-embedded complete graph.

When obstacles are introduced, many edges of the complete graph become occluded. The relevant graph in this case is the visibility graph of the scene. If we compute a Minimum Spanning Tree of the visibility graph, we will have a spanning tree that avoids the obstacles. This tree also spans every vertex of every obstacle in the scene, so it is really more than we want.

What we want is a tree that spans *some* of the vertices of the visibility graph (the set $V$ to be precise) and optionally any of the vertices of the obstacles. We would additionally like the tree to have minimum weight. This is know as the *Graphical Steiner Minimum Tree Problem*(GSMT) [?], and, though it really has little in common with the Euclidean Steiner Minimum Tree Problem, it is also known to be NP-complete. Hence, we must seek solutions that are approximate. Several heuristic algorithms are presented in the literature [?, ?, ?, ?, ?].

**$t$-Spanners.** To augment $t$-spanner algorithms to handle obstacles, we will follow the same reasoning as we do with the EMST. Call the set of points $V$ together with the set of obstacles $O$ the *scene*. Define the *complete visible graph* to be the union of all $n(n-1)$ pairwise shortest paths through the visibility graph of the scene. It seems reasonably straightforward to extend the Greedy algorithm to solve this problem: define a graph $G' = (V, E')$ where there is an edge in $E'$ for each pair of points in $\binom{V}{2}$ and its weight is

the length of the corresponding shortest path in the complete visible graph. Given $G'$ as input, the Greedy algorithm will produce the desired spanner.

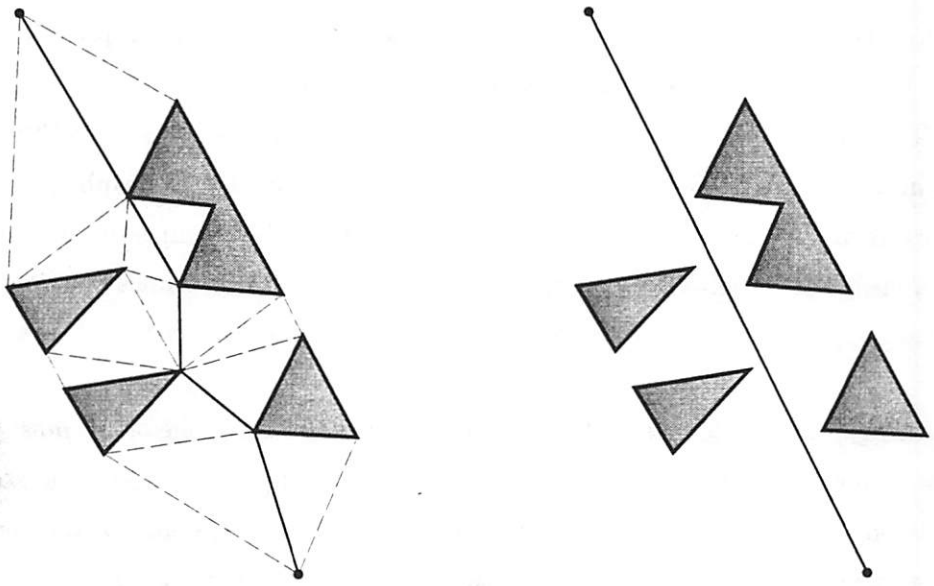
It is not clear to us if either Salowe's or Ruppert and Seidel's algorithms can be extended in a similar way, because those algorithms do not construct graph spanners but rather Euclidean spanners. It may be possible to redefine the distance metric to be the length of a visible path between two points, or, alternatively, the distance metric might be redefined to be infinite whenever the two points can not see each other.

**The Delaunay Triangulation.** The Delaunay Triangulation is one of the most promising skeletons for our purposes because it it efficiently computes sparse, stiff skeletons. These skeletons may be used as-is, or may be used as the starting point for further refinement using one of the optimization techniques discussed in Chapter 3. In many ways the DT is an approximation of the complete graph [?], yet its complexity is $O(n)$ rather than $O(n^2)$. We believe that we can invent a definition that describes a triangulation that is similar in spirit to the Delaunay Triangulation, but that avoids obstacles.

Our first hope was that the *Constrained Delaunay Triangulation* (CDT) would provide this. The CDT modifies the DT problem by adding edges to the problem specification. The constructed triangulation is required to utilize these edges. The remainder of the constructed triangulation must obey the circle criterion.

At first glance it appears that we can set up a Constrained Delaunay Triangulation problem by defining the constraint set to be all of the edges of all the obstacles. The CDT will essentially triangulate the insides of the obstacles independently from the surrounding free space. We simply discard those portions of the result that lie inside obstacles. The shortcoming of this approach is much like the one we encounter with the Visibility Graph/Spanning Tree approach discussed above: the result contains all the edges and vertices of all the obstacles. This is clearly more than we want. In both cases, we can imagine pruning this result by throwing away as many edges as possible while ensuring that $V$ is still spanned.

It is easy, however, to construct an example that shows this to be non-ideal. Figure 5.1(a) shows such a result. It is obvious that the edge we would want is simply the straight line connecting the two nodes (Figure 5.1(b)). This edge is not a member of the CDT, however. What we get instead is a zigzag line that spans several obstacle vertices along the way. This may, however, be suitable as a starting point for an optimization-



(a) (b)

Figure 5.1: Computing a Delaunay Triangulation in the Presence of Obstacles

based strategy. It is easy to imagine "tightening" the result in (a) so the the zigzag path straightens out. This would be a natural result of applying a continuous optimization. We will call the result the *Delaunay Triangulation with Obstacles* (DTO).

We also note that the standard way of efficiently computing the Euclidean Minimum Spanning Tree is to first compute the Delaunay Triangulation and then find the MST of that. (It is a fact that the EMST is a subgraph of the DT.) If we can find a suitable way of generating the DTO, then we merely need to find its MST to have an "EMSTO."

### 5.1.2 Adding a dimension to CSPACE for the width parameter

In Chapter 4 we noted that the Parametric Visibility Graph is of no use for planning paths whose width is nonuniform. If, for example, tapered spars are desired, we must find another method. One possible approach is to add a dimension to Configuration Space to represent the width of the path. For a 2-D design problem, CSPACE becomes 3-D.

Figure 5.2 illustrates such a configuration space for the path planning problem of Figure 4.4. The obstacles in this generalized space are frustums, whose tops are the

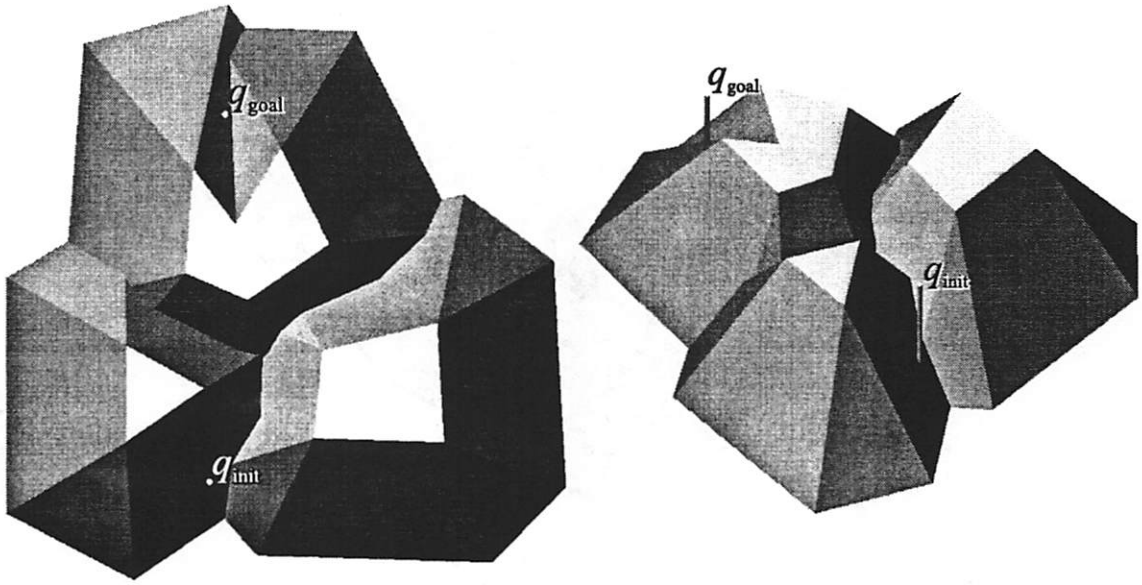



Figure 5.2: Configuration Space With a Third Dimension Parameterized by Path Width.

polygonal physical-space obstacles. The "vertical" dimension represents the width parameter, $r$. Higher elevations correspond to smaller values of $r$. Lower elevations correspond to larger values of $r$. Accordingly, the slope of the obstacle sides is unity.

Path planning in this CSPACE is analogous to designing a road that winds through a mountain range. We want to find the shortest road that will connect two cities, each of which is at a different elevation. We are not allowed to cut into the mountain, but we are allowed to fly through open space. Thus, some sections of the road lie along the sides of the mountain and other sections jump across valleys as though supported by a bridge.

**Is there such a thing as a vgraph?** We can imagine a variety of techniques for solving the spar synthesis problem, depending on how it is posed. If the path planner is given a value of $r$ to be used at $q_{\text{init}}$ and another value to be used at $q_{\text{goal}}$, then we can assign these values as the third coordinate in CSPACE, giving $q_{\text{init}}$ and $q_{\text{goal}}$ concrete 3-D locations. A crude way of finding a path would be to define a plane that intersects these two points in CSPACE, and is otherwise "as horizontal as it can be." In other words, the angle that this plane makes with the $r = 0$ plane is the same as the angle between the line joining $q_{\text{init}}$ to $q_{\text{goal}}$ and the $r = 0$ plane. If we then intersect this plane with the 3-D CSPACE obstacles,

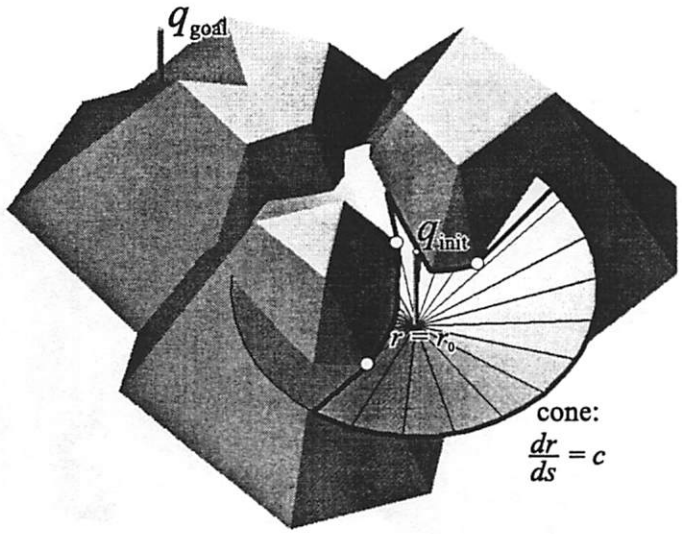



Figure 5.3: Configuration Space Cone Representing Constant Taper.

we get a 2-D configuration space, in which we can use visibility graph techniques.

The problem with this approach is that any resulting path will not have constant taper. In fact, if any segments of the path are perpendicular to the line from $q_{\text{init}}$ to $q_{\text{goal}}$, they will have constant width. The closer a segment is to being parallel to the line from $q_{\text{init}}$ to $q_{\text{goal}}$, the greater its taper.

**The Constant Slope (Initial Value) Problem.** In other cases the path planner may be given a value of $r$ for $q_{\text{init}}$ and a desired value of $\frac{dr}{ds}$, where $s$ is distance along the path. The problem is to find a path from $q_{\text{init}}$ to $q_{\text{goal}}$ that has the constant taper dictated by $\frac{dr}{ds}$. Clearly, a given problem may have no solution if $\frac{dr}{ds}$ is too large. The length of the path may cause $r$ to shrink to zero or to grow too large before $q_{\text{goal}}$ is reached.

Assuming that a solution does exist, we can represent the set of straight path segments emanating from $q_{\text{init}}$ by the surface of a cone, as in Figure 5.3. The tip of the cone is at the 3-D location of $q_{\text{init}}$. If the large end of the path is to be at $q_{\text{init}}$, then the cone opens upward. The cone is intersected with the 3-D CSPACE obstacles and trimmed to the innermost boundary (heavy outline) defined by this intersection. Trimming forces the cone to represent only those line segments that do not interfere with any obstacles. Vertices are defined wherever the trimmed cone intersects an obstacle edge and the line from $q_{\text{init}}$ is tangent (white circles). The first segment of the path will have one of these

vertices as its endpoint, thus we have a finite number of possible first segments for the solution.

Each vertex is labeled with its (2-D) distance from $q_{\text{init}}$. The process is repeated by defining a similar cone at each of the new vertices, which is then intersected with the CSPACE obstacles to yield an additional set of vertices. These vertices are labeled with their distances from $q_{\text{init}}$. The algorithm is a modification of Dijkstra's algorithm for finding the shortest path through a graph. In this case, we construct the graph as we go along. At each step the algorithm focuses its attention on the vertices whose cost is the lowest found thus far. Eventually, cones would intersect $q_{\text{goal}}$, yielding paths and corresponding total lengths. As each new solution is found, the label on $q_{\text{goal}}$ is updated to reflect the shortest solution found thus far. All possible paths must pursued until they either reach $q_{\text{goal}}$ or their length exceeds the current shortest solution.

The act of "relaxing" a vertex's cost must be modified. When Dijkstra's algorithm is applied to a 2-D graph, a vertex of the graph is, in general, reachable by multiple paths. As the algorithm traverses the graph, it labels each vertex with the length of the path to that vertex. If a vertex is visited via more than one path, it's length is always updated to reflect the length of the shortest such path. This is called *relaxing* the vertex's cost.

For the algorithm we are proposing here, vertices are 3-D points that lie on the downward-sloping ridges of the 3-D CSPACE obstacles. When two different cones intersect the same ridge, they will, in general, do so at two distinct elevations. Thus, in general, a given 3-D vertex is reachable usually by only one path. The elevation of each is proportional to its cost because all path segments have the same slope ($\frac{dr}{ds}$). Dijkstra's algorithm must be modified as follows: Call two vertices *compatible* if they lie on the same ridge. A set of compatible vertices is relaxed by discarding all but the lowest cost vertex. Only that vertex (and its corresponding path) will be considered as a possible part of the solution during the remaining processing of the algorithm.

**Specified Begin And End Widths (Boundary Value) Problem.** Finally, if the problem is posed as in the first example, above, but with the additional stipulation that the taper must be constant over the length of the path, we have a boundary-value problem. The path planner is given a value of $r$ to be used at $q_{\text{init}}$, another value to be used at $q_{\text{goal}}$, and $\frac{dr}{ds}$ must be constant. This is the problem that most closely mirrors the problem of planning a road through the mountains.

We can imagine using shooting techniques, wherein a candidate value of $\frac{dr}{ds}$ is chosen, then the solution to the initial-value problem is found with the above technique. With iteration, a solution to the boundary-value problem may evolve. This will, in general, have convergence problems, because different values of $\frac{dr}{ds}$ may cause topologically different routes between the obstacles to be taken. Even when convergence is obtained, efficiency may be poor. It would be interesting to discover an exact (non-iterative) solution to this problem.

## 5.2 Material Synthesis

For this dissertation, little study of material synthesis techniques has been conducted. In the two software implementations, `graphlab` and `racerx`, only two techniques have been coded, uniform-width spars and tapered spars, so there are plenty of methods that remain to be explored. Some simple ideas, such as the bounding box and the convex hull, were mentioned in Chapter 1. It is likely that numerous approaches exist for computing tapered spars and other nonuniform-width paths. Finally, perhaps the most important topic we have left unexamined is the incorporation of manufacturing process knowledge in the material synthesis step.

### 5.2.1 The Simple Techniques

**Modified Rectilinear Bounding Box.** The modified rectilinear bounding box, as shown in Figure 1.3(b), is actually rather interesting. There are two issues to be addressed: what are the principal axes, and what is to be done about obstacles? Given an orientation, it is easy to find the bounding box of the application features. Then we must subtract material to make room for the obstacles. In the spirit of rectilinearity, the material subtracted should be the union of rectangles wherever possible.

To choose an orientation for the axial directions, we can imagine several criteria that may be useful. A simple one would be to minimize the area of the resulting part. Another would be to minimize the number of edges so that the part can be made with few cuts. This corresponds to finding an angle such that the obstacles can be covered by a small number of rectangles without interfering with the thick skeleton.

To do this well may require tighter integration between the skeleton and material synthesis processes. For example, the visibility-graph planner will often place the thick

skeleton so that it hugs obstacles. This will likely preclude cutting a large rectangular notch to accommodate an obstacle. As has been suggested, iteration is one way to address this problem. A circumscribing rectangle could be substituted for the obstacle, and the skeleton synthesizer run again. If a solution is found, then the material synthesizer can use the bounding box of the new thick skeleton, and can accommodate the obstacle by subtracting the rectangle. In general, however, we are not guaranteed of finding a solution this way. Some clever ideas will be needed to solve this problem in an optimal way.

**Modified Convex Hull** The modified convex hull, as shown in Figure 1.3(c), is another potentially useful design style. Really the only issue here is what to do about obstacles. A simple solution is to find the convex hull of the thick skeleton and subtract the obstacles, but this will, in general, leave excess material.

A better solution is illustrated by envisioning the convex hull as the boundary defined by wrapping a rubber band around the the thick skeleton. We then let the obstacles push in the rubber band so that the boundary does not interfere with them. Such a solution is probably a straightforward modification of existing algorithms for computing the convex hull of a 2-D set.

### 5.2.2 Manufacturing in Two Dimensions

We have repeatedly opined that perhaps the most important topic relating to material synthesis is the incorporation of manufacturing process knowledge. In two dimensions the machining problem is fairly easy because there are few limitations to the shape of a plate that can be cut on a three-axis milling machine. Three manufacturing constraints that can be applied in 2-D and have some real-world relevance are fillets, roundness and rectilinearity.

**Make Room for Fillets** An interesting problem to be solved is how to synthesize skeletons that ensure sufficient clearance for fillets to be added without creating interferences. Obviously, we can add clearance equal to the tool radius when we compute the configuration space, but this is conservative. In many cases this clearance may be unacceptably large.

The challenge of finding solutions without specifying a large clearance is illustrated by a simple example. The problem is that we are given a rectangle of dimensions

$l \times w$, and we are asked to find a rounded-corner rectangle that just fits around it. The radius for the rounded corners is $r$. There are infinitely many solutions, ranging from $l \times (w + 2r)$ to $(l + 2r) \times w$. An interesting question is whether CSPACE techniques can be adapted to include a representation of this continuum of fillet solutions at each obstacle vertex.

**Maximize Roundness** The roundness of a 2-D shape can be defined to be the ratio of its area to the square of its perimeter. For a 2-D part, this estimates the complexity of machining as a dimensionless number. A lower value of roundness indicates that the part has a large perimeter. Since the perimeter of the part is defined by cutting, this translates into higher cost to manufacture the part. For convenience, we can add a coefficient of $4\pi$, i.e.:

$$R = 4\pi \frac{A}{P^2}. \tag{5.1}$$

where $A$ is the area of the shape, and $P$ is its perimeter. Thus a circle has a roundness value of 1. If we use an optimization framework to synthesize material for the part, we can thus use $R$ as a term in the objective function to favor designs that are cheap to manufacture.

**Maximize Rectilinearity** Rectilinearity expresses the quality that many or all of the part's edges are straight and axis-aligned. This improves handling, assembly and inspection of the part. For example, straight, parallel surfaces can be clamped in a vice to hold the part for further machining.

The rectilinear bounding box is one possible design, but there are infinitely many more. Figure 1.3(h) illustrates a rectilinear solution. It is likely that all methods incorporated by the the shape synthesizer will have to be modified. Completely new skeleton synthesis, and obstacle avoidance techniques may be required.

### 5.2.3 Manufacturing in Three Dimensions

The Holy Grail of this research is to be able to design three dimensional parts that are, among other things, cheap to manufacture. In this dissertation, we have repeatedly stressed that an advantage of assembly-centric design is that the software can apply manufacturing process knowledge during shape synthesis. For this dissertation, we have

not even begun to investigate how this can be done, but it seems clear that the material synthesizer will have to reason in terms of the kinds of operations used in the selected manufacturing process. If a subtractive process such as machining is to be used, the material synthesizer must reason in terms of subtractive geometric boolean operations. This would be essentially automation of the technique known as Destructive Solid Geometry (DSG) [?].

In the DSG method, a designer interactively creates a part design using specialized software. The modeling operations he uses directly correspond to manufacturing operations appropriate to the chosen process. At each step, he is limited to only those operations that could physically be carried out by a real machine. If we can program an automatic material synthesizer to obey these same rules, we will be guaranteed of yielding part designs that are manufacturable.

## 5.3 Conclusion

This dissertation has presented a vision for what we believe will be genuinely useful new paradigm for mechanical design. It has described in broad strokes the various elements required to realize this vision. Some of this research has been devoted to applying existing techniques to this new domain. These include graph algorithms such as the Steiner Minimum Tree, $t$-Spanners, Delaunay Triangulation, and Conjugate-Gradient optimization. A few new techniques have been presented, including Hierarchical Nearest Neighbor Trees, the Switch-Topology optimizer, chamfered CSPACE, and the Parametric Visibility Graph.

In Chapter 1 we show that shape synthesis of a single part can proceed in a static environment. We conjecture that the shape synthesis problem is not difficult for simple parts. In Chapter 2 we show several simple algorithms that generate reasonable skeletons. Some of these benefit from applying numerical optimization, and an efficient method for doing this is presented in Chapter 3. It is also feasible to guarantee that synthesized part shapes do not interfere with other parts in the assembly. Chapter 4 describes the approach we use in the `racerx` software, and presents a data structure that may be useful when planning for paths of varying widths.

Many fascinating topics remain to be studied. A few have been mentioned in this chapter, including skeleton generation in the presence of obstacles, planning for

nonuniform-width paths, and material synthesis.

# Bibliography

[1] ALTHÖFER, I., DAS, G., DOBKIN, D., AND JOSEPH, D. Generating sparse spanners for weighted graphs. In *Proceedings of the 2nd Scandinavian Workshop on Algorithm Theory* (July 1990), Springer-Verlag, pp. 26–37.

[2] ALTHÖFER, I., DAS, G., DOBKIN, D., JOSEPH, D., AND SOARES, J. On sparse spanners of weighted graphs. *Discrete & Computational Geometry 9*, 1 (1993), 81–100.

[3] ARBAB, F. *Requirements and Architecture of CAM oriented CAD systems for Design and Manufacture of Mechanical Parts.* PhD thesis, University of California, Los Angeles, 1982.

[4] BHARATH-KUMAR, K., AND JAFFE, J. M. Routing to multiple destinations in computer networks. *IEEE Transactions on Communications COM-31* (1983), 343–351.

[5] CHANDRA, B., DAS, G., NARASIMHAN, G., AND SOARES, J. New sparseness results on graph spanners. In *Proceedings of the Eighth Annual Symposium on Computational Geometry* (June 1992), ACM, pp. 192–201.

[6] CHEW, L. P. There are planar graphs almost as good as the complete graph. *Journal of Computer and System Sciences 39*, 2 (1989), 205–219.

[7] CORMEN, T. H., LEISERSON, C. E., AND RIVEST, R. L. *Introduction to Algorithms.* MIT Press, Cambridge, Mass., 1989.

[8] CRAIG, J. J. *Introduction to Robotics*, second ed. Addison-Wesley, Reading, MA, 1989.

[9] CUNNINGHAM, J. J., AND DIXON, J. R. Designing with features: The origin of features. In *Intl. Computers In Engineering Conference* (Aug. 1988), pp. 237–243.

[10] CUTKOSKY, M. R., BROWN, D. R., AND TENENBAUM, J. M. Working with partially specified designs in concurrent product and process design. Tech. Rep. 19891214, Center for Design Research, Stanford University, Stanford, CA, 1989.

[11] DAS, G., AND HEFFERNAN, P. Constructing degree-3 spanners with other sparseness properties. In *Proceedings of the 4th International Symposium on Algorithms and Computation* (December 1993), Springer-Verlag, pp. 11–20.

[12] DUFFEY, M. R., AND DIXON, J. R. Automating extrusion design: A case study in geometric and topological reasoning for mechanical design. *Computer Aided Design 20*, 10 (Dec. 1988), 589–596.

[13] EDELSBRUNNER, H., AND SHAH, N. R. Incremental topological flipping works for regular triangulations. Tech. Rep. UIUCDCS-R-92-1726, Department of Computer Science, University of Illinois at Urbana-Champaign, 1992.

[14] EL-ARBI, C. Une heuristique pour le probleme de l'arbre de steiner. *RAIRO(Operations Res.) 12* (1978), 207–212.

[15] GRAHAM, P. V., AND ULRICH, K. T. Structural synthesis of sheet metal parts: An analogy to path planning using manufacturability as a guide. In *Advances in Design Automation* (Sept. 1989), American Society of Mechanical Engineers, pp. 289–294, Vol. 1.

[16] HESSER, J., MANNER, R., AND STUCKY, O. Optimizations of steiner trees using genetic algorithms. In *Proceedings of the Third International Conference on Genetic Algorithms* (1989), pp. 231–236.

[17] HWANG, F., AND RICHARDS, D. Steiner tree problems. *Networks 22*, 1 (Jan. 1992), 55–89.

[18] IWAINSKY, A., CANUTO, E., TARASZOW, O., AND VILLA, A. Network decomposition for the optimization of connection structures. *Networks 16* (1986), 205–235.

[19] KEIL, J. M., AND GUTWIN, C. A. Classes of graphs which approximate the complete euclidean graph. *Discrete & Computational Geometry 7*, 1 (1992), 13–28.

[20] KORTSARZ, G., AND PELEG, D. Generating sparse 2-spanners. In *Proceedings of the 3rd Scandinavian Workshop on Algorithm Theory* (July 1992), Springer-Verlag, pp. 73–82.

[21] KOU, L., MARKOWSKY, G., AND BERMAN, L. A fast algorithm for steiner trees. *Acta Info. 15* (1981), 141–145.

[22] LATOMBE, J.-C. *Robot Motion Planning.* Kluwer Academic Publishers, Norwell, MA, 1991.

[23] LEVCOPOULOS, C., AND LINGAS, A. There are planar graphs almost as good as complete graphs and as cheap as minimum spanning trees. *Algorithmica 8*, 3 (1992), 251–256.

[24] LING, Z.-K., AND CHASE, T. R. A technique for the design of an interference free complex planar mechanism. In *Advances in Design Automation* (1991), American Society of Mechanical Engineers, pp. 433–441.

[25] LOZANO-PÉREZ, T., AND WESLEY, M. A. An algorithm for planning collision-free paths among polyhedral obstacles. *Communications of the ACM 22*, 10 (1979), 560–570.

[26] LUNDY, M. Applications of the annealing algorithm to problems in statics. *Biometrica 72* (1985), 191–198.

[27] LUNDY, M., AND MEES, A. Convergence of an annealing algorithm. *Math. Prog. 34* (1986), 111–124.

[28] MACULAN, N. The steiner problem in graphs. *Annals of Discrete Mathematics 31* (1987), 185–222.

[29] MEHRA, R. K., AND SEEREERAM, S. Flight control system design optimization using adaptive partitioned random search and genetic algorithms. Tech. rep., Scientific Systems Company, Inc., 500 West Cummings Park, Suite 3950, Woburn MA 01801, Nov. 1995.

[30] OVERMARS, M. The forms library: a package for building graphical interfaces on silicon graphics computers. Public Domain Software Library, ftp://sol.cs.ruu.nl/pub/SGI/FORMS/forms2.3.tar.Z, 1995.

[31] PARAMETRIC TECHNOLOGY CORPORATION. *Pro/ENGINEER Modeling User's Guide.* 128 Technology Drive, Waltham, MA 02154, 1993.

[32] PELEG, D., AND SHÄFFER, A. Graph spanners. *Journal of Graph Theory 13*, 1 (1989), 99–116.

[33] PLESNIK, J. A bound for the steiner tree problem in graphs. *Math. Slovaca 31* (1981), 155–163.

[34] PREPARATA, F., AND SHAMOS, M. *Computational Geometry*, second ed. Springer Verlag, 1989.

[35] PRESS, W. H., FLANNERY, B. P., TEUKOLSKY, S. A., AND VETTERLING, W. T. *Numerical Recipes in C: The Art of Scientific Computing*, $1^{st}$ ed. Cambridge University Press, New York, NY, 1988.

[36] RUPPERT, D., AND SEIDEL, R. Approximating the d-dimensional complete euclidean graph. In *Proceedings of the Canadian Conference on Computational Geometry* (1991), pp. 207–210.

[37] SALOWE, J. S. Constructing multidimensional spanner graphs. *International Journal of Computational Geometry & Applications 1*, 2 (June 1991), 99–107.

[38] SALOWE, J. S. On euclidean spanner graphs with small degree. In *Proceedings of the Eighth Annual Symposium on Computational Geometry* (1992), ACM, pp. 186–191.

[39] SHAH, J. J. Assessment of features technology. *Computer Aided Design 23*, 5 (June 1991), 331–343.

[40] SHAH, J. J., SEN, S., AND GHOSH, S. An intelligent cad environment for routine mechanical design. In *Intl. Computers In Engineering Conference* (1991), American Society of Mechanical Engineers, pp. 111–117.

[41] SHAPIRO, V., AND VOELKER, H. On the role of geometry in mechanical design. *Research in Engineering Design 1*, 1 (1989), 69–73.

[42] SHIMADA, K., AND GOSSARD, D. Automated shape generation of components in mechanical assemblies. In *Advances in Design Automation* (1992), American Society of Mechanical Engineers, pp. 51–58, Vol. 1.

[43] SMITHERS, T. AI-based design versus geometry based design, or why design cannot be supported by geometry alone. *Computer Aided Design 21*, 3 (April 1989), 141–150.

[44] VAIDYA, P. An $O(n \log n)$ algorithm for the all-nearest-neighbors problem. *Discrete & Computational Geometry 4*, 2 (1989), 101–115.

[45] WILLIAMS, J. D. STICKS—a new approach to LSI design. Master's thesis, Massachusetts Institute of Technology, 1977.

# Appendix A

# Graphlab

The `graphlab` program is invoked by typing `lab` at the UNIX command prompt. There are no options or arguments to be typed on the command line. All options and actions are invoked via the graphical user interface when the program is running. When started, the program presents a large main window in which part skeletons can be built and tested. User interaction is provided via menus, dialog boxes, and the mouse.

## A.1 Keyboard and Mouse

Keyboard and mouse buttons are given user-defined functions by commands listed in an initialization file (see Section A.4). There are no default assignments—**without an initialization file, the keyboard and mouse do nothing.** The commands listed in the following sections of this appendix are all accesible via menus or dialog-box buttons. Assigning keyboard keys to them provides shortcuts, but is not required to be able to invoke the actions. Each command is listed with its unique identifier in parentheses, which is used in the initialization file when assigning the command to a key sequence.

There is a special set of actions that must be assigned to keys: the ones intended to be associated with the mouse. If these actions are not bound to keys by the initialization file, they will not be available. Though there is no restriction on the keys to which they can be assigned, the user will likely want to bind these actions to the three mouse buttons. Since there are more than three actions, some should be bound with modifier keys, e.g. `(bind-key "M-rmouse" 'window-pan)`. See Section A.4 for a suggested minimal initialization file.

The mouse commands are as follows:

- **Delete Edge** (`edge-delete`): *Interactively delete an edge from the Edit graph.*
- **Delete Point** (`point-delete`): *Interactively delete a point.*
- **Drag Point** (`point-drag`): *Interactively drag a point.*
- **Make Edge** (`edge-make-steiner`): *Interactively make a new edge in the Edit graph between two points.* First click on a point. It will be highlighted. Clicking on a second point adds an edge between those two points. Clicking on free space or on an existing edge creates a new Steiner vertex.
- **Make Edge** (`edge-make-edit`): *Interactively make a new edge in the Edit graph between two points.* First click on a point. It will be highlighted. Clicking on a second point adds an edge between those two points. Clicking on free space or on an existing edge creates a new Edit vertex.
- **Make Steiner Point** (`point-make-steiner`): *Interactively make a new steiner point in the Edit graph.*
- **Make Point** (`point-make`): *Interactively make a new point.*
- **Pan Window** (`window-pan`): *Pan the view in the current window.*
- **Rotate Window** (`window-rotate`): *Rotate the view in the current window.*
- **Zoom Window** (`window-zoom`): *Zoom the view in the current window.*

## A.2 Menus

The Main Window of the `graphlab` program conains a menu bar along its top edge. On that there are five menus, labeled *File*, *Window*, *Debug*, *Tools*, and *Help*. These menus contain the following commands:

### A.2.1 File Menu

- **Quit** (`quit-program`): *Quit the program.*

- **Dump Key Bindings** (`dump-key-bindings`): *Dump the current key bindings in a format (almost) suitable as an initialization file.* This file is called 'keybindings' and lists all program actions, including those that are not bound to keys (see Section A.4). The unbound actions are listed as

  `(bind-key "" 'command)`

  With no key sequence specified between the two double-quotes, this line will cause the `graphlab` program to complain. Therefore, the dumped file is not quite usable as an initialization file. To bind such a command to a key sequence, copy the line from the 'keybindings' file and add it to your actual initialization file. Add the desired key sequence between the two double-quotes.

- **Dump Help** (`dump-help`): *Dump the help blurbs for all user interface items.* This includes menu items, buttons, mouse clicks, etc. Output is dumped to two files: 'Help.txt,' and 'Help.tex.' The latter is (almost) suitable for inclusion in LaTeX documents (such as this one).

- **Save As...** (`graph-file-save-as`): *Save the Edit Graph under a new filename.*

- **Open...** (`graph-file-open`): *Open a graph file.*

- **Export IGES...** (`graph-export-iges`): *Export the Edit Graph as an IGES file.*

### A.2.2 Window Menu

- **Lights...** (`show-lights`): *Adjust the lights.* This command displays a dialog that allows you to adjust the color and position of the lights. See Section A.3.1.

- **Window Controls...** (`show-window-controls`): *Set window options.* This command displays a dialog that allows you to adjust various aspects of the current window. See Section A.3.2.

- **Zoom All** (`window-zoom-all`): *Zoom the scene to fill the current window.*

- **Screen Dump...** (`window-screendump`): *Save a copy of the screen to a .rgb file.* The image is saved using 'scrsave(6D)'. The coordinates of the corners of the viewport are passed as the region to save. An exact copy of the screen within that region is

saved. Thus, for a clean screen dump, make sure that no other windows on the screen overlap the program window.

- **Export Idraw...** (`graph-export-idraw`): *Export the Edit Graph as an Idraw (PostScript) file.*

### A.2.3 Debug Menu

- **Lists Debug Submenu**
  - **Before** (`debug-lists-before`): *Toggle consistency checks on lists.* When enabled, checks are made at the entry of each operation.
  - **After** (`debug-lists-after`): *Toggle consistency checks on lists.* When enabled, checks are made at the exit of each operation.
  - **Data** (`debug-lists-data`): *Toggle consistency checks on lists.* When enabled, checks are made for null data pointers.
- **Mem Manager Debug Submenu**
  - **Alloc** (`debug-mem-alloc`): *Toggle memory manager debugging output.* When enabled, calls to allocate and free are reported.
  - **Blocks** (`debug-mem-blocks`): *Toggle memory manager debugging output.* When enabled, construction of MemBlocks objects are reported.
  - **Header** (`debug-mem-header`): *Toggle memory manager debugging output.* When enabled, calls to HeaderBlock new and delete are reported.
- **Window Debug Submenu**
  - **Clear: Print** (`debug-winclear-print`): *Toggle printing of debugging information during window clear.* When enabled, the clear operation prints the scene and clear rectangles, as well as the vertices of the clear rectangle.
  - **Clear: Inset 1** (`debug-winclear-inset1`): *Toggle insetting of the 'clear' rectangle of all windows.* When enabled, the cleared rectangle of the window is inset by one pixel on all four sides, leaving the outer pixels unerased during each window refresh. (Does not apply if window background is black or white.)

- **Clear: Inset 2** (`debug-winclear-inset2`): *Toggles insetting of the 'clear' rectangle of all windows.* When enabled, the cleared rectangle of the window is inset by two pixels on all four sides, leaving the outer pixels unerased during each window refresh. (Does not apply if window background is black or white.)
- **Clear: Border** (`debug-winclear-border`): *Toggle bordering of the 'clear' rectangle of all windows.* When enabled, the cleared rectangle of a window is outlined by a one-pixel-wide red border. (Does not apply if window background is black or white.)
- **Clear: Don't** (`debug-winclear-dont`): *Toggle clearing of all windows.* When enabled, windows are not cleared during refresh.
- **Clear: White First** (`debug-winclear-whitefirst`): *Toggle pre-clearing of all windows.* When enabled, windows are first cleared to white, then cleared again using the currently selected clear.
- **BBox: Draw** (`debug-winbbox-draw`): *Toggle display of the scene bounding box.* When enabled, the bounding box that surrounds the objects in the scene is displayed. This bounding box is used to set the clipping planes.
- **Late Mouse** (`debug-win-late-mouse`): *Toggle late sampling of the mouse position.* When enabled, the mouse position is read as late as possible prior to redrawing the window. Otherwise, the most recent mouse event from the Forms Library queue is used.
- **Output Profile** (`window-profile`): *Output profiling information to standard out.* This information shows the time required for various operations involved in redrawing the window.

- **Graph Debug Submenu**
  - **Print Optimization** (`graphopt-print`): *Toggle printing of debug information during optimization.*
  - **Print Compliance** (`graphopt-debug-compliance`): *Toggle printing of debug information during compliance calculation.*
  - **Display Super** (`graphopt-debug-display`): *Update display after each step during Super Optimize.*

### A.2.4 Tools Menu

- **Rendering Options...** (`show-rendering-options`): *Set rendering options.* This command displays a dialog that allows you to adjust various aspects of how parts are rendered, including, thickness and edge rounding. See Section A.3.3.
- **Graph Controls...** (`show-graph-controls`): *Control graph options.* This command displays a dialog allows that you to adjust various aspects of graphs, spanners, and optimization. See Section A.3.4.
- **GraphOpt Status...** (`show-graph-optstat`): *Display graph optimization statistics.* This command displays a dialog that displays the number of iterations of the most recent run of the optimizer, as well as the weight of the resulting graph.

### A.2.5 Help Menu

- **What is...** (`what-is`): *Display help about an input.* Select from a menu, type a key sequence, or click the mouse in a window. The terminal window will display help information about the action that is currently bound to that input.

## A.3 Dialogs

Much of the `graphlab` program is controlled via dialog boxes (forms). These include *Lights*, *Window Controls*, *Rendering Options*, *Graph Controls*, and *Relaxation Controls* dialogs:

### A.3.1 Lights Dialog

The *Lights* Dialog is accessed by the **Lights...** command on the **Window** menu. Figure A.1 shows the dialog. It contains the following controls:

1. **Activate**: The eight lights can be turned on and off individually with this group of buttons:
   - **Light 1 On** (`light-1-on`): *Turn light number 1 on and off.*
   - **Light 2 On** (`light-2-on`): *Turn light number 2 on and off.*
   - **Light 3 On** (`light-3-on`): *Turn light number 3 on and off.*

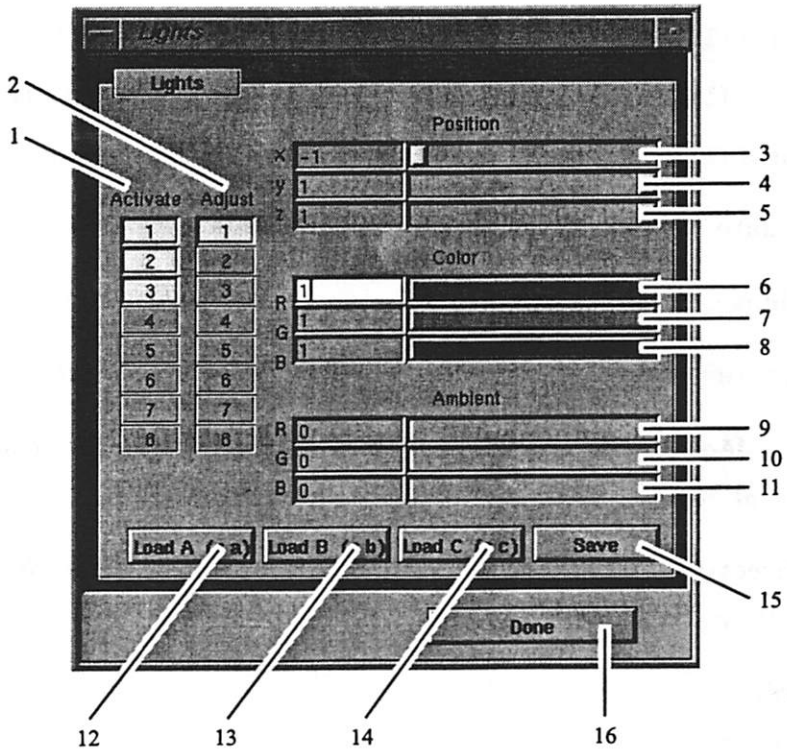



Figure A.1: The *Lights* Dialog.

- **Light 4 On** (`light-4-on`): *Turn light number 4 on and off.*
- **Light 5 On** (`light-5-on`): *Turn light number 5 on and off.*
- **Light 6 On** (`light-6-on`): *Turn light number 6 on and off.*
- **Light 7 On** (`light-7-on`): *Turn light number 7 on and off.*
- **Light 8 On** (`light-8-on`): *Turn light number 8 on and off.*

2. **Adjust**: To adjust a light's position and color, it must first be selected by pressing one of the buttons in this group:

   - **Light 1** (`light-1`): *Select light number 1 for adjusting color and position.*
   - **Light 2** (`light-2`): *Select light number 2 for adjusting color and position.*
   - **Light 3** (`light-3`): *Select light number 3 for adjusting color and position.*
   - **Light 4** (`light-4`): *Select light number 4 for adjusting color and position.*
   - **Light 5** (`light-5`): *Select light number 5 for adjusting color and position.*

- **Light 6** (`light-6`): *Select light number 6 for adjusting color and position.*
- **Light 7** (`light-7`): *Select light number 7 for adjusting color and position.*
- **Light 8** (`light-8`): *Select light number 8 for adjusting color and position.*

3. **x Coordinate** (`light-x`): *Set the x coordinate of the current light.*
4. **y Coordinate** (`light-y`): *Set the y coordinate of the current light.*
5. **z Coordinate** (`light-z`): *Set the z coordinate of the current light.*
6. **Red Directional Component** (`light-red`): *Set the RED directional component of the current light.*
7. **Green Directional Component** (`light-green`): *Set the GREEN directional component of the current light.*
8. **Blue Directional Component** (`light-blue`): *Set the BLUE directional component of the current light.*
9. **Red Ambient Component** (`light-ambient-red`): *Set the RED ambient component of the current light.*
10. **Green Ambient Component** (`light-ambient-green`): *Set the GREEN ambient component of the current light.*
11. **Blue Ambient Component** (`light-ambient-blue`): *Set the BLUE ambient component of the current light.*
12. **Load Lights A** (`lights-load-a`): *Load light colors and positions from file 'lights_a.h'.* The file must be in the same format as the file 'lights.h' created by the 'Save Lights' command.
13. **Load Lights B** (`lights-load-b`): *Load light colors and positions from file 'lights_b.h'.* The file must be in the same format as the file 'lights.h' created by the 'Save Lights' command.
14. **Load Lights C** (`lights-load-c`): *Load light colors and positions from file 'lights_c.h'.* The file must be in the same format as the file 'lights.h' created by the 'Save Lights' command.

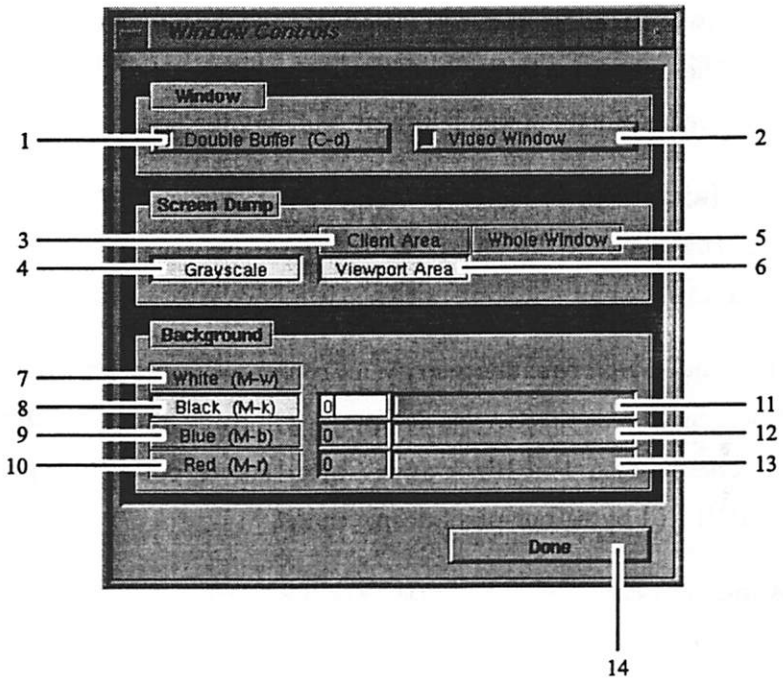



Figure A.2: The *Window Controls* Dialog.

15. **Save Lights** (`dump-lights`): *Save the current light colors and positions.* A file called 'lights.h' is created containing the lighting data. If this file is renamed to 'lights_a.h', 'lights_b.h', or 'lights_c.h', then it can be loaded by the respective 'Load Lights' command at any time. Alternately, this file can be moved to the skel directory and named 'deflights.h'. The values will be integrated into the program the next time 'lighting.cc' is recompiled.

16. **Done** (`hide-lights`): *Close the Lights form.*

### A.3.2 Window Controls Dialog

The *Window Controls* Dialog is accessed by the **Window Controls...** command on the **Window** menu. Figure A.2 shows the dialog. It contains the following controls:

1. **Double Buffer** (`window-double-buffer`): *Toggle the current window between single and double buffering.* Double buffering makes animation smooth, but degrades display quality. Single buffering makes display quality as good as it can be, but animation will flicker.

2. **Video Window** (`window-video`): *Toggle the window configuration between 'large' and 'video'.* The 'large' window is 950 x 925 pixels. The 'video' window is 605 x 479, and positioned in the lower left corner of the display.

3. **Client Area** (`window-screendump-client`): *Set the mode for saving screendumps to Client.* In this mode, the client area of the window is saved. This includes the viewport area and menu bar, but not the title bar or frame of the window.

4. **Grayscale** (`window-screendump-gray`): *Toggle saving of screendumps in grayscale format.* When enabled, the Grayscale option will cause the Screen Dump command to save the screen data as a Black and White (1 byte per pixel deep) image rather than as an RGB (3 bytes per pixel deep) image.

5. **Whole Window** (`window-screendump-window`): *Set the mode for saving screendumps to Window.* In this mode, the entire window is saved. This includes the viewport area, menu bar, and the title bar and frame of the window.

6. **Viewport Area** (`window-screendump-viewport`): *Set the mode for saving screendumps to Viewport.* In this mode, only the viewport area of the window is saved. No menus, title bars, window frames, etc., are included in the image file.

7. **White** (`window-background-white`): *Set the background of the current window to white.*

8. **Black** (`window-background-black`): *Set the background of the current window to black.*

9. **Blue Wash** (`window-background-blue`): *Set the background of the current window to a wash from black to dark blue.*

10. **Red Wash** (`window-background-red`): *Set the background of the current window to a wash from black to dark red.*

11. **Red Component** (`window-comp-red`): *Set the RED component of the current window's background color.*

12. **Green Component** (`window-comp-green`): *Set the GREEN component of the current window's background color.*

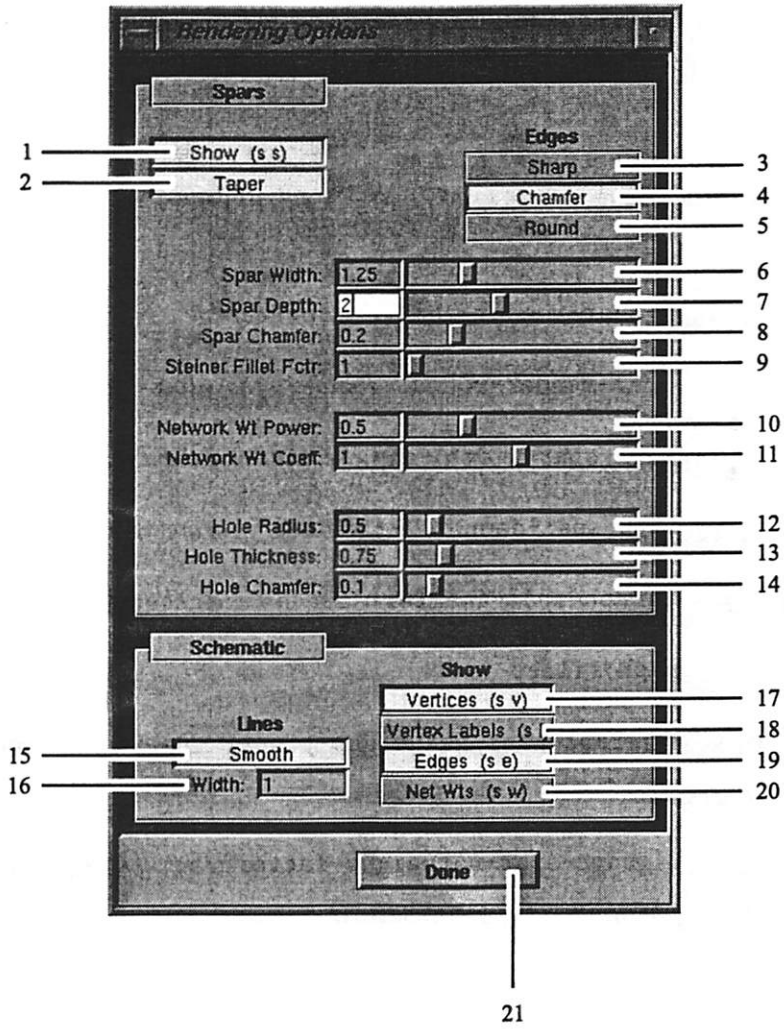



Figure A.3: The *Rendering Options* Dialog.

13. **Blue Component** (`window-comp-blue`): *Set the BLUE component of the current window's background color.*

14. **Done** (`hide-window-controls`): *Close the Window Controls form.*

### A.3.3 Rendering Options Dialog

The *Rendering Options* Dialog is accessed by the **Rendering Options...** command on the **Tools** menu. Figure A.3 shows the dialog. It contains the following controls:

1. **Spars** (`graph-display-spars`): *Toggle display of spars.*

2. **Taper** (`graph-taper-spars`): *Toggle tapering of spars.* When enabled, spar sections are tapered so that their widths at each end match the diameters of the associated nodes.

3. **Sharp** (`graph-spar-render-sharp`): *Render spar edges sharp, with no chamfer or rounding.*

4. **Chamfer** (`graph-spar-render-chamfer`): *Render spar edges chamfered.*

5. **Round** (`graph-spar-render-round`): *Render spar edges rounded.*

6. **Spar Width** (`graph-spar-width`): *Set the width of the spars.*

7. **Spar Depth** (`graph-spar-depth`): *Set the depth of the spars.*

8. **Spar Chamfer** (`graph-spar-chamfer`): *Set the edge chamfer of the spars.*

9. **Fillet Extra** (`graph-fillet-extra`): *Set the width of the spar fillets.*

10. **Net Wt Power** (`graph-spar-netweight-power`): *Set the exponent of influence that network weights have on spar thickness.*

11. **Net Wt Coeff.** (`graph-spar-netweight-factor`): *Set the coefficient of influence that network weights have on spar thickness.*

12. **Hole Radius** (`graph-spar-radius`): *Set the radius of the holes.*

13. **Hole Thickness** (`graph-spar-hole-thickness`): *Set the thickness of the material around the holes.*

14. **Hole Chamfer** (`graph-hole-chamfer`): *Set the edge chamfer of the holes.*

15. **Smooth Lines** (`graph-display-smooth`): *Toggle smoothing of lines.*

16. **Line Width** (`graph-line-width`): *Set the width that lines will be drawn in the current window.*

17. **Vertices** (`graph-display-edit-verts`): *Toggle display of Vertices in the Graph.*

18. **Vertex Labels** (`graph-display-vertex-labels`): *Toggle display of Vertex Labels in the Graph.* When enabled, vertices will be labeled with the address of the vertex object in memory.

19. **Edges** (`graph-display-edit-graph`): *Toggle display of the graph edges.*

20. **Network Weights** (`graph-display-network-weights`): *Toggle display of Network Weights of the graph edges.* This only works if the graph is a tree. If it isn't, no weights are displayed.

21. **Done** (`hide-rendering-options`): *Close the Rendering Options form.*

### A.3.4 Graph Controls Dialog

The *Graph Controls* Dialog is accessed by the **Graph Controls...** command on the **Tools** menu. Figure A.4 shows the dialog. It contains the following controls:

1. **Delete All Edges** (`graphed-del-edges`): *Delete all edges in the Edit Graph.*

2. **Delete Vertices** (`graphed-del-all-verts`): *Delete all vertices in the Edit Graph.*

3. **Delete Steiner** (`graphed-del-steiner`): *Delete all Steiner vertices in the Edit Graph.*

4. **N** (`graphed-random-n`): *Set the number of vertices for the 'Random Vertices' command.*

5. **Random Vertices** (`graphed-random-verts`): *Generate a random set of vertices.* The number of vertices generated is determined by the 'N' input field.

6. **Once** (`graph-compute-once`): *Compute a new initial graph based on the selected type.* Any existing graph will be replaced. The available types are Centroid, Nearest Neighbor, Principle Axis, Spanner, Delaunay Triangulation, and the Complete Graph.

7. **Continuous** (`graph-compute-contin`): *Continuously compute the initial graph based on the selected type.* The initial graph is recomputed after every change that is made to the input set or parameters. The available types are Centroid, Nearest Neighbor, Principle Axis, Spanner, Delaunay Triangulation, and the Complete Graph.

8. **Group Centroids** (`graph-neighbor-group`): *Toggle centroid grouping for Nearest Neighbor and Heirarchical initial graphs.* This setting affects where new steiner nodes

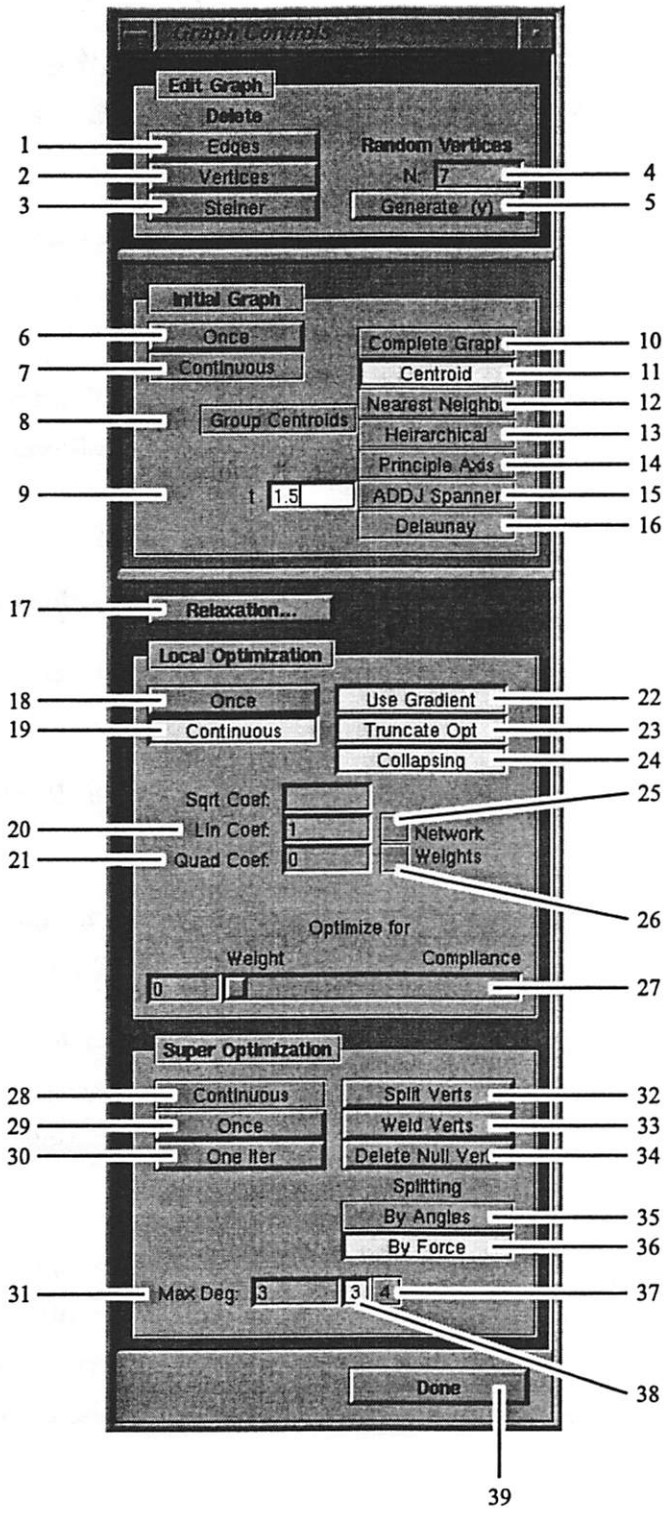



Figure A.4: The *Graph Controls* Dialog.

are placed during thegraph-generation process. When disabled, a new node is placed midway between the two nodes it connects. When this setting is enabled, a new node is placed at the centroid of the new connected component that it creates.

9. **t** (`graph-addj-t`): *Set the value of 't' for the ADDJ Spanner algorithm.*

10. **Complete Graph** (`graphtype-complete`): *Use a Complete Graph as the initial graph for the skeleton.* The Complete Graph has an edge between every pair of vertices in the set.

11. **Centroid** (`graphtype-centroid`): *Use a Centroid graph as the initial graph for the skeleton.* This graph is a spanning tree created by adding a single vertex at the centroid of the vertex set.

12. **Nearest Neighbor** (`graphtype-nearest`): *Use a Recursive Nearest Neighbor graph as the initial graph for the skeleton.* This graph is created by repeatedly finding the closest pair of unconnected vertices and creating a Steiner vertex between them.

13. **Heirarchical** (`graphtype-heirarchical`): *Use a Heirarchical Recursive Nearest Neighbor graph as the initial graph.* This graph is created by repeatedly finding the closest pair of unconnected vertices and creating a Steiner vertex between them.

14. **Principle Axis** (`graphtype-principle`): *Use a Principle Axis graph as the initial graph for the skeleton.* This graph is based on the principle axis of the vertex set.

15. **ADDJ Spanner** (`graphtype-addj`): *Use an ADDJ Spanner as the initial graph for the skeleton.*

16. **Delaunay Triangulation** (`graphtype-delaunay`): *Use a Delaunay Triangulation as the initial graph for the skeleton.*

17. **Relaxation Controls...** (`show-relaxation-controls`): *Control graph Relaxation options.* This dialog allows you to control the relaxation engine for graph optimization.

18. **Once** (`graphopt-locally-once`): *Optimize the Edit Graph once, by Conjugate-Gradient method.*

19. **Continuous** (`graphopt-locally`): *Toggle optimization of the Edit Graph.* When enabled, the Edit Graph is continuously driven to a local minimum by Conjugate-Gradient optimization.

20. **Linear Coefficient** (`graphopt-lin-coef`): *Set the linear coefficient for computing the cost of an edge.* The cost of an edge is computed by the function $c = al + bl^2$ where l is the length of the edge. This input sets the coefficient a.

21. **Quadratic Coefficient** (`graphopt-quad-coef`): *Set the quadratic coefficient for computing the cost of an edge.* The cost of an edge is computed by the function $c = al + bl^2$ where l is the length of the edge. This input sets the coefficient b.

22. **Use Gradient** (`graphopt-use-gradient`): *Toggle use of gradient information in the Conjugate-Gradient optimizer.*

23. **Trucate Optimization** (`graphopt-truncate`): *Toggle trunctation of optimizer runs.* When truncation is on, the optimizer is aborted whenever the gradient becomes undefined, e.g., when the length of an edge becomes zero. This seems to be useful. When the gradient becomes undefined, the optimizer can burn up a lot of iterations while making little progress.

24. **Collapsing** (`graphopt-collapsing`): *Toggle use of collapsing in the local optimizer.* When enabled, the local optimizer may change the topology of the graph by removing edges whose length becomes zero. Vertices at the ends of such edges are coalesced into a single vertex.

25. **Network Weights Quadratic** (`graphopt-use-network-quadratic`): *Toggle use of network weighting of graph edges in the optimizer.* When enabled, the quadratic term in the edge-weight cost function is multiplied by the number of node-to-node paths that include that edge. This setting is ignored if the graph contains loops.

26. **Network Weights Linear** (`graphopt-use-network-linear`): *Toggle use of network weighting of graph edges in the optimizer.* When enabled, the linear term in the edge-weight cost function is multiplied by the number of node-to-node paths that include that edge. This setting is ignored if the graph contains loops.

27. **Optimize for** (`graph-opt-lambda`): *Choose the relative importance of weight and compliance in the optimization.*

28. **Super** (`graphopt-super`): *Toggle super optimization of the Edit Graph.* When enabled, the Edit Graph is optimized using continuous and discrete methods. At each step a local minimum is found for the continuous optimization problem, then topology switching is applied and the process repeated.

29. **Super Once** (`graphopt-super-once`): *Super-optimize the Edit Graph once.* The Edit Graph is optimized using continuous and discrete methods. At each step a local minimum is found for the continuous optimization problem, then topology switching is applied and the process repeated. The process terminates when no topology switching is possible.

30. **One Iter** (`graphopt-super-one-iter`): *Perform one iteration of the super-optimization algorithm.*

31. **Maximum Steiner Degree** (`graphopt-steiner-degree`): *Set the Super Optimizer's maximum allowable degree for Steiner nodes.* The Super Optimizer will not split Steiner nodes having this many or fewer incident edges.

32. **Split Vertices** (`graphopt-split-tens-verts`): *Split all vertices in the edit graph that are under tension.*

33. **Weld Vertices** (`graphopt-collapse-edges`): *Find all edges in the edit graph of length zero and collapse them.*

34. **Delete Null Vertices** (`graphopt-delete-null-verts`): *Delete all degree-two Steiner vertices in the Edit graph.*

35. **Split Vertices By Angles** (`graphopt-splitting-byangles`): *Set the Super Optimizer's Vertex Splitting Method to 'Angles.'*

36. **Split Vertices By Force** (`graphopt-splitting-byforce`): *Set the Super Optimizer's Vertex Splitting Method to 'Force.'* Note that this only effects Steiner vertices. Other vertices are still split by angles.

37. **Maximum Steiner Degree = 4** (`graphopt-steiner-degree-4`): *Set the Super Optimizer's maximum degree for Steiner nodes to 4.* The Super Optimizer will not split Steiner nodes having this many or fewer incident edges.

38. **Maximum Steiner Degree = 3** (`graphopt-steiner-degree-3`): *Set the Super Optimizer's maximum degree for Steiner nodes to 3.* The Super Optimizer will not split Steiner nodes having this many or fewer incident edges.

39. **Done** (`hide-graph-controls`): *Close the Graph Controls form.*

## A.4 Initialization File

When the `graphlab` program starts up, it reads an initialization file called '`.racerxrc`'. It looks for this file first in the user's home directory, then in the current directory. The syntax of this file is Lisp, and there are two commands that may be used, *load-file*, and *bind-key*. Comments may be added to any line. A comment begins with a semicolon (;), and continues to the end of the line. All text in the comment is ignored by the program.

- **Load File** (`load-file`): *Load an additional initialization file.*

  Syntax: `(load-file "otherfile")`

  This command loads the additional initialization file named "otherfile". The file will be searched for in the same way as was `.racerxrc`. First the user's home directory is checked, then, if the file is not found there, the current directory is checked. The `load-file` command can be used as many times as necessary, and may also appear in the additional files that are loaded. Any depth of nesting is allowed.

- **Bind Key** (`bind-key`): *Bind a key sequence to a program action.*

  Syntax: `(bind-key "sequence" 'command)`

  This command binds a keyboard or mouse key sequence to a program action so that the action will be carried out whenever the user presses that sequence. There are no default bindings in the program; all bindings must be specified in the initialization files.

The "*sequence*" argument must be a string denoting a sequence of key names and modifiers. For example, the string "C-d" denotes the sequence **control-d**. That is, when the user holds down the **control** key and presses the **d** key, the bound program action will be invoked. The string "c a" denotes the sequence of pressing the **c** key followed by pressing the **a** key. The string "M-x C-c" denotes holding down the **meta** (or **alt**) key and pressing the **x** key, followed by holding down the **control** key and pressing the **c** key.

The '*command* argument must be the unique lisp identifier denoting the action to be bound to the sequence. These identifiers are shown in parentheses for every program action in this appendix.

Example: `(bind-key "M-w w" 'window-background-white)`

As noted in Section A.1, the mouse functions must be bound to keys (preferably mouse buttons), for them to be available when the program is run. A suggested minimum initialization file is as follows:

```
(bind-key "M-rmouse" 'window-pan)
(bind-key "M-mmouse" 'window-rotate)
(bind-key "M-lmouse" 'window-zoom)
(bind-key "lmouse" 'point-make)
(bind-key "S-lmouse" 'edge-make-edit)
(bind-key "S-rmouse" 'edge-make-steiner)
(bind-key "rmouse" 'point-make-steiner)
(bind-key "mmouse" 'point-drag)
(bind-key "C-rmouse" 'point-delete)
(bind-key "C-mmouse" 'edge-delete)
```

This file should be named `.racerxrc` and kept in the user's home directory.

## A.5 Graphlab Implementation Notes

The `graphlab` program's user interface is supported by a centralized event processing model. This minimizes modality and provides some user-configurability, while at the same time requiring minimal work on the part of the programmer. User-configurable key bindings are modeled after the user interface of GNU Emacs—bindings of key sequences to program actions are not hardcoded. Assignment of key sequences to actions is

defined in an ASCII text file that is read at program startup. The result is that all user functions can be available at all times (except where it doesn't make sense). For example, viewpoint navigation with the mouse can be performed at any time by holding down the **alt** key. Switching to a navigation mode is not required. Further, the assignment of the **alt** key to this behavior is not hardcoded, but is defined in the text file.

In addition to the advantage of user convenience, development effort is reduced because key-assignment collisions are eliminated: programmers do not define key-assigments in code, so conflicts do not arise. The syntax of the user key-assignment file is Lisp, and may be extended in a regular way in the future to support additional user options.

The programmer's interface to this system is an object-oriented library written in C++. Programmers expose a program option or action by creating an instance of a user-interface class object in the source code. This typically requires six lines of code for one object. All relevant information is thus maintained in a single location in the code, including help strings that explain the use of the user feature. A help menu is available to the user and provides a *Whatis* command. The user can click any menu, button, entry field, slider, or other interface object, and receive online help to explain its function. This programming model encourages the developers to evolve the documentation along with the software, rather than after the fact.

A large portion of this C++ library is dedicated to interfacing with the Forms Library [?]. Programmers create user interface screens using the `fdesign` program provided with the Forms Library distribution. In the source code, programmers instantiate one user-interface object for each Forms Library widget, using the same C++ interface as above. The connection is made by initializing the C++ object with a pointer to the Forms Library widget. The C++ object is also initialized with help strings, so that the same online help services are available with widgets drawn by the Forms Library. Further, the C++ object automatically makes itself available for key-sequence binding, so that the user can create a keyboard shortcut for any button, menu, text field, or any other user-interface widget drawn by the Forms Library. Thus the user interface event processing model is uniform from both the programmer's and user's points of view.