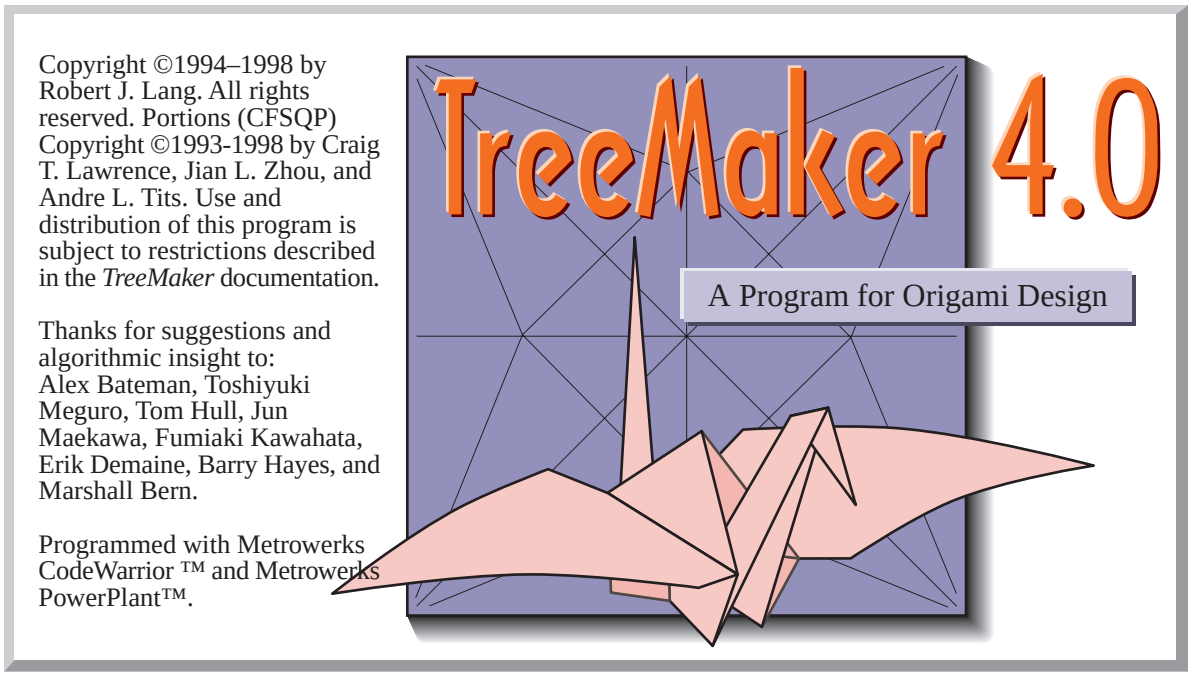


Terms of License

TreeMaker contains copyrighted code (CFSQP) that may only be used for non-profit purposes. *TreeMaker* may not be freely distributed.

CFSQP's Conditions for External Use include the following

1. The CFSQP routines may not be distributed to third parties. Interested parties should contact the authors directly.

2. Due acknowledgment must be made of the use of the CFSQP routines in research reports or publications. Whenever such reports are released for public access, a copy should be forwarded to the authors of CFSQP.

3. The CFSQP routines may only be used for research and development, unless it has been agreed otherwise with the authors in writing.

For more information on CFSQP, see http://www.isr.umd.edu/Labs/CACSE/FSQP/fsqp.html

To obtain a copy of the *TreeMaker* executable, please provide the following information:

Your name

Your affiliation (must be a non-profit organization)

Your e-mail address

An affirmation that you will abide by CFSQP's Conditions for External Use.

An affirmation that you will only be using *TreeMaker* for non-profit purposes and will not redistribute the executable

Confirmation that you are using a Macintosh™ computer (it won't run on anything else)

sent to: rjlang@aol.com

A copy of the executable will be emailed to you within a few weeks.

If you like the program, drop me a line with your comments and attaboys. If you find bugs, drop me a line with a description and I'll try to fix it in the next version. This software is provided as-is with no implied warranties of fitness or usability (but I hope you will find it to be be both fit and usable).

Table of Contents

TreeMaker 4.0

1.0 Introduction

TreeMaker is a tool for origami design. Starting from a description of a desired origami model, *TreeMaker* computes a crease pattern for folding a base for the model from an uncut square of paper. This document describes *TreeMaker*, how you use it to design origami, and how it works.

1.1 Background

Origami is the Japanese name for the art of paper-folding. The purest form of origami stipulates that you fold one sheet of paper, which must be square, and no cutting is allowed. These rules might seem restrictive, but over the hundreds of years that origami has been practiced, tens of thousands of origami designs have been developed for birds, flowers, animals, fish, cars, planes, and almost everything else under creation. Despite the age of the art, the vast majority of origami models have been designed within the last thirty years. This relatively late bloom of the art is due in part to a spread of the art worldwide and an increase in the number of its practitioners, but it is

due in large part to the development of a body of techniques for the design of origami models. Notable among these developments is the concept of a generalized base. Growing out of the four classic bases of origami — the bird, fish, frog, and kite bases — the concept of a base forms the foundation for nearly all sophisticated origami designs.

What, exactly, do we mean by "a base?" A base is a geometric shape that contains flaps corresponding to all of the appendages of the origami model. For example, the base for a dog would have six flaps, corresponding to four legs, a head, and a tail. The base for a beetle would have nine flaps, corresponding to six legs, two antenna, and an abdomen. As origami subject matter has moved from relatively simple birds to complex insects with legs, wings, horns, and antennae, the advances that enabled this transition were the development of geometric and other mathematical techniques for the design of the underlying origami base.

Fundamentally, *TreeMaker* is a program for designing origami bases. It calculates the crease pattern for a base that has any arbitrary number of points of arbitrary size and distribution. *TreeMaker* specializes in bases that can be represented by a tree graph, *i.e.*, an acyclic (no loops) line graph (or stick figure) in which each segment is labeled with a specified length. Each segment of the tree corresponds to a flap of the base. *TreeMaker* will compute a crease pattern showing how to fold a square (or rectangle, for non-purists) into a multi-pointed base whose distribution of flaps match the lengths and connections of the tree.

I have been working on *TreeMaker* for several years and have posted some earlier versions at public sites on the Internet, updating it as I discover or prove new algorithms and design rules. Earlier versions solved the fundamental design problem but only found a few of the creases. The current version, version 4.0 finds and fills in all of the creases for the full crease pattern.

To use *TreeMaker*, you represent the subject as a tree, or stick figure, that defines all of its appendages and their relative lengths. You construct the tree using a simple graphical point-and-click interface to define the segments of the tree and to set the lengths of the flaps. You can also include constraints that enforce symmetry requirements among the appendages — for example, forcing the model to be bilaterally symmetric, or forcing particular points to come from a corner or edge of the square (to control thickness or to allow color-changes). After you have defined the stick figure, *TreeMaker* will find an optimally efficient arrangement of points on a square that correspond to the nodes of the tree and will identify the major creases of the base. The crease pattern so computed is guaranteed to be foldable into a flat base with the proper proportions and is in fact a locally optimum solution for the base. ("Locally optimum" means that for a given starting configuration, you get the largest possible base for a given square, but an entirely different starting configuration might give a different base.) You can subsequently add points to the tree to simplify the crease pattern (I'll explain more about this later); when you have sufficiently simplified the pattern, a single command computes the rest of the creases. The full pattern can be printed (and cut out and folded) at arbitrary size or copied to the Clipboard and pasted into your favorite drawing program for further editing.

While anyone can use *TreeMaker* to construct a crease pattern for a base, effectively using it takes some practice and understanding of the process of origami design. You should start by working your way through the tutorials, which will give you a feeling for the capabilities (and limitations) of *TreeMaker*.

1.2 Installing TreeMaker

To install *TreeMaker*, drag the "*TreeMaker ƒ*" folder onto your hard disk. *TreeMaker* is a fat binary Macintosh application and will run native on both 68K and Power Macintosh. (I have no plans to port *TreeMaker* to other platforms. I am told there is a Mac emulator for Linux, DOS/Win95, and OS/2 called Executor. There is a sample version of it on http://www.ardi.com.)

Much of the information in *TreeMaker* is color-coded; although you can run it on a gray-scale system, it will be easier if you have a color monitor with at least 256 colors.

Although you can jump right into *TreeMaker* if you like to experiment (or you're the type who never reads manuals), I recommend that you work your way through the following tutorials to familiarize yourself with the capabilities of *TreeMaker*. A basic familiarity with Macintosh applications is assumed throughout.

1.3 Conventions

Throughout this manual, menu titles and menu commands will be emboldened.

2.0 Tutorials

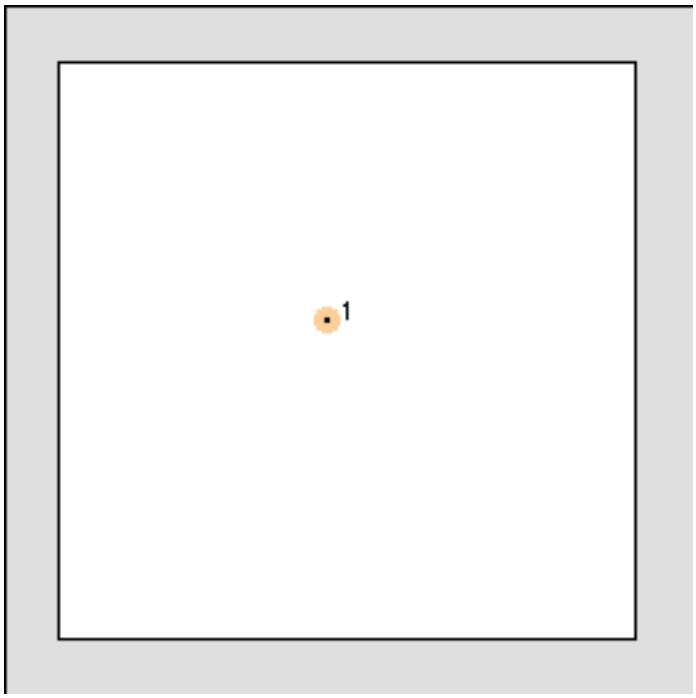
2.1 Tutorial 1: Designing a Base

In this tutorial, you will work through the design of a 4-legged animal with a head and tail and will learn the fundamentals of designing a model with *TreeMaker*. Before starting it, you should understand the fundamentals of origami design, such as the difference between corner, edge, and middle flaps and be familiar with the basic origami procedures and terms such as valley fold, mountain fold, reverse fold, rabbit ear, et cetera.

Begin by double-clicking on the *TreeMaker* icon to launch it. On startup, *TreeMaker* creates a new, untitled window that displays a square — the default shape of paper. Later, you will see how to change the size and shape of the paper (you can even design for a rectangle if you like) but for now we'll use a square.

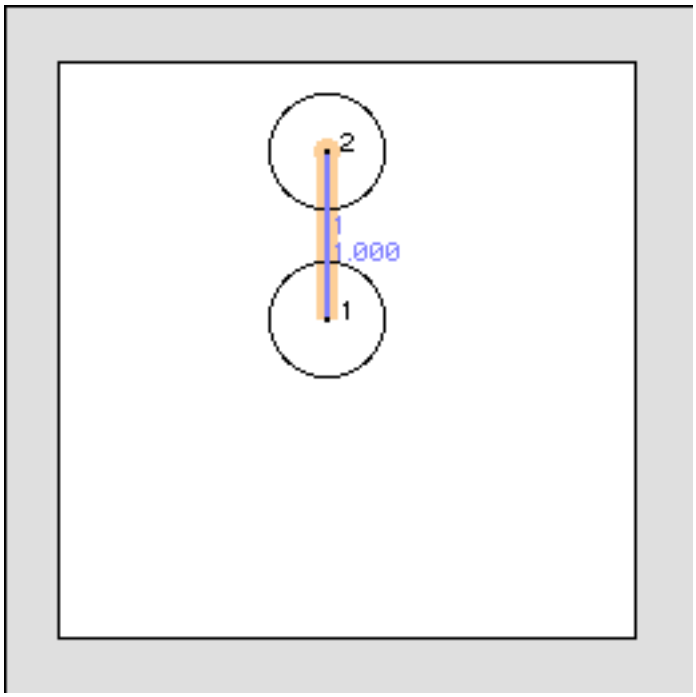
TreeMaker designs an origami base that can be represented by a stick figure. You specify the desired proportions of your base by drawing a stick figure in the square and specifying its dimensions. If you have ever used a drawing program on a Macintosh computer, you should have no trouble in figuring out how to draw a figure, but if you haven't, we'll walk you through it.

Click once in the middle of the square. You will see a dot appear with a number above it. This dot is a *node*; the number is the index of the node, which is used to identify the node later on. Each node is assigned a unique index. You can ignore the index for now. If you clicked just once, the node has a highlight around it, which means that it is *selected*. If the node has no highlight, it is *deselected*. The color of the highlight is the color set on your computer, so it may be different from the orange color shown in figure 2.1.1.



2.1.1

If exactly one node is selected and you click somewhere else, a new node will be placed at the click location with a line connecting the nodes. The line between the two nodes is called an *edge*. We'll do this now. Make sure the node is selected (click once on it so that it is highlighted) and then click once above the node; you have now created an edge connecting the two nodes. Note that the new node is now selected (highlighted), as is the edge, as shown in figure 2.1.2.



2.1.2

The edge displays two numbers near its midpoint: the index, which serves to identify the edge, and another number below it, which is the desired length of the edge. The length of the edge is the same as the length of the corresponding flap on the base.

- Note: all lengths on the tree are defined relative to each other, not relative to the size of the square.

All edges start out with a length of 1.000 by default; later you'll learn how to change the length of an edge. Note too that the displayed length of the edge doesn't change when you drag nodes and edges around; the displayed length is not the *actual* length of the drawn line; it's the *desired* length of the corresponding flap.

Note that each node has a circle around it. That's because they are *terminal nodes*. Terminal nodes are nodes that have only one edge attached to them, and they correspond to the tips of flaps of the base.

As mentioned earlier, if you click on an existing node, it becomes selected. If you hold down the mouse button after clicking you can drag the node around on the square. If exactly one node is already selected and you click elsewhere in the square, you create a new node at the click location and create a new edge connected to the selected node. (If zero or two or more nodes are selected and you click elsewhere, everything gets deselected.) You can also click on edges and drag them; dragging an edge drags the nodes at each end. You can extended the selected (select more than one node or edge at a time) by holding down the shift key when you click on nodes or edges. The collection of selected nodes and edges is called the *selection*.

The following are all the things you do with clicks:

- Click on a node or edge to select it and deselect everything else

- Shift-click on a deselected node or edge to add it to the selection

- Shift-click on a selected node or edge to remove it from the selection

• Double-click on a node or edge to edit its individual characteristics (we'll get into this later)

By clicking and dragging, you can change the position of a node or edge. As you build up constraints on the position of a node, a node can become "pinned" and you won't be able to drag it around any more. However, if you hold down the option key, you can override this behavior and drag a pinned node.

To delete a node or edge, you can either hit the backspace key or select **Clear** from the **Edit** menu. Although you can't drag a pinned node without holding down the option key, you can delete any node at any time.

Keyboard commands for editing are:

• <delete> or <forward delete> to delete selected nodes or edges

• <tab> to deselect everything

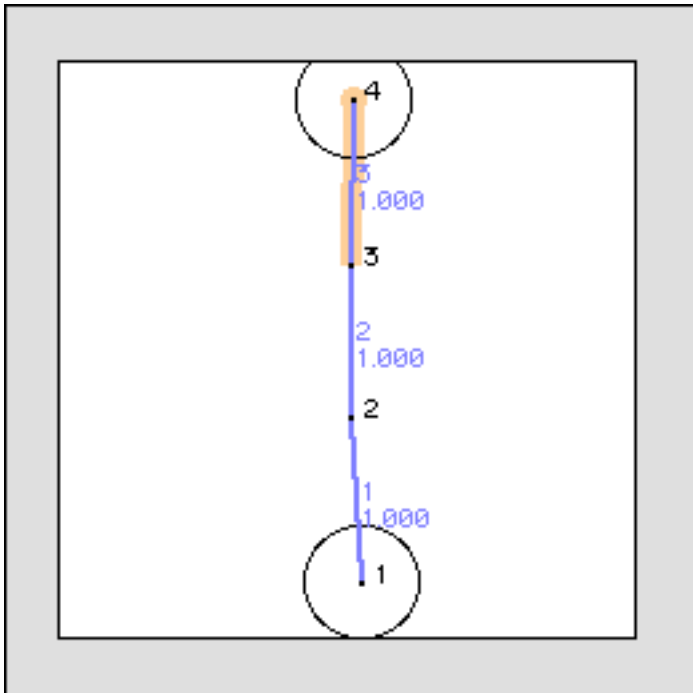
• Command-A to select everything

Deleting a terminal node also removes the attached edge. Similarly, deleting an edge also deletes any nodes that would otherwise be orphaned.

• Note: you can't delete a node or edge if it would break the tree into two pieces. You'll get an error message if you try.

Now we will build a tree. Choose **Select All** from the **Edit** menu and then choose **Clear** from the **Edit** menu (or hit <delete>) to get rid of what's currently on the square. Now, starting near the bottom of the square, click once to place a node; then click once about 1/3 of the way up; click once again about 2/3 of the way up; and click once near the top of the square. You will have created a line of four nodes and three edges as shown in figure 2.1.3.

This line of nodes represents the head (top), body (middle) and tail (bottom) of the animal. Note that only terminal nodes get circles around them.



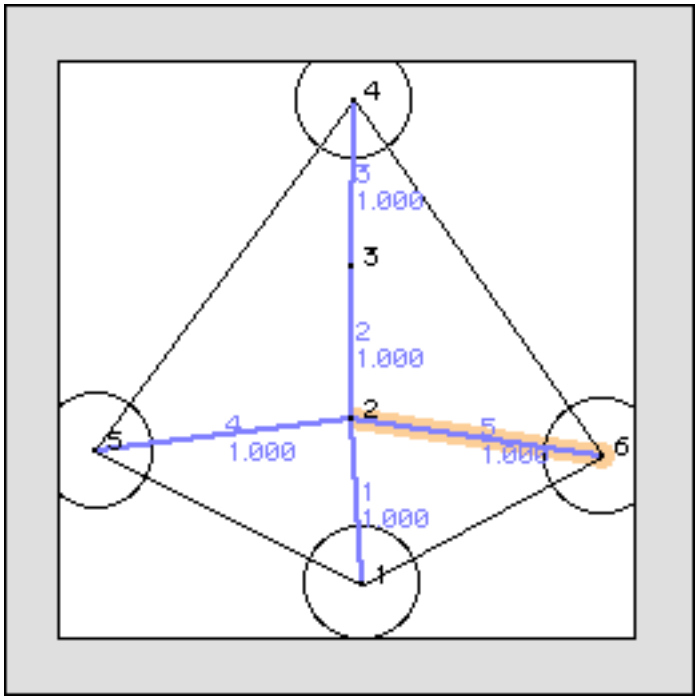
2.1.3

Now we will add side branches to the tree. Click once on node 2 to select it, then click once to the left of node 2; this adds a leg. Click again on node 2 to select it and click once to the right of node 2; this adds another leg, as shown in figure 2.1.4.

Got the hang of it? Now try to add two more legs to node 3, so that you wind up with something that looks like figure 2.1.5.

This stick figure represents an animal with a head, two front legs, a body, two rear legs, and a tail. It will be used to define a base with flaps for the head, legs, body, and tail. Since all of the edges of the stick figure have the same relative length, each of the flaps of the base (and the body) will be the same length in this example.

Note that the nodes are outlined by lines. These lines are called border paths; they'll define the usable region of the square when we're done.



2.1.4

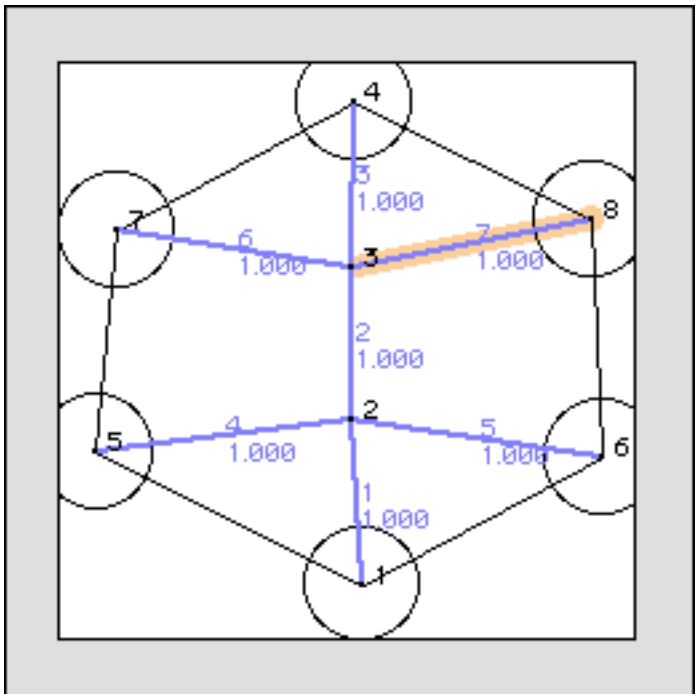
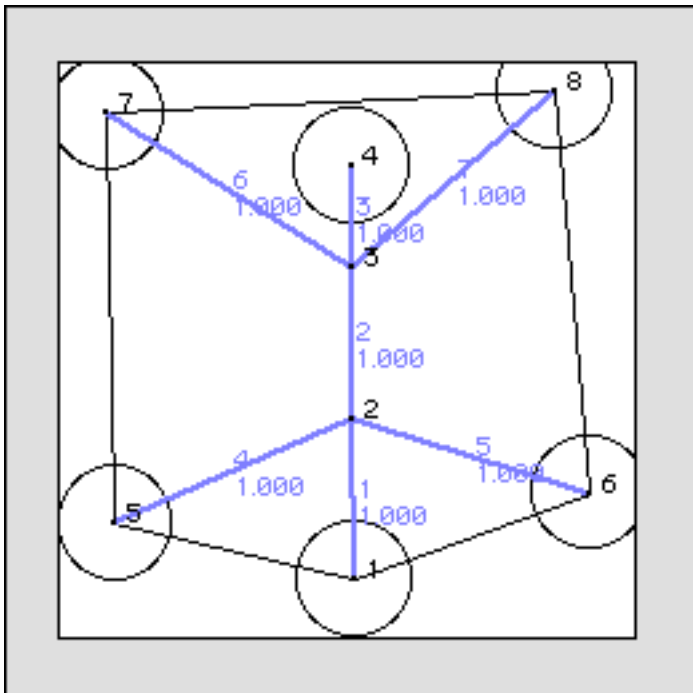


Figure 2.1.5

At this stage of the game, you can click on nodes and edges and move them about rather easily because none of them are pinned. Click on nodes 7 and 8 and drag them upward so that they lie above node 4, as shown in figure 2.1.6.



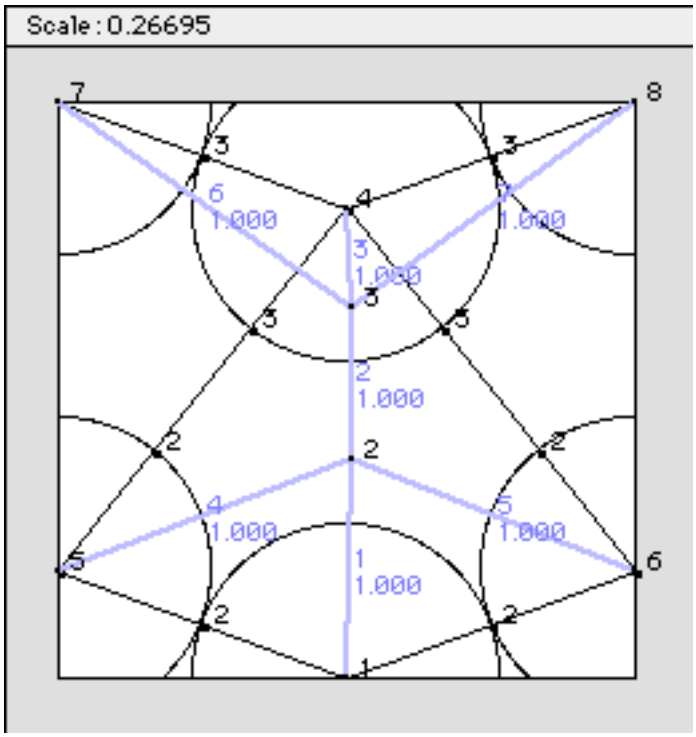
2.1.6

When you have defined the stick figure, it's time to start computing the crease pattern. Go to the **Action** menu and select the command, **Optimize Scale**. This starts the computation of the crease pattern. You can tell that the calculation is running from two things: first, the positions of the nodes will change and move around, and second, the cursor turns into a piece of paper that is folding and unfolding itself. The calculation will take anywhere from several minutes (on a Mac IIcx) to a few seconds (on a Power Mac).

• Note: if you get tired of waiting during a calculation, you can always interrupt it by pressing Command-Period.

• Note: most of the actions in *TreeMaker* are not easily undoable (the Undo command in the Edit menu usually isn't enabled). It's a good idea to save figures before you run an optimization, in case you don't like the results.

Let this calculation run until the cursor turns back into an arrow and things stop moving around. You should see a pattern something like figure 2.1.7. (Your example might look like figure 2.1.7 turned upside-down; if it does, don't worry about it.)



2.1.7

There is a lot of information in this figure, so let me spend a few minutes describing what we're seeing. Note that the circles surrounding each terminal node have expanded and moved so that several are touching. Several black lines have appeared that connect terminal nodes and they have dots and numbers along them. And the stick figure has been distorted (because the terminal nodes have moved) and has changed to a slightly lighter blue. Also, the number labeled "Scale" at the top of the window has changed from 0.10000 to 0.26695.

Here's what this all means. The circles represent the paper used by flaps that correspond to terminal nodes. The black lines between them are valley folds (most of the time) in the base. They are called *active paths* (for reasons that will come later). The dots and numbers along the active paths are the locations of the *internal nodes* (non-terminal nodes) along the active paths.

The stick figure has been distorted, which is because each of the terminal nodes of the stick figure is now located in its actual location on the square. That is, when you fold the base, the tip of the head flap will come from the point on the paper where the terminal node that corresponds to the head is located.

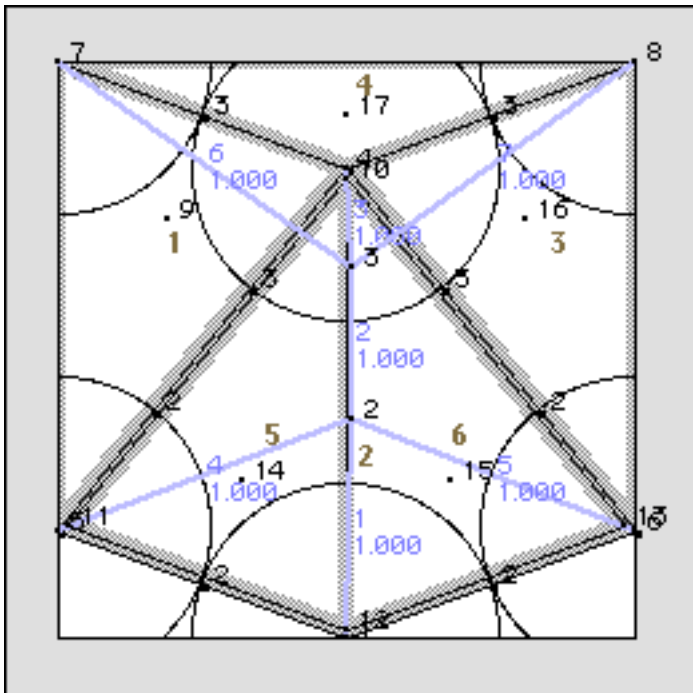
What **Optimize Scale** actually does is to move around the terminal nodes to try to find an arrangement that gives the largest possible base, subject to the proviso that all of the edges maintain the same lengths relative to each other. Although all of the edges are enlarged by the same amount, at the optimum configuration, at least some of the edges cannot be enlarged any more. Such edges are called "pinned" edges are are indicated by turning a lighter blue. In this optimization, all of the edges have turned light blue, which is an indication that no edge can be made any longer. In general, after you have run the **Optimize Scale** command, all or nearly all of the edges will be in this state.

Most of these lines, numbers, and circles are here for illustrative purposes and you will learn later how to turn on and off their display. The black lines, however, are actual creases in the base; they are (usually) valley folds. Bases are built in two stages. First, you **Optimize Scale**, which identifies the valley folds of the crease pattern that form a network of polygons. This defines the overall structure of the base. Then, you fill in the polygons with the remaining creases and you have several options at this point.

We'll do this second stage now via the simplest route. Go to the **Action** menu and select **Build Polygons**. You will see the polygons become inlined with a thick gray border, as shown in figure 2.1.8.

Then select the command **Build Creases**. You will see a bunch of new lines appear, as shown in figure 2.1.9.

This is the full crease pattern. The black lines are valley folds; the medium gray lines are mountain folds. You might wonder why I don't use dashed and chain lines for mountain and valley folds, since they are standard origami usage. I've found that with the low resolution of computer screens, in very complex crease patterns, it's hard to distinguish the two.



2.1.8

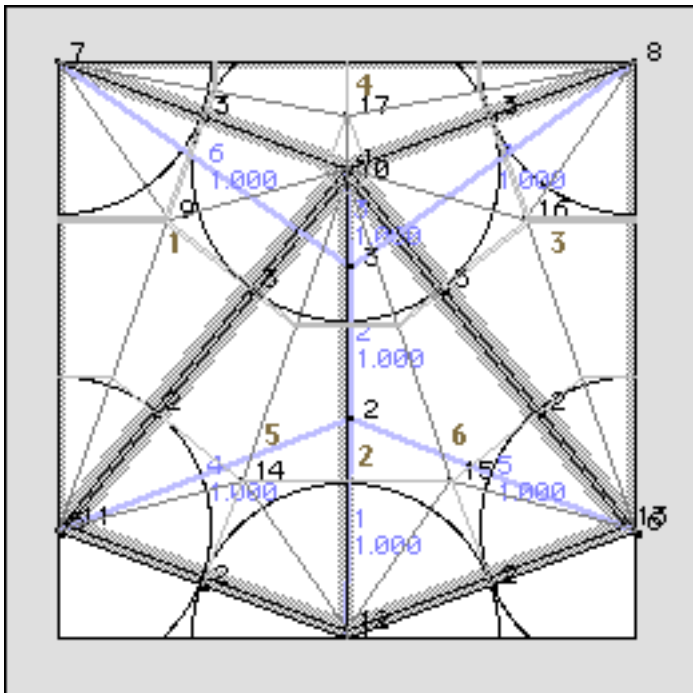
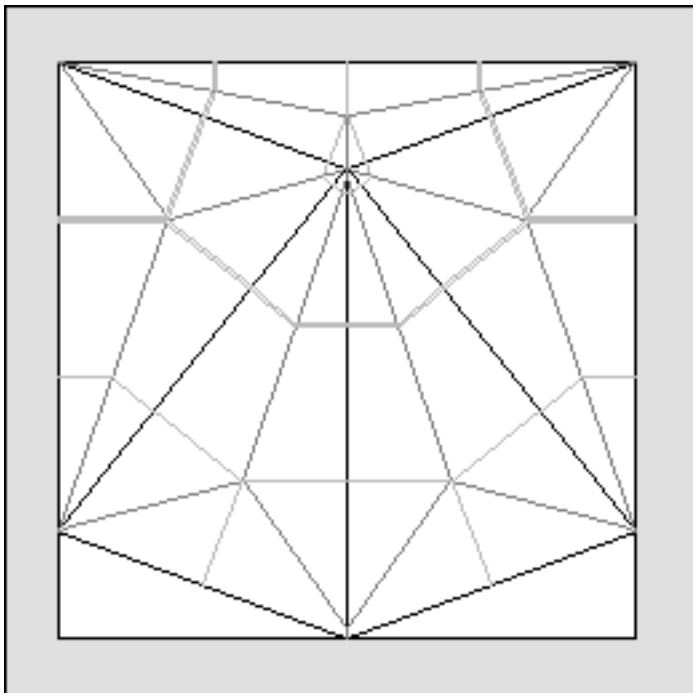


Figure 2.1.9

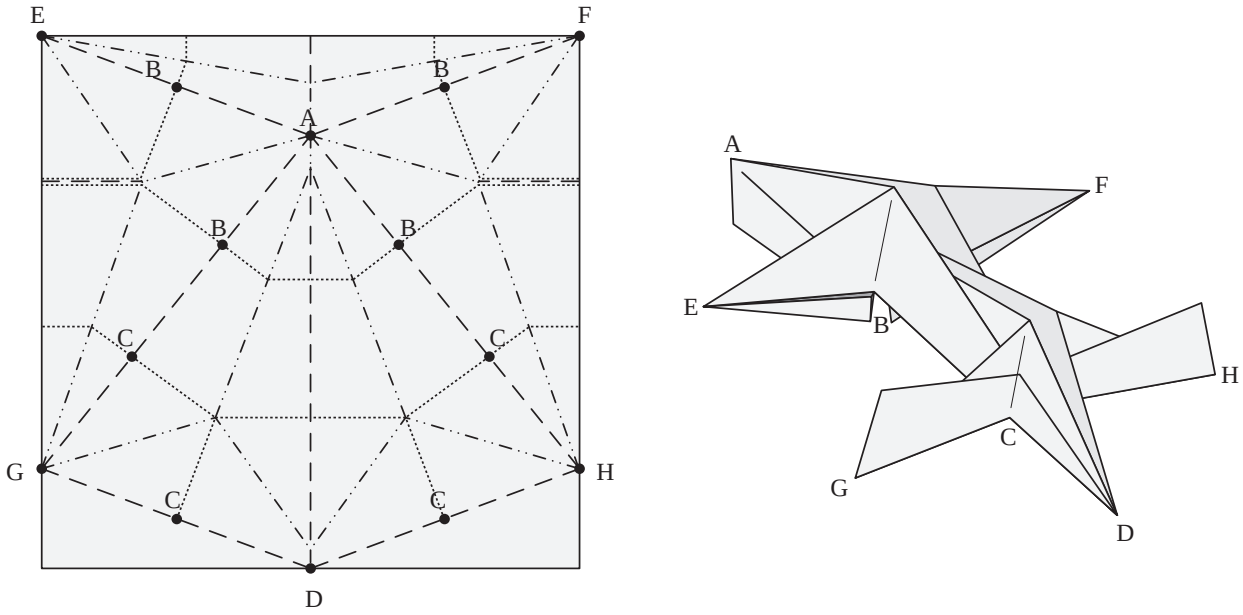
At this point, you are seeing several different sets of information overlaid on one another. The tree (shades of blue) displays the abstract structure of the tree. The gray polygons display the underlying structure of the crease pattern. The crease lines display the creases themselves. Because the creases are overlaid on top of everything else, the display gets rather busy when all of the creases are present. You can choose to display just the creases by choosing the **Show Creases Only** command from the **View** menu, getting the result shown in figure 2.1.10.



2.1.10

Finally, we have the crease pattern for the base — the thing we were looking for all along. In general, the black lines are valley folds, the medium gray lines are mountain folds, and the light gray folds can be either; but there are a few exceptions. The bottom corners of the square aren't used for anything, so you would probably mountain-fold them underneath, rather than valley-folding them upward. A major exception is the following. Around a middle point, the creases alternate around the point as mountain, valley, mountain, valley… and one of the valleys needs to be turned into a mountain fold to satisfy the Kawasaki theorem $M=V\pm2$. Since there are, in general, multiple valleys that could be turned into mountains, I don't make that choice for you. The light gray creases can be either mountain, valley, or unfolded, depending upon which way you direct the flaps when you flatten the model. These folds are, using the terminology of tree theory, tristate creases. So you will still need to add a few creases when you try to collapse the crease pattern into a base, but it should be pretty obvious where the additional creases go.

There are two ways to get the crease pattern from the screen onto a piece of paper. First, you can simply print it using the **Print** command in the **File** menu. Second, at any time you can copy the contents of the window to the Clipboard using the **Copy** command in the **Edit** menu. You can then paste the image of the crease pattern into your favorite drawing program for further touch-up and subsequent printout. In either case, if you print out the crease pattern and fold it up, you will arrive at the base shown in figure 2.1.11 (which also shows which creases are mountain and valley folds using the conventional line patterns). As promised, its flaps have the same number, connections, and relative size as the original stick figure.



2.1.11

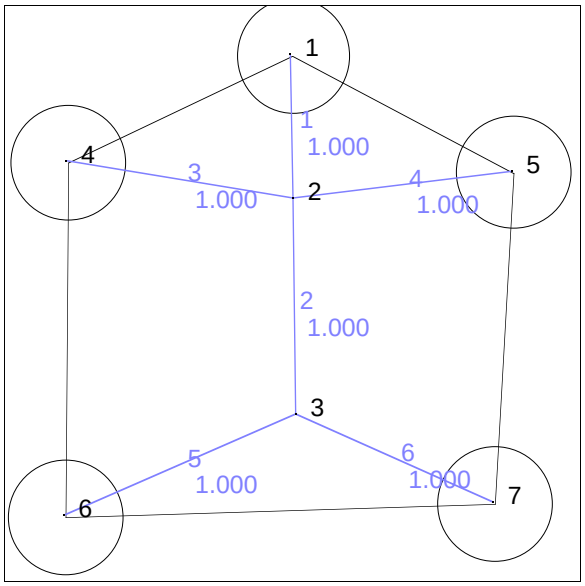
That's all for this example. You have learned the basics of *TreeMaker*: how to define a stick figure using the point-and-click interface and how to construct a basic crease pattern. In the next example, you'll learn how to modify the stick figure to alter proportions, how to incorporate symmetry, and how to customize the screen and printed image.

2.2 Tutorial 2: Imposing symmetry

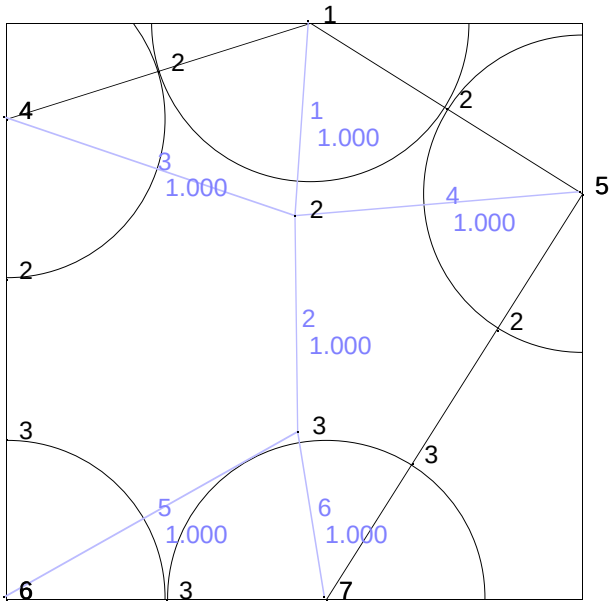
In this tutorial, you will learn how to use some more of the settings in *TreeMaker* and how to incorporate bilateral symmetry into your design.

Open the *TreeMaker* application if it is not already open and create a new square. Make a five-limbed stick figure as shown in figure 2.2.1. As you did in the previous tutorial, set the body segment to have a length of 0.8, but this time we'll leave the head to have a length of 1.

Now select **Optimize Scale** from the **Action** menu to optimize the distribution of nodes. You should see the nodes move around as usual, and then suddenly the two bottom nodes will shift over toward one side or the other. When the optimization stops, the pattern of nodes will be asymmetric, as shown in figure 2.2.2.

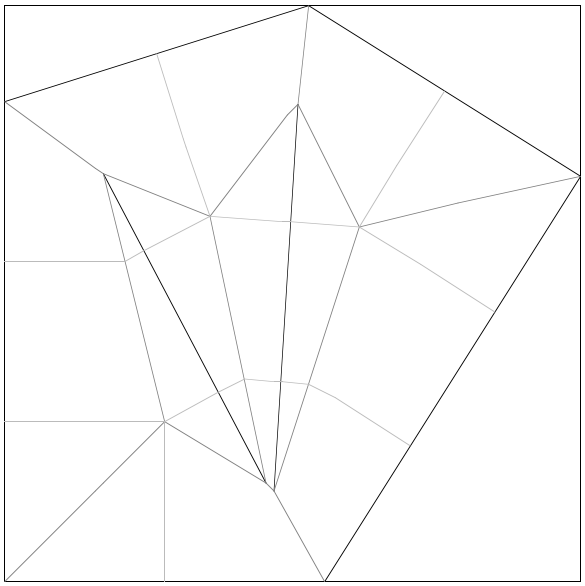


2.2.1.



2.2.2

This is rather surprising; the node pattern is less symmetric than the tree from which it is derived. The tree has bilateral symmetry — that is, the left side is the mirror image of the right side. However, the node pattern does not have bilateral symmetry. We'll see this in the crease pattern as well. Select **Build Polygons** from the **Action** menu; then **Build Creases** from the **Action** menu. Then convert to **Creases Only** view from the **View** menu. You will see that the resulting crease pattern has no line of symmetry at all (figure 2.2.3.)



2.2.3

This is an example of a phenomenon called "spontaneous symmetry-breaking," in which a system that is fundamentally symmetric at high energy settles into an asymmetric state at lower energy. (In the analogy, a larger scaling factor corresponds to a lower energy.) In this case, a slightly larger base is obtained for an asymmetric distribution of nodes than is obtained for a symmetric distribution of nodes. While symmetry-breaking is a wonderful phenomenon for generating Ph.D. theses, it can be very undesirable in origami design. The crease pattern in figure 2.2.3 can indeed be folded into the tree we started with and left and right flaps will have the same lengths, but they will have different widths and different distributions of layers, so the base will not have mirror symmetry.

An asymmetric base is not necessarily a bad thing. It depends on the position of the subject. If you were making a running human, for example, in which the left and right arms and legs were in different positions, it might not matter if paired flaps had different numbers of layers. But in a lot of models, it does matter, and in this case at least, we would like the folded base and therefore the underlying crease pattern to have the same bilateral mirror-symmetry as the subject.

Fortunately, *TreeMaker* offers us this option. The first thing to do is to define a line of the symmetry for the square. This need provides us with the opportunity to meet the Tree Editor dialog, which lets us change settings that affect the entire structure. Select **Edit Tree…** from the **Edit** menu, which brings up the dialog shown in figure 2.2.4.

2.2.4

This dialog gives information about the entire structure. It shows us the shape of the paper; the default paper is a square whose width and height are both 1 unit, but you can also design for rectangles by changing either the width or height. (But with *TreeMaker'*s capabilities for design, why would you ever want to use anything but a square?) The third number is the *scale*, which is the ratio between one unit on the tree and the size of the paper. (This is the same number that appears at the top of the window.) In this example, we see that the scale is 0.277, which means that a 1-unit edge on the tree will turn into a flap whose length is 0.277 times the side of the square from which it is folded.

- Note: when we select the command **Optimize Scale**, *TreeMaker* tries to make the scale as large as possible.

The controls and fields on the right are the ones that interest us. The "Symmetry" check box lets us define a line of symmetry on the square. A line of symmetry is defined by the *x* and *y* coordinates of any point on the symmetry line and by the angle of the symmetry line with respect to the paper. A square has two natural lines of symmetry, that is, two different ways of dividing it into mirror-symmetric halves. If you crease the square along the diagonal, you get two mirror-symmetric triangles and the diagonal is the line of symmetry. If you book-fold the square, you get two mirror-symmetric rectangles and the book fold is the line of symmetry. The two buttons named "diag" and "book" are presets for these two symmetries. Click on the "book" button. You'll see that the "Symmetry" check box is turned on, and we now have values entered for *x*, *y*, and the angle of the symmetry line as shown in figure 2.2.5.

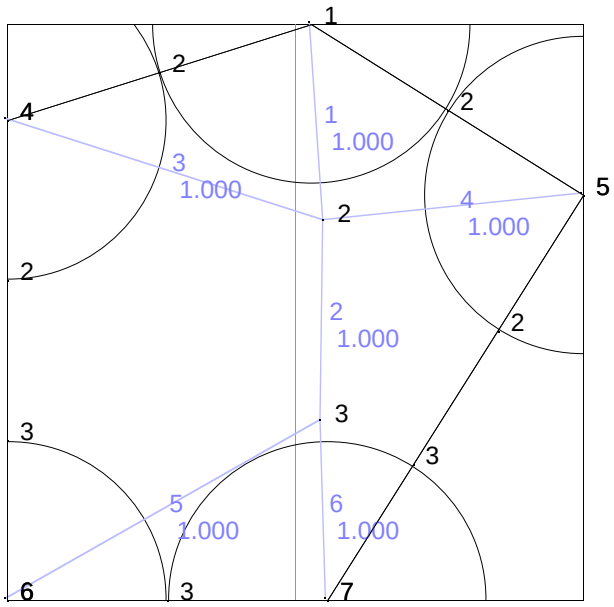
2.2.5

Click "OK" to make the dialog box go away.

- Note: you can, if you like, enter your own values to either offset the line of symmetry from the center of the paper or to create a line of symmetry for a rectangle.

If you built creases to make your paper look like figure 2.2.3, select **Remove Polygons** from the **Action** menu and make sure you are in **Default View** (in the **View** menu). You'll see that the paper now has a green line, which indicates the line of symmetry as shown in figure 2.2.6.

Now we need to establish a relationship between individual nodes and the line of symmetry. A node can have one of two possible relationships with the line of symmetry: it actually lie on the line of symmetry (for example, the head or tail of an animal) or it can be one of a pair of nodes arranged symmetrically about the line of symmetry. In our model, the head lies on the symmetry line of the tree. Thus, we need to force the node that corresponds to the head (node 7) to lie on the symmetry line of the square.



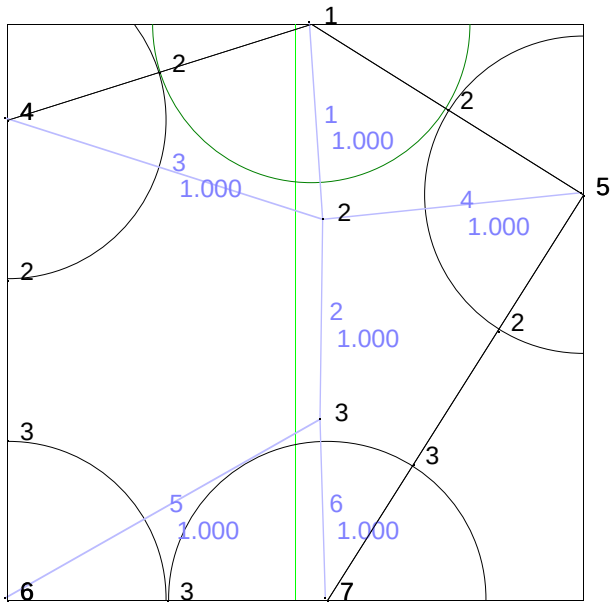
2.2.6

When we want to impose additional requirements on a tree, we are imposing conditions on the various parts of the tree. We can impose conditions in several ways. The easiest is to simply select node 1, and from the **Conditions** menu, select the command **Node fixed to symmetry line**.

What we have done is impose a condition on node 1. The tangible evidence of this is that the circle around node 1 has turned green, as shown in figure 2.2.7. Also, if you click on the **Conditions** menu, you'll see that the **Node fixed to symmetry line** command is now checked. (Don't select the command again, or you'll toggle off this condition.)

- Note that node 1 hasn't moved and isn't on the symmetry line; conditions don't take effect until you perform another optimization.

Next, we need to tell *TreeMaker* that pairs of flaps should be mirror-symmetric. For example, nodes 4 and 5 correspond to the left and right arms of the figure. Thus, each of nodes 4 and 5 should be the mirror image of the other.



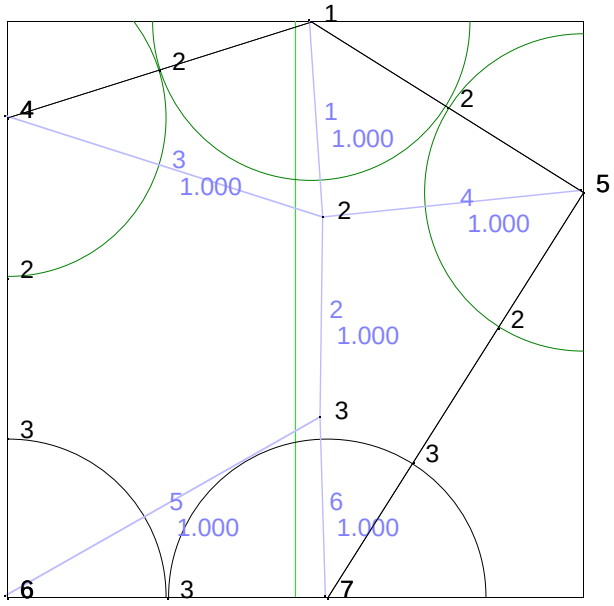
2.2.7

To set up this condition, click on node 4 and shift-click on node 5 to select them both. Then go to the **Conditions** menu and select the **Nodes paired** command.

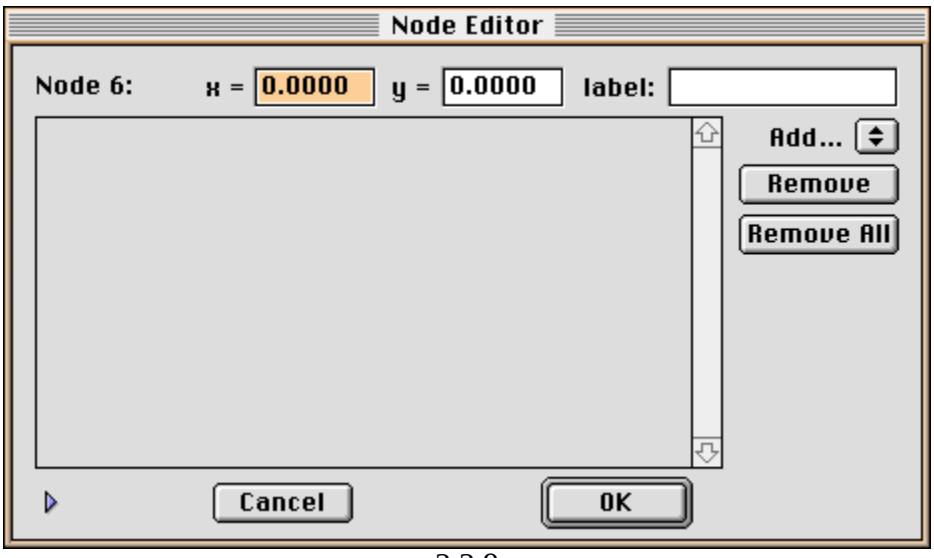
Now that nodes 4 and 5 have a condition on them, their node circles are also green as shown in figure 2.2.8. If you still have them selected and click on the **Conditions** menu, you'll see that the **Nodes paired** command is checked.

We also need to pair nodes 6 and 7. You could do this the same way — select nodes 6 and 7 and then select the **Nodes paired** command — but there's more than one way to place conditions on nodes. So, just for the experience, we'll do these a different way.

Double-click on Node 6. This brings up the Node Editor dialog, which is shown in figure 2.2.9.

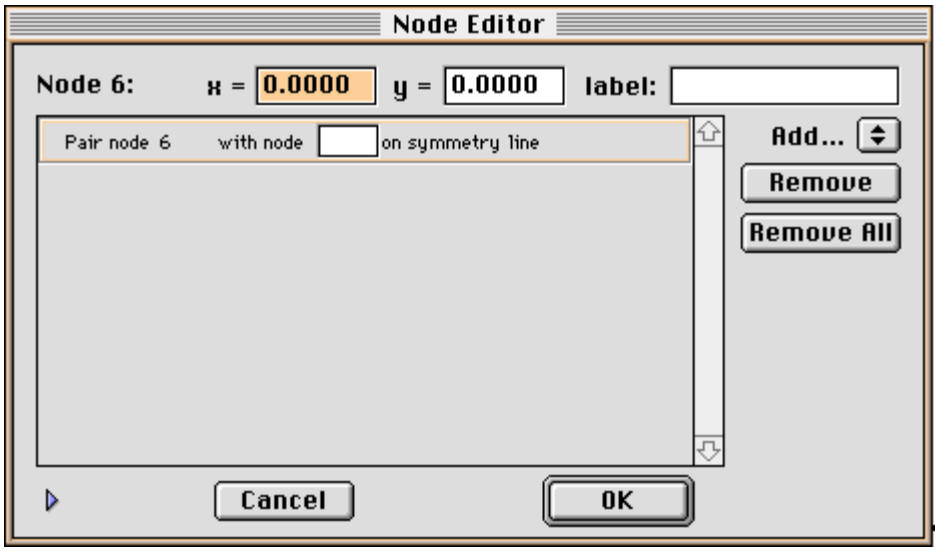


2.2.8



2.2.9

The (empty) scrolling list is a list of all conditions that apply to node 6. It's empty because there are no conditions on node 6. The popup menu at the right is for adding conditions. If you click on the popup menu it will give you a list of conditions to choose from. Select **Nodes paired about symmetry line**. A new condition will appear in the scrolling list, as shown in figure 2.2.10.



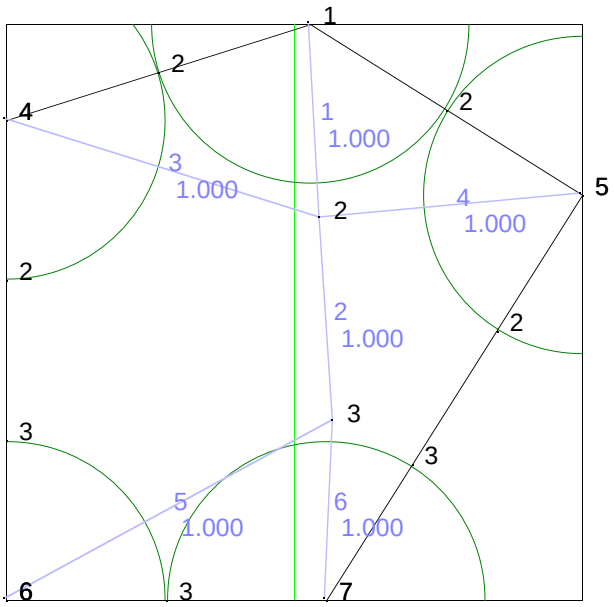
2.2.10

You see that a condition has been added to the scrolling list. You need to fill in which node is paired with node 6. Type a 7 into the field and click OK.

The result is shown in figure 2.2.11. After the dialog box is closed, you'll see that the circles for nodes 6 and 7 circle have turned green.

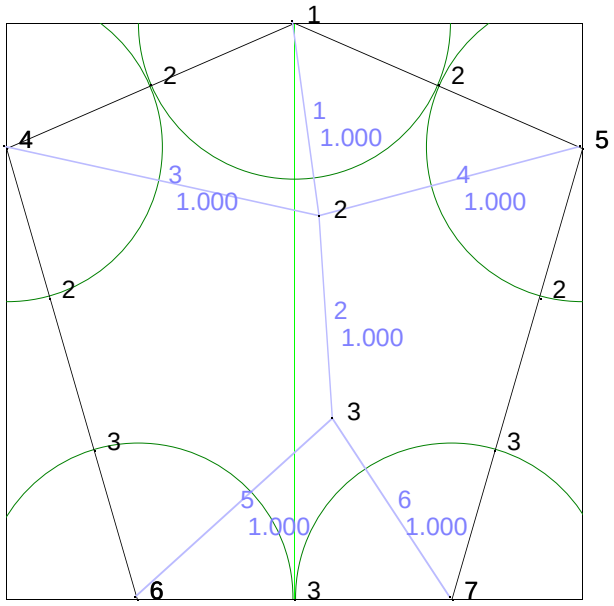
• Note: if you haven't defined a line of symmetry, the symmetry-related conditions have no effect.

• Note: terminal nodes are the only nodes you can set conditions on. You don't have to set up any conditions on internal nodes (and in fact any conditions you might set up on them are ignored) since the position of internal nodes in the tree are irrelevant to the final crease pattern.



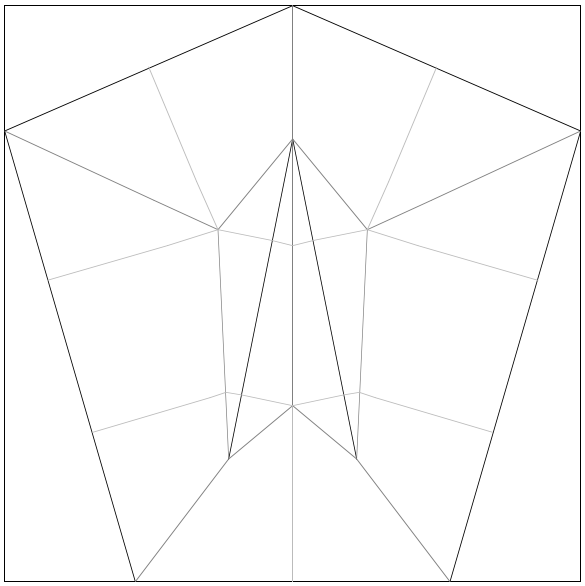
2.2.11

Now we have set up the conditions that enforce symmetry. As mentioned above, the conditions don't take effect until we re-optimize the pattern. Select **Optimize Scale** from the **Action** menu to re-optimize the crease pattern taking the new conditions into account. Figure 2.2.12 shows the result.



2.2.12

When you ran the optimization, you saw that the figure almost instantly became symmetric and the final pattern of terminal nodes is now symmetric. (Observe, though, that the internal nodes and edges are right where we left them.) If you select **Creases Only** (from the **View** menu) and **Build Polys and Creases** (from the **Action** menu), you will see the full crease pattern, which is also mirror-symmetric, as shown in figure 2.2.13.

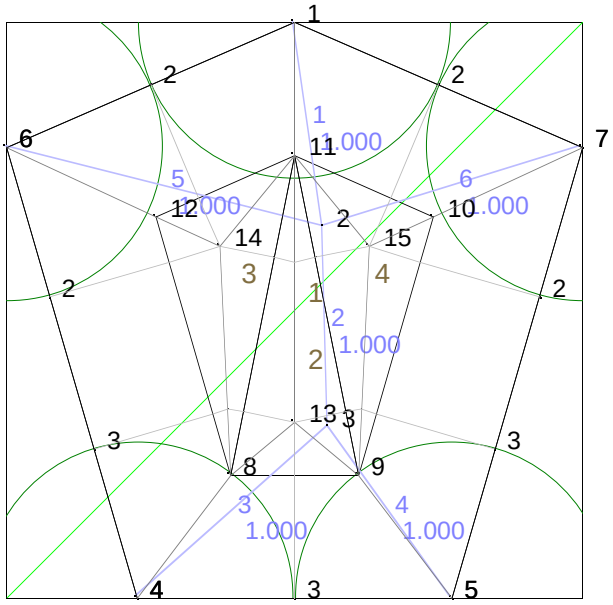


2.2.13

Note the scale of the model (shown at the top of the window) is now 0.272 (it was 0.277 with the asymmetric structure). The scale has dropped, meaning that the base folded from this pattern is slightly smaller than the asymmetric base. In general, when you add constraints to a pattern, you will reduce the scale and thus the size of the finished model.

We can also try the other line of symmetry. Select **Edit Tree…** from the **Edit** menu to open the Tree Editor. Click on the "diag" button, which sets up a line of symmetry along one of the diagonals of the square and click OK. Also, put the square back into **Default View** if it wasn't there already.

Now the node pattern is still symmetric about the book fold and the conditions on the nodes relating to symmetry haven't changed, but we have moved the line of symmetry; observe that the green line of symmetry in figure 2.2.14 is now along the diagonal. Therefore, the conditions we imposed are no longer satisfied. Nothing has moved because conditions aren't enforced until we re-optimize. Again select the command **Optimize Scale** from the **Action** menu.



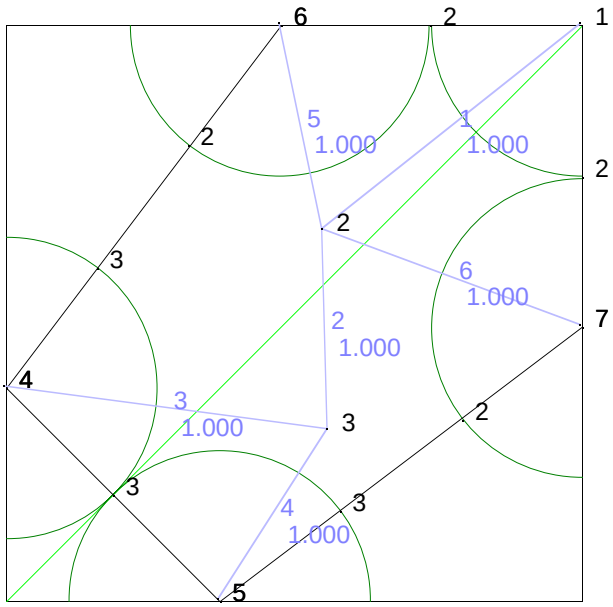
2.2.14

Observe that, as shown in figure 2.2.15, the nodes have moved to new positions consistent with the new symmetry conditions. Also, the polygons and creases have disappeared. Whenever you change the node configuration by moving a node, the polygons that used to include that node are destroyed, as are any creases that may have been associated

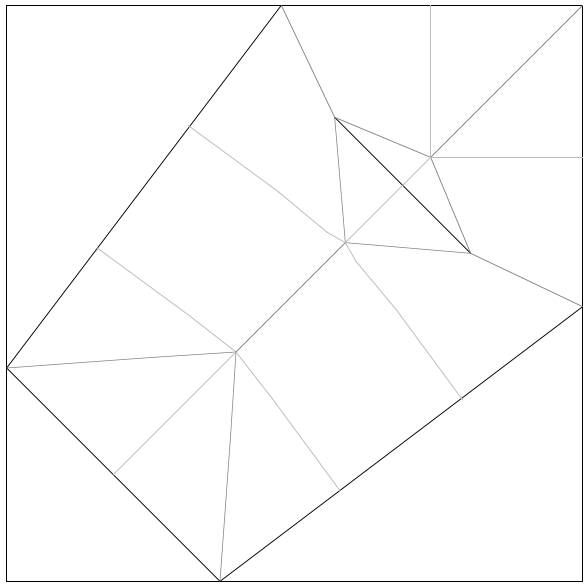
with the polygon. You'll have to rebuild the polygons yourself, which you can do by selecting the **Build Polys and Creases** command from the **Action** menu. Then put the model back into **Creases Only** view (**View** menu).

The new crease pattern, shown in figure 2.2.16, is mirror-symmetric about the new symmetry line and is completely different from the crease pattern that has book symmetry.

Also, if you check the scale (at the top of the window), you'll see that the scale has fallen once again to 0.2626. So the diagonal symmetry in this example is less efficient than book symmetry. In general, you can orient a model along either symmetry line and will have very different crease patterns for both configurations. Sometimes book symmetry will be more efficient, sometimes diagonal symmetry will be more efficient. With *TreeMaker*, you can easily check them both.



2.2.15

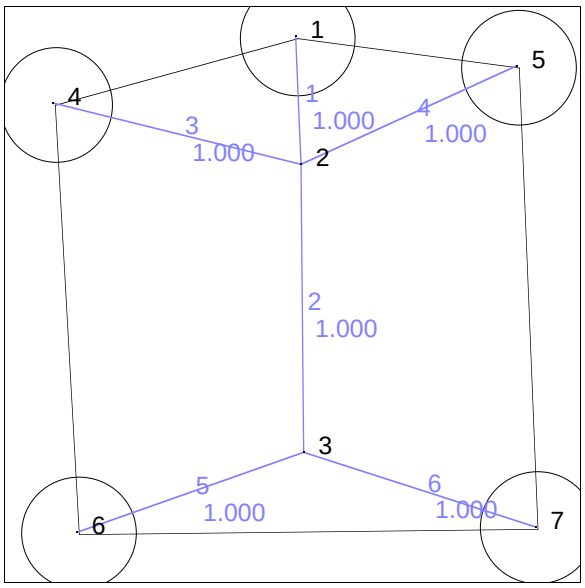


2.2.16

2.3 Tutorial 3: Changing edge lengths

In this tutorial, you will work through the design of a base that could be used for a human figure. In the process, you will learn how to change the lengths of flaps.

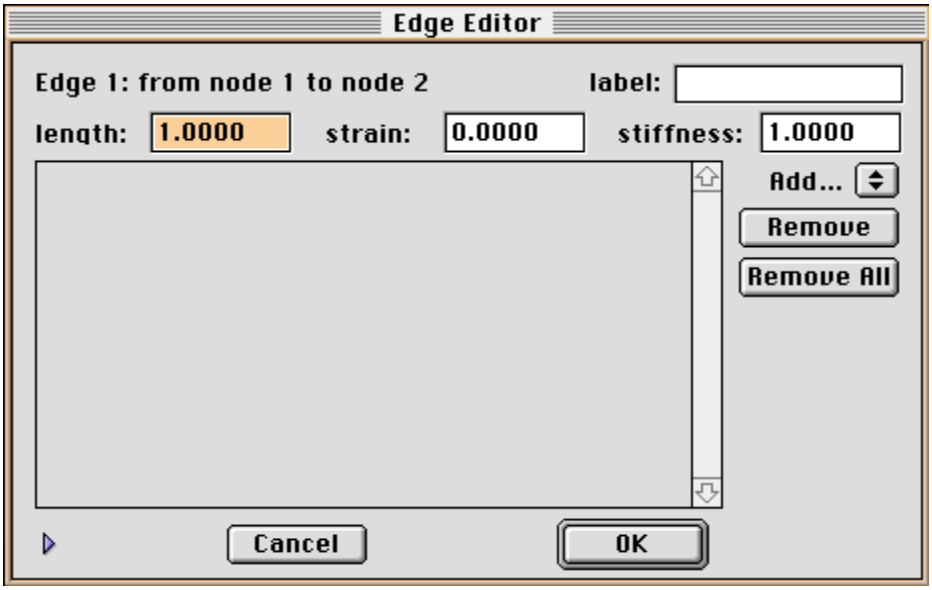
If you have quit *TreeMaker*, start it up again or select **New** from the **File** menu to create a new square. We will be making a human figure, so using the techniques described in the previous tutorial, draw a stick figure of a human as shown in figure 2.3.1.



2.3.1

Initially, all of the edges of the tree graph have the same length, which is defined to be 1 unit. The length of the edge is written near the middle of the edge. For many designs, it is desirable to change the lengths of some of the edges to make them longer or shorter than other edges. For example, if you were making a grasshopper, you'd want the back legs to be much longer than the front legs. In *TreeMaker*, you can choose the length of any flap. That's what we'll learn how to do now.

Each edge of the tree has a set of attributes, one of which is its length. To change the length of the "head" (i.e., edge number 1), double-click on it. A dialog box will appear as shown in figure 2.3.2.



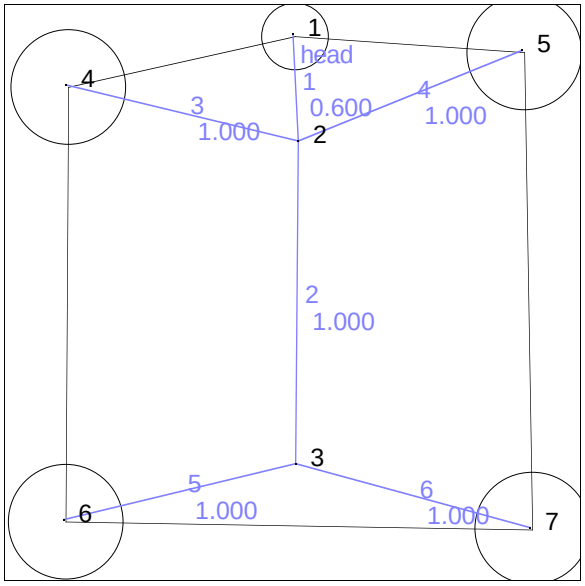
2.3.2

The dialog box tells the index of the edge and which nodes the edge runs between; the desired length of the edge; two quantities called strain and stiffness (more on these later); an optional label; and whether there are any condition on the edge. We'll change the length of the "head" flap by giving it a new length. Enter "0.6" in the Length field, which will make the head flap a bit over half the length of the arms and legs. If you like, you can also enter a label such as "head" in the Label field as well. Then click the OK button.

You'll notice that the circle around node 1 has gotten smaller, as shown in figure 2.3.3. The radius of the circle around a terminal node is equal to the length of the adjacent edge using the current scale of the model (which is 0.1000). A terminal node attached to a 1-unit-long edge has a circle of radius 1×0.1 = 0.1 times the side of the square. Since we shrank the edge to a length of 0.6, the circle shrinks accordingly to a radius of 0.6×0.1 = 0.06.

You can also see that the length of the edge (the number near the middle of the edge) is now 0.6. The label, if you entered one, also appears above the index of the edge.

It's probably worth a reminder that the length of the blue line segment that represents the edge hasn't changed a whit; what matters is the *numerical* length that appears next to it.

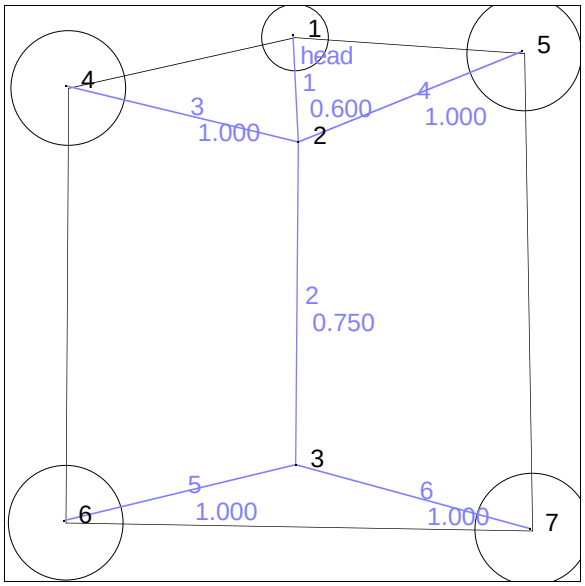


2.3.3

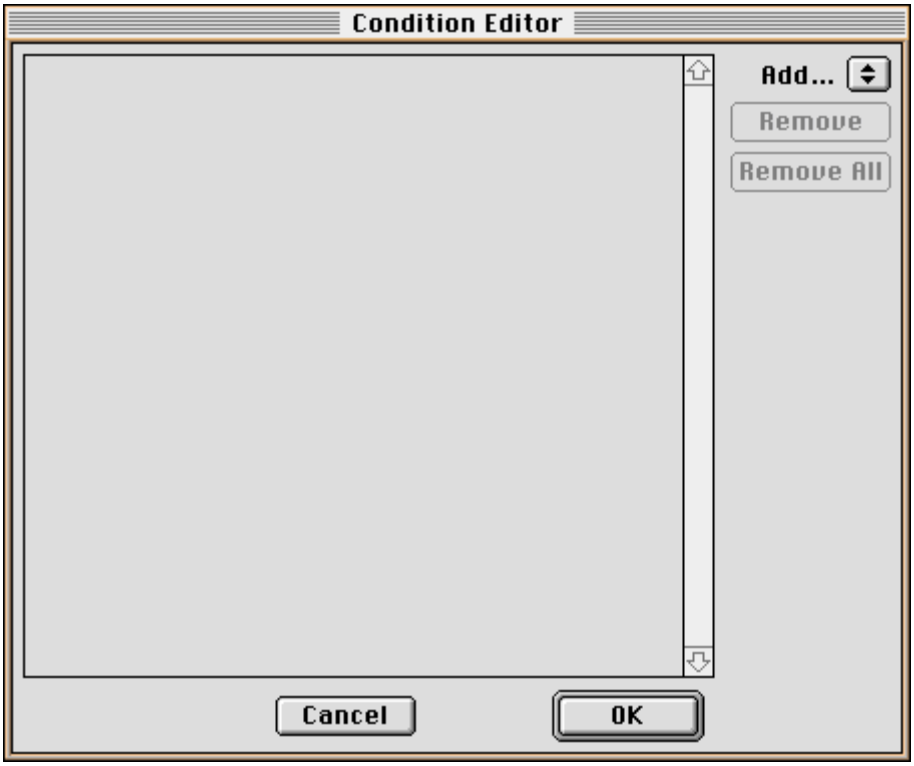
Now we'll also shrink the torso a bit. Double-click on the torso and enter a length of 0.75 in the Length field. Then click OK. The result is shown in figure 2.3.4.

We'll also make this model symmetric and will take this opportunity to learn a third way of applying conditions. Open up the Tree Editor by selecting the **Edit Tree…** command from the **Edit** menu and turn on book symmetry by clicking on the "book" button.

Now we'll apply conditions, but rather than selecting specific menu commands or editing particular nodes as we did before, we'll create all the conditions at once. Select the **Edit Conditions…** command from the **Conditions** menu. You will be presented with a dialog as shown in figure 2.3.5.

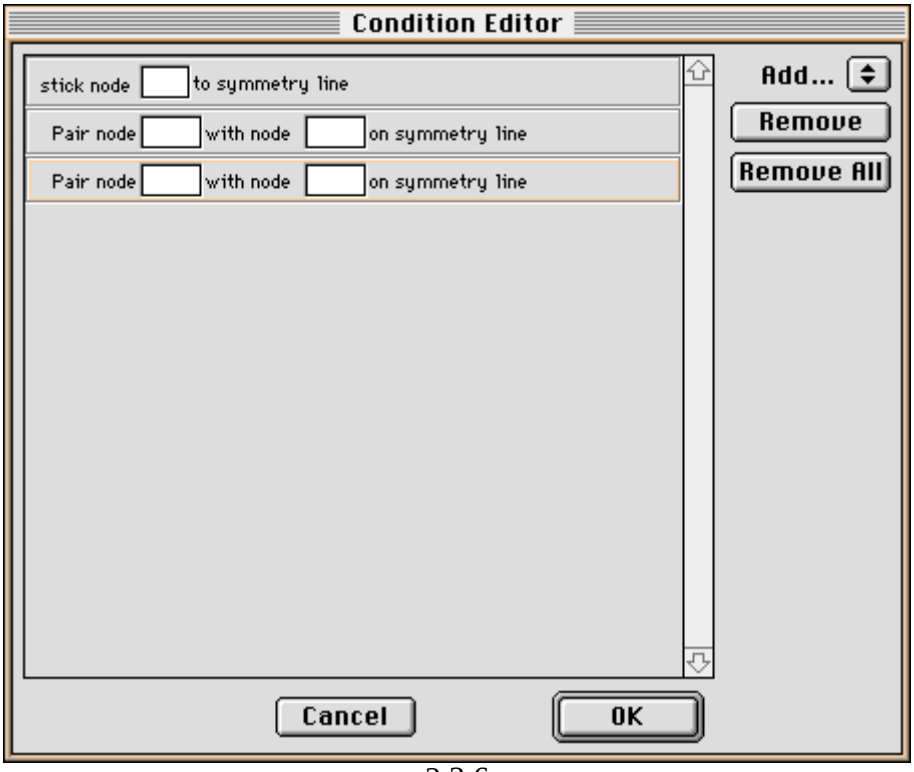


2.3.4



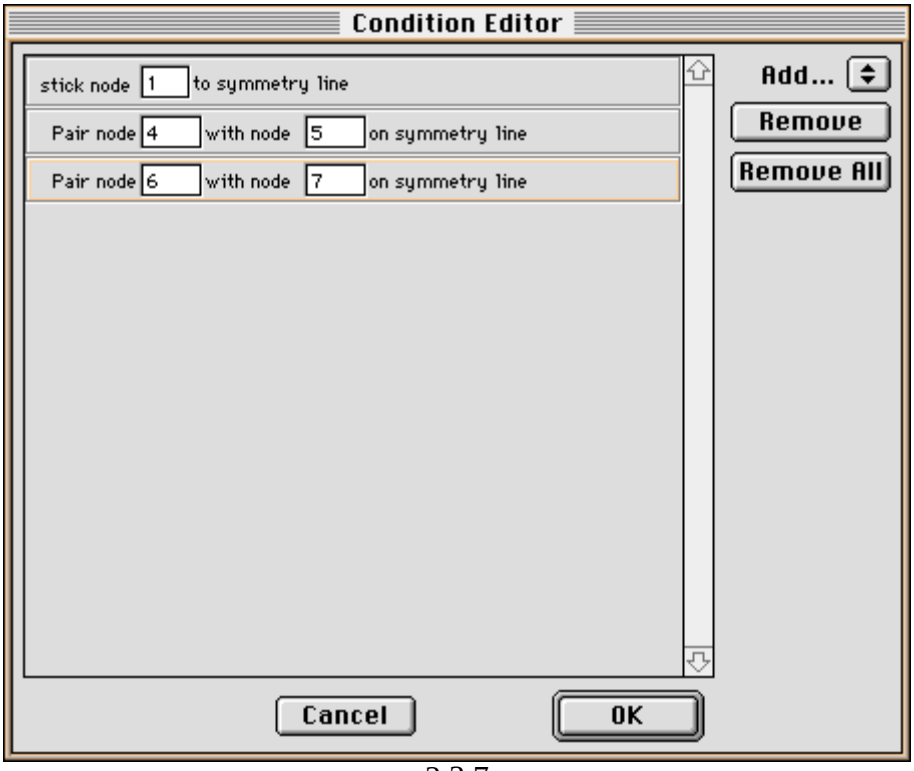
2.3.5

The Condition Editor shows every condition that applies to every part. With it, you can create all three conditions within the same dialog. Select **Node symmetric** from the "Add…" popup menu, followed by selecting **Nodes paired about symmetry line** twice from the same popup.



2.3.6

You will have created three blank conditions. Fill in the fields with the indices of the nodes (1, 4, 5, 6, and 7, respectively):



2.3.7

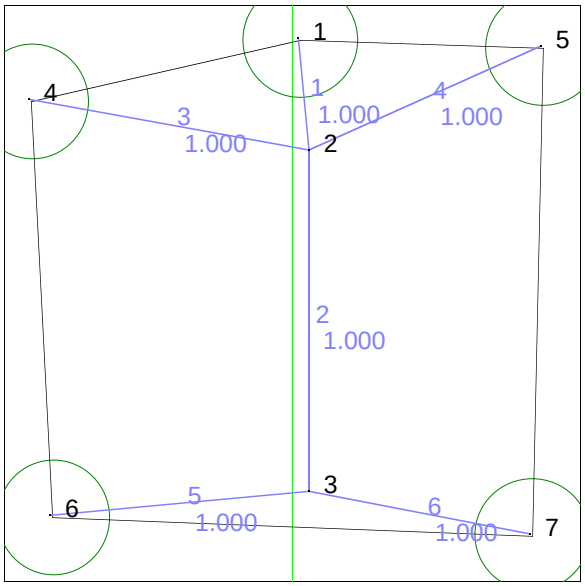
Then click OK. The node circles will have turned green, indicating that the nodes have conditions on them, as shown in figure 2.3.8.

> •Note: if you make a mistake, or for example, accidentally create the wrong type of condition, select the condition by clicking on it, and then click the "Remove" button. If you want to start over, click the "Remove All" button.

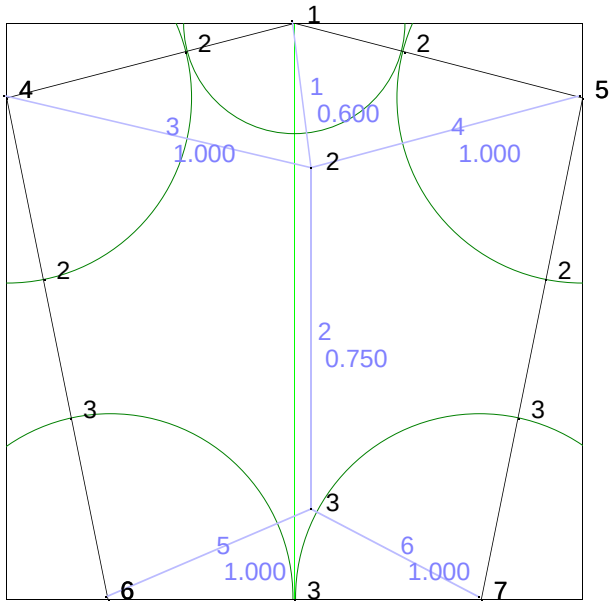
Now we'll optimize to find a base. Select **Optimize Scale** from the **Action** menu. When the run is complete, you should have something that looks like figure 2.3.9.

The scale of this is 0.3388 and you can generate the crease pattern if you like by selecting **Build Polys and Creases**. However, it's often possible to find a different crease pattern with possibly a larger scale (which gives a larger base) for exactly the same tree and conditions. You can find other patterns by starting from a different initial configuration of nodes.

In particular, whenever you have a large, many-sided polygon as we do here, it's often possible to get a larger crease pattern if you drag one of the nodes inside the polygon. In this case, node 1 is the obvious candidate. If you try to click and drag directly, node 1 won't move because it's pinned, but you can always option-click and drag even a pinned node. So, option-click on node 1 and drag it down into the middle of the crease pattern, as shown in figure 2.3.10.



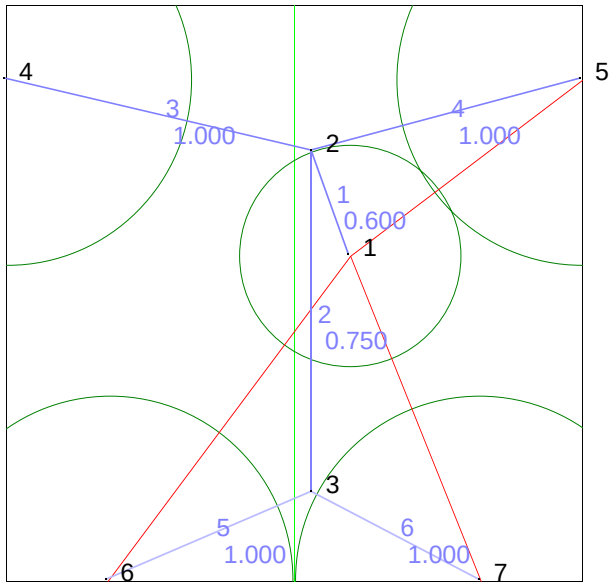
2.3.8



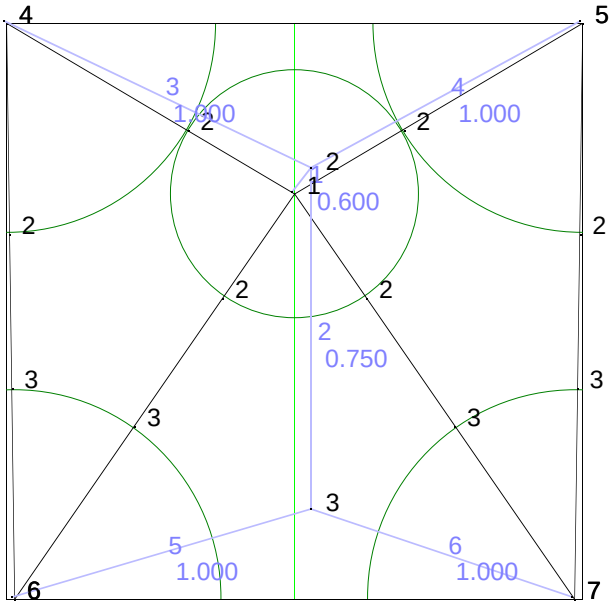
2.3.9

Your figure should look something like this but it probably won't look exactly the same. You should see some red lines, however. The red lines are paths that are not valid, meaning that their actual length is less than their minimum length. They are a sign that no valid crease pattern is possible with this arrangement of nodes.

Again, select **Optimize Scale** from the **Action** menu to find a new configuration. The result should look like figure 2.3.11.

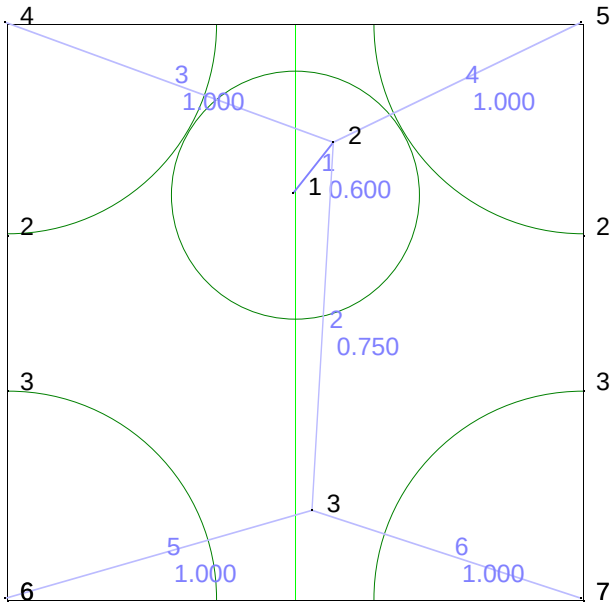


2.3.10



2.3.11

The new scale is 0.36367, somewhat larger; so this configuration is more efficient than the last. However, it's not very elegant, because nodes 6 and 7 are close to a corner but are not precisely in the corner. Option-drag each of those two nodes into the corner as shown in figure 2.3.12.



2.3.12

> •Note: you can put a node exactly into the corner by dragging it slightly beyond the corner. If you drag a node beyond an edge, when you let go, *TreeMaker* will snap the node to the nearest edge. Drag it beyond a corner and *TreeMaker* will snap it into the corner.

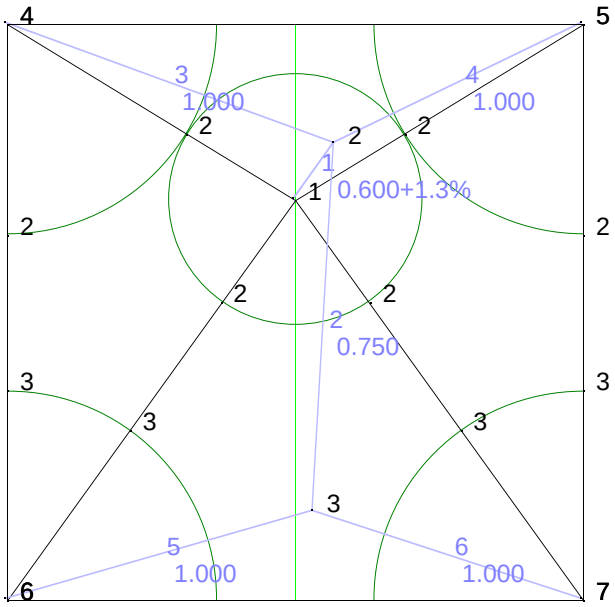
Now select **Build Polys and Creases** from the **Action** menu. Nothing happens. What's wrong?

TreeMaker cannot build polygons or creases if there are *unpinned*, or movable, terminal nodes that are inside what would be a polygon. What do we mean by *pinned*? A pinned node is one that cannot be moved in any direction without either violating a path constraint or going outside the square.

Nodes aren't the only things that are pinned; an edge whose length cannot be made larger without causing some path to be violated is also said to be pinned. A pinned edge is shown in a lighter blue than an unpinned edge (initially, all edges are unpinned). Looking at figure 2.3.12, you can see that all edges except edge number 1, and all terminal nodes except node 1, are pinned.

An edge that isn't pinned can be made longer without reducing the lengths of other edges, which means that the flap in the base that corresponds to the edge can be made longer. Sometimes lengthening a flap of the base is esthetically acceptable; sometimes it isn't. For this example, if we lengthen the head flap, we'll just have a bit more paper with which to make hair, facial features, et cetera. So we can, and will, length edge number 1.

But how long should we make edge 1? *TreeMaker* will find that out for you. Click once on edge 1 and shift-click on node 1 to select them both. Then go to the **Action** menu and select the command **Scale Selection**. This command will move any unpinned selected nodes in order to maximize the length of the selected edge or edges. (If you have selected more than one edge, they will all be lengthened proportionally.) When the optimization ends, you should have something like figure 2.3.12.



2.3.12

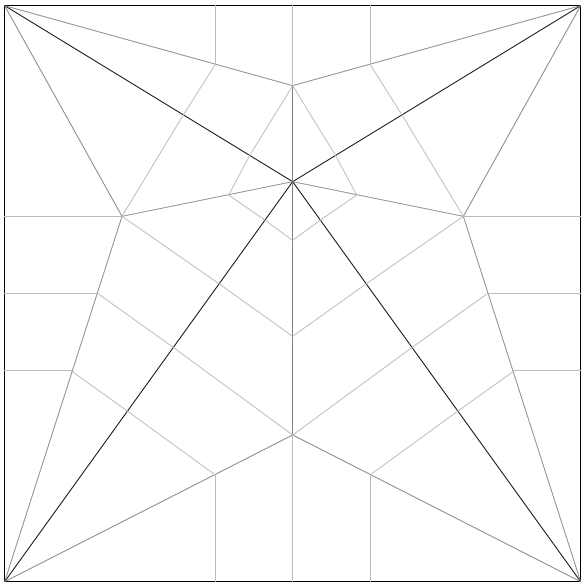
• Note: you must select both some unpinned nodes to move and some unpinned edges to enlarge; otherwise, nothing happens because *TreeMaker* simply ignores pinned nodes and edges. If you want all unpinned nodes and unpinned edges to be considered, just hit command-A (**Select All**) to select everything; *TreeMaker* will weed out the pinned nodes and edges before running the optimization.

Look closely at the length of edge 1: its length is "0.600+1.3%". It is said to be *strained*. Strain is a deviation of an edge from its original desired value.

Strain is an important concept, particularly when we start imposing lots of symmetry conditions on a base. A strained edge behaves as if it were actually longer or shorter. However, an edge "remembers" its original ("unstrained") length, a behavior that we utilize elsewhere.

There are two strain-related commands in the **Action** menu: **Remove strain** sets all strain to zero; it undoes any strain changes. **Relieve strain** does something quite different: it absorbs length changes due to strain into the edge and resets the strain to zero. Select **Relieve strain** now. You'll see that the length of edge 1 has now changed from 0.6 to 0.608.

Now, all edges are pinned and we can proceed with the construction of the creas pattern. Select **Build Polys and Creases** from the **Action** menu to construct the crease pattern and select **Creases Only** from the **View** menu to show just the creases, as shown in figure 2.3.13.



2.3.13

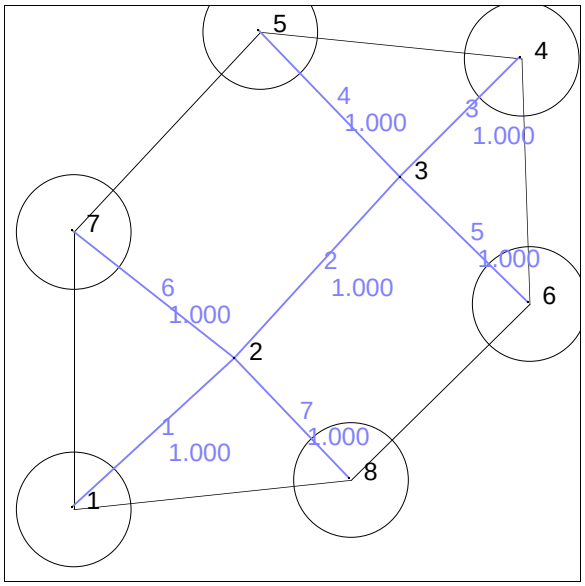
You now know how to set up a crease pattern for a base with an arbitrary number of flaps and how to change the lengths of the flaps. You've also seen in this tutorial how to set up and enforce bilateral symmetry in the base. This gives you 90% of what you need to know to compute crease patterns with *TreeMaker*, and I encourage you at this point to go off and experiment. To keep things simple (and to keep the optimization times down), I have kept to small examples with a small number of flaps. But you can try much more complicated bases with much more complicated assemblies of flaps with *TreeMaker*; you are limited only by the computer memory and speed (and of course, your own patience!). The next section will address some of the subtler issues of origami design using *TreeMaker*. You can continue working through it, or go off, experiment, and then come back when you're ready.

3.0 Tips and Techniques

This section describes several tips and techniques that will improve your productivity when working with *TreeMaker*.

3.1 Adding nodes

Once you have defined a tree and found an optimum configuration of nodes, you have several options for filling in the remaining creases. In this section, I'll demonstrate several different treaments of the same tree. Construct a "generic mammal" tree with equal-length head, tail, four legs, and a body, as shown in figure 3.1.1.

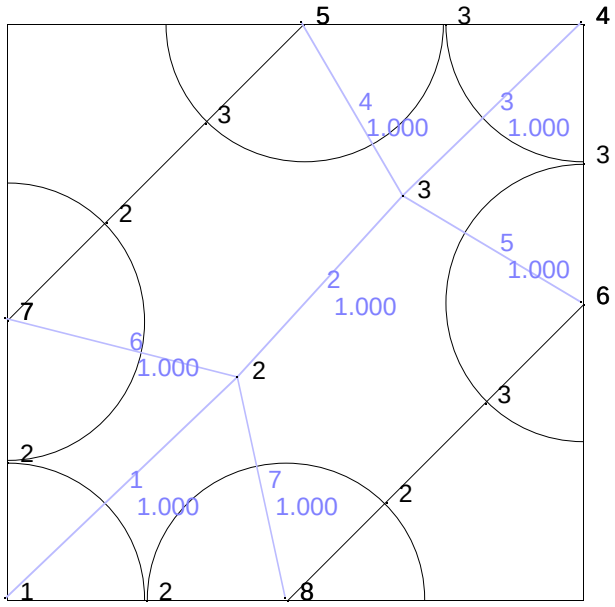


3.1.1

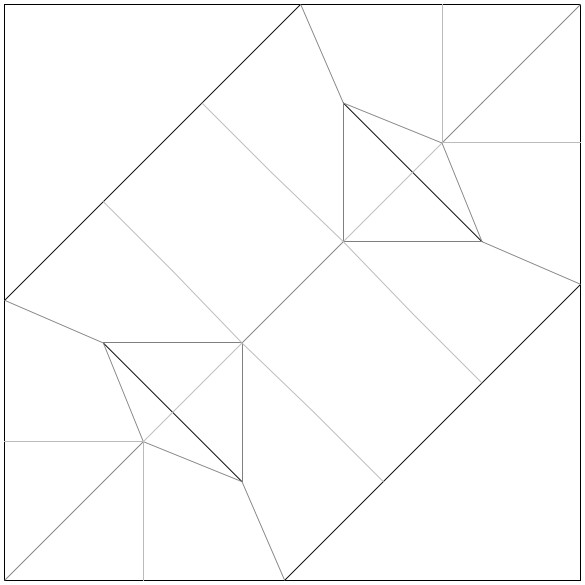
Select command **Optimize Scale** from the **Action** menu; you should arrive at a distribution of terminal nodes as shown in figure 3.1.2.

This crease pattern gives a single active polygon, which is a hexagon. Now, as I said, there are several ways to add creases to this pattern to complete the base. The simplest is the one we have already learned; simply select **Build Polys and Creases** from the **Action** menu. This selection fills in the active polygon with the "universal molecule" crease pattern. This is the simplest pattern and produces the least total crease length.

Although the universal molecule is the simplest pattern, it produces a base with wide flaps that must be repeatedly sunk in and out to narrow them.



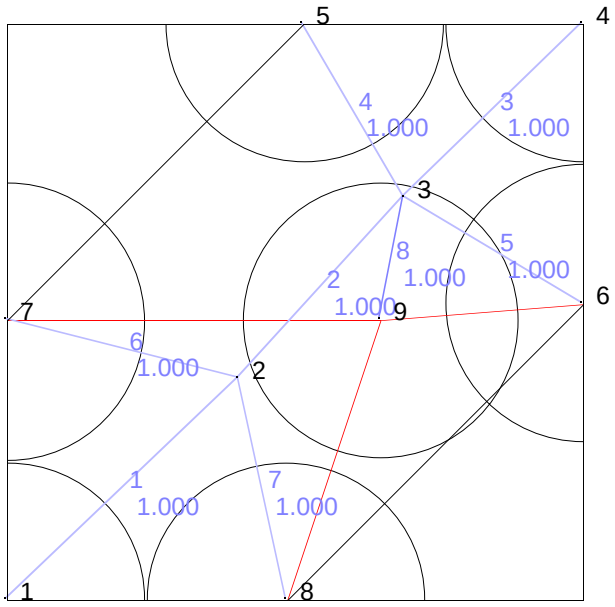
3.1.2



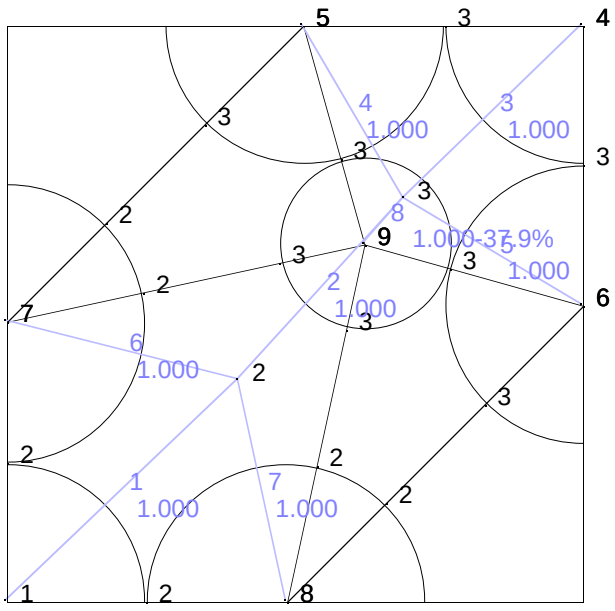
3.1.3

It is possible to narrow flaps by adding new nodes to the tree and inflating them to break up the active polygon. For example, in the tree shown above, if you add a point to node 3, as shown in figure 3.1.4,

and then, making sure both the new edge and node are selected, choose **Optimize Selection**. This will expand the new edge to its limit, creating new active paths that effectively break up the hexagon into smaller active polygons — in this case, two triangles and two quadrilaterals, as shown in figure 3.1.5.



3.1.4



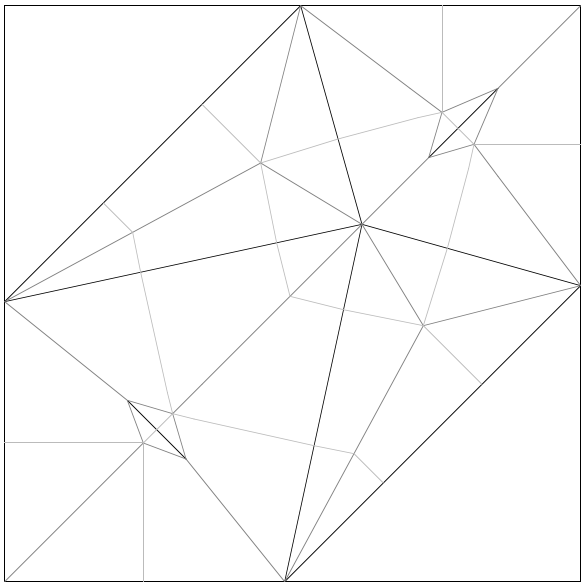
3.1.5

Building creases gives the crease pattern shown in figure 3.1.6.

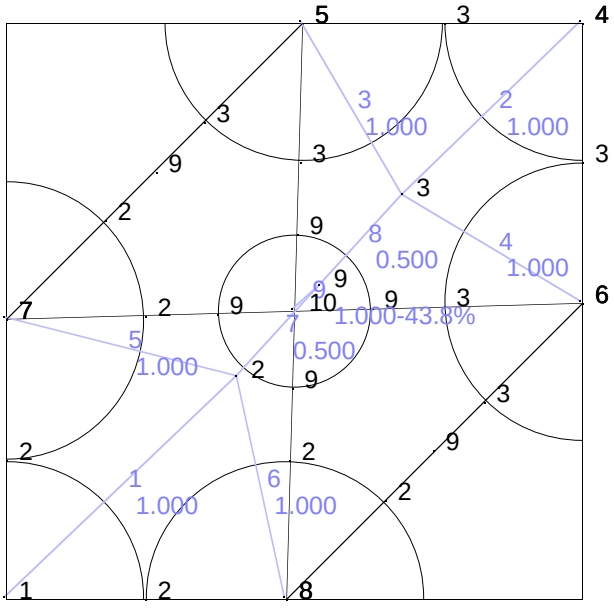
Alternatively, you could have added the new edge to node 2, which would give you the same crease pattern flopped end-for-end.

With either of these two patterns, you get a crease pattern in which the “head” and “tail” are folded differently. For reasons of symmetry, you might want the head and tail to be symmetric, in which case, if you add a new edge, you need to put it in the middle of edge 2.

A third possibility is to split edge 2, using the **Split selected edge** command. If you split the edge, add a new node and edge, and then optimize the length of the new edge, you will get the node arrangement shown in figure 3.1.7.

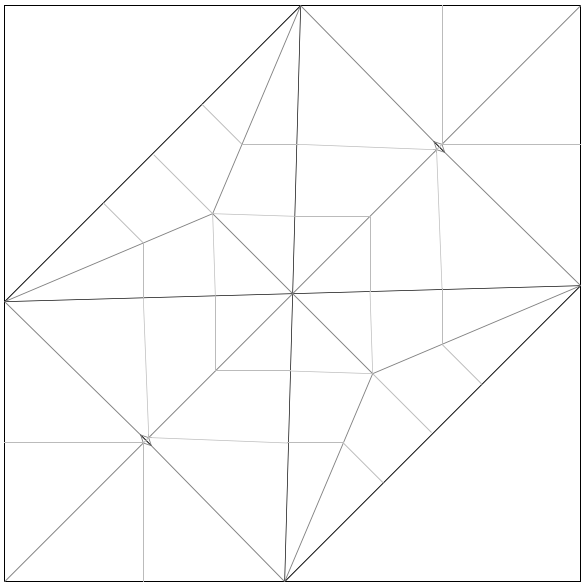


3.1.6



3.1.7

This yields the crease pattern shown in figure 3.1.8.



3.1.8

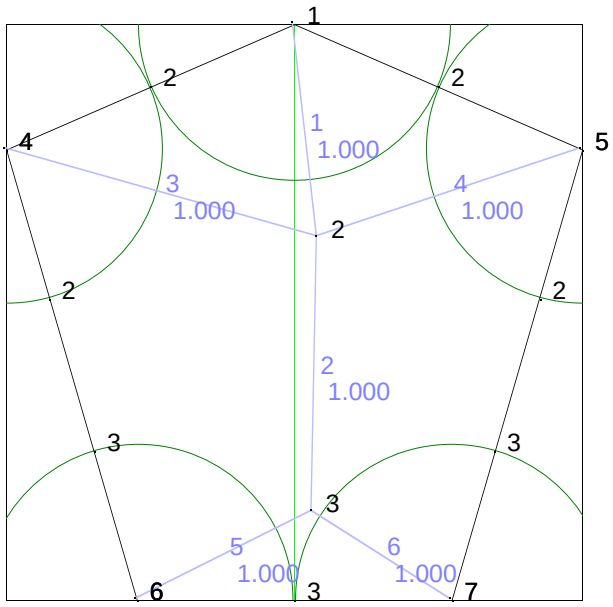
Suppose after seeing the result, we didn't want to do this after all. You can remove the node by deleting node 10. But you will still be left with an extra node in the middle of the model. A node with only two edges connected is called a "redundant node." You can remove a single redundant node by selecting it and choosing the command **Absorb Selected Node** from the **Action** menu. Alternatively, you can select the command **Absorb Redundant Nodes** to remove all redundant nodes from a tree.

3.2 Making a plan view model

Flat origami models come in two distinct types, which correspond to two different views of the subject. The most common is a "side view" model, which resembles the silhouette of the subject viewed from the side. (You can often make a side view model three-dimensional by opening out the body from the underside, but the base is still fundamentally side view.) Using *TreeMaker*, it is very easy to make side view models.

However, many models — especially insects — look better in "plan view," that is, viewed as if you were looking down on them from above. Some of the time, you can make a plan view model out of a side view base by spreading the layers of the model to the left and right and flattening the result. But oftentimes, the base has an odd number of layers, or the layers don't spread sufficiently far apart, or it is necessary to spread-sink one or more corners in a way that ends up reducing the length of the attached points. For example, let's take the regular human tree (equal-length head, arms, legs, and body) oriented along the book direction.

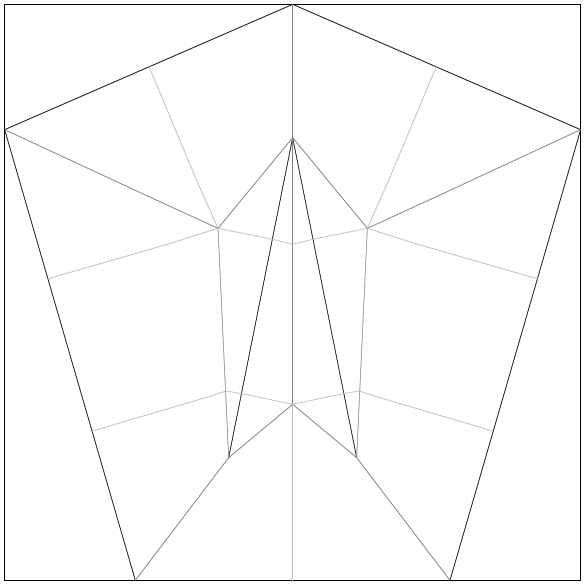
If you set up the tree and symmetry line and run an optimization, you will arrive at the arrangement of nodes shown in figure 3.2.1.



3.2.1

Select **Build Polygons and Creases** from the **Action** menu, followed by **Creases Only** from the **View** menu to get the crease pattern shown in figure 3.2.2.

Now the problem with this crease pattern is that if you fold it up into a side view base, there is only one layer on the top, and if you try to squash-fold the layer to spread the arms and legs to either side and make a plan view base, you'll find that the base of the squash fold reduces the effective length of the legs considerably.



3.2.2

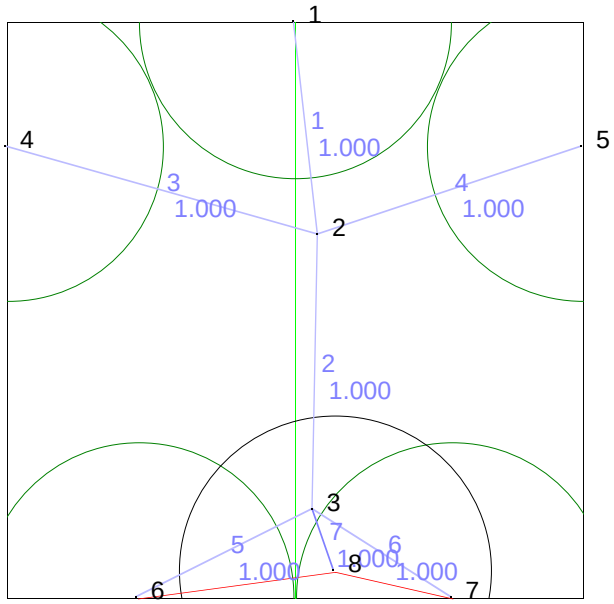
That's because we have overlooked an important requirement for making a plan view base. The way you make a side view base into a plan view base is to open the model along the line of symmetry, which means that all of the paper along the line of symmetry must lie in the same plane to form the "hinge" of the base. Put differently, the line of symmetry of the paper must lie in the plane of projection of the base. This means that to make a plan view base, *the line of symmetry of the square must consist entirely of active paths* (since active paths are the only parts of the base that lie in the plane of projection of the base).

So we need to get active paths running along the line of symmetry. But since active paths only run between nodes of the tree and there is currently only one node on the line of symmetry (the head), we need to add more nodes to the tree solely for the purpose of creating active paths. These nodes don't correspond to features of the model; they're only there to make the base a plan view base. In particular, we need to add a node to the bottom of the square so that there is a terminus on the bottom edge of the square for the active paths that are to run along the line of symmetry.

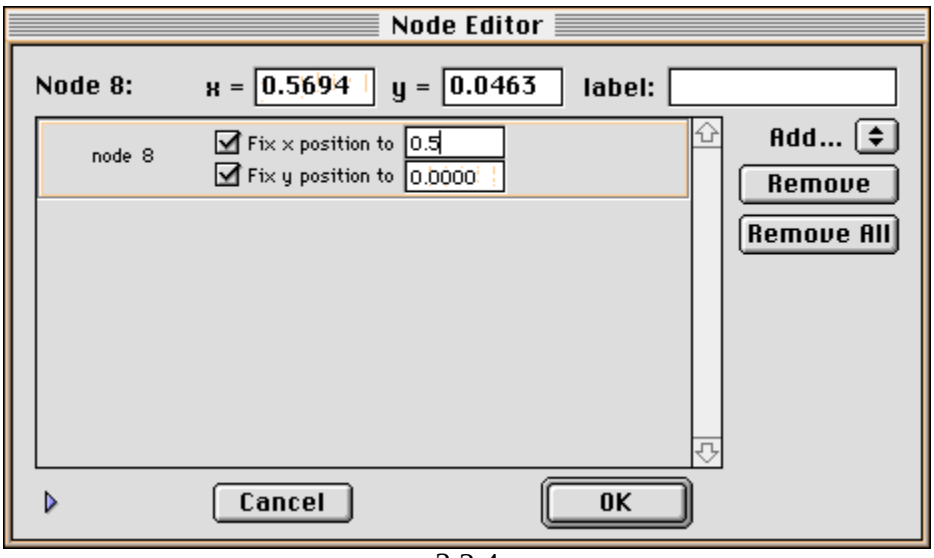
Now any time we add a new node, it brings with it a node circle that cannot overlap any other node circle. In the current configuration, there is no room between node circles 1 and 3 to add a new circle. So we will have to add a new node and re-optimize in order to make room for the new node. This is a very important point: there is no way to modify the original configuration of nodes to make a plan view base without rearranging the entire pattern of nodes!

So, we'll add a new node attached to node 3. We'll want the node to be located on the centerline between nodes 6 and 7. We'll make this forcing using conditions. There are a couple of ways we could do it: we could apply a **Node fixed to symmetry line** and **Node fixed to edge** condition to the same node; but since we know exactly where the node should lie, we can just use a **Node fixed to position** condition. So, create a new node attached to node 3 as shown in figure 3.2.3.

Then double-click on the node to bring up the Node Editor. Apply a **Node fixed to position** condition, affixing the node to location x=0.5, y=0 (turn on both checkboxes).

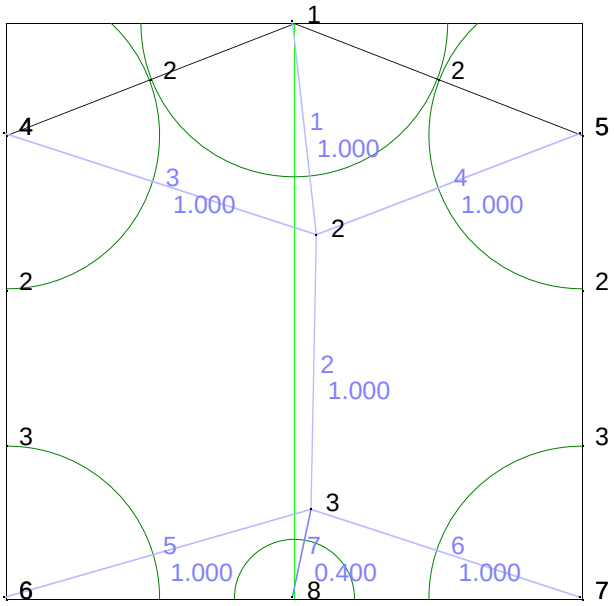


3.2.3



3.2.4

There is still the question of how long to make the associated edge. For now, give it a length of 0.4 — we'll discuss how length matters in a moment. After you re-run the optimization (**Optimize Scale**), you should get a modified arrangement of node circles shown in figure 3.2.5.



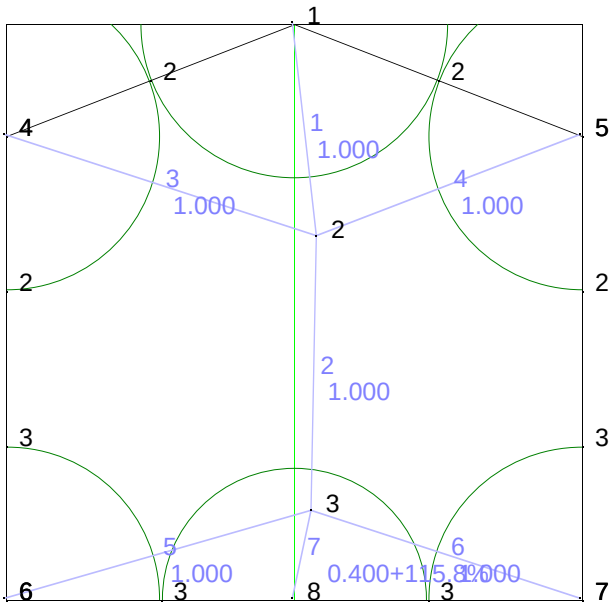
3.2.5

The new node — node 8 — is now on the bottom of the square. The darker blue of its edge tells you that it isn't pinned. Choose **Select all** from the **Edit** menu, then **Optimize selection** from the **Action** menu. This will expand the edge to its maximum possible length.

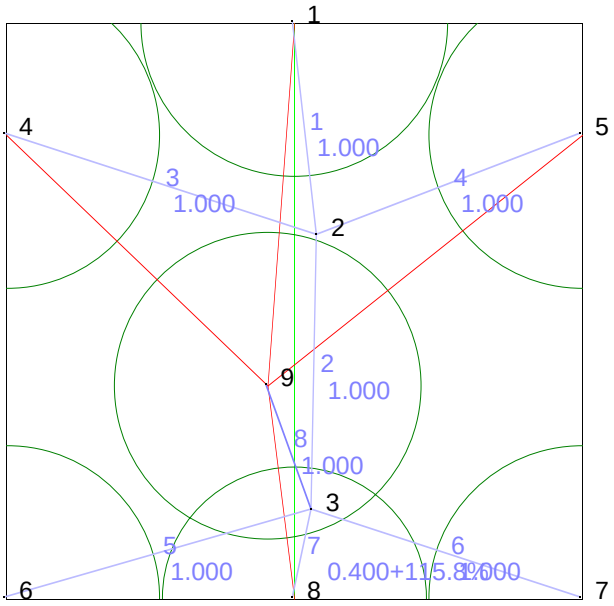
So now we have nodes 1 and 8 at the top and bottom of the symmetry line — but there is still no active path between them.

The reason is that the distance between nodes 1 and 8 is larger than the minimum allowed path length between them. To get active paths in place, we need to add another node to the symmetry line somewhere between 1 and 8 to "soak up" this excess length. The longer the new node, the more excess length it will require. For this new node, we don't know how large it must be, so we'll use the **Optimize Selection** command to optimize its size. Add a new node connected to node 3 and apply a **Stick to symmetry line** condition (which by now, you know how to do). The result is shown in figure 3.2.7.

Then select both the node and edge and select **Optimize Selection** from the **Action** menu to optimize the position of the node and the length of the edge.



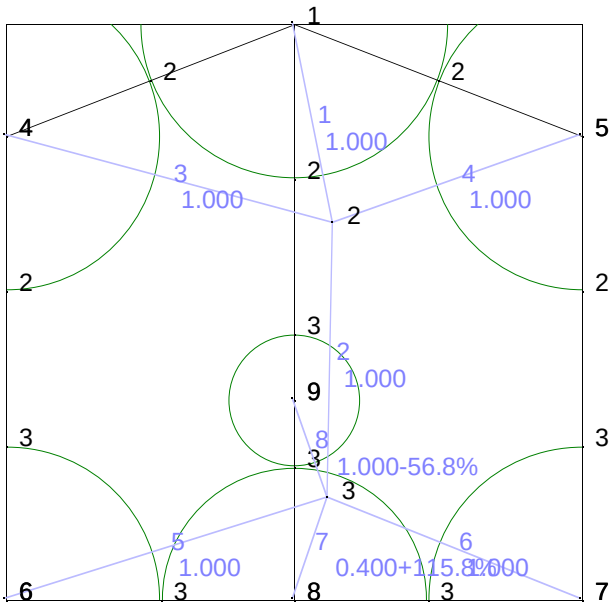
3.2.6



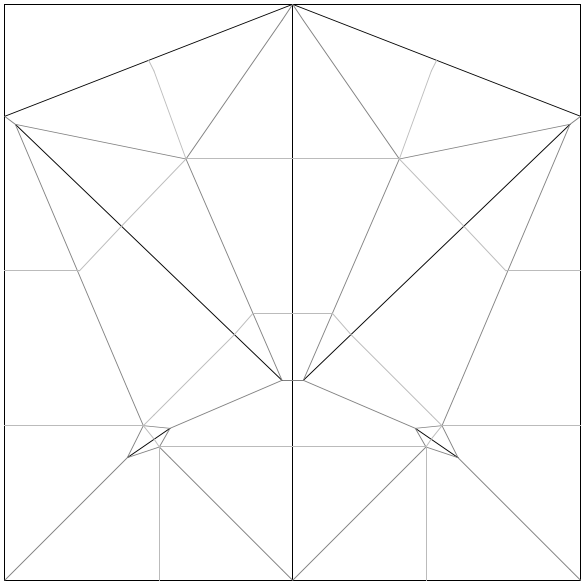
3.2.7

The result is shown in figure 3.2.8. Now there are active paths between nodes 6 and 9 and between nodes 9 and 8, filling the entire symmetry line. Thus, the condition for making a plan view model is satisfied.

Select **Build Polys and Creases** to complete the pattern and **Creases Only** to display it. The crease pattern is shown in figure 3.2.9. And if you fold this pattern into a base, you will indeed get a plan view base.



3.2.8



3.2.9

When adding the new node to the crease pattern, we didn't necessarily have to add it to node 2, which put the extra point between the legs of the base; we could have added it to node 4, which would put the extra point at the shoulders of the base. If you do that and run the optimization, you'll get the pattern shown in figure 3.2.10.

This gives a different crease pattern, shown in figure 3.2.11. This crease pattern is entirely different: yet it results in a plan view base with exactly the same number and size of flaps as the first example.

In both cases, the extra flap "sops up" excess paper between the top and bottom of the model. The difference between the two is that in the first example, the extra paper emanates from the junction of the top group of points. In the second, the excess flap emanates from the bottom group of points.

There's another possibility: we could collect the excess paper in the middle of the model. This would be preferable because it distributes layers more evenly through the model. We can do this by adding the new flap in the middle of edge 2.

To do this, delete the new node you just added. Then select edge 2. Choose the command **Split selected edge…** from the **Action** menu, which puts up the dialog shown in figure 3.2.12.

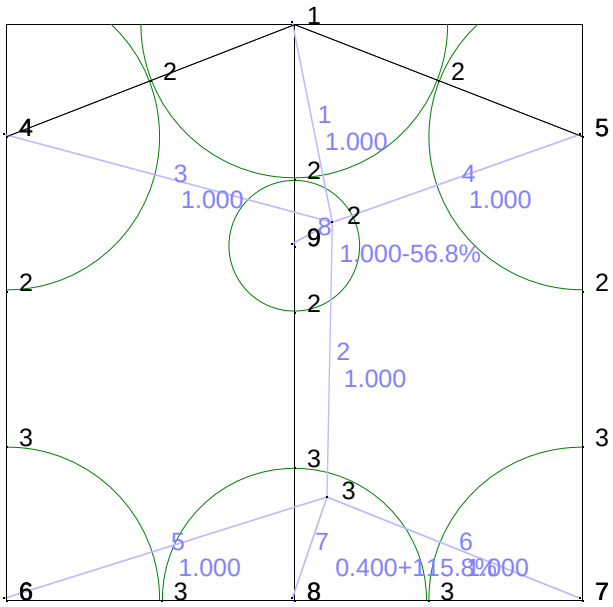
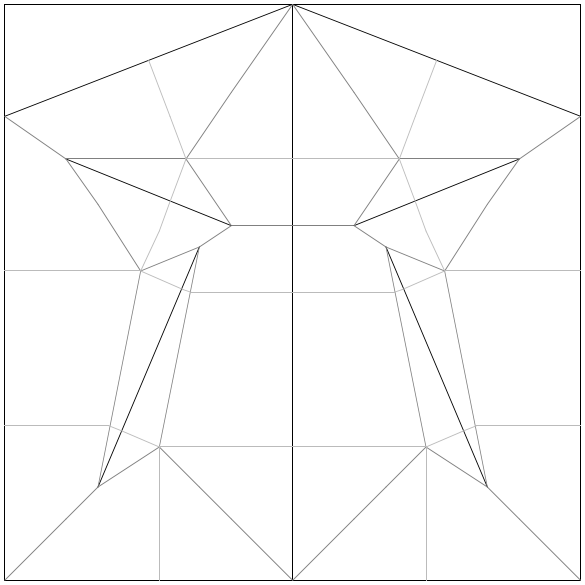
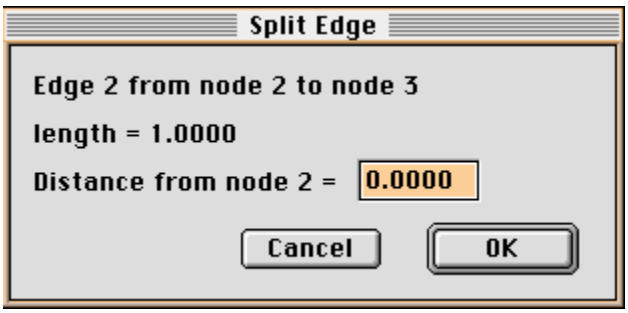


Figure 3.2.10

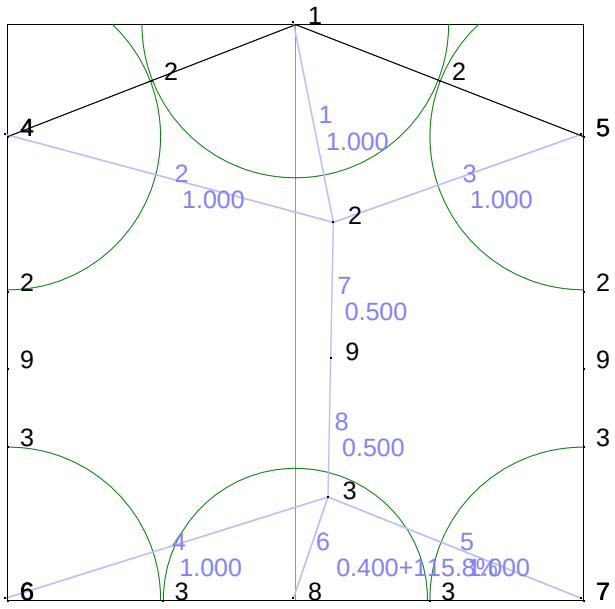


3.2.11



3.2.12

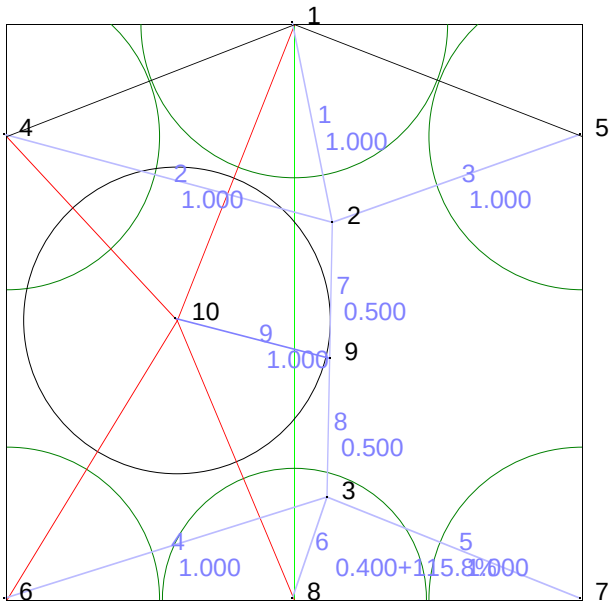
This lets you add a node in the middle of the edge (or anywhere along it). Enter 0.5 in the field (to split the edge at its middle) and click OK. There is now a new node, node 9 in the middle of the edge, as shown in figure 3.2.13.



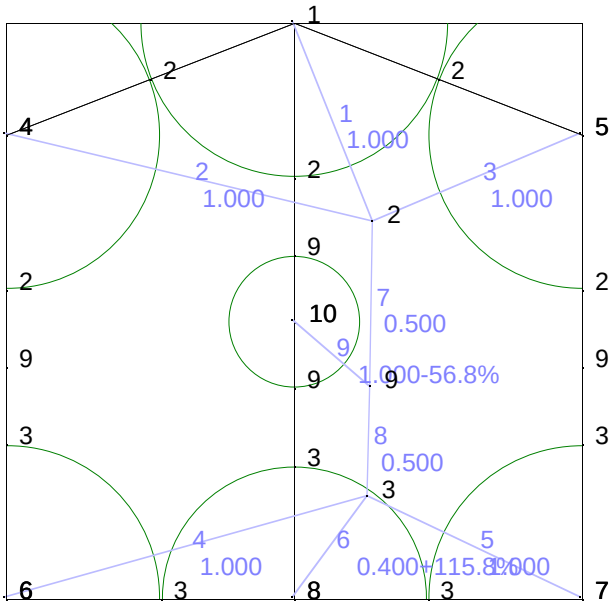
3.2.13

Click on node 9 to highlight it. Then click next to it to create a new node and edge attached at node 9 as shown in figure 3.2.14.

Apply a **Node fixed to symmetry line** condition to node 10, then select edge 9 and node 10 and choose the **Optimize selection** command from the **Action** menu. Again, you will create an active path between nodes 1, 8, and the new node, as shown in figure 3.2.15.

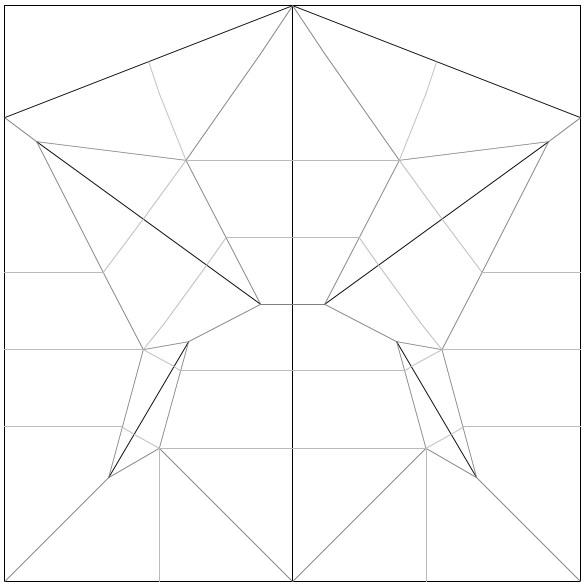


3.2.14



3.2.15

Again, select **Build Polys and Creases** from the **Action** menu, followed by **Creases Only** from the **View** menu to get the crease pattern shown in 3.2.16.



3.2.16

So you see, there are many ways of adding circles to force a plan view base; yet another demonstration that there are many different ways of achieving a base for the same basic design.

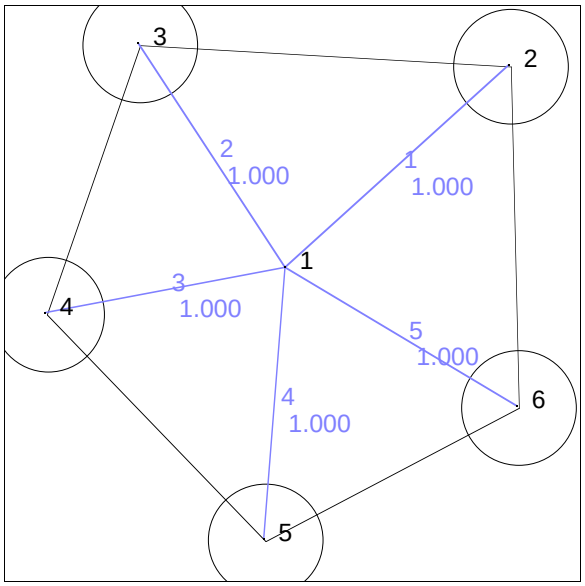
One final note: if you have two circles in the interior of the model that are touching along the center line, you'll have to add a circle between them to force them apart. The way to do this is to use a **Nodes collinear** condition to force the new node to lie between the other two.

3.3 Forcing edge flaps

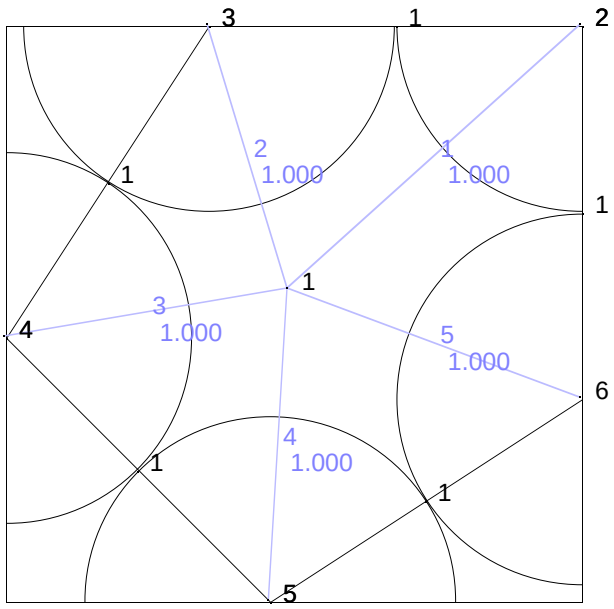
Once you have run several *TreeMaker* designs and folded them, you will eventually come to the realization that some crease patterns are easy to collapse into bases while others are incredibly difficult to fold up. Closer examination of this phenomenon reveals a common thread: the more middle points a base has, the harder it is to fold up. In addition, while edge points can almost always be separated from the rest of the base by simply being folded out to the side (perpendicular to the plane of the the base), middle points are often surrounded by layers of paper and can only be separated out by being reverse-folded up and out of the model (lying in the plane of the base). For these reasons, it is often desirable to force most of the flaps in the base to be edge flaps.

With only 3 or 4 flaps in the tree, they will almost inevitably be edge flaps. However, even with as few as 5 flaps, you will find that one or more flap will tend to move into the middle. For example, suppose we want a base with five equal-length flaps. Construct a tree as shown in figure 3.3.1.

Depending on the initial configuration of nodes, there are two optimum arrangments of node circles, shown in figure 3.3.1. If you start with the initial configuration above, then you will probably wind up with the solution shown in figure 3.3.2, which has a scale of 0.323.



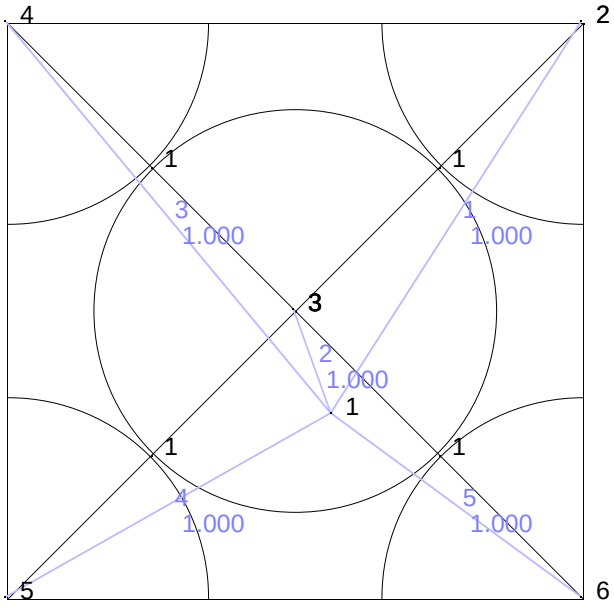
3.3.1



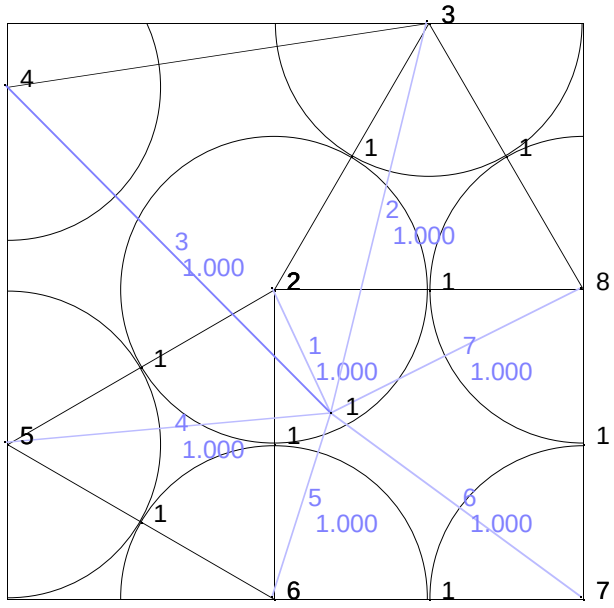
3.3.2

If you allow one of the nodes to start in the middle of the square, you will probably arrive at the solution in figure 3.3.3, which is significantly larger with a scale of 0.353.

In general, for 5 or more flaps, the most efficient crease pattern will have at least one middle flap. In this example, we were able to force all of the flaps to be edge flaps simply by starting from an initial configuration in which all of the terminal nodes were on the edges. This is not always possible, however. For example, if you try the same problem for seven equal-length flaps, there is no initial configuration that will avoid putting a circle into the middle of the paper, creating a middle flap. A typical circle pattern (with a scale of 0.268) is shown in figure 3.3.4.

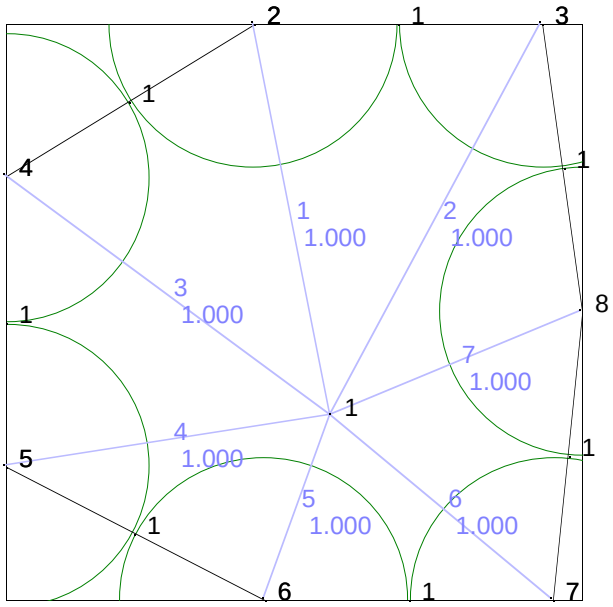


3.3.3



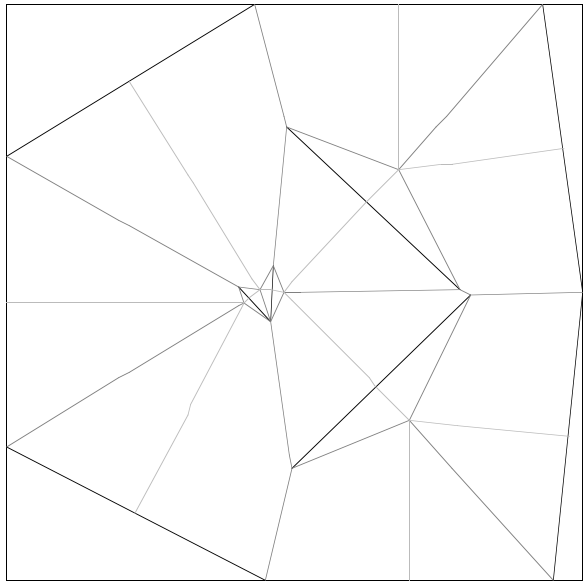
3.3.4

However, we can force nodes to lie on the edge of the paper, thereby turning the associated flaps into edge flaps, by setting conditions on them. Select all the nodes, either by shift-clicking on each in turn, or by choosing the **Select All** command from the **Edit** menu. Then select the **Node fixed to edge** command from the **Condition** menu, which applies this condition to each node in the selection, as shown in figure 3.3.5.



3.3.5

Re-run the optimizer. (Because of a quirk in the optimization code, you may have to re-run it several times sequentially to force convergence.) When you do, you will find that all the nodes now lie on the edge of the paper, essentially forming one giant polygon. Select **Build Polys and Creases** from the **Action** menu and **Creases Only** from the **View** menu. This pattern results in the crease pattern shown in figure 3.3.6.



3.3.6

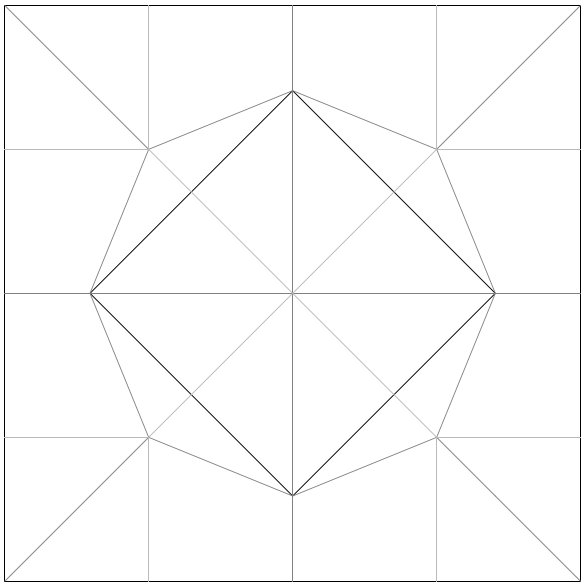
Ease of collapsing the crease pattern into a base is a good reason to force flaps to be edge flaps; another good reason is the number of layers. For a given width flap, edge flaps have only half the number of layers of middle flaps, and so if a flap is to be folded further, it may be desirable to make it be an edge flap to it will be easier to fold.

Another reason to make a flap be an edge flap is for color-changing. The bases produced by *TreeMaker* are inherently single-color. If you are making a model that exploits the colors of both sides of the paper — a bald eagle, for example, which would have white head, tail, and feet — then you will need to make sure that the flaps to be color-changed are edge flaps, so that they may be turned inside-out around the raw edges of the flap. (It is extremely difficult to turn a middle flap inside-out!) Of course, color-changing individual flaps is only the most rudimentary way of exploiting two-tone models and is insufficient for models with stripes or spots, for example. For zebras, tigers, and giraffes, you'll have to use different techniques than those found in *TreeMaker*.

All bases designed by *TreeMaker* are what I call a “uniaxial” base. A uniaxial base is one in which all the flaps lie along a single line. A base in which all of the flaps are edge flaps has the very useful property that the layers can be arranged so that with all of the layers in the same half-plane, no major flap is enclosed inside of another flap. I call a base with this latter property a “simple” uniaxial base. Simple bases can also be designed by the “string-of-beads” algorithm, and it is fairly easy to prove that the perimeter of an edge-flap-only base cannot exceed the perimeter of the square.

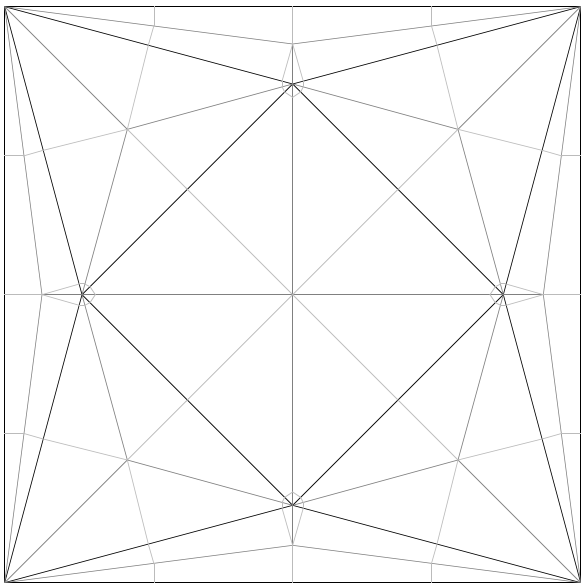
It is tempting to suppose that any simply stacked base must be an edge-flap-only base, but it is often possible to add auxiliary nodes to a base with middle points to make them “quasi-edge-like”. An example will make this clear.

An eight-pointed base in which all of the flaps are edge flaps is shown in figure 3.3.7. It has a scale of 0.250. If we don’t force all the points to be edge flaps, we’ll find the arrangement shown in figure 3.3.8, in which four of the points have become middle flaps.



3.3.7

Although four of the flaps are middle flaps and will be trapped inside layers, it's possible to squash-fold the raw-edged flaps attached to each node in such a way that a simply-stacked base is obtained. And clearly, the perimeter of this base exceeds the perimeter of the square by the factor (sec(15°)), or by about 3.5%.



3.3.8

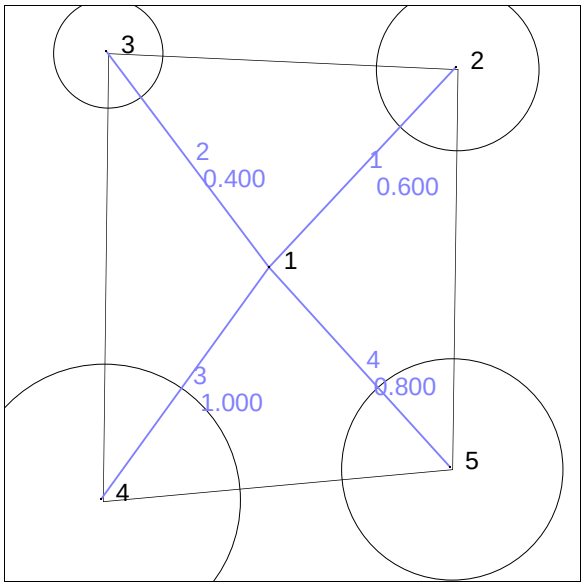
3.4 Fracturing polygons

Once you have completed an optimization, the simplest crease pattern that collapses into a base is produced by choosing **Build Polys and Creases**, which creates all creases using the "universal molecule" algorithm. You can also break up high-order active polygons into lower-order active polygons by adding new edges to existing nodes. When you create a new edge and scale it using **Optimize selection**, it will be expanded until it forms typically three new active paths.

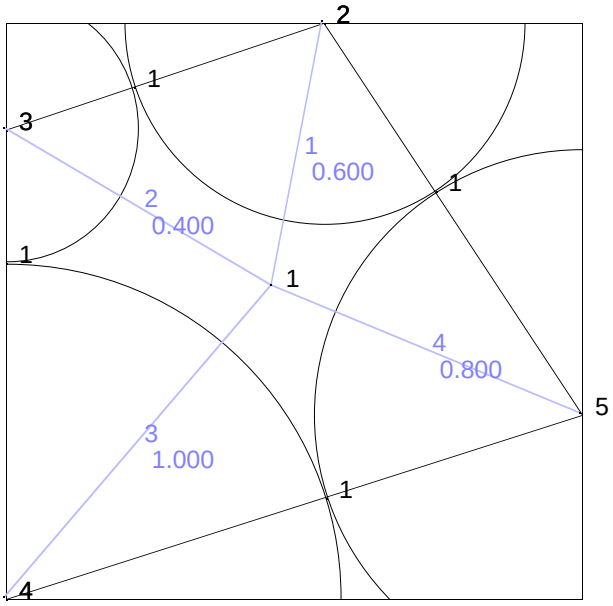
This process will break any polygon of order 5 or higher into polygons of lower order. In general, the lower the order of the active polygon, the simpler it is to fold, so this is often a desirable process. However, if you try to break up a polygon of order 4 (a quadrilateral), unless it happens to have a line of symmetry, you will form two triangles and another quadrilateral. Thus, you cannot in general reduce a crease pattern to all triangles simply by adding new edges to existing nodes.

As an example, construct the tree shown in figure 3.4.1 that has four legs of different lengths.

When you optimize this tree, you will get an irregular quadrilateral. Building creases gives you a “gusset quadrilateral” as shown in figures 3.4.2 and 3.4.3.

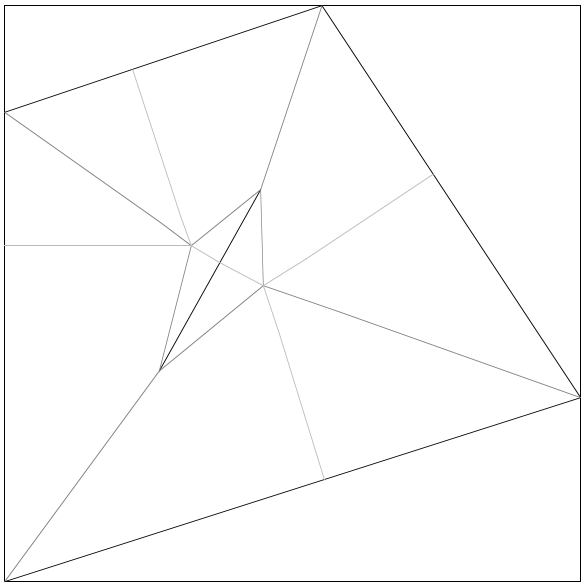


3.4.1

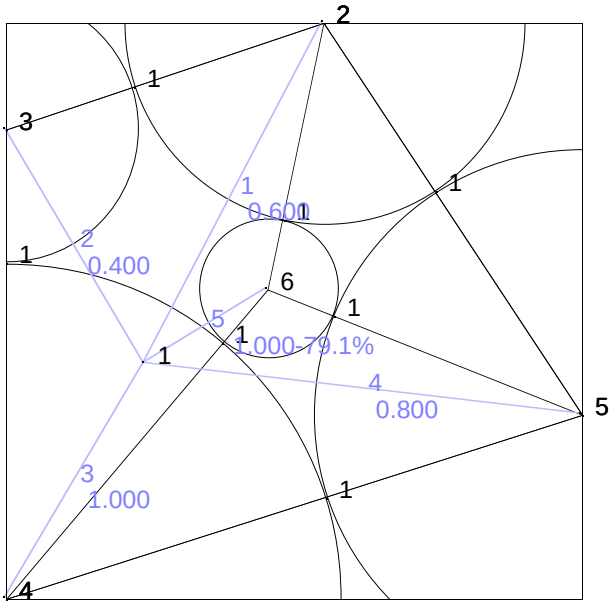


3.4.2

If you add a new edge to node 1 and expand it, you'll get something like the polygon pattern shown in figure 3.4.4. Indeed, the quad has been broken into two triangles — and another quad.

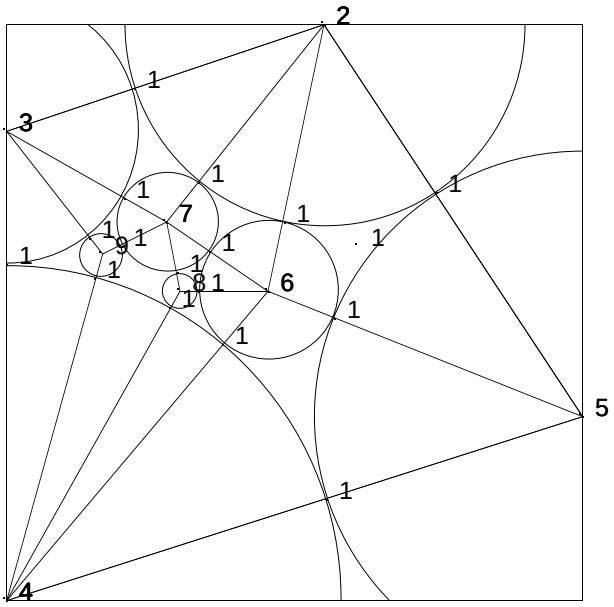


3.4.3



3.4.4

The process can be repeated essentially infinitely; you can make the quad smaller and smaller, but you'll never get rid of it! Figure 3.4.5 continues the process, but there is still one quadrilateral.



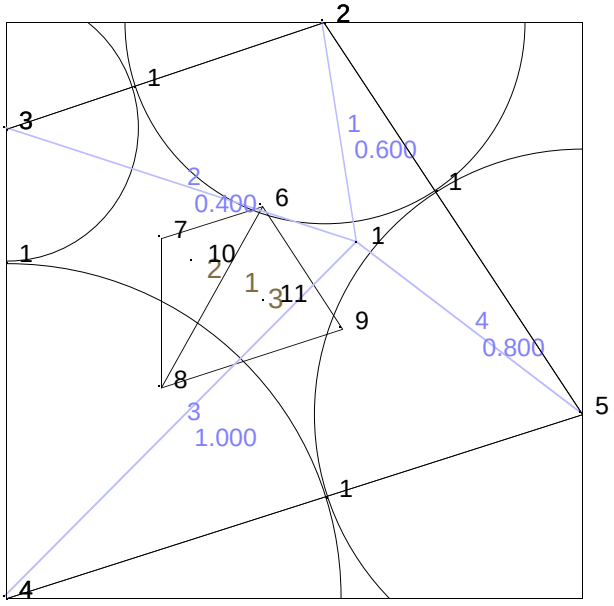
3.4.5

There is a fundamental reason for this limitation: breaking up an arbitrary quad into four triangles requires the formation of *four* active paths. But when you add a new edge to an existing node, you only have *three* degrees of freedom: the *x* and *y* coordinates of the new node and the length of the edge. To create four active paths, you need a fourth degree of freedom.

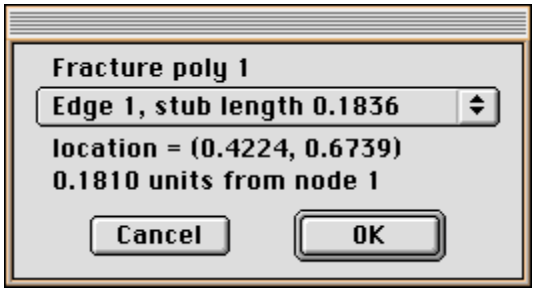
If we add the new edge to a newly-created node formed by splitting an edge, we have a fourth degree of freedom, namely, the location of the split along the edge. There is a special command for adding a node that forms four active paths by splitting an edge; it is the **Fracture Poly…** command in the **Action** menu. This command lets you break up an active polygon by splitting one of the edges in such a way as to form *four* active paths.

To use this command, you need to build the polygons and select the one to be fractured. Select the **Build Polys** command from the **Action** menu, as shown in figure 3.4.6.

This builds all of the polygons for the crease pattern. There are two kinds of polygons: "top-level" polygons, which are formed from the nodes of the tree, and "subpolygons," which are smaller polygons located inside of the top-level polygons. In the figure, the largest polygon — the quadrilateral — is a top-level polygon. The two small triangles are sub-polygons, which are fully enclosed by the top-level polygon.



3.4.6



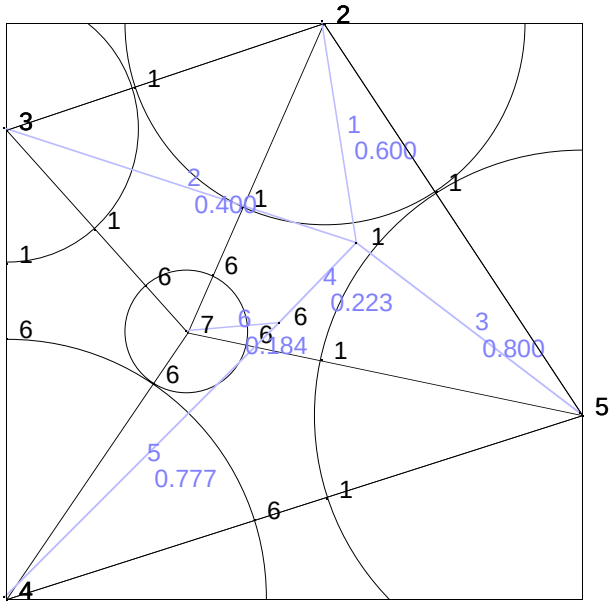
3.4.7

Only top-level polygons can be fractured in this version of *TreeMaker*. Click inside the largest polygon (but outside of the two sub-polygons) to select it. Then select **Fracture Poly…** from the **Action** menu, which puts up the dialog shown in figure 3.4.7.

There is frequently more than one way to fracture a poly. This dialog lists all possibilities in a popup menu, cataloged by which edge is split and what the length of the new edge would be. Below the popup menu is listed the location of the new node and the distance that the split is from one of the nodes along the edge. Choose edge 3 from the popup and click OK.

Enter the index of the polygon to fracture. (Since we only have one active polygon in this figure, its index is 1.) Then click OK. The result is shown in figure 3.4.8.

Observe that the former edge 3 has been split into two with a new node at the junction. A new edge — called a "stub" — has been attached to the node with another new node at its end. The new node makes active paths with all four of the corners of the original quadrilateral, thereby breaking it into four active triangles.

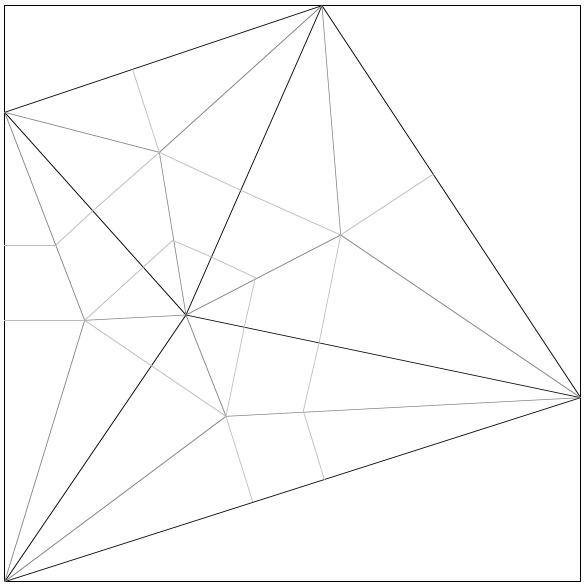


3.4.8

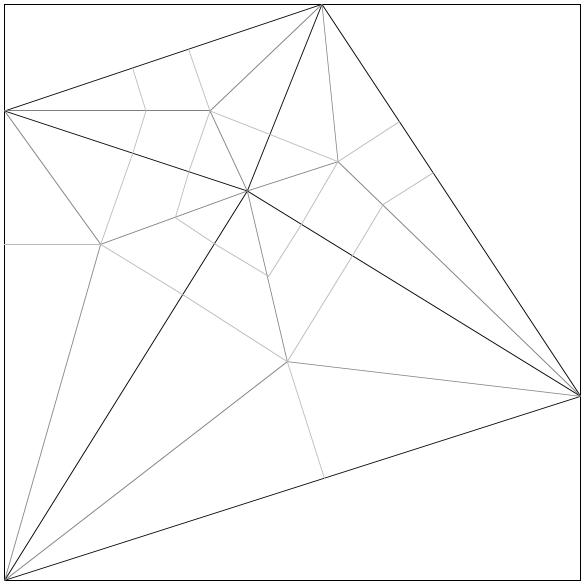
Building the creases gives a distorted Bird Base–like shape shown in figure 3.4.9 whose flaps remain the proper length for the specified tree.

If instead we had chosen to make the stub on edge 1, we would have achieved the crease pattern shown in figure 3.4.10.

Both crease patterns result in the 4 initial flaps coming out with the same length and scale. In both cases, there is an extra flap that corresponds to the stub. The extra flap “soaks up” excess paper within the polygon.

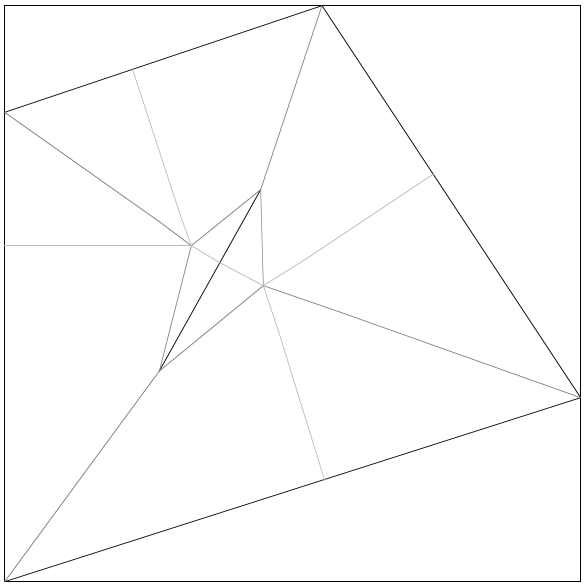


3.4.9



3.4.10

Of course, you can always simply create creases directly, which results in the universal molecule, as shown in figure 3.4.11.



3.4.11

This shows, once again, that there are many ways to get the same number of flaps.

3.5 Forcing symmetric angles

One of the major enhancements in *TreeMaker* 4.0 is that you can force active paths to lie at specific angles. Why would you want to do this?

Once you have computed and folded a number of *TreeMaker* designs, you'll discover that only rarely will the edges of each flap line up with each other. That's because a flap is composed of a bunch of triangles coming together and when *TreeMaker* optimizes a base, there's no particular reason for it to make the angles of one triangle equal to those of the adjacent layers.

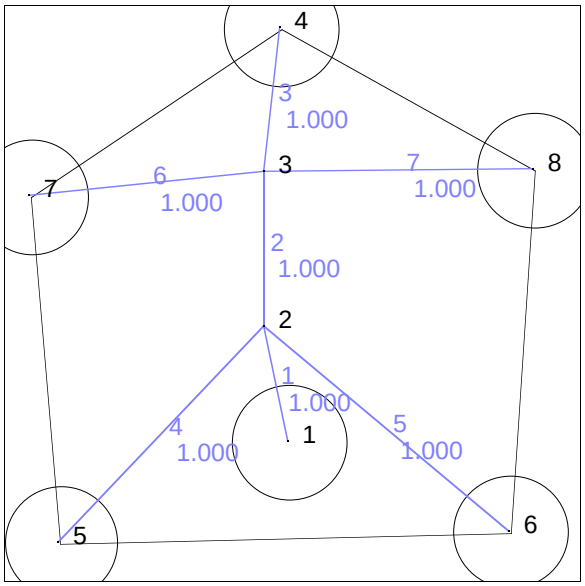
If you could force all angles to be the same, or one of a small set of similar angles, then you'd get a base in which more layers tend to line up.

An additional benefit of this is that if all the major creases run along fairly standard angles, you run a better chance of being able to construct reference points by folding and/or a sequential folding sequence, rather than measuring for reference points and a folding sequence that consists of a single "bring all these creases together at once."

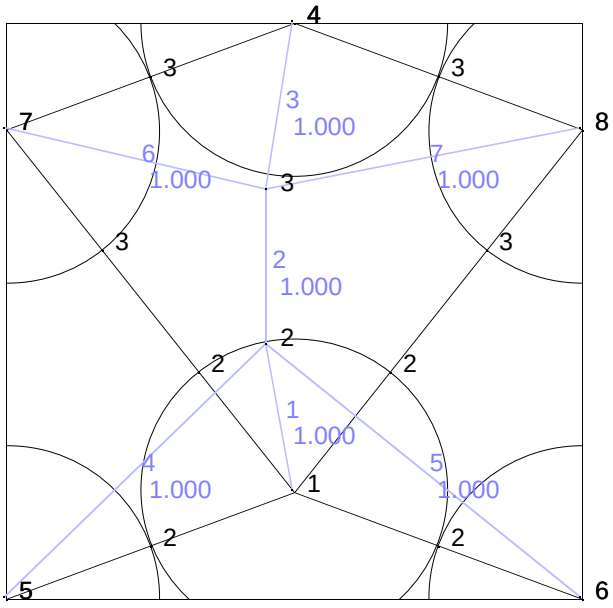
If we are to force major creases to lie at "standard" angles, which angles should we choose? A natural set of angles are those that are obtained by dividing a circle into an integral number of equal divisions: for example, the common 22.5° degree angle found in the Bird Base is obtained by dividing a circle into 16 parts. While this is the most common division, several others provide useful and interesting bases: division into 12 parts gives multiples of 30° and crease patterns with many equilateral triangles in them.

Let's work through an example. Create a new document and construct the six-legged tree as shown in figure 3.5.1.

Now, select **Optimize Scale** from the **Actions** menu to create the first iteration of the design.



3.5.1

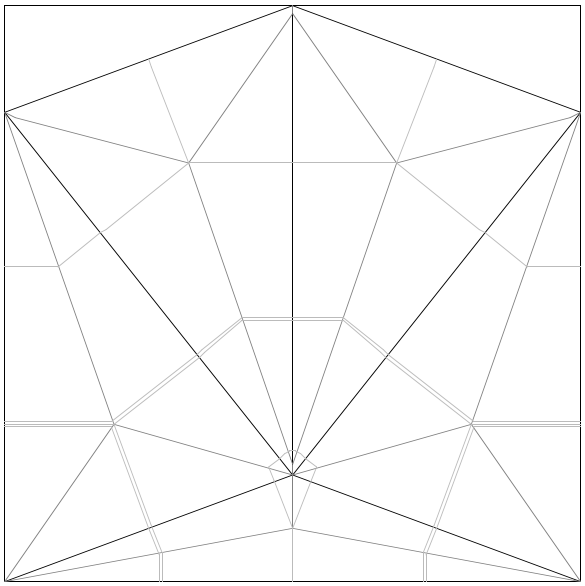


3.5.2

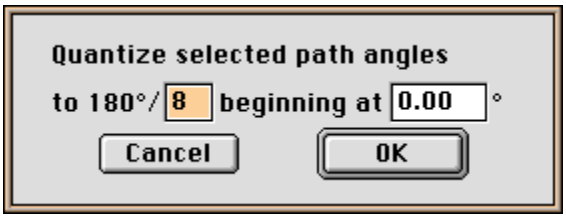
If you **Build Polygons and Creases** you will get the crease pattern shown in figure 3.5.3.

The creases in this design don't precisely fall on standard angles. We'll modify it so that they do. We do this by putting a "Path angle quantized" condition on all of the paths that correspond to major creases.

Put the model back into **Default View** and select the **Kill Polygons** command, **Actions** menu, to put in back into the configuration shown in figure N. In this case, only the polygon paths are shown. Select the **Select all polygon paths** command, **Edit** menu. Then select **Path angle quantized…** from the **Conditions** menu. This puts up the dialog shown in figure 3.5.4.



3.5.3



3.5.4

This will place a condition that forces each of the selected paths to be active (so it will be a major crease) and to lie at an angle that is a multiple of 180°/*N*, where *N* is an integer. Common values of *N* and the corresponding angle are:

N	*angle*
4	45°
6	30°
8	22.5°
12	15°
16	11.25°

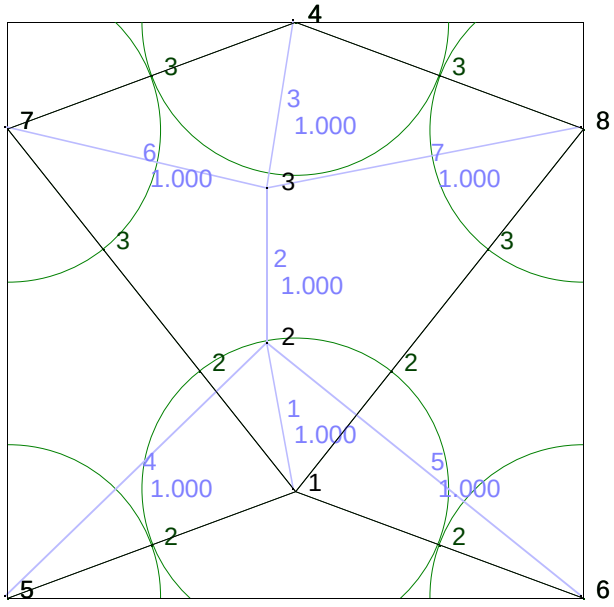
The most common choice is *N*=8, which corresponds to the angles you find in a Bird Base, but there are many other useful possibilities to be found in the other angles, and I have found interesting structures for *N*=10 as well.

We'll work with *N*=8 for now. Type in 8 and click OK.

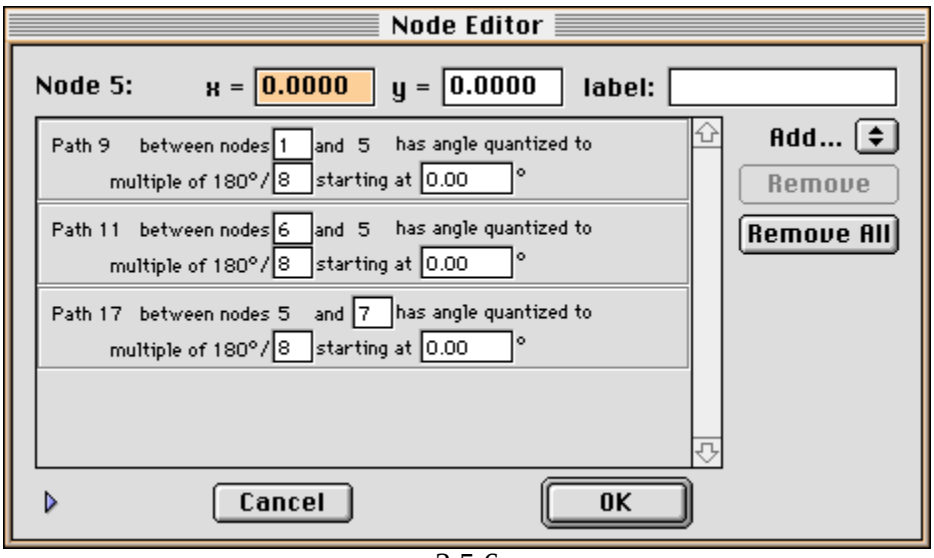
Note that the node circles have turned green, indicating that there's a condition imposed on the nodes (actually, on the paths connected to the nodes). However, both ordinary polygon paths and polygon paths with conditions on them are displayed in black.

In principle, you should be able to re-optimize; however, due to a quirk in *TreeMaker*'s nolinear optimizer, *TreeMaker* has a tendency to settle onto a spurious solution in which the entire crease pattern has shrunk into oblivion. To prevent this tendency, we'll need to stitch some of the nodes to the edges of the square.

The two easiest to do this to are to stick nodes 5 and 6 to their present locations. Double-click on node 5 to bring up the Node Editor.

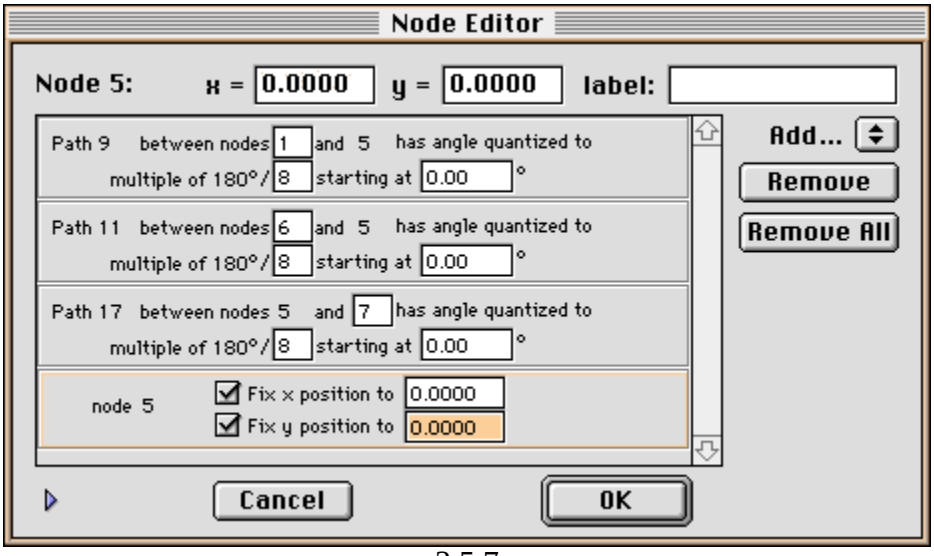


3.5.5



3.5.6

Notice that the node already has three conditions on it — conditions on the paths that emanate from it. From the popup menu, select a "Node fixed to position" condition, and turn on the conditions for both the x and y coordinates, fixing both to the value 0.0000.



3.5.7

Then click on OK.

Repeat on node 6, this time fixing it to x=1.0000, y=0.0000.

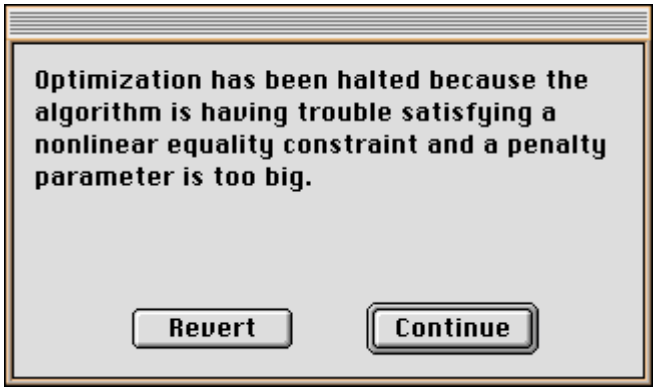
Now **Select All** (from the **Edit** menu) and choose the command **Optimize Strain** from the **Action** menu.

This is a different optimizer than we chose before. The reason is that once we start putting a lot of conditions on the crease pattern (and particularly path conditions), it becomes impossible to satisfy all the conditions without distorting the original tree. The **Optimize Strain** command

allows all of the edge strains to become variables as well as the node positions, but it seeks the minimum amount of distortion (specifically, it minimizes the RMS strain of the tree).

This type of optimization is one of the most difficult from a numerical analysis perspective, as there are a great many equality constraints and they are frequently not independent (i.e., you'll find that internally, there are two different constraints that are saying essentially the same thing). In this case, the optimizer becomes very sensitive to round-off errors (even in double-precision math) and *TreeMaker* will frequently think it is unable to find a solution.

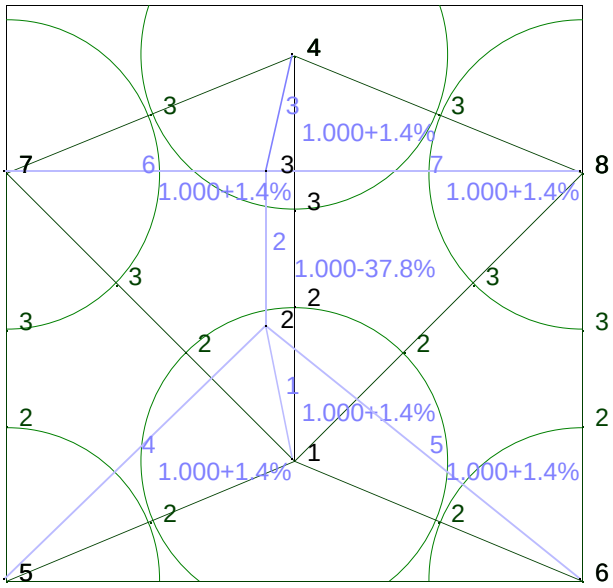
Such is the case here. Although you will see the nodes move around, you'll probably be presented with the following dialog:



3.5.8

In this case, *TreeMaker* actually did find a solution. So click on the "Continue" button.

You'll see that the pattern of nodes shown in figure 3.5.9 is very similar to what it was before, but now the edges are slightly strained. All of the paths we put conditions on are now running at multiples of 22.5°.

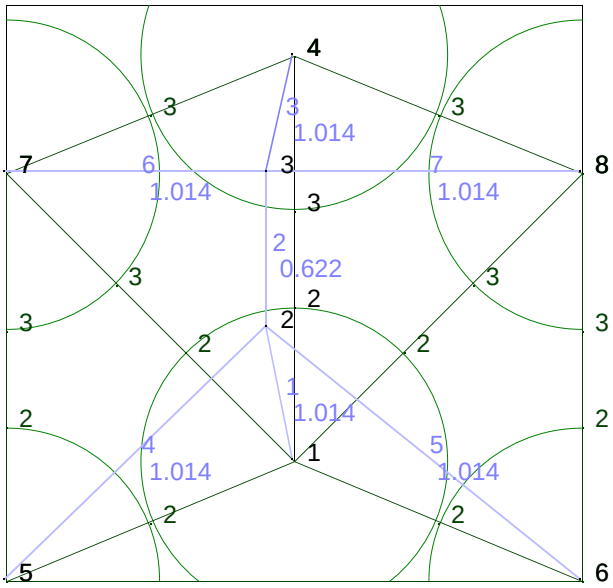


3.5.9

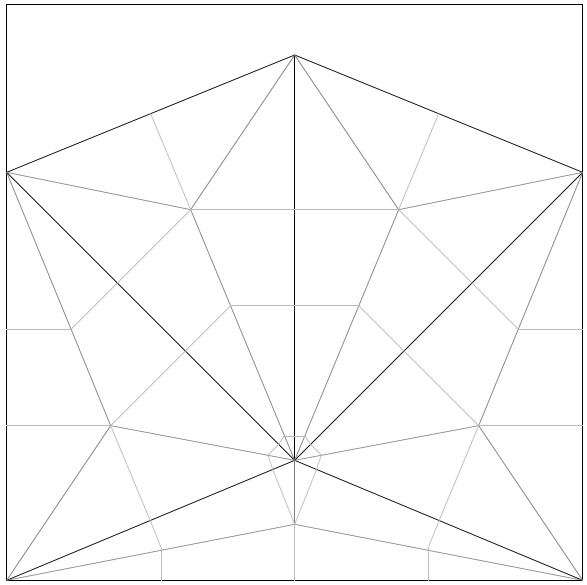
If you're happy with the results of the strain optimization, you can make it permanent by selecting the **Relieve strain** command, **Action** menu. This absorbs the strain into the length of each edge and resets the strain to zero, as shown in figure 3.5.10.

You can again **Build Polygons and Creases** from the **Action** menu to get the new crease pattern shown in figure 3.5.11.

You'll see that the new crease pattern is somewhat simpler and cleaner than the previous pattern. Perhaps more useful is that with the standardization of the angles, you can find all of the reference points by folding alone.



3.5.10



3.5.11

4.0 Reference

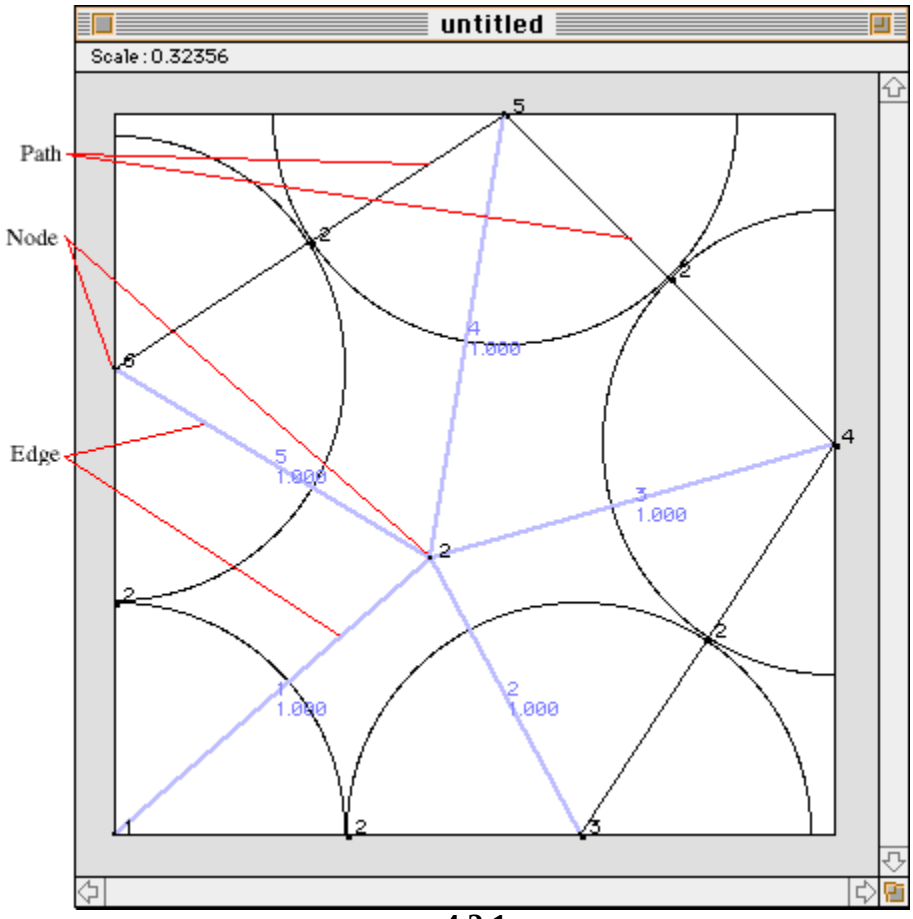
4.1 Introduction

This section describes the user interface of *TreeMaker*; it gives a comprehensive description of the main window, how you construct a tree, and lists all of the menu commands and describes all of the dialogs. The reference complements the tutorials in the previous section. While you should start with the tutorials, they only cover the most important parts of *TreeMaker;* it would be a good idea at some point to read through all of the reference to make sure you are aware of all the capabilities of the program.

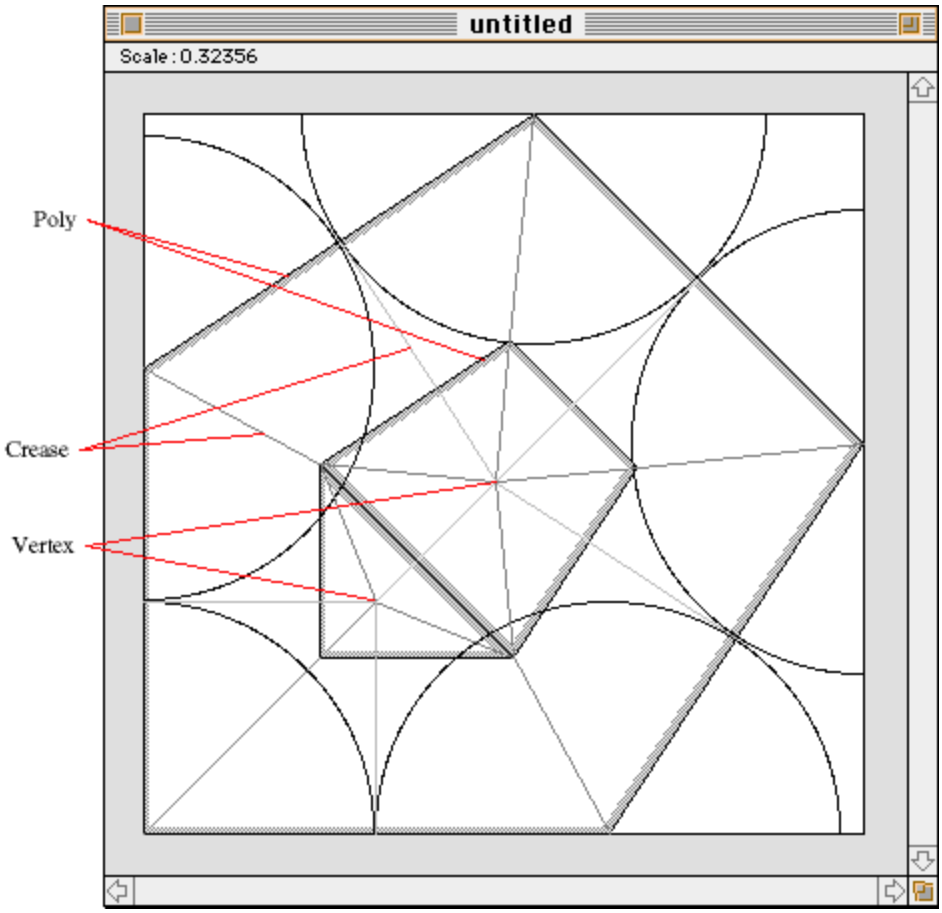
4.2 Main window

Each origami design is its own document and is displayed within a single window. An origami design is composed of a number of interrelated "parts"; the 7 types of parts are called: node, edge, path, poly, vertex, and crease — each of which has a graphical representation — and conditions, which modify nodes, edges, or paths, and don't have a direct visual representation.

Figures 4.2.1 and 4.2.2 show two views of a typical window that displays all of the different parts.



4.2.1



4.2.2

These are two different ways of viewing the same design. As is shown in the next section (Menu Commands), you can use the **Custom View…** command in the **View** menu to select which parts are shown and how they are shown.

Some parts are color-coded to display their attributes. Color coding is summarized in the following tables.

terminal node	**cond'd node**	**Node circle color**
false		no circle
true	false	black
true	true	green

pinned edge	cond'd edge	Edge color
true	false	light blue
true	true	light cyan
false	false	medium blue
false	true	medium cyan

sub path	terminal path	valid path	cond'd path	polygon path	active path	Path color
true				false		pale red
true				true		light red
false	false					medium light red
false	true	false				bright red
false	true	true	false	true		black
false	true	true	false	false	true	dark red
false	true	true	false	false	false	medium red
false	true	true	true	true		dark green
false	true	true	true	false	true	dark orange
false	true	true	true	false	false	medium orange

crease type	Crease color
valley	black
mountain	medium gray
tristate	light gray

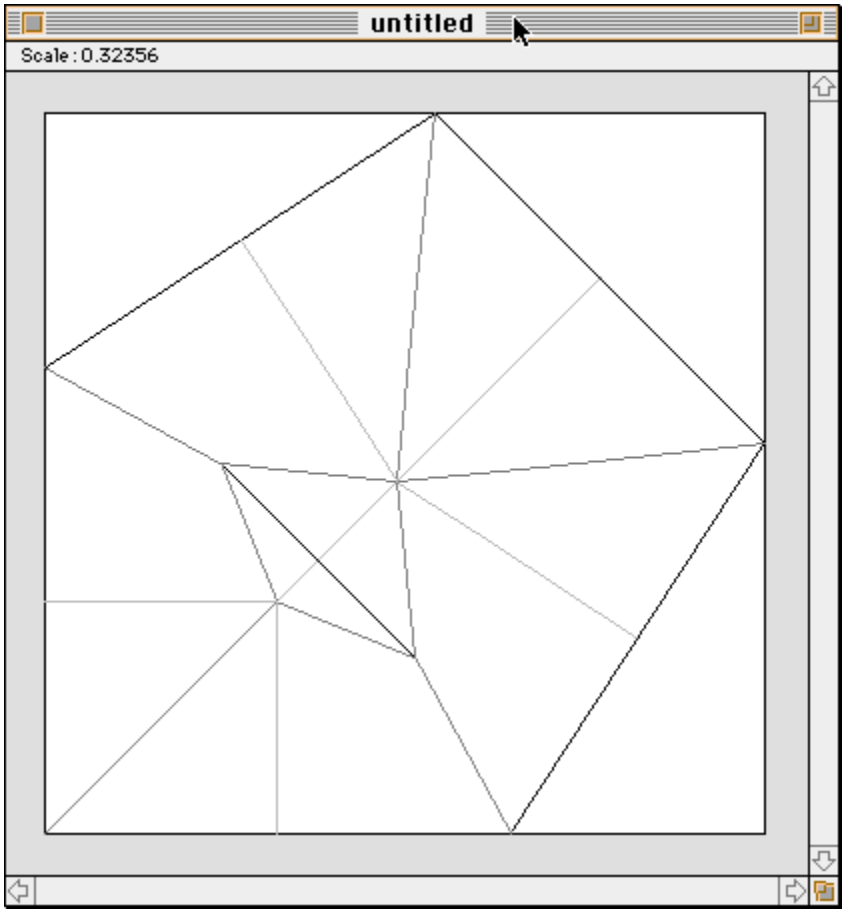
The top of the window shows the *scale* of the current design (the scale is the relationship between one edge unit and the size of the square). The rest of the window shows all the parts.

The collection of nodes and edges is known as a tree. Lines between nodes are paths. There is a path between most pairs of nodes, however, most paths are invisible in the windows above. Groups of certain paths form polys ("poly" is short for "polygon" and I'll use the term more or less interchangeably). Polys are filled in with vertices and creases. The vertices and creases form the crease pattern for the origami base. The design of an origami base is accomplished in three steps:

(1) Construction of a tree and specification of the desired attributes of the base (this is done by you)

(2) Placement of the nodes in a valid configuration (this is computed by *TreeMaker*)

(3) Construction of the polys, vertices, and creases, forming the crease pattern for the base. (This is also computed by *TreeMaker*.)



4.2.3

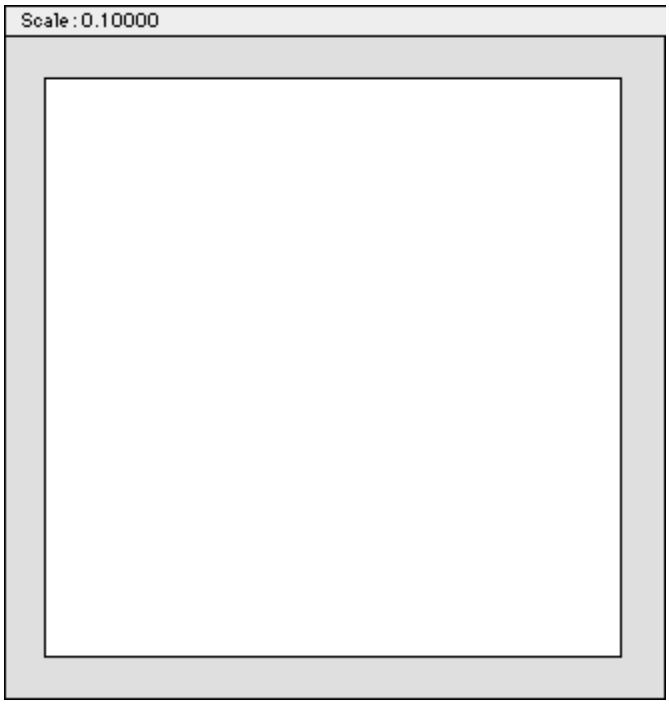
Figure 4.2.33 shows a finished crease pattern.

4.3 Creating a new Tree

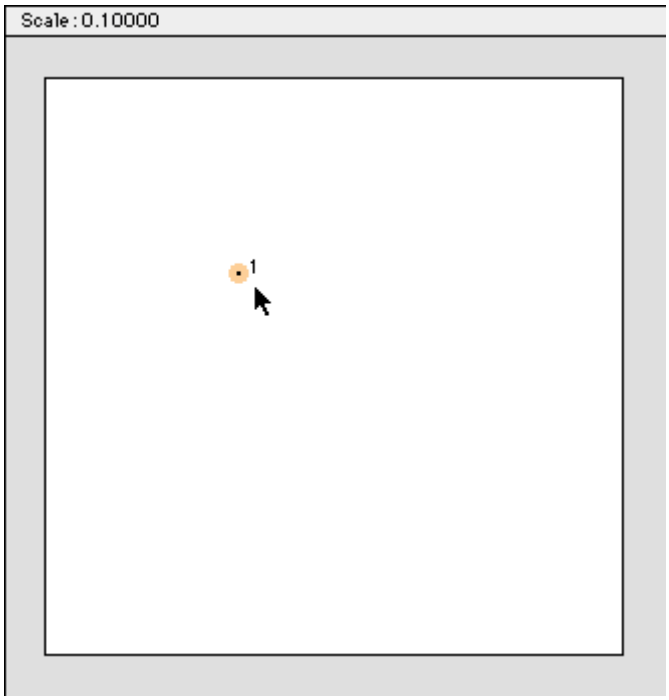
When a window is first opened, it will be in Default View showing a blank square, as shown in figure 4.3.1.

You define the tree by pointing and clicking in the square to draw a stick figure that defines the topology of the desired origami base. Clicking once in the square produces a node at the location of the click, as shown in figure 4.3.3. When a node is selected it is highlighted: it has a spot of color around it in the highlight color of the computer *TreeMaker* is running on.

If a single node is highlighted, clicking in another location will add a node with an edge connecting the two, as shown in figure 4.3.3.



4.3.1



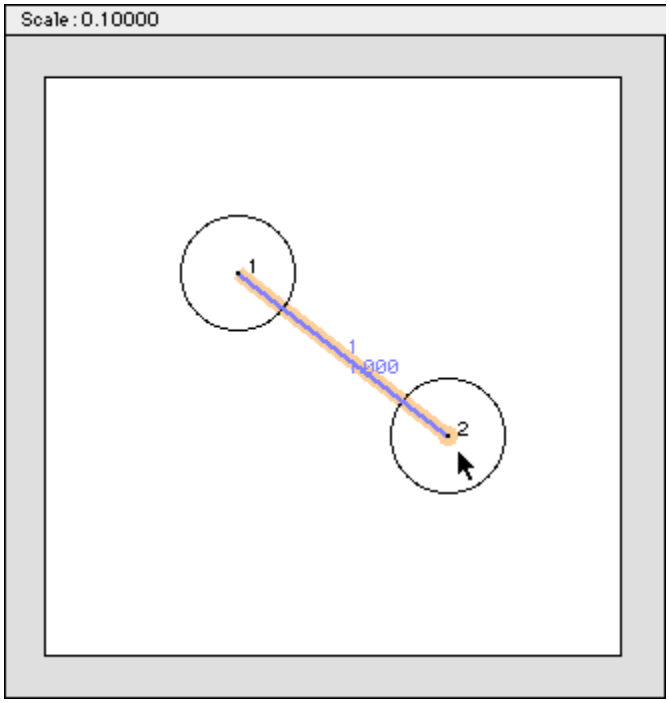
4.3.2

When an edge is first created, it will be highlighted and will have a default length of 1 unit.

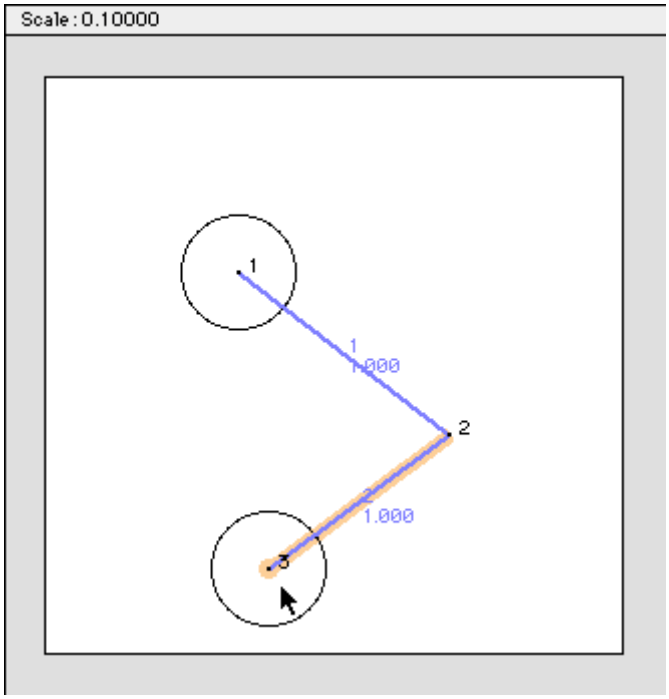
Once there are two or more terminal nodes, all terminal nodes will have a circle shown around them whose radius is equal to the scaled length of the attached edge.

In this example, node 2 is selected, so clicking in a new location will create another new node and edge as shown in figure 4.3.4.

Internal nodes (those with two or more edges emanating from them) do not display circles around them. However, you can add branches to the tree that emanate from an internal node by selecting the internal node and then clicking elsewhere in the square, as shown in figure 4.3.5 and 4.3.6.



4.3.3



4.3.4

Once a third node has been created, the convex hull of the nodes (border paths) will be outlined as is shown in figure 4.3.6.

Each edge of the tree corresponds to a flap of the base. By repeatedly selecting nodes and clicking, you can build up arbitrarily complex trees with any number and arrangement of flaps.

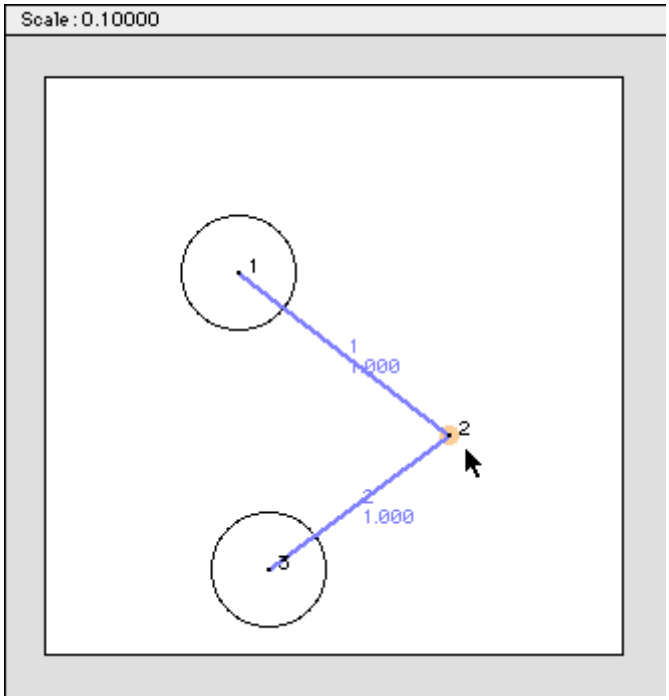
To remove a node or edge, simply select it by clicking on it and do one of the following:

Select the **Cut** command, **Edit** menu

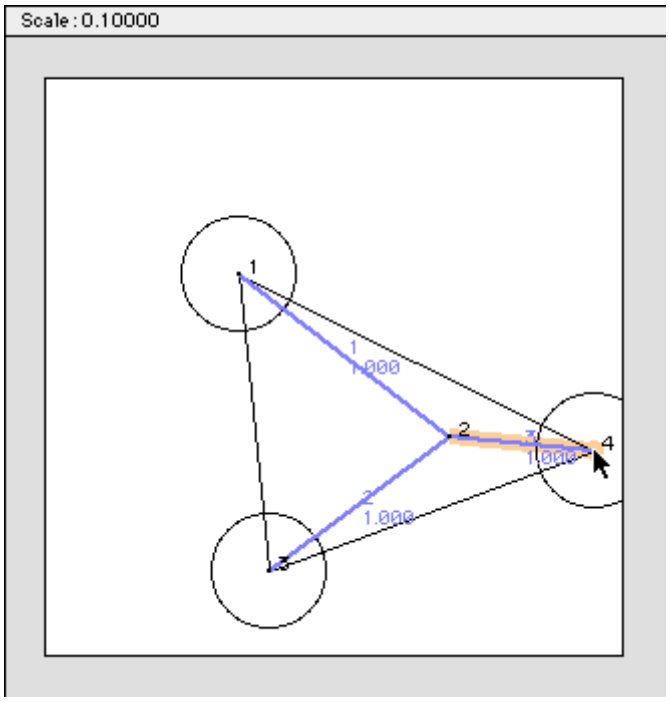
Select the **Clear** command, **Edit** menu

Hit the <delete> or <forward delete> key

Nodes, edges, and paths are some examples of "parts." Attributes of parts — the location of a node, the length of an edge, and so forth — can be altered by editing the part. How this is done is the subject of the next section.



4.3.5



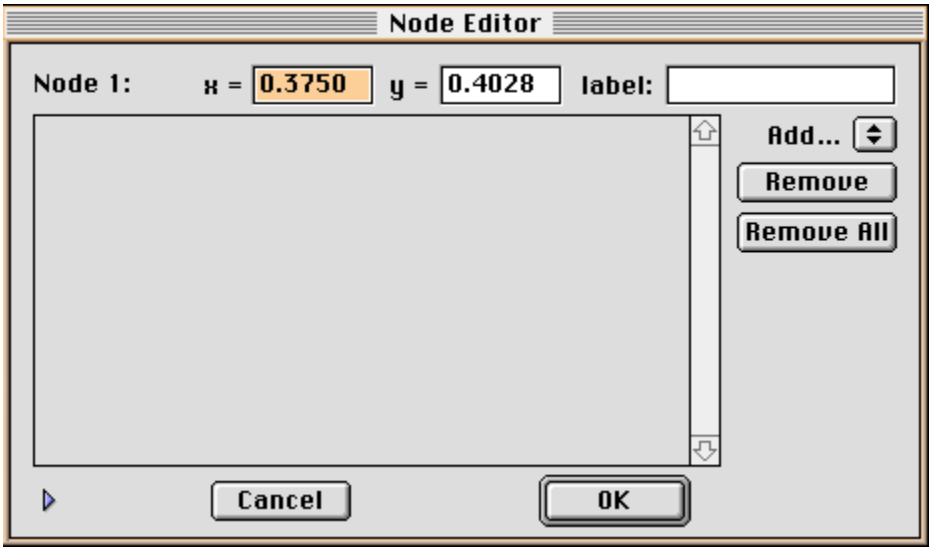
4.3.6

4.4 Editing Parts

All parts (nodes, edges, paths, polys, vertices, and creases) have attributes associated with them. Some are numerical settings; some describe relationships to other parts. You can edit many of these attributes in several ways. The most direct is to double-click on the part. Double-clicking brings up a dialog that displays all of the attributes of the part whether they are editable or not. This section describes all of the editor dialogs for each of the parts.

Nodes

If a node is double-clicked, the Node Editor is put up as shown in figure 4.4.1.



4.4.1

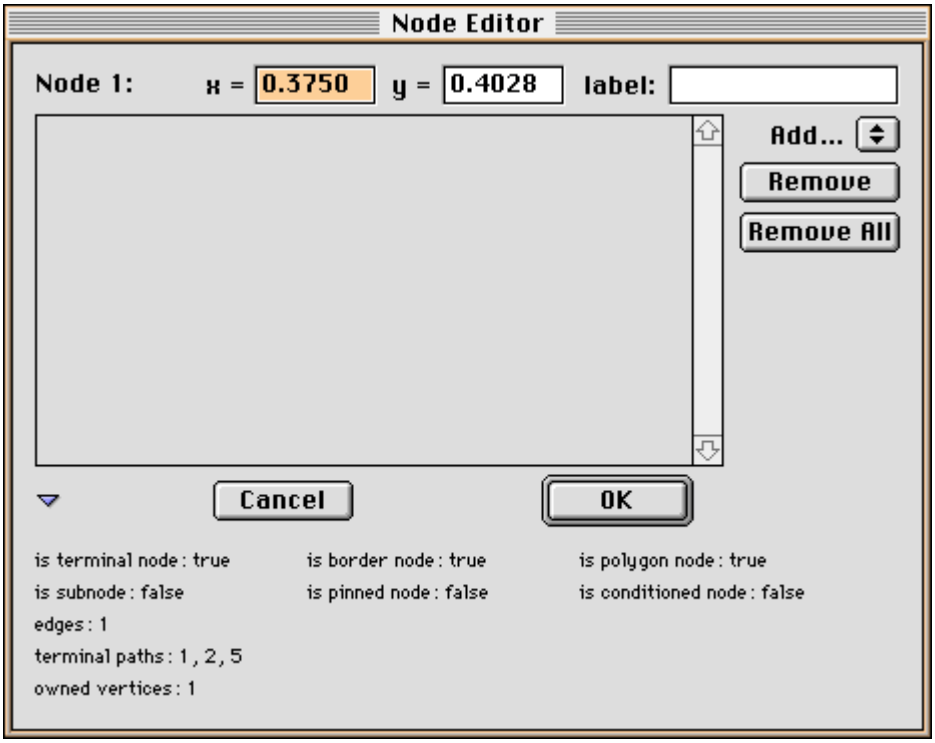
A node is always identified by an index (node 1 in this example). The fields of the Node Editor are:

x= ___, *y*= ___ — the coordinates of the node

label — an optional label for the node, which you may use any way you wish

The large list (empty in the example), the popup menu labeled “Add…”, and the two buttons titled “Remove” and “Remove All” are used to place conditions on the node. See **Edit Conditions…** command, **Conditions** menu, for a description of these tools (which work the same in every dialog in which they appear).

If you click on the disclosure triangle at the lower left corner of the dialog, it toggles between a standard dialog and an Expert Mode dialog shown in figure 4.4.2.

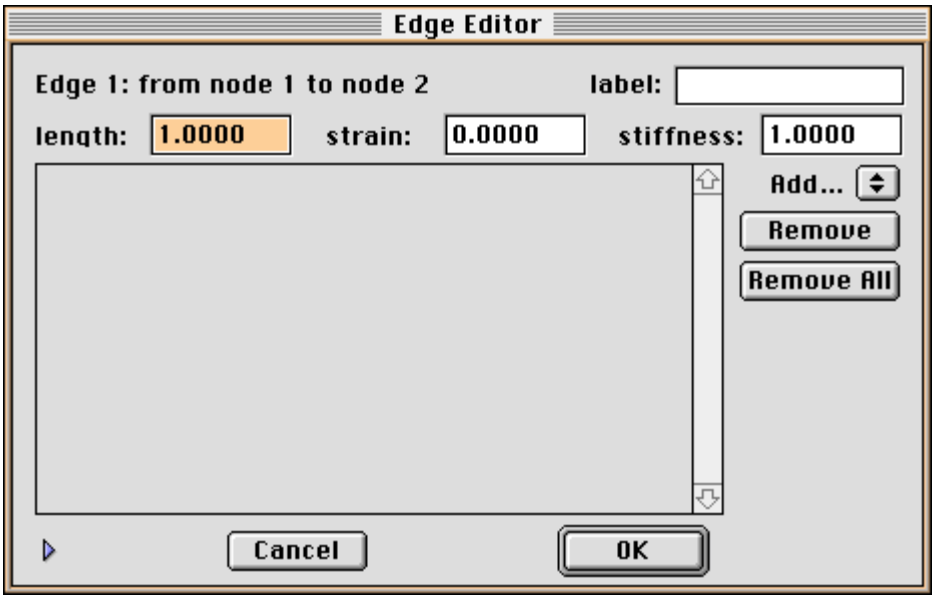


4.4.2

Expert Mode displays additional noneditable information about the node at the bottom of the dialog. You don't really need to know this information to design an origami base, but you might find it interesting; it tells you a little about how the data structure is put together.

Edges

If an edge is double-clicked the Edge Editor is put up as shown in figure 4.4.3.



4.4.3

Like the Node Editor, the Edge Editor displays the index of the edge as well as the indices of the nodes at each end of the edge. The fields of the Edge Editor are:

label — an optional label for the edge, which you can use any way you want

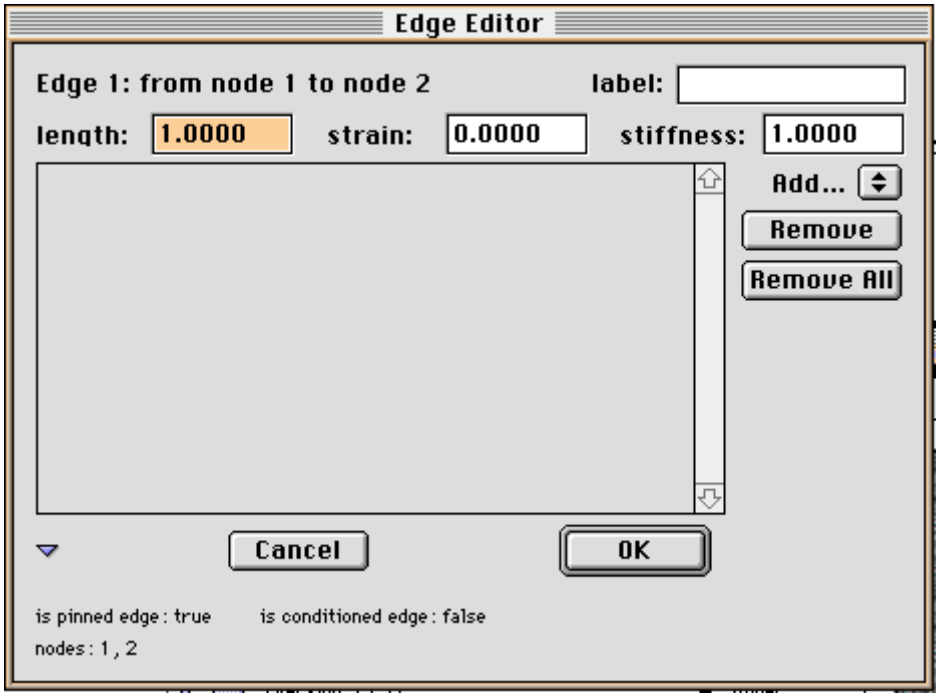
length — the desired length of the edge, relative to all the other edges

strain — the current strain of the edge, i.e., deviation from its desired length

stiffness — the stiffness of the edge, i.e., how resistant it is to being strained

There is also a set of controls for adding conditions to the edge. Again, look at the description of the **Edit Conditions…** command, **Conditions** menu, for a description of these controls.

At the lower left corner, clicking on the disclosure triangle toggles between normal and Expert Mode as shown in figure 4.4.4.



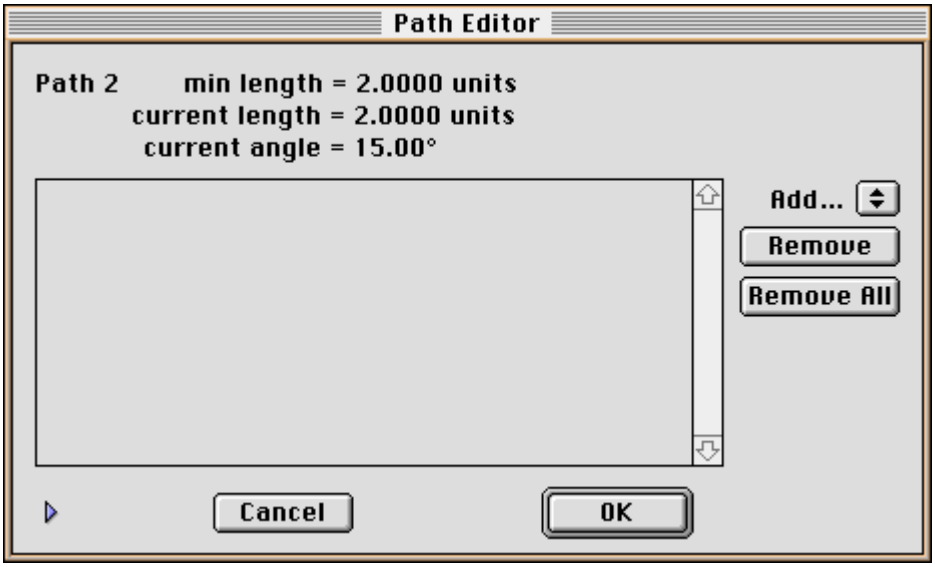
4.4.4

As with nodes, the information shown in Expert Mode isn't really necessary, but you might find it interesting.

Paths

If you double-click on a path, the Path Editor is put up as shown in figure 4.4.5.

•Note: most paths are not visible, but you can call up the editor for any path (or any other part) using the **Select Part…** command, **Edit** menu.



4.4.5

There are no editable fields of a path, other than the controls for adding conditions. The uneditable fields of the Path Editor do tell you some useful information; they are:

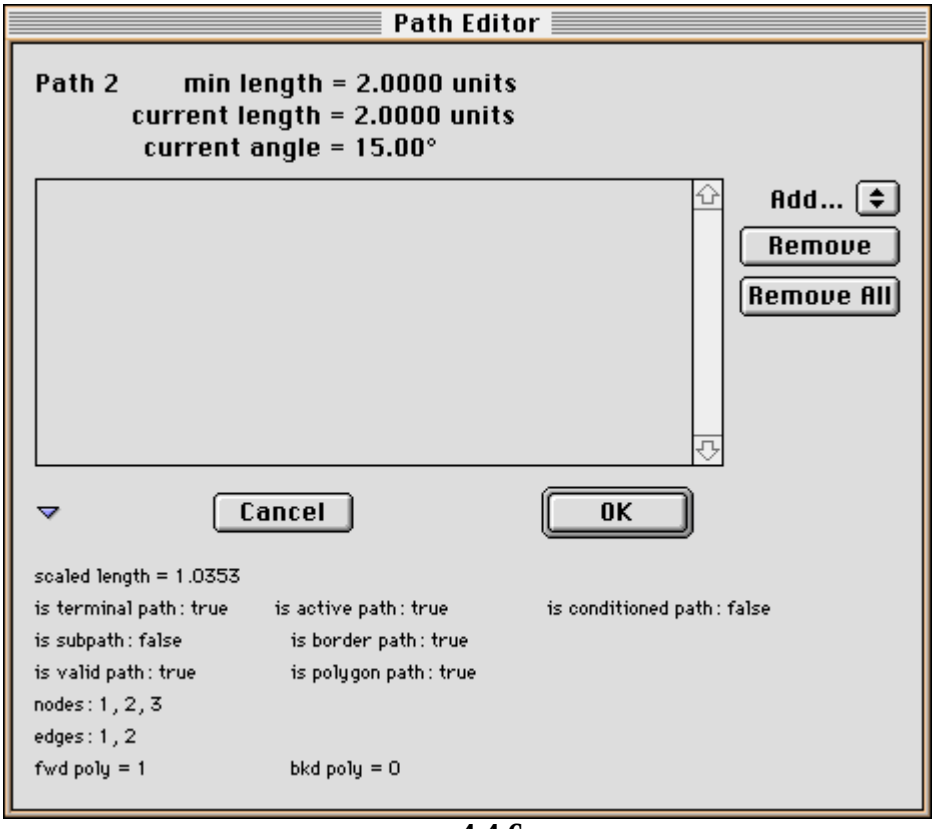
min length — the minimum length (in unscaled tree units) this path is allowed

current length — the current length of the path (in unscaled tree units)

current angle — the current angle of the path with respect to the *x*-axis.

There is also a set of controls for adding conditions to the path.

At the lower left corner, clicking on the disclosure triangle toggles between normal and expert mode as shown in figure 4.4.6.



4.4.6

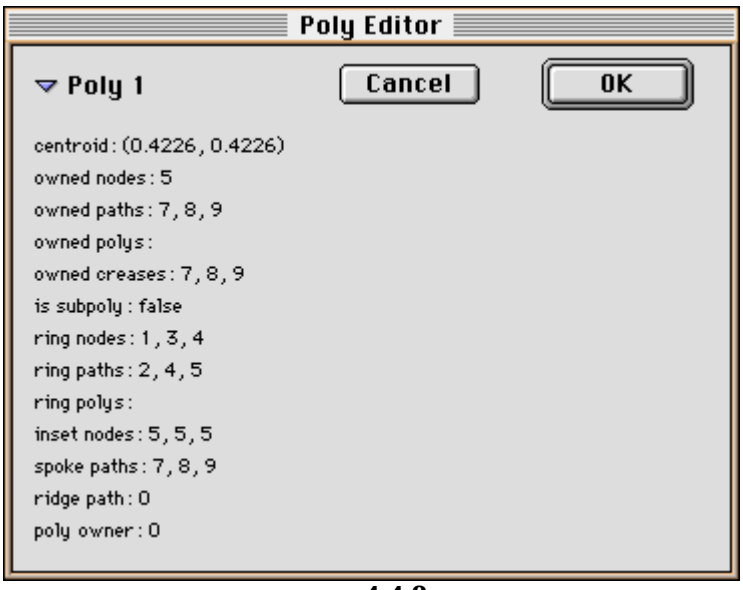
Polys

If you double-click on a poly, the Poly Editor is put up, as shown in figure 4.4.7.

4.4.7

There are no editable fields of a poly and you can't apply conditions to one, so the only thing the Poly Editor really tells you is the index of the poly.

However, the disclosure triangle toggles to Expert Mode, which gives some of the structural information of the poly, as shown in figure 4.4.8. A poly is, internally, a fairly complex beastie.



4.4.8

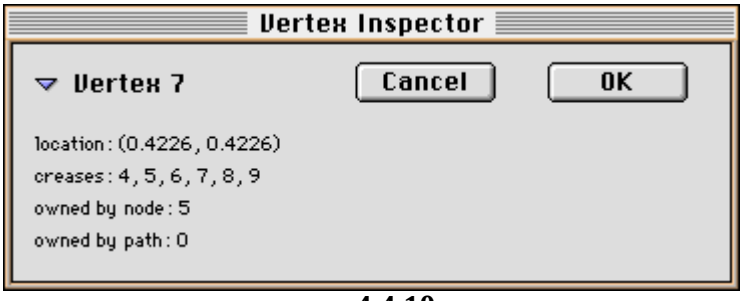
Vertices

If you double-click on a vertex, the Vertex Editor is put up, as shown in figure 4.4.9.

4.4.9

There are no editable fields of a vertex and you can't apply conditions to one, so the only thing the Vertex Editor tells you ins the index of the vertex.

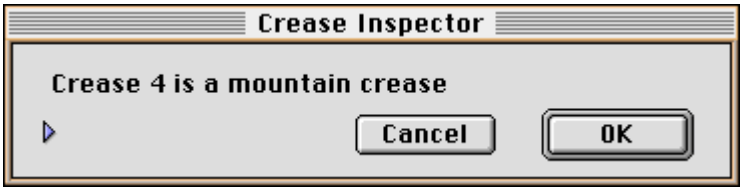
However, the disclosure triangle toggles to Expert Mode, which shows some of the structural information of the vertex, as shown in figure 4.4.10.



4.4.10

Creases

If you double-click on a crease, the Crease Editor is put up as shown in figure 4.4.11.



4.4.11

There are no editable fields of a crease and you can't apply conditions to one. However, the Crease Editor does tell you the index and the type of crease (mountain, valley, or tristate).

The disclosure triangle toggles to Expert Mode, which shows some of the structural information about the crease, as shown in figure 4.4.12.

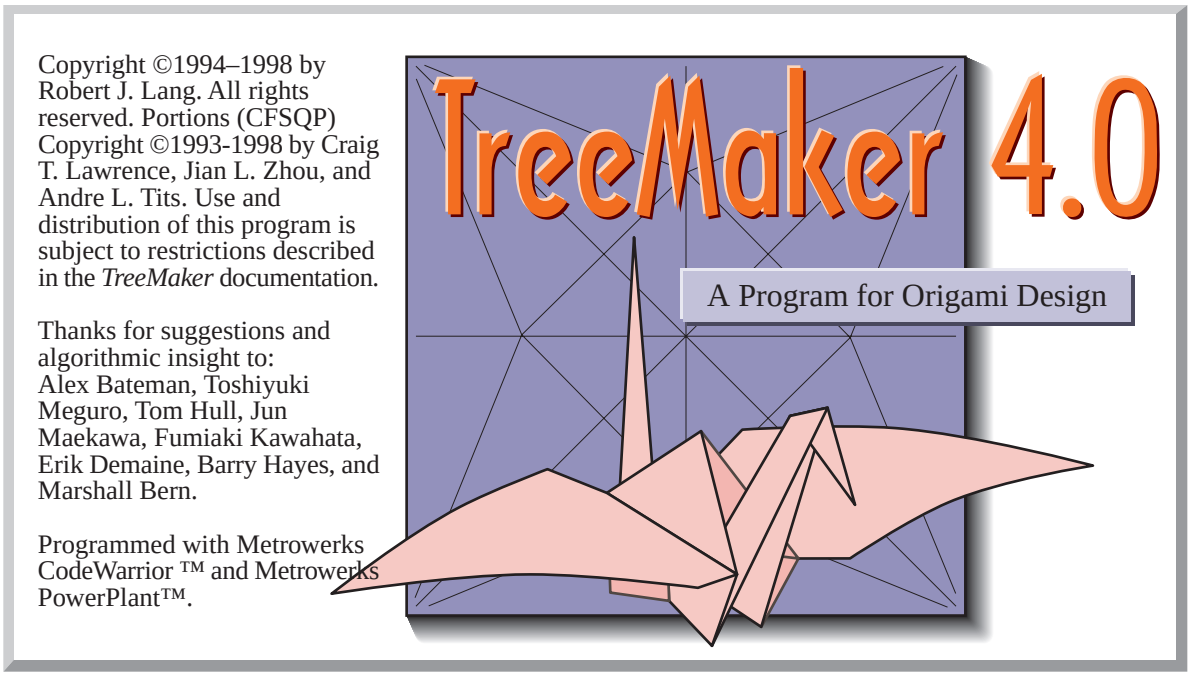
4.4.12

These editors are brought up by double-clicking on a visible part, but they can also be called up for any part (visible or invisible) by selecting the **Edit Part…** command, **Edit** menu (see below).

4.5 Menu commands

This section summarizes all of the menu commands of TreeMaker. TreeMaker has five menus: **Apple**, **File**, **Edit**, **View**, **Actions**, and **Conditions**. The commands in each menu are given below.

Shows you the version and credits for the program as shown in figure 4.5.1.



4.5.1

File Menu

All of these commands do what they usually do in a Macintosh application.

New ⌘N

Create a new, blank square to work on.

Open ⌘O

Open an existing tree.

Close ⌘W

Close the current tree. You will be prompted to save it if you have made any changes.

Save ⌘S

Save the current tree to disk.

Save As...

Save the current tree under a new name.

Revert

Revert the current tree to the last saved version.

Page Setup...

Select the page orientation and other printing options.

Print... ⌘P

Print the current tree. The printout is exactly the same size as what's on the screen and the same elements will be shown as are displayed. If you are in **Default View**, you'll see all the nodes, edges, circles, and so forth; if you are in **Creases Only View**, then you'll print just the crease pattern.

If the image is too large to fit on a single sheet, it will be tiled on multiple pages with crop marks at the joints. There will be a header on each page consisting of the name of the file on the left and the current scale on the right.

Print One...

Print a single page.

Quit ⌘Q

Quit the program.

Edit Menu

Undo ⌘Z

Undo is currently not supported except for text entries in dialogs. The structure of an origami design has many complex interrelationships between parts; if you do something as simple as moving a node, many data structures are created, destroyed, and re-connected. Implementation of Undo would be difficult.

The next four commands are the standard Macintosh editing commands that behave normally for text in dialog boxes.

Cut ⌘X

In a text field, **Cut** cuts the current selection to the Clipboard. In the main window, **Cut** delete the current selection of nodes or edges.

Copy ⌘C

In a text field, copy the current selection to the Clipboard. In the main window, copy the contents of the window to the Clipboard as a picture (PICT) with embedded Postscript commands for line weight and polygon formation. You can paste the crease pattern into another drawing program to touch it up, change line weights or styles, or color it in. Many of the images in this document, for example, were generated by Copying the image of the main window to the Clipboard and pasting them into the document.

Paste ⌘V

In a text field, past the contents of the Clipboard into the field. This command does nothing in the main window.

Clear

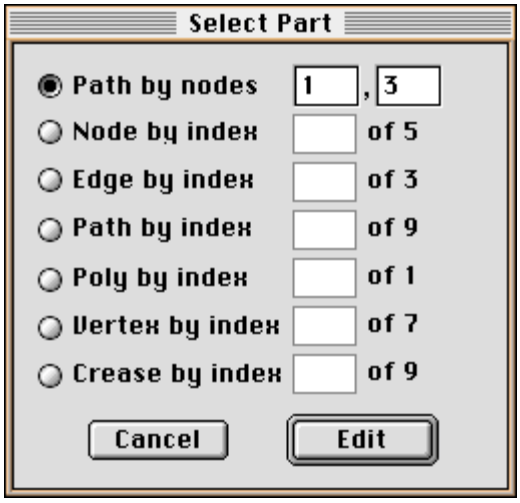
In a text field, delete the current selection. In the main window, delete the current selection of nodes or edges.

Select All ⌘A

This command selects all visible nodes and edges. It's most useful for deleting everything and starting over and, with operations that are constrained to the selection (like *Optimize Edges* and *Fracture Poly*), for giving the largest number of degrees of freedom possible.

Edit Part... ⌘I

This command puts up a dialog, the Part Selector, that lets you select a part to edit by several different criteria. This is useful if several parts are close together and you can't easily click on the one you want.



4.5.2

The fields of the Part Selector are:

Path by nodes — specifies a path according to the indices of the nodes at each end of the path

Node by index — specifies a node according to its index

Edge by index — specifies an edge according to its index

Path by index — specifies a path according to its index

Poly by index — specifies a poly according to its index

Vertex by index — specifies a vertex according to its index

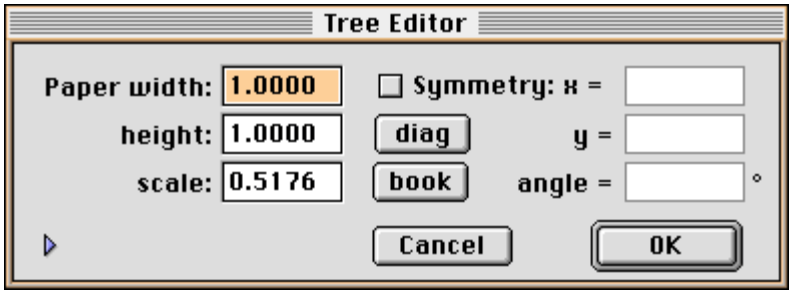
Crease by index — specifies a crease according to its index

This command is used to edit a node, edge, path, poly, vertex, or crease when you can't easily double-click on it. I find that I use it most often to get access to a particular path when I know the nodes at each end of the path, so that's the default selection criterion.

After you close this dialog, an editor will be put up for the selected part.

Edit tree… ⌘J

Puts up the Tree Editor, which lets you alter attributes of the overall pattern, as shown in figure 4.5.3.



4.5.3

The fields of the Tree Editor are:

Paper width — gives the width of the paper in *dimensionless* units. This value is normally set to 1, no matter how big the screen or printout is. To change the actual size of the screen (or printed) image, select **Set Paper Size…** in the **View** menu.

Paper height — gives the height of the paper in dimensionless units. This value is normally set to 1 for a square but you can change it to other values if you like. To work out designs on a rectangle, you can change either the width or height to a different value. (You can, if you like, change both to a different value, but it won't make any difference.)

Scale — this sets the relationship between the units of the tree (edge lengths) and the dimensionless size of the square. This quantity is determined when you invoke the **Optimize Scale** command. You will ordinarily not change this number, but if you are creating a very complicated tree, you might want to set it to a very small number to keep from cluttering up the screen with a lot of circles.

Symmetry — on = define a line of symmetry. The line is defined by a point on the line and an angle. Two common lines of symmetry are the point (0.5, 0.5) and angle 45° (for a diagonal model) and the point (0.5, 0.5) and angle 90° (for a book-folded model). The two buttons immediately below the check box are preset for these values.

diag — preset button for diagonal symmetry

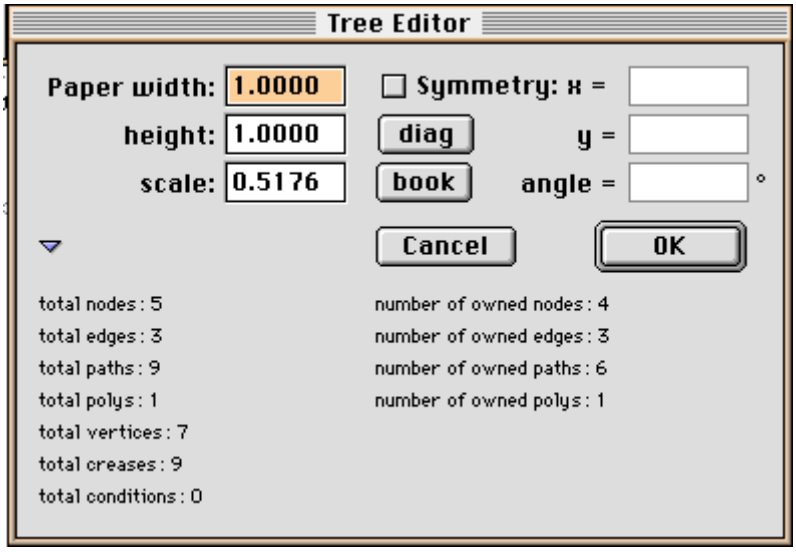
book — preset button for book-fold symmetry

x = ___ — the *x*-coordinate of a point on the line of symmetry.

y = ___ — the *y*-coordinate of a point on the line of symmetry.

angle = ___ — the angle (in degrees) of the line of symmetry.

Clicking on the disclosure triangle toggles to Expert Mode, which displays additional structural information about the Tree as shown in figure 4.5.4.



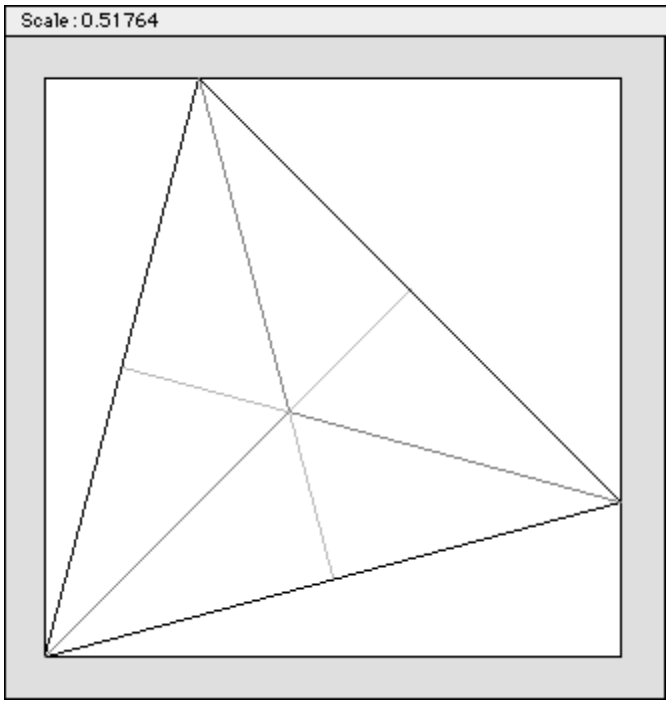
4.5.4

View Menu

The View menu contains commands that affect the way the tree and crease pattern are displayed on the screen or printout. Commands are:

Default View ⌘D

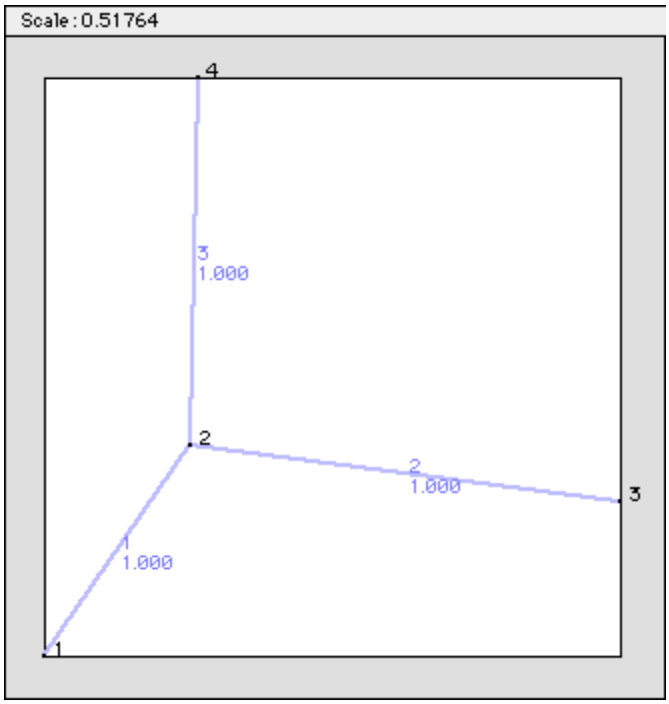
Alters settings so that both the tree (the stick figure) and creases are visible. This is usually the most useful combination of settings for working through a design. Here is an example shown in figure 4.5.5.



4.5.5

Show Tree Only ⌘T

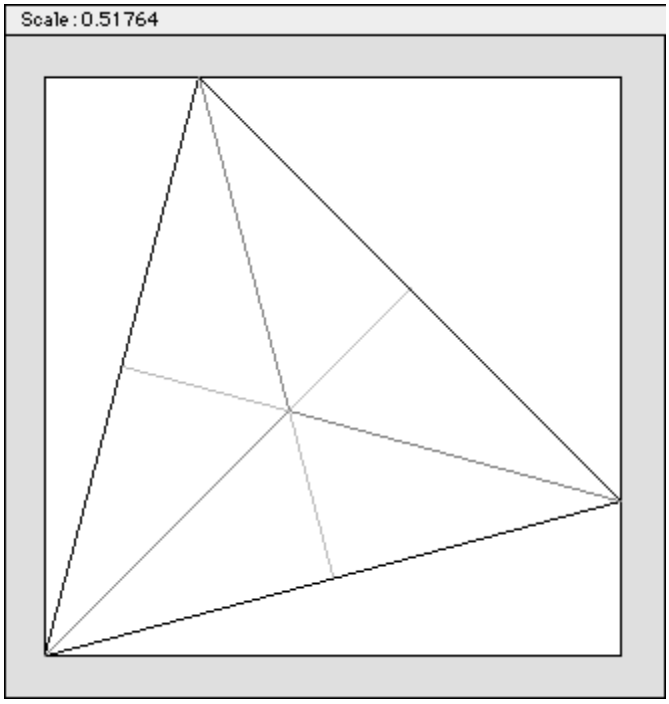
Show only the stick figure. Here is an example, shown in figure 4.5.6.



4.5.6

Show Creases Only ⌘L

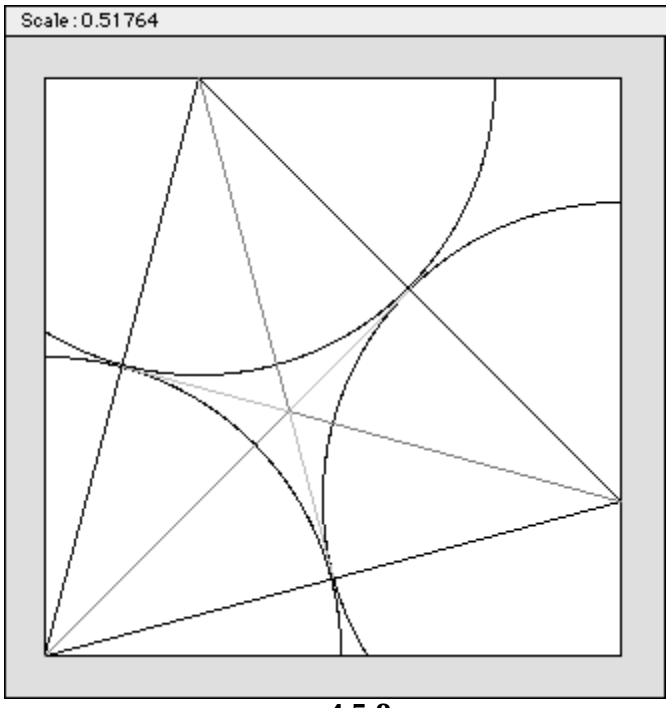
Show only the crease pattern. Use these settings when you want to print out the crease pattern without any other extraneous elements. Here is an example shown in figure 4.5.7.



4.5.7

Show Creases and Circles ⌘U

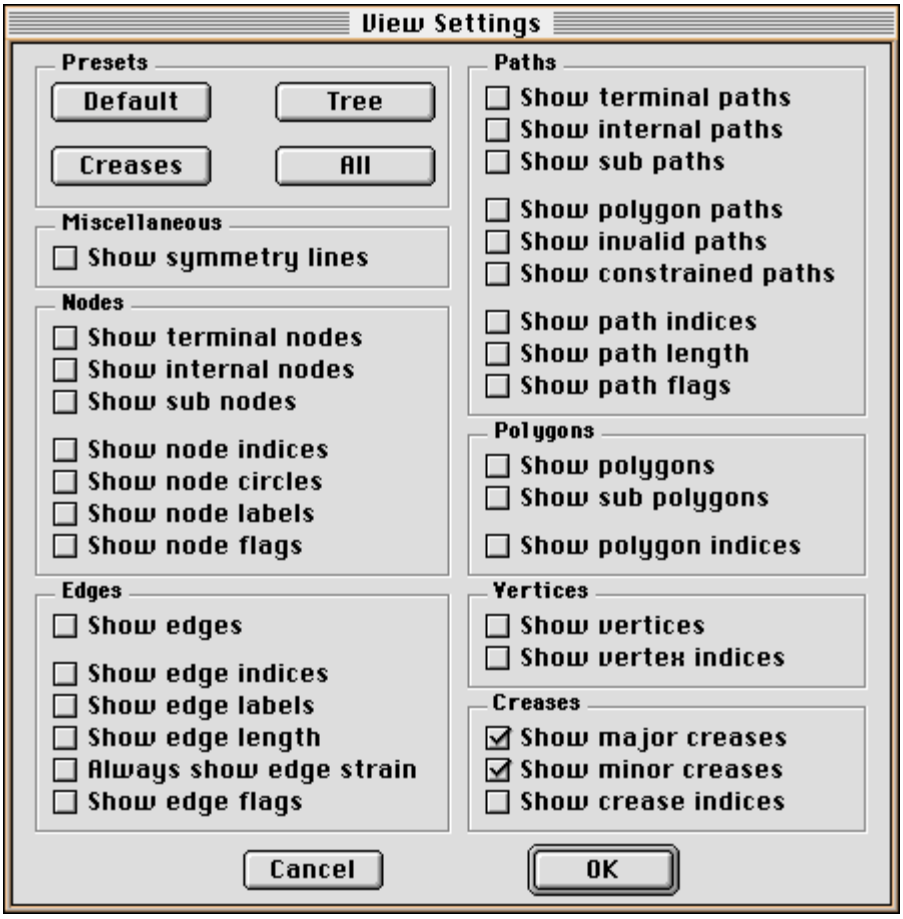
Show the crease pattern plus circles around terminal nodes. This group of settings shows some of the structure of the base superimposed over the crease pattern. It's a good compromise between showing just the crease pattern but also giving some indication of where paper goes into flaps. Here is an example shown in figure 4.5.8.



4.5.8

Custom View... ⌘G

puts up the View Editor, which lets you change how the image is displayed on the screen and display any combination of settings, as shown in figure 4.5.9.



4.5.9

The View Editor is a movable dialog and its settings take effect immediately so you can see what the results of your settings are.

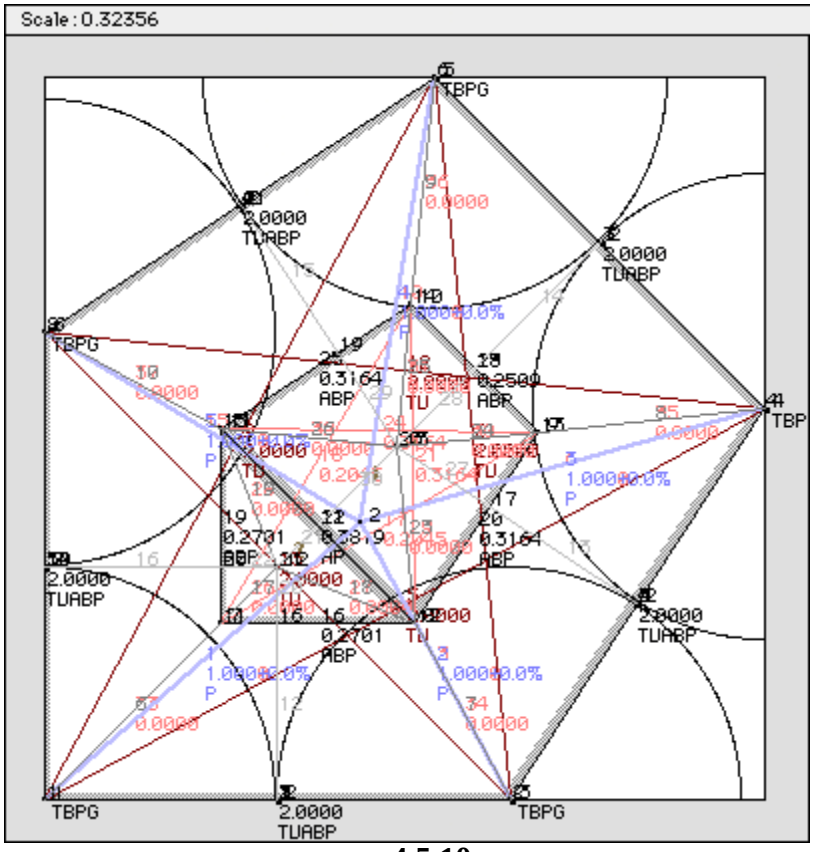
There are four buttons for preset combinations of settings:

Default — show the default view, which displays both the tree, node circles, polygons, and creases.

Tree — show just the tree (stick figure)

Creases — show just the crease pattern.

All — show everything. For all but the simplest figures, this is too busy to be useful (see figure 4.5.10), but if you ever wanted to see all the "guts" of a TreeMaker design, this is the closest thing to it.



4.5.10

The other fields of the View Editor are:

Show symmetry lines — draw the line of symmetry on the square

Show terminal nodes — display nodes that have only one edge attached. All nodes are displayed as dark black dots at the location of each node. Nodes with conditions attached are shown in dark green.

Show internal nodes — display nodes that have more than one edge attached

Show sub nodes — display nodes that are not part of the Tree but that are contained by polys

Show node indices — display the indices of the nodes

Show node circles — display a circle around each terminal node. The circles around nodes with conditions attached are shown in dark green.

Show node labels — display any labels you have defined for nodes

Show node flags — display single-letter codes for the flags attached to each node, which are:

t — terminal node: node has only one edge attached

b — border node: node lies on the convex hull of the group of nodes

p — pinned node: node can't be moved without violating a path constraint

g — polygon node: node is a vertex of a valid polygon

c — conditioned node: node has one or more conditions applied to it

Show edges — display the edges of the tree. Edges are shown as thick light blue lines. Pinned edges are shown in a lighter blue.

Show edge indices — display the index of each edge at its midpoint

Show edge labels — display any labels defined for edges

Show edge length — display the length of each edge

Always show edge strain — display the edge strain even if it's equal to zero

Show edge flags — display single-letter codes for the flags attached to each edge, which are:

p — pinned edge: edge can't be lengthed without violating a path constraint

c — conditioned edge: edge has one or more conditions applied to it

Show terminal paths — display all paths between terminal nodes (nodes with only 1 edge). Paths are shown as single-pixel-thick lines between nodes. Terminal paths are shown in red.

Show internal paths — display all paths that connect to at least one internal node. Internal paths are shown in pale red.

Show sub paths — display paths between sub nodes. Sub paths are shown in very pale red.

Show invalid paths — display all paths whose actual lengths exceeds their computed minimum length. Invalid paths are shown in dark red.

Show polygon paths — display all paths that are part of the network of active paths and polygons. Polygon paths are shown in black.

Show conditioned paths — display paths that have conditions applied. Conditioned paths are shown in orange.

Show path indices — display the index of each visible path at its midpoint

Show path length — display the minimum length of each visible path

Show path flags — display single-letter codes for the flags attached to each path, which are:

t — terminal path: the path runs between two terminal nodes.

v — valid path: this path's actual length is greater than or equal to its minimum length

a — active path: this path's actual length is exactly equal to its minimum length

b — border path: this path lies on the border (the convex hull) of the set of nodes

g — polygon path: this path should be part of a poly

c — conditioned path: this path has one or more conditions applied to it

Show polygons — display major polygons. Polygons are displayed by thick gray borders inlining each polygon.

Show sub polygons — display sub polygons (reduced polygons)

Show vertices — display all vertices as black dots.

Show vertex indices — display the index of each vertex next to it

Show major creases — display mountain and valley fold creases. Valley folds are shown in black; mountain folds are shown in dark gray.

Show minor creases — display tri-state creases. Tristate creases are shown in light gray.

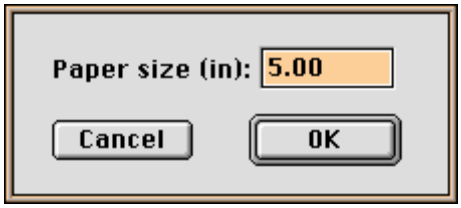
Show crease indices — display the index of each crease

Fit to Screen ⌘F

This menu command changes the size of the paper and the window to fill up the main screen. On a 17" monitor, you will get about a 7.25" square (which, incidentally, fits neatly on an 8.5×11" sheet of paper when it is printed).

Set Paper Size... ⌘Y

puts up the dialog shown in figure 4.5.11 to change the paper size, which is given in inches. The screen image is exactly the same size as the printed image.



4.5.11

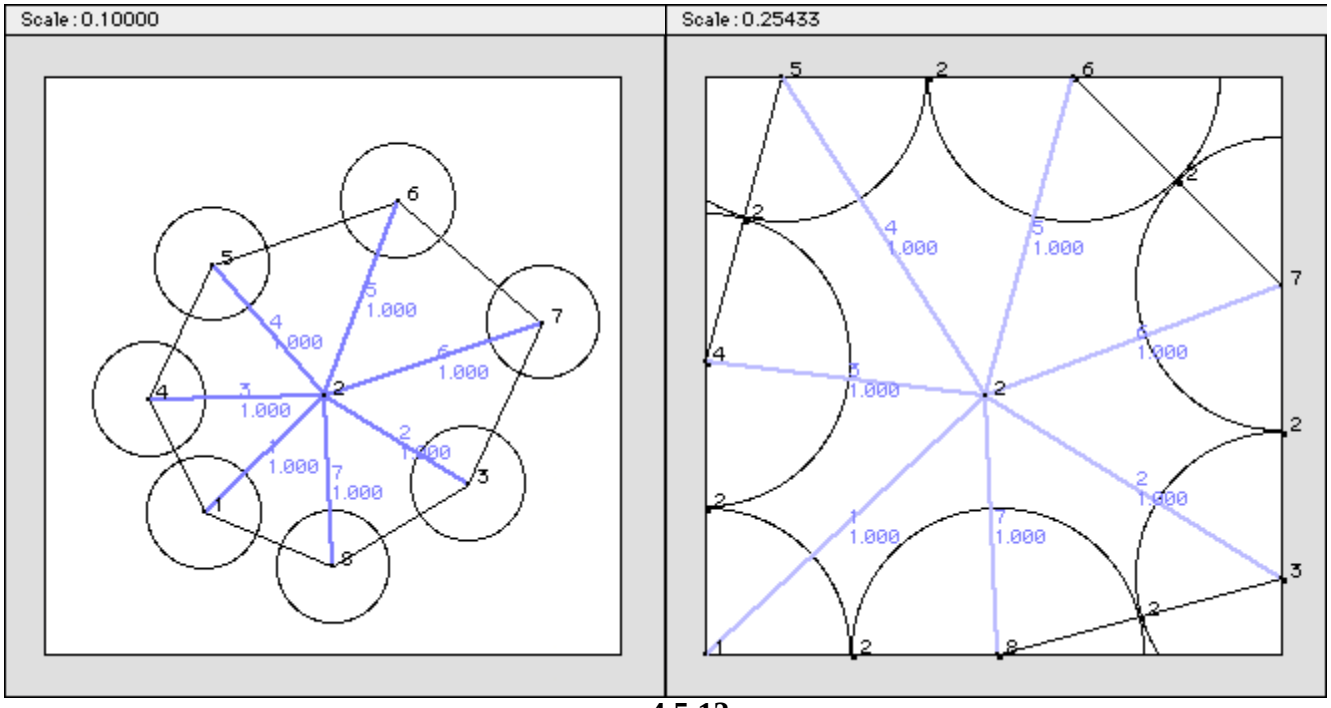
Action Menu

The commands in the **Action** menu perform the calculations that determine the node positions and the locations of creases. Normally you will define a stick figure, set up conditions on the nodes, edges, or paths, and then alternate between refining the condition set and selecting commands from the **Action** menu.

Scale Everything ⌘1

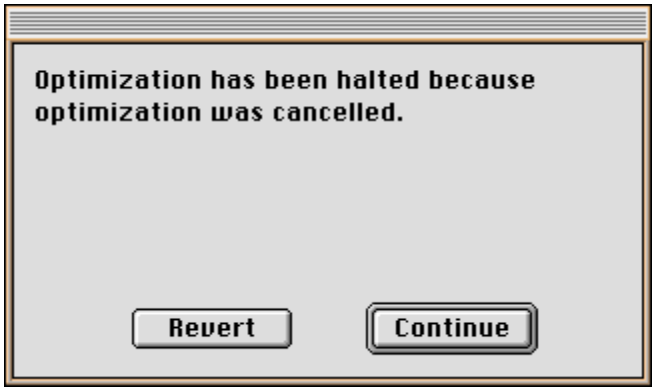
Adjust the positions of all nodes and the overall scale of the model to find the position of the nodes that satisfies all conditions and that gives the largest possible base. The image is updated periodically as the calculations are performed so you can see progress. Type Command-Period to

interrupt the calculation. Figure 4.5.12 shows a typical configuration before and after “Scale Everything.”



4.5.12

If you interrupt the calculation, you will be presented with the following dialog:



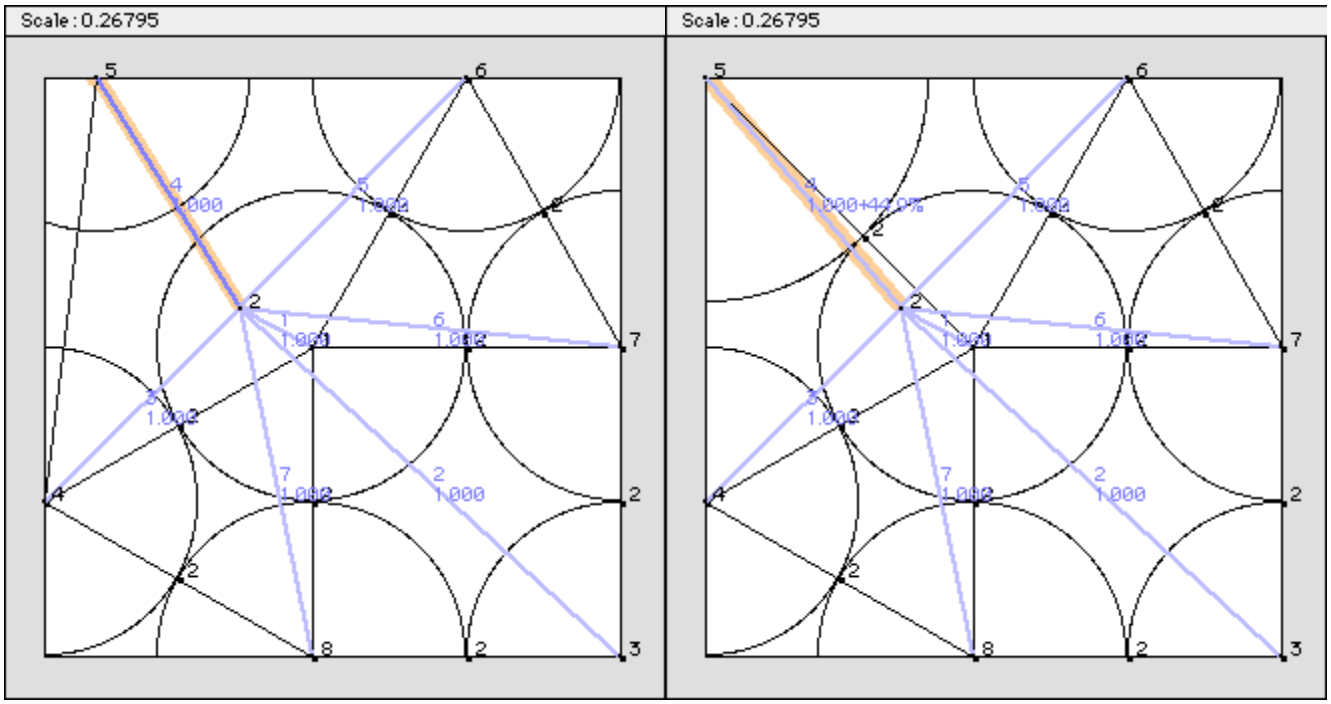
4.5.13

Similarly, if the optimizer runs into problems, it may also interrupt the calculation and present you with a dialog of explanation. You have the choice of continuing with the nodes in their present positions or reverting to the state the nodes were in before the calculation. Even if you revert, any creases you may have generated are irretrievably lost, but they are easily reconstructed.

Scale Selection ⌘2

This command is used to remove degrees of freedom in a pattern after a **Optimize Scale** command. By selecting unpinned nodes (shown in gray) and unpinned edges (shown in darker

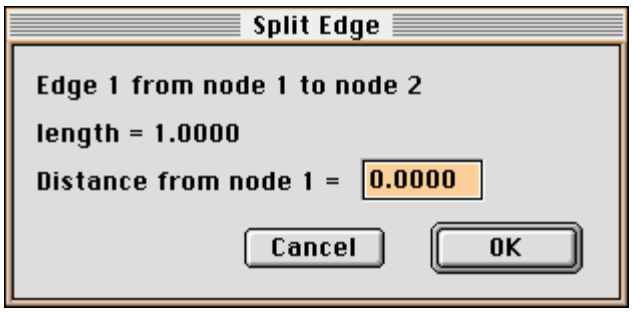
blue) and selecting this command, the lengths of all of the unpinned selected edges will be lengthened proportionally by increasing their strain, optimizing the position of the unpinned selected nodes. If you happen to have selected pinned nodes or edges, they will be ignored. Since only the selected nodes and edges will be considered, an easy way to increase all possible nodes and edges is to select all (command-A) and then select this command. Figure 4.5.14 shows a typical before and after configuration for scaling a single edge.



4.5.14

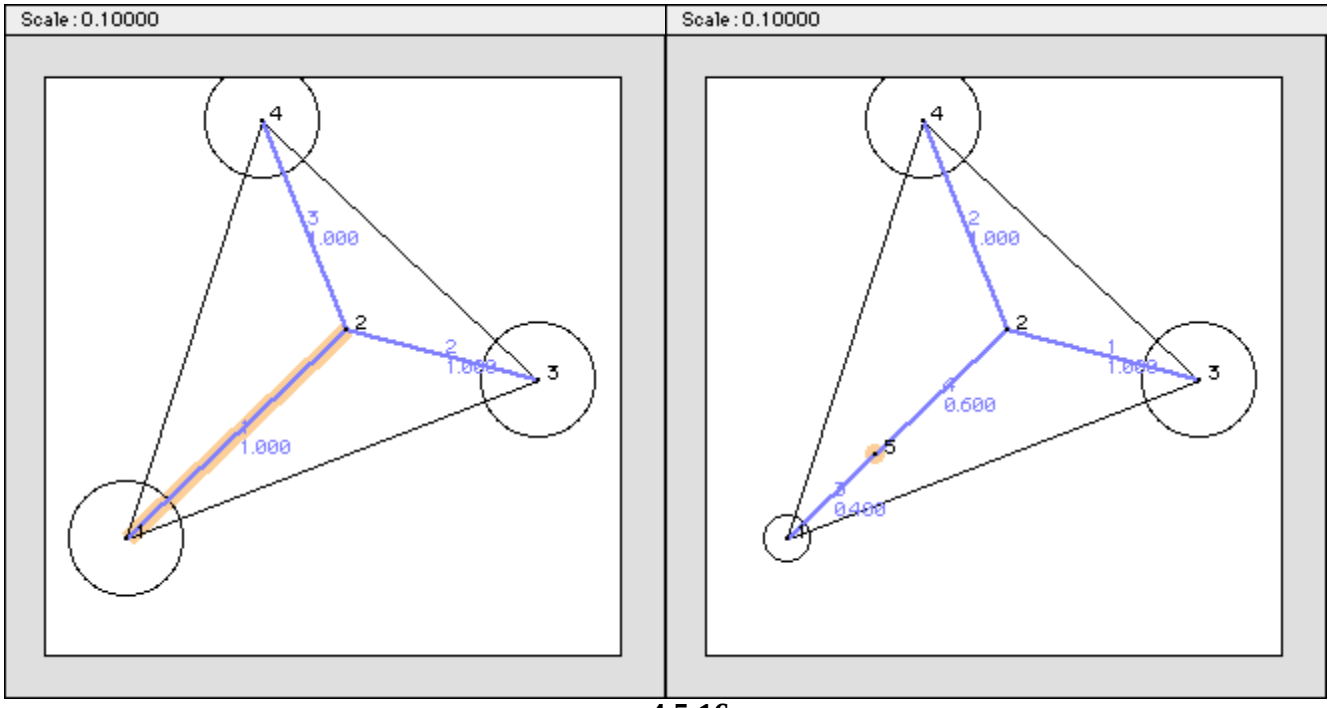
Split Selected Edge... ⌘H

This command lets you add a node somewhere in the middle of an existing edge. Selecting this comands puts up a dialog shown in figure 4.5.15.



4.5.15

You enter the distance the soon-to-be-created node should be from the indicated node (in tree units). You must enter a positive value less than the length of the edge. A new node will be added along the edge, breaking it into two separate edges. Just adding a node to an edge doesn't accomplish anything, but you can use the new node to add new branches to the tree. Figure 4.5.16 shows a before and after configuration.



4.5.16

Absorb Selected Node ⌘K

Nodes that have exactly two branches are unnecessary and are called “redundant” nodes. (the node added by the previous command is a redundant node, at least until you attach something to it.) If you select a redundant node, this command absorbs it into the edges it connects and joins the edges into a single edge. This is the inverse of **Split Selected Edge**.

Absorb Redundant Nodes ⌘-

This command absorbs all redundant nodes in the tree. It’s useful for cleaning up a tree if you fractured one or more polys and then subsequently deleted all of the stubs.

Remove Strain ⌘]

This command sets the strain of all edges to zero while leaving their lengths unchanged.

Relieve Strain ⌘[

This command changes the lengths of the edges to be equal to their strained length and then sets the strain equal to zero.

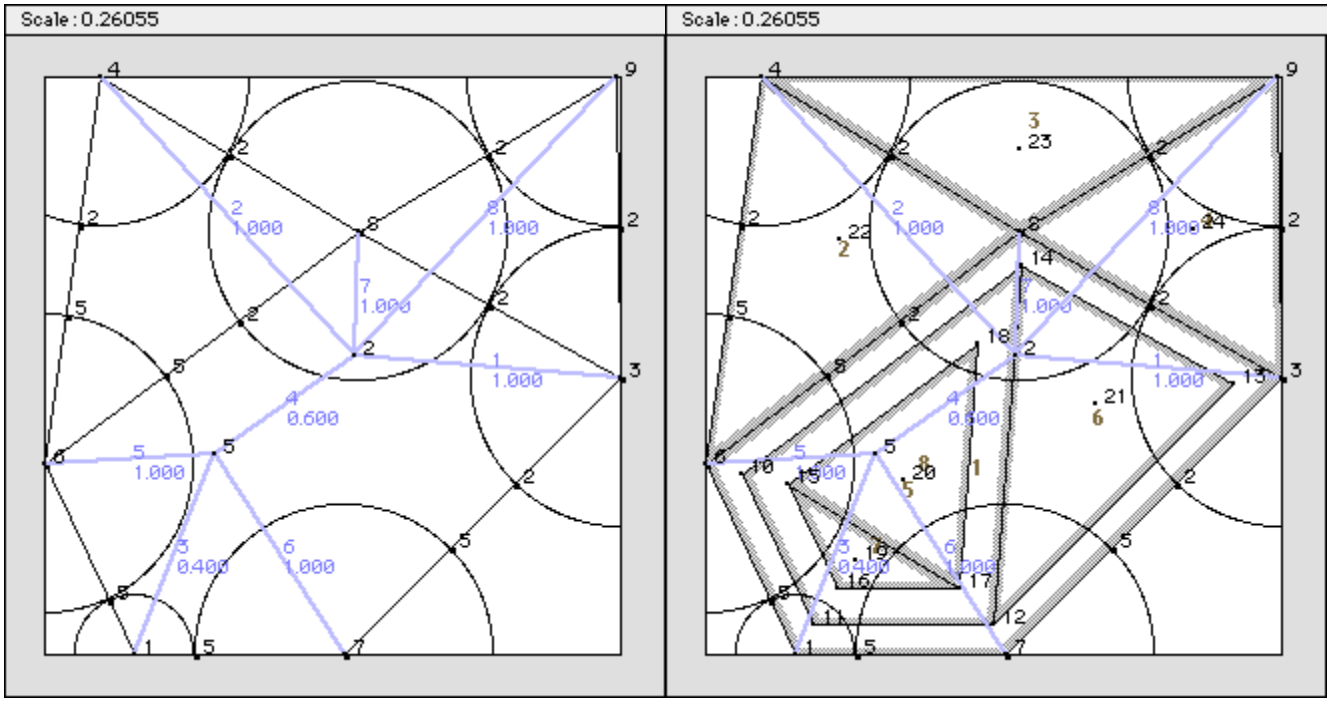
Minimize Strain ⌘3

This command does another optimization that minimizes the RMS strain of the node configuration. This is the command to use when many path active conditions have been placed on the nodes, particularly if they are coupled with quantized angles; often it is not possible to satisfy all conditions without straining different edges by different amounts.

Note: it is very easy to set up conditions that admit no solution at all. A sign that you have done this is that the optimizer will suddenly set the scale nearly equal to zero.

Build Polygons ⌘4

Build all of the polygons in the crease pattern. This command constucts polys from active and border paths; it also computes all the reduced paths and reduced polygons, according to the theory of the "universal molecule" construction. (The algorithm that does this is one of the centerpieces of my origami design theory.) Figure 4.5.17 shows a typical before and after for this command.



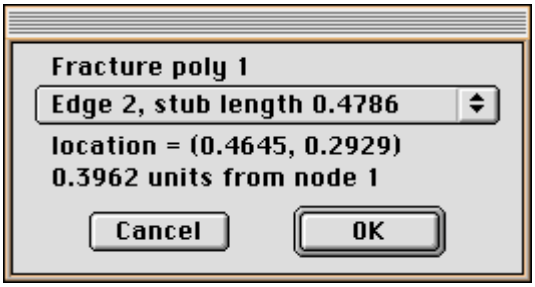
4.5.17

In the example above, the triangles are converted to triangle polygons, while the irregular pentagon at the bottom contains a set nested subpolygons.

Fracture Poly... ⌘;

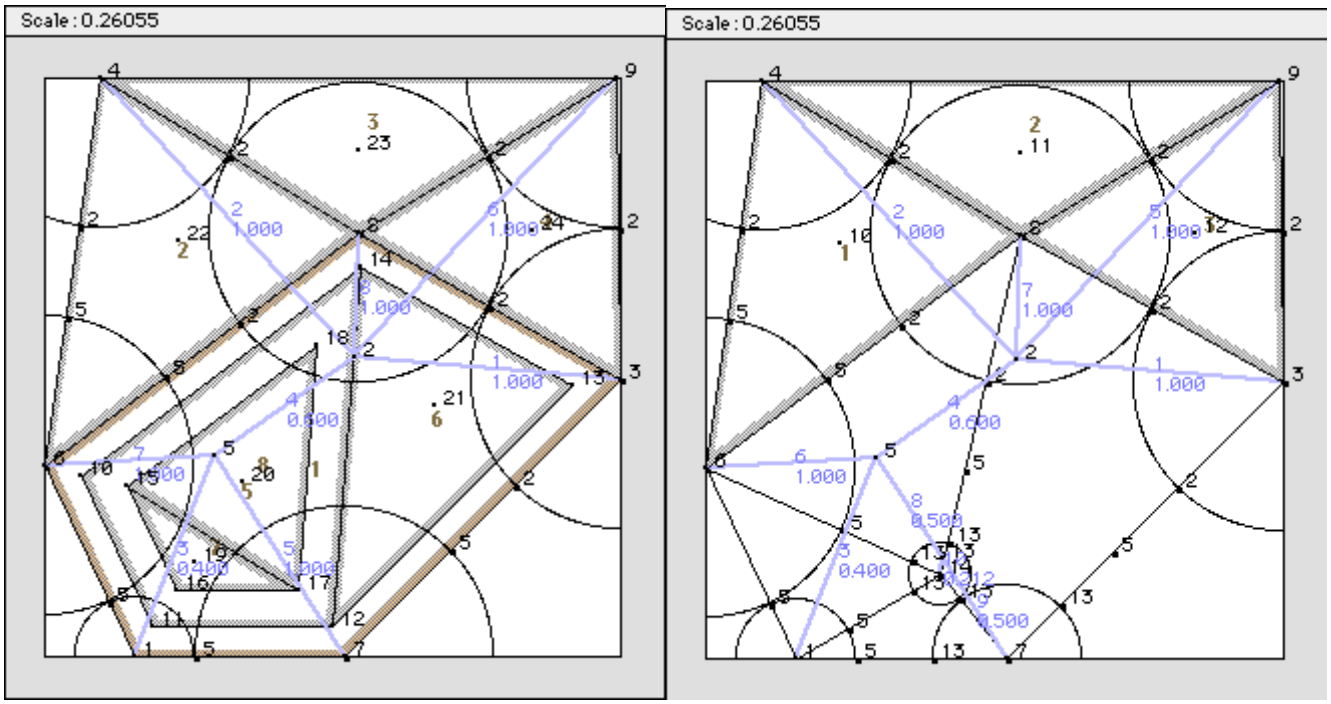
Any active polygon can be broken up into smaller active polygons by adding a stub to the corresponding subtree (this is a generalization of Toshiyuki Meguro's overlapping circle algorithm). If you add the stub to an existing node and optimize it using the **Optimize Selection** command, you can usually only break the polygon into three smaller polys. However, if one adds the degree of freedom to create a new node along an edge, then it's possible to break the poly into four smaller polys. (This is also a new algorithm I developed.) This command implements this improved algorithm.

You must select a major polygon to fracture, then select this command. A dialog is put up as shown in figure 4.5.18.



4.5.18

The dialog computes all possible ways to fracture the poly and displays them in a popup menu, cataloged by which edge is split and the length of the stub to be added. The location of the new node and the distance of the split from the end of the edge is shown immediately below the popup menu. Clicking OK results in the addition of a new terminal node and edge (a stub) to the tree, which fractures the polygon into lower-order polygons. Figure 4.5.19 shows before and after examples.

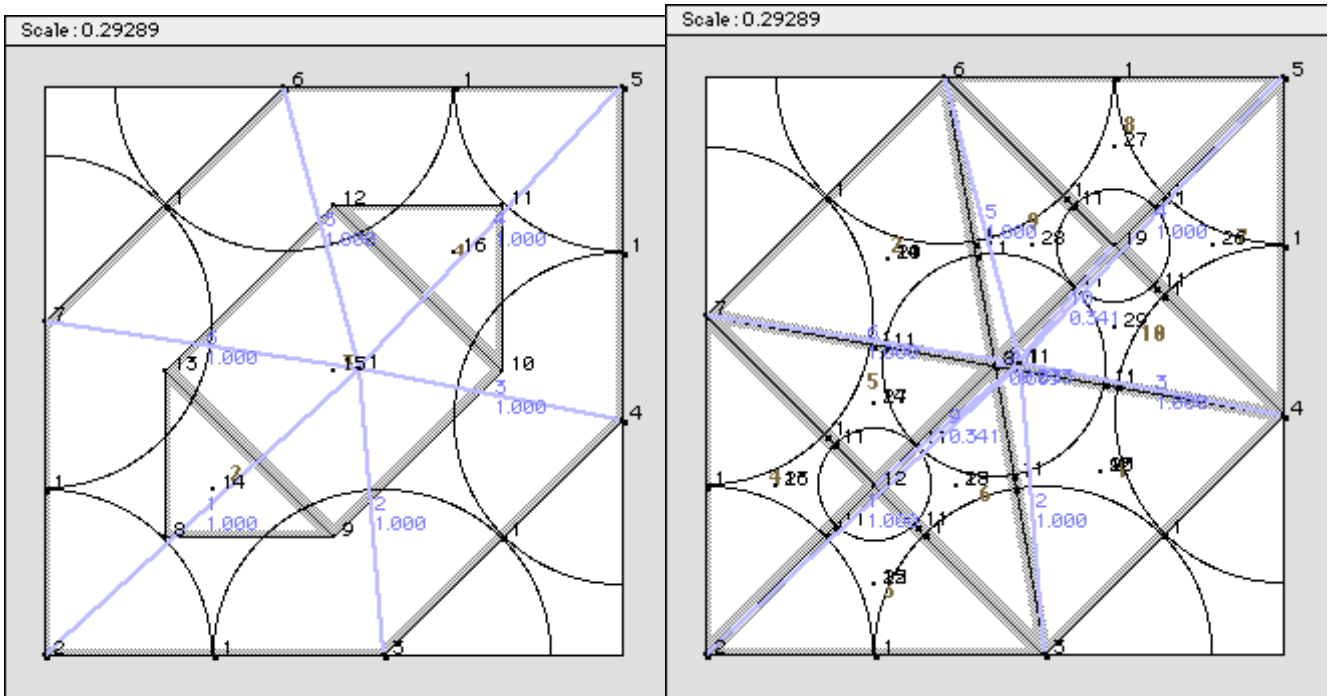


4.5.19

Note: you must explicitly call **Build Polys** again after fracturing a poly to build the new polys.

Triangulate Tree ⌘/

This command repeatedly tries to fracture *every* active polygon of order-4 or higher until all active polygons are triangles. If you then build creases, you will get all mountain, valley, and tristate creases in the model making rabbit-ear patterns. Figure 4.5.20 shows a typical before and after example.



4.5.20

Remove Polygons ⌘B

This command removes all polygons. If you are working with arranging nodes, the presence of polygons can be a distraction; this command gets rid of them all.

Build Creases ⌘B

Build all of the creases in the crease pattern. Creases will only be built for existing polygons, so you need to have built all polygons prior to calling this routine.

A typical sequence of commands after defining the tree might be:

1. **Optimize Scale** to maximize the scale and get the largest possible base.

2. **Select All** followed by **Optimize Selection** to maximize any unpinned edges.

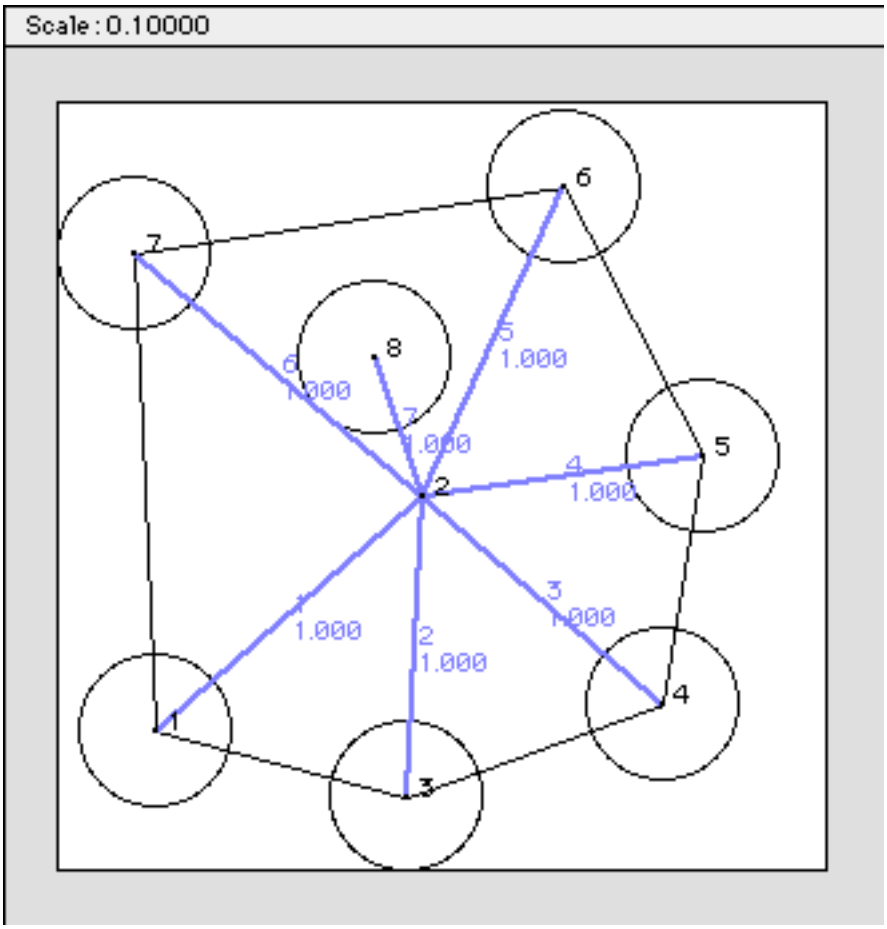
3. **Build Polygons**.

4. **Triangulate Tree** (optional)

5. **Build Creases**.

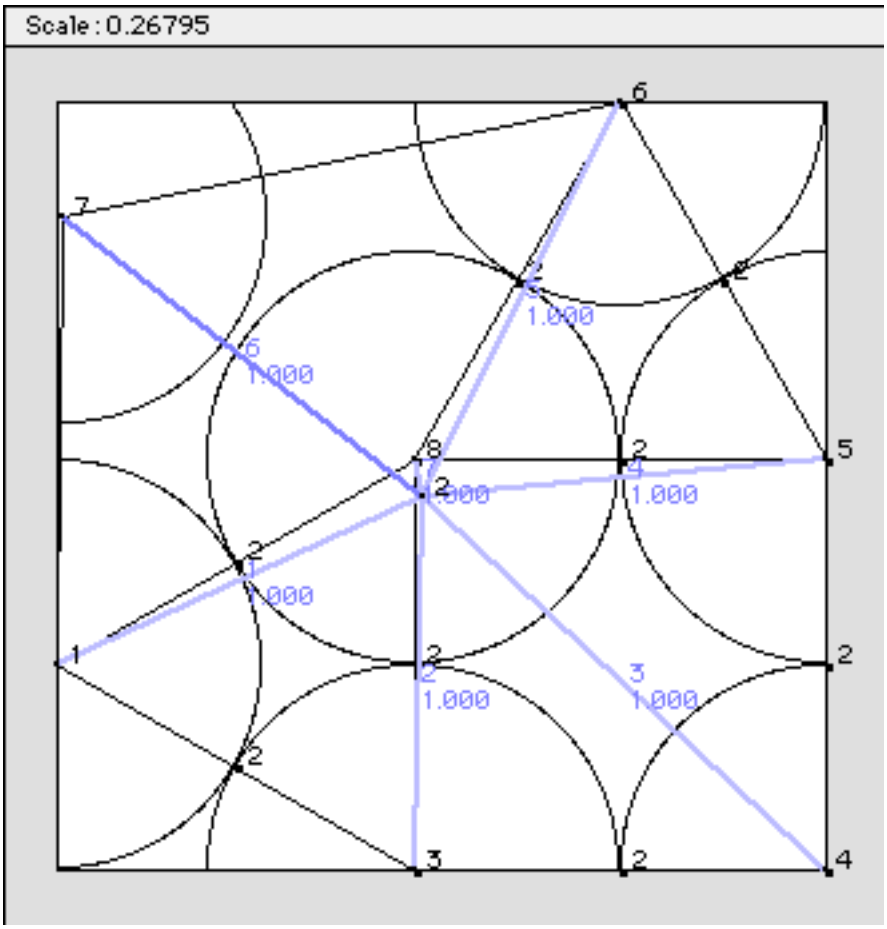
By the time you are done, the main window is likely to be rather cluttered, so it's a good idea to select **Creases Only** to display just the crease pattern. The figures below illustrate this entire sequence.

First, draw the initial tree as shown in figure 4.5.21.



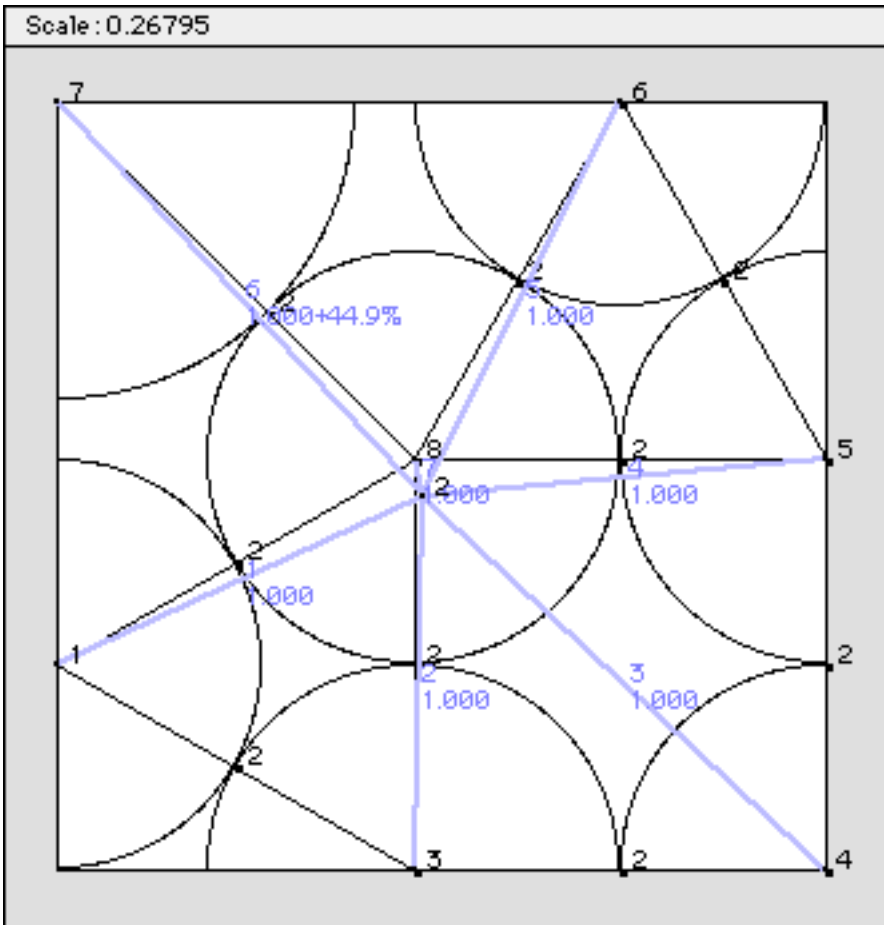
4.5.21

Then select **Optimize Scale**, **Action** menu, to find the configuration of nodes that gives the largest possible scale, as shown in figure 4.5.22.



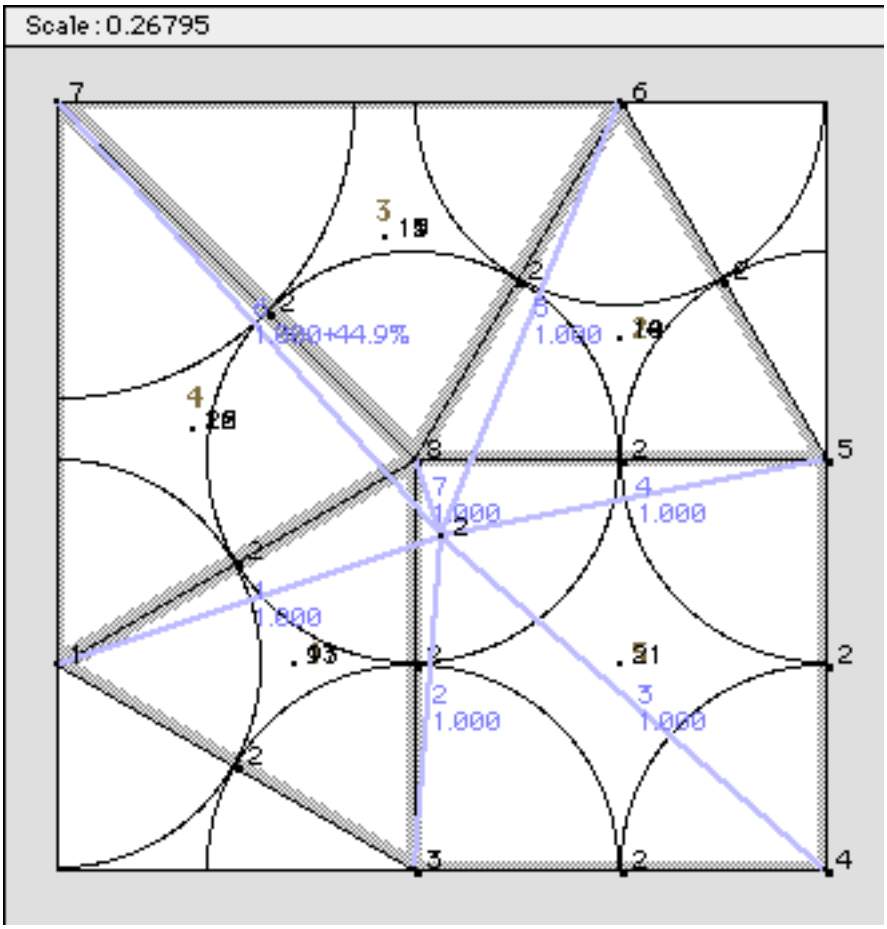
4.5.22

Most of the nodes and edges are pinned, but node 7 isn't pinned and edge 6 could be larger. So, select node 7 and edge 6 and then select the **Optimize Selection** command, **Action** menu, giving the result shown in figure 4.5.23.



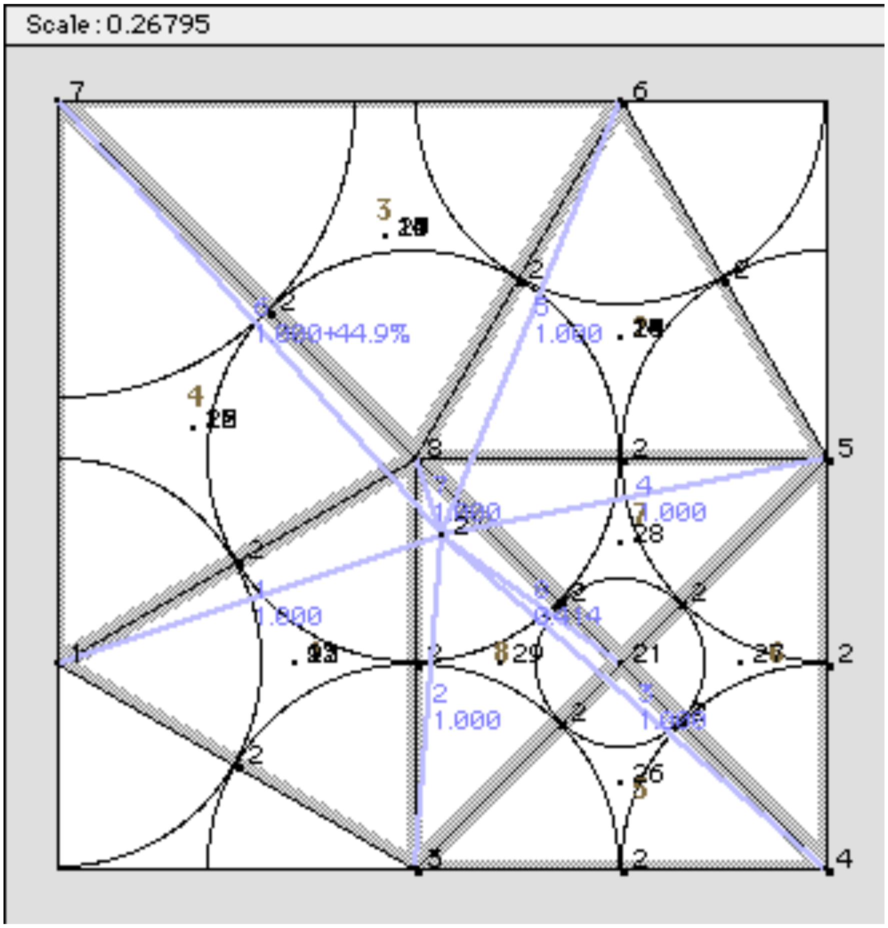
4.5.23

Edge 7 has been strained (lengthened) by 44.9%, which now puts edge 7 in the corner of the paper. Now select the **Build Polys** command, **Action** menu, to construct polygons from the active paths, as shown in figure 4.5.24.



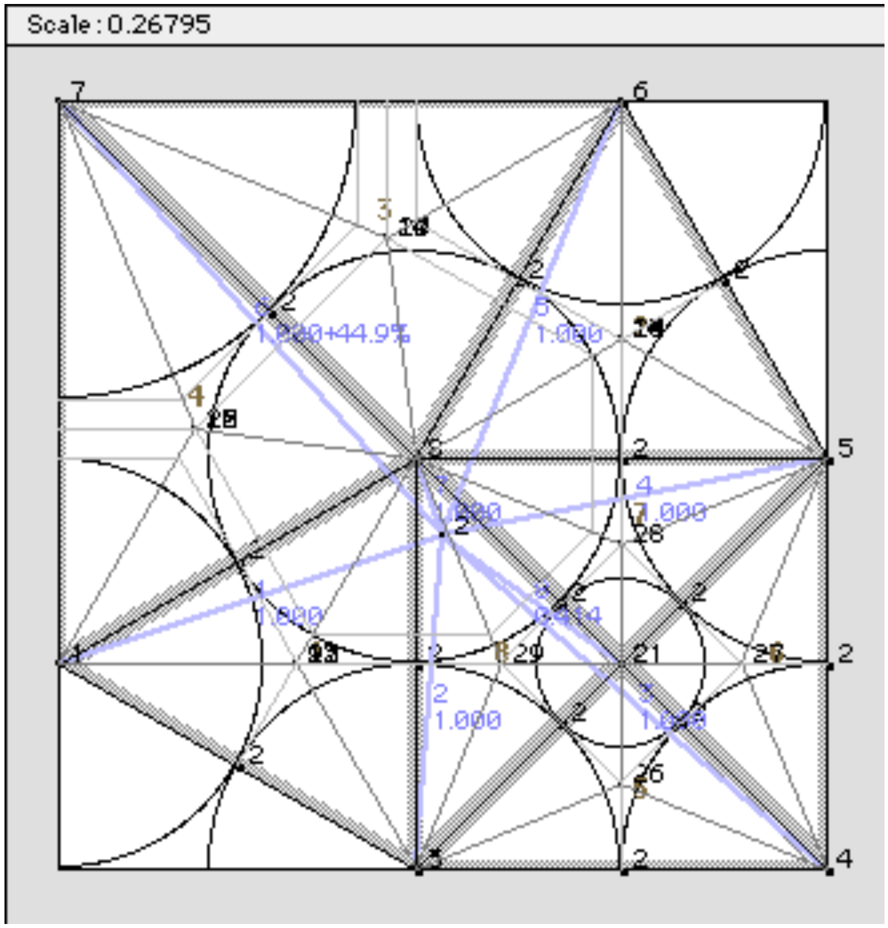
4.5.24

There are five polygons: four triangles and a quadrilateral. Although we could build a crease pattern from this, we can also break the entire crease pattern into triangles. Select the **Triangulate Tree** command, **Action** menu, which adds a stub to the tree and breaks the lower polygon into four triangles, as shown in figure 4.5.25.



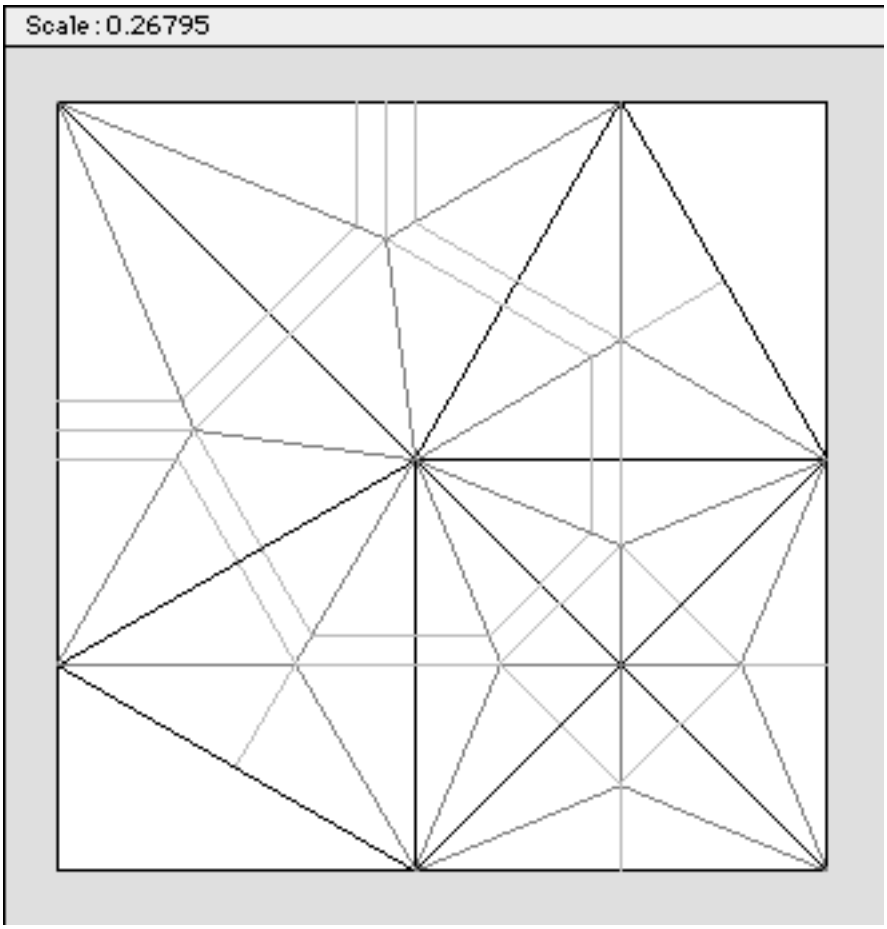
4.5.25

Now select **Build Creases**, **Action** menu, to construct the crease pattern, as shown in figure 4.5.26.



4.5.26

The origami design is complete. This view is very busy because it shows all of the sub-structure of the design. Select **Creases Only** command, **View** menu, to display only the crease pattern, shown in figure 4.5.27.



4.5.27

The origami design is complete!

Remove Creases	⌘R

Removes all creases from the image.

Build Polys & Creases	⌘6

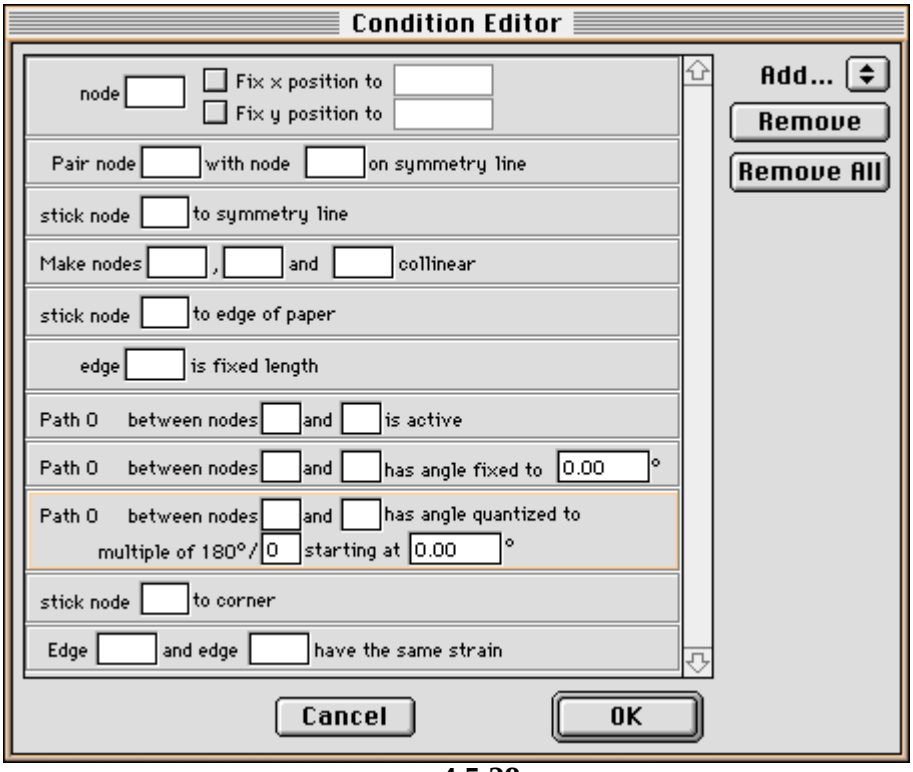
Builds both polygons and creases in a single step.

Conditions

The Conditions menu contains commands for applying or removing conditions from the origami design. A condition is a way of imposing additional requirements on the crease pattern beyond the bare minimum required for foldability. For example, you might want to require that the crease pattern be bilaterally symmetric (to match the subject), that certain flaps be corner or edge flaps (to minimize the total thickness of the flap); that certain flaps be the same length no matter how the tree is distorted due to strain; or that certain paths fall at particularly symmetric angles. The requirements are met by creating objects called “conditions” and linking them to the tree.

Edit Conditions...	⌘=

The **Edit Conditions…** command puts up a dialog that displays all conditions and their settings, as shown in figure 4.5.28.



4.5.28

Most of the Condition Editor is filled with a scrolling list of all conditions. The "Add…" popup menu lets you add conditions to the origami design. When you first create a condition, it has blank fields, which you must fill in with the indices of affected parts and any settings that apply to those parts.

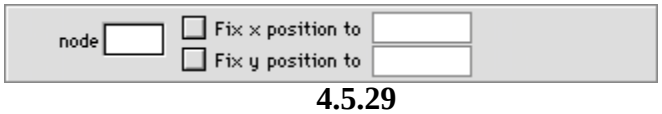
If you click on or in a condition, it will be highlighted (the third from the bottom is currently highlighted). If you then click on the "Remove" button, the highlighted condition will be deleted.

If you click on the "Remove All" button, all conditions will be deleted.

Conditions can be applied to nodes, edges, and paths. The Node Editor, Edge Editor, and Path Editor each contain a reduced version of the Condition Editor that will display only the conditions that apply to the particular node, edge, or path being edited.

The types of conditions that can be applied and their fields are the following:

Node fixed to position



4.5.29

This condition fixes one or both coordinates of a node to a set location. The fields are:

— the index of the node

— a checkbox if the *x* coordinate is to be fixed

— the desired value of the *x* coordinate

— a checkbox if the *y* coordinate is to be fixed

— the desired value of the *y* coordinate

Node fixed to corner

4.5.30

This condition insures that a node lies on one of the four corners of the paper. It's used to make a flap a corner flap. The fields are:

— the index of the node

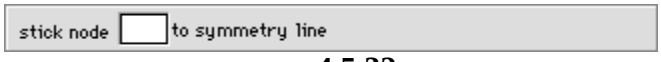
Node fixed to edge

4.5.31

This condition insures that a node lies on one of the edges of the paper. It's used to make flap an edge flap. The fields are:

— the index of the node

Node fixed to symmetry line



4.5.32

This condition insures that a node lies on the symmetry line of the model. It's used to make flaps that lie on the center line of a bilaterally symmetric subject. The fields are:

— the index of the node

Nodes paired

4.5.33

This condition insures that two nodes are mirror images of each other about the symmetry line of the base. It's used to make two identical flaps on the left and right sides of the model. The fields are:

— the index of the first node

— the index of the second node

Nodes collinear

4.5.34

This condition is used to force three nodes to be collinear. It's used to make crease patterns more symmetric. The fields are:

— the index of the first node

— the index of the second node

— the index of the third node

Note: a more effective way of making a crease pattern symmetric is to use the "Path Angle Fixed/Quantized" conditions.

Edge length fixed

4.5.35

This condition is used to force an edge to have zero strain in a strain optimization. The fields are:

— the index of the edge

Edges same strain

4.5.36

This condition is used to force two edges to have the same strain in a strain optimization. The fields are:

— the index of the first edge

— the index of the second edge

Note: you can force a group of edges to all have the same strain by binding each edge in the group to a common edge with this condition.

Path active

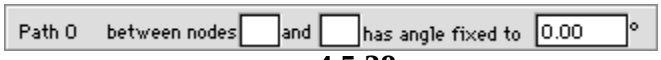
4.5.37

This condition is used to force a particular path to be active, i.e., to have its actual length equal to its minimum length. This will insure that there is a crease between the two nodes that runs along the axis of the model. The fields of this condition are:

— the index of the first node

— the index of the second node

Path angle fixed



4.5.38

This condition is used to force a particular path to be both active and to lie at a given angle. It's used to make a crease pattern more symmetric. Its fields are:

— the index of the first node

— the index of the second node

— the angle that the path is forced to lie at

Note: if you have very many path angle fixed or path angle quantized constraints, it is very easy to create incompatible constraints. Sometimes incompatible constraints can be relieved by performing a strain optimization; other times you will simply have to remove some conditions.

Path angle quantized

4.5.39

This condition is used to force a particular path to be active and to lie at one of a group of angles, which are integer divisions of 180°. It's used to make a crease pattern more symmetric by requiring that major creases lie at "nice" angles. The fields are:

— the index of the first node

— the index of the second node

— the integer divisor for the angle quantization

— the first angle in the group

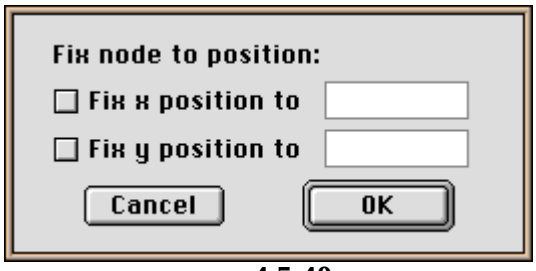
By applying this condition to most or all of the active paths in a model and performing a strain optimization (which is usually necessary to satisfy all conditions), you can create a very symmetric crease pattern, which is usually necessary if you want a sequentially foldable design.

The subsequent commands in the conditions menu apply and give the status of each of the types of conditions. These commands may have a mark next to them giving the status of the current

selection. Selecting one of the commands toggles on/off the given condition as applied to the current selection.

Node fixed to position...

This command applies/removes a “Node fixed to position” condition to the selected node. It is only enabled if exactly one node is selected. If the command is checked, there is already a condition and selecting this command will remove it. If the command is unchecked, selecting this command will put up a dialog shown in figure 4.5.40.



4.5.40

The fields of the dialog are:

Fix *x* position to — on = this node’s *x* coordinate will be set to a fixed value

Fix *y* position to — on = this node’s *y* coordinate will be set to a fixed value.

The command has the following marks and effects:

No mark: the selected node does not have this condition. Selecting the command applies a condition to the selected node.

Check mark: the selected node has this condition. Selecting the command removes this type of Condtion from the selected node.

Node fixed to corner

This command applies/removes a “Node fixed to corner” condition to the selected node(s). It is enabled if at least one node is selected. The command has the following marks and effects:

No mark: none of the selected nodes have this condition. Selecting the command applies a condition to each of the selected nodes.

Diamond mark: some of the selected nodes have this condition. Selecting the command applies a condition to all of the selected nodes.

Check mark: all of the selected nodes have this condition. Selecting the command removes this type of Condtion from each of the selected nodes.

Node fixed to edge

This command applies/removes a “Node fixed to edge” condition to the selected node(s). It is enabled if at least one node is selected. The command has the following marks and effects:

No mark: none of the selected nodes have this condition. Selecting the command applies a condition to each of the selected nodes.

Diamond mark: some of the selected nodes have this condition. Selecting the command applies a condition to all of the selected nodes.

Check mark: all of the selected nodes have this condition. Selecting the command removes this type of Condtion from each of the selected nodes.

Node fixed to symmetry line

This command applies/removes a “Node fixed to symmetry line” condition to the selected node(s). It is enabled if at least one node is selected. The command has the following marks and effects:

No mark: none of the selected nodes have this condition. Selecting the command applies a condition to each of the selected nodes.

Diamond mark: some of the selected nodes have this condition. Selecting the command applies a condition to all of the selected nodes.

Check mark: all of the selected nodes have this condition. Selecting the command removes this type of Condtion from each of the selected nodes.

Nodes paired

This command applies/removes a “Nodes paired about symmetry line” condition to a selected pair of nodes. It is enabled only if you have selected exactly two nodes. The command has the following marks and effects:

No mark: neither of the selected nodes has this condition. Selecting the command applies a condition binding the two nodes together.

Diamond mark: one or both of the selected nodes have this condition but not with each other. Selecting the command removes the existing pair conditions and applies a condition binding the two nodes together.

Check mark: the two nodes currently have this condition applied to them. Selecting the command removes the condition.

Nodes collinear

This command applies/removes a “Nodes Collinear” condition to a selected triplet of nodes. It is enabled only if you have selected exactly three nodes. The command has the following marks and effects:

No mark: none of the selected nodes has this condition. Selecting the command applies a condition binding the three nodes together.

Diamond mark: one of the three selected nodes have this condition but they don’t form a triplet. Selecting the command removes the existing triplet conditions and applies a condition binding the three nodes together.

Check mark: the three nodes currently have this condition applied to them. Selecting the command removes the condition.

Remove selected Node conditions

This command removes all conditions that apply specifically to this node. It does not affect Edge conditions or path conditions.

Remove all Node conditions

This command removes all conditions that apply to all nodes. It does not affect edge conditions or path conditions.

Edge length fixed

This command applies/removes a “Edge length fixed” condition to the selected edge(s). It is enabled if at least one edge is selected. The command has the following marks and effects:

No mark: none of the selected edges have this condition. Selecting the command applies a condition to each of the selected edges.

Diamond mark: some of the selected edges have this condition. Selecting the command applies a condition to all of the selected edges.

Check mark: all of the selected edges have this condition. Selecting the command removes this type of Condtion from each of the selected edges.

Edges same strain

This command applies/removes a “Edges same strain” condition to a selected pair of edges. It is enabled only if you have selected exactly two edges . The command has the following marks and effects:

No mark: neither of the selected edges has this condition. Selecting the command applies a condition binding the two edges together.

Diamond mark: one or both of the selected edges have this condition but not with each other. Selecting the command removes the existing pair conditions and applies a condition binding the two edges together.

Check mark: the two edges currently have this condition applied to them. Selecting the command removes the condition.

Remove selected Edge conditions

This command removes all conditions that apply specifically to this edge. It does not affect node conditions or path conditions.

Remove all Edge conditions

This command removes all conditions that apply to all edges. It does not affect node conditions or path conditions.

Path active ⌘,

This command applies/removes a “Path active” condition to the selected path(s). It is enabled if at least one path is selected. The command has the following marks and effects:

No mark: none of the selected paths have this condition. Selecting the command applies a condition to each of the selected paths .

Diamond mark: some of the selected paths have this condition. Selecting the command applies a condition to all of the selected paths .

Check mark: all of the selected paths have this condition. Selecting the command removes this type of Condtion from each of the selected paths.

Path angle fixed...

This command applies/removes a “Path angle fixed” condition to the selected path(s). It is enabled if at least one path is selected. The command has the following marks and effects:

No mark: none of the selected paths have this condition. Selecting the command applies a condition to each of the selected paths. It puts up a dialog for the path angle shown in figure 4.5.41.

4.5.41

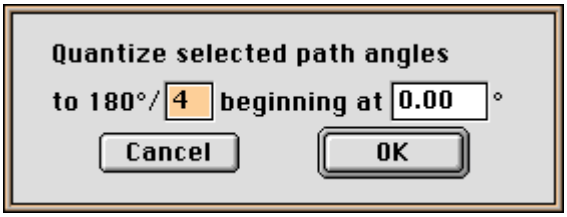
Diamond mark: some of the selected paths have this condition. Selecting the command applies a condition to all of the selected paths. It also puts up the dialog for the path angle.

Check mark: all of the selected paths have this condition. Selecting the command removes this type of Condtion from each of the selected paths.

Path angle quantized... ⌘

This command applies/removes a “Path angle quantized” condition to the selected path(s). It is enabled if at least one path is selected. The command has the following marks and effects:

No mark: none of the selected paths have this condition. Selecting the command applies a condition to each of the selected paths. It puts up a dialog for the path angle quantization shown in figure 4.5.42.



4.5.42

Diamond mark: some of the selected paths have this condition. Selecting the command applies a condition to all of the selected paths. It also puts up the dialog for the path angle quantization.

Check mark: all of the selected paths have this condition. Selecting the command removes this type of Condtion from each of the selected paths.

Remove selected Path conditions

This command removes all conditions that apply specifically to this path. It does not affect node conditions or edge conditions.

Remove all Path conditions

This command removes all conditions that apply to all paths. It does not affect node conditions or edge conditions.

Remove all conditions ⌘`

This command removes all conditions in the design.

5.0 The Tree Method of Origami Design

This section describes the mathematical ideas that underly the tree method of origami design — and technical origami, or *origami sekkei* — in particular. The Japanese art of origami, though centuries old, has undergone a renaissance in the last few decades from new advances in origami design. The simple stylized birds, flowers, and fish of old have been replaced by incredibly complex replicas of insects, crustacea, mammals, amphibians, dinosaurs, humans, objects, and just about everything under the sun. The explosion of subject matter was driven in part by new discoveries in design techniques. One of the most significant of these discoveries was a series of mathematical algorithms for designing origami bases in terms of the number, size, and arrangement of their points. This document section some of those algorithms.

Even before mathematics came into the picture, the concept of "points" — flaps that become appendages of an origami model — and their number, length, and connections have figured heavily in the origami designer's approach to creating new folds. Over the past few years, I and several folders have been formulating mathematical techniques for designing origami models on the basis of their points. We have discovered several general algorithms for design, algorithms that enable one to compute the crease pattern for a base containing any number of points of arbitrary complexity. A complete description of such algorithms and all of their nuances would fill a book — in fact, I am working on that very task — but the basic technique can be quickly outlined and may be of interest to folders interested in exploring new bases and symmetries.

I am going to describe an algorithm using terminology and concepts that I developed in my own research. However, I should acknowledge major contributions from other mathematical folders. In any scientific endeavor, the rate of progress is vastly accelerated through the interaction and cross-fertilization of ideas among several investigators and those who have contributed to this theory are many: they include Jun Maekawa, Fumiaki Kawahata, Toshiyuki Meguro, Alex Bateman and Tom Hull, with additional insights from John Montroll, Peter Engel, Issei Yoshino, Brian Ewins, Jeremy Shafer, and Martin and Erik Demaine. My thanks and acknowledgment to these, as well as the many more folders whose works formed both inspiration and guide for my work over the years.

And now, on with design.

The name "technical folding" has come to be applied to origami designs that, whether intentionally or not, display a strong geometrical elegance and sophistication. The target concept for much of technical folding is the **base**. A base is simply a geometric shape that resembles the subject to be folded — usually, a shape having the same number and length of flaps as the subject has appendages. For example, a base for a bird might have four flaps, corresponding to a head, tail, and two wings. A slightly more complicated subject such as a lizard would require a base with six flaps, corresponding to the head, four legs, and a tail. And an extremely complicated subject such as a flying horned beetle might have six legs, four wings, three horns, two antennae and an abdomen, requiring a base with sixteen flaps. You needn't stop there, either. Throw in mouthparts and a separate flap for the thorax, if you like, for a total of nineteen flaps. The number of flaps required in a model depends on the level of anatomical accuracy desired by the designer. Historically, much origami design was performed by trial and error — manipulating a piece of paper until it began to resemble something recognizable. However, for a complex subject, trial and error is a highly inefficient approach, since one is unlikely to stumble upon a

nineteen-pointed base with flaps of the right size in the right places purely by luck. For such models, a more directed approach is called for.

For a given number and distribution of flaps, there are always many possible bases that can be folded that differ in various ways: the overall size of the base, which flaps have how many layers, how the flaps are positioned in the plane of the paper, the stacking order of the layers, and so forth. I doubt that there is a universal algorithm that could come up with every possible base for a given subject; at any rate, most folders would be satisfied with an algorithm that could at least come up with *one*. It turns out that such an algorithm exists for a particular class of base. We will consider bases that have a special property: their flaps and layers can be arranged such that (1) one edge of every flap lies in a common plane and (2) when the edges all lie in a plane, all of the layers of the base are perpendicular to the plane. I call this type of base a **uniaxial base** (because when the base is flattened, all of the flaps lie along a single common axis). An example of a hypothetical uniaxial base for a hypothetical four-legged (and hence six-flapped) creature is shown in figure 5.1.

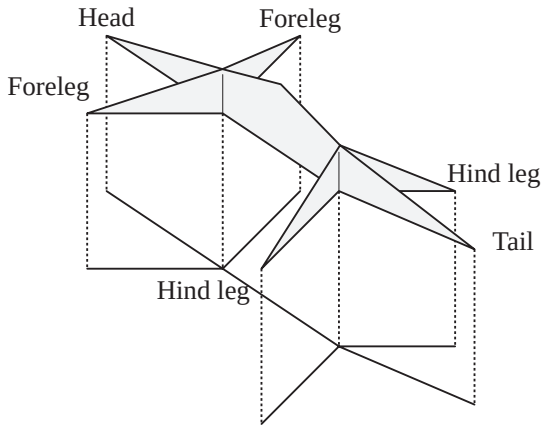


Figure 5.1. Schematic of a uniaxial base for an animal with four legs, a head, body, and tail. It's a uniaxial base if it can be manipulated so that all of the flaps lie in a plane and all of the layers are perpendicular to the plane. The shadow of the base consists entirely of lines.

While a great many origami bases can be represented as uniaxial bases including the four classic bases — the kite, fish, bird, and frog bases — many others cannot. For example, John Montroll's Dog Base and its derivatives do not fall into this category. But there are many bases that do, and most importantly from the standpoint of design, for a great many subjects, a uniaxial base is a sufficiently good approximation of the subject that you can easily transform it into a very accurate and detailed replica of the subject.

We can represent any uniaxial base by the stick figure formed by the projection of the base into the plane, i.e., the shadow cast by the base if illuminated from above. This stick figure has the same number and length sticks as the base has flaps, and therefore it is a convenient abstraction of the base, even as the base is an abstraction of the subject which we are trying to fold. The problem to be solved is: for a given stick figure, can one find a way of folding a square into a base whose projection is that stick figure?

Let's first define some terms describing this stick figure. The mathematical term for a stick figure is a **graph**. I'll call the particular stick figure for a uniaxial base the graph of the uniaxial base.

Each line segment in the graph is called an **edge** of the graph; each endpoint of a line segment (or point where two segments come together) is called a **node**. Each flap of the base corresponds to a distinct edge of the graph, and since each flap has a specified length, each edge of the graph has a length as well, which is the length of its associated flap. Thus, the planar graph for the six-legged animal base shown in figure 5.2 has eight nodes and seven edges. In figure 2, I have assigned a length of 1 to each edge of the graph, which implies that each segment in the base has the same length. One can, of course, assign any desired combination of lengths — if you wanted an extra-long tail, you could give it a length of 2 or 3 — but for simplicity, we'll make all of the edges the same in this example.

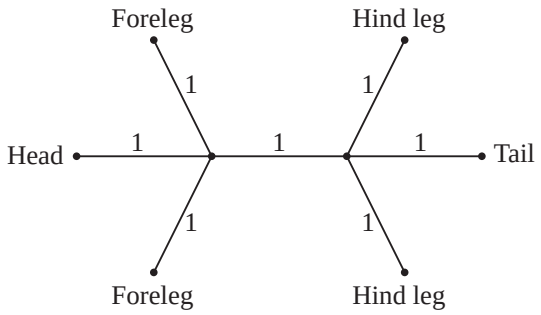


Figure 5.2. Planar graph for the six-legged base. Each edge of the graph has a length of 1 unit in this example.

We will also restrict ourselves to bases that don't contain any loops of paper, which correspond to planar graphs that don't have any cycles in them. If a base has a closed loop, that implies it has a hole in it somewhere, which would imply cutting — a no-no, in origami design. (Of course, we can always simulate a loop by joining two flaps together and hiding the joint.) A graph with no cycles is called a **tree graph** or just **tree**. It's easy to see that for any tree , the number of nodes is always exactly one more than the number of edges. Thus, we will be searching for bases whose projections are trees; for this reason, I call this design approach the *tree method* of design.

We will also divide the nodes into two types: **terminal nodes** are nodes that come at the end of a single edge. Terminal nodes correspond to the tips of legs, wings, and other appendages. Nodes formed where two or more edges come together are called **internal nodes**.

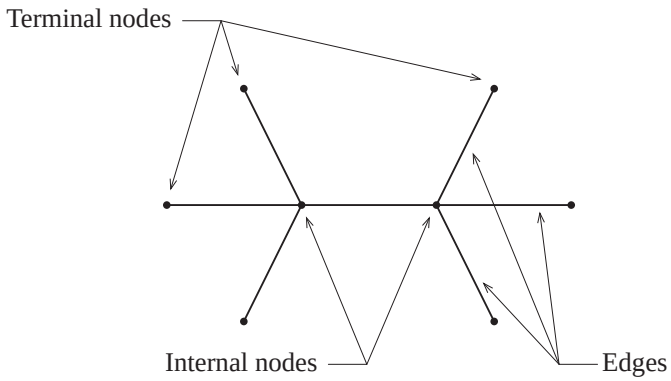


Figure 5.3. Parts of a tree graph. Line segments are called edges of the graph; points where lines end or come together are nodes. Terminal nodes are nodes with only one edge; internal nodes have two or more edges.

Suppose we have a base that has been folded from a square and we construct its tree. If we unfold the base, we get a square with a crease pattern that uniquely defines the base. The folding sequence that transforms the square into the base can be thought of as a mapping between points on the square and points on the tree.

If you think of the tree as the shadow of the base, you can see from figure 1 that wherever you have vertical layers of paper, there are many points on the base that map to the same point on the tree. However, at the terminal nodes of the tree, there is exactly one point on the square that maps to the node. Thus, we can uniquely identify the points on the square that become the tips of the flaps of the base.

Now consider the following thought experiment: suppose an ant were sitting at the tip of one of the legs of the base and wished to travel to another part of the base — say, the tail — without leaving the paper. It would have to walk down the leg to the body, down the body, and back out the tail. The distance it traveled would be (length of the leg) + (length of the body) + (length of the tail).

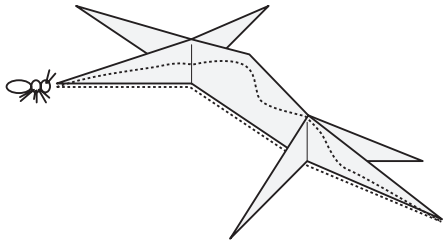


Figure 5.4. An ant wishes to go from a foreleg to the tail along the base. It can take several different paths, but the most direct path is the path that lies in the plane of projection.

Now, let's think about what the path of the ant would look like on the unfolded square (you can imagine dipping the ant into ink so that it left a trail soaking through the paper as it walked). On the square, the path might meander around a bit or it might go directly from one point to the other, depending on the course of the ant. One thing is certain, however: the distance traveled on the unfolded square must be at least as long as the minimum distance traveled along the base.

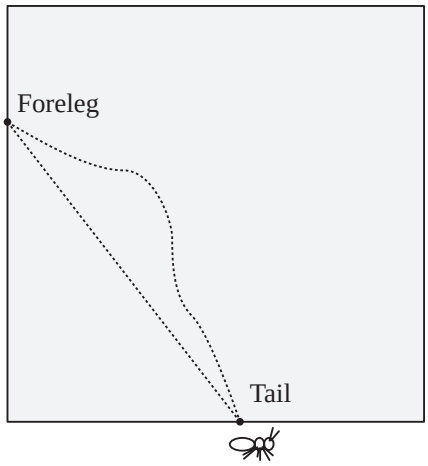


Figure 5.5. The trail of the ant. The path of the ant might wander around on the square once you've unfolded it, but there is no way that the path can be shorter on the square than the path was on the base. Thus, the distance between two nodes on the square must be at least as large as the distance between the two nodes measured along the edges of the tree.

This illustrates an extremely important property of any mapping from a square to a base: the distance between *any* two points on the square must be greater than or equal to the distance between the two corresponding points on the base. And in particular, this relationship must hold for any two points on the base that correspond to nodes on the tree. Now while this condition must hold for *any* pair of points on the base, it turns out that if it holds for any pair of terminal nodes, it will hold for *every* pair of points on the base. That is, if you identify a set of points on the square corresponding to terminal nodes of a tree and the points satisfy the condition that the distance between any pair of points on the square is greater than or equal to the distance between the points as measured on the graph, then *it is guaranteed that a crease pattern exists to transform the square into a base whose projection is the tree*.

This is a remarkable property. It tells us that no matter how complex a base is sought, no matter how many points it may have and how they are connected to one another, we can *always* find a crease pattern that transforms the square (or any other shape paper, for that matter) into the base. Putting this into mathematical language, we arrive at the fundamental theorem of the tree method of design (which I call the "tree theorem" for short):

> Define a simply connected tree P with terminal nodes P_i, i=1, 2,…N. Define by l_{ij} the distance between nodes P_i and P_j as measured along the edges of the tree; that is, l_{ij} is the sum of the lengths of all the edges between nodes P_i and P_j. Define a set of points $\mathbf{u}_i$ in the unit square $u_{i,x} \in [0,1]$, $u_{i,y} \in [0,1]$. Then a crease pattern exists that transforms the unit square into a uniaxial base whose projection is P if and only if $|\mathbf{u}_i - \mathbf{u}_j| \leq l_{ij}$ for every i,j. Furthermore, in such a base, P_i is the projection of $\mathbf{u}_i$ for all i.

Although the proof of the tree theorem is beyond the scope of this article[1], we won't let that stop us from using it. The tree theorem tells us that if we can find a distribution of nodes on a square for which the distance between any two nodes is greater than or equal to the distance between the nodes on the tree, then a crease pattern exists that can transform that node pattern into a base. For a given tree, there are often several possible node patterns that satisfy the tree theorem, each of which yields a different base. For our six-pointed base, a little doodling with pen and paper will reveal that the pattern of nodes shown in figure 5.6 fits inside of a square of side $2\sqrt{((121+8\sqrt{179})/65)} \approx 3.7460$.

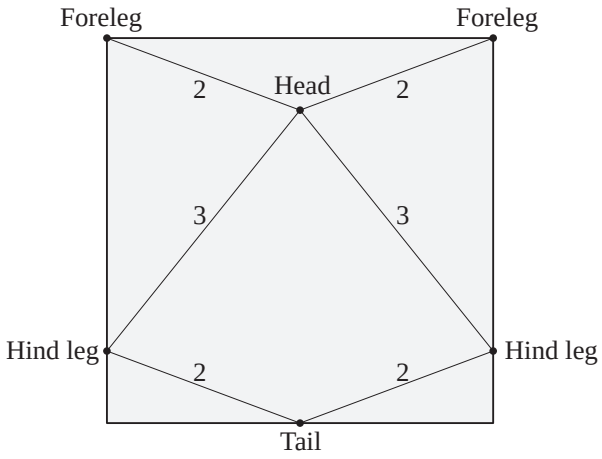


Figure 5.6. Node pattern that satisfies the tree theorem for the six-legged tree.

The tree theorem is an existence theorem; it says that a crease pattern exists but it doesn't tell us what this supposed crease pattern actually *is*. However, it does provide a strong clue. The tree theorem says that the points on the square that correspond to terminal nodes of the tree become the tips of the flaps on the folded base. Are there any other parts of the square that we can identify on the base?

Consider the inequality in the tree theorem. Two points corresponding to terminal nodes must be separated on the square by a distance *greater than or equal to* the distance between them on the tree. In the special case where equality holds, we can uniquely identify the line between the two points. We will call a line on the square that runs between any two terminal nodes a **path**. Every path has a **minimum length**, which is the sum of the edges of the tree between the terminal nodes that define the path. (In the symbolism of the tree theorem, l_{ij} is the minimum length of path ij.) The **actual length** of a path is given by the distance between the terminal nodes as measured upon the square ($|\mathbf{u}_i - \mathbf{u}_j|$ in the tree theorem). Any path for which its actual length is equal to its minimum length is called an **active path**.

On the base, the only route between two flap tips that is equal to the distance between the terminal nodes lies in the plane of the projection. Thus, any active path between two terminal

[1]For a proof of the tree theorem, see "A Computational Algorithms for Origami Design," 12th Annual Symposium on Computational Geometry, May 22–26, 1996, Philadelphia, PA.

nodes on the square becomes an edge of the base that lies in the plane of projection. Consequently, we have another important result:

> All active paths between terminal nodes map to an edge of the base in the plane of projection of the base.

Active paths on the square lie in the plane of projection of the square, but the plane of projection is where the vertical layers of paper in the base meet each other. Thus, active paths are not only edges of the base: they are major creases of the base.

So now we have the rudiments of the crease pattern for the base. We know the points on the square that correspond to terminal nodes of the tree become the tips of the flaps of the base, and we know that active paths on the square become major creases of the base. Furthermore, they are creases that lie in the plane of projection of the base. In many cases, this information alone is enough to find the complete crease pattern for the uniaxial base that corresponds to the tree. In many cases, though, it isn't. So we'll keep going.

If you draw all of the active paths on a square, you get a network of lines, each of which is a major crease of the base. Figure 6 includes the active paths for the node pattern for the six-legged animal, labeled with their length. Each of those lines will turn out to be a crease in the base.

Since the active paths map one-to-one to paths on the tree, any features on the tree can be mapped directly onto the active paths. Specifically, we can locate where on an active path each internal node falls. If our hypothetical ant travels from one terminal node to another encountering internal nodes at distances d_1, d_2, d_3, and so forth along the way, then when we draw the crease pattern, we can identify each internal node along the active path connecting the nodes at the same distances from the terminal nodes. Thus, we can add all of the internal nodes to our budding crease pattern. In figure 5.7, I've identified all of the nodes, terminal and internal, by a letter on the tree, and have added their locations to the active paths in the crease pattern on the square. Note that in general, an internal node will show up on more than one active path.

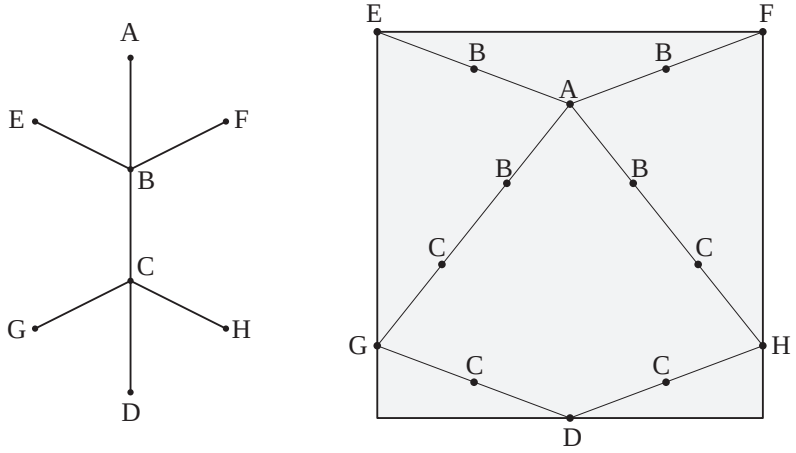


Figure 5.7. (Left). Tree with all nodes lettered. (Right) Crease pattern with terminal nodes, internal nodes, and active paths.

It's also worth pointing out that we don't show any terminal nodes along the edges of the square because the paths between nodes *G* and *E*, *E* and *F*, and *F* and *H* are not active paths.

It can be shown that active paths never cross each other, so the only places where lines come together on the square are terminal nodes. This pattern of lines breaks up the square into a set of polygons. In some of the polygons, all of their sides are active paths (like the inverted-kite-shaped quadrilateral in the center of figures 5.6 and 5.7). If one of the sides of a polygon lies on the edge of a square, it may or may not be an active path (in figure 5.6 and 5.7, each triangle has one side on the edge of the square that is not an active path.)

We'll call a polygon whose corners are terminal nodes and whose sides are active paths or the edge of the square an **active polygon**. Each active polygon has the property that all of its sides map to the plane of projection of the base when the square is folded into a base. Consequently, to find a crease pattern that collapses the square into the base, it is necessary to find a crease pattern that maps the network of active paths onto the plane of projection of the base.

This sounds like we have added requirements and made our job harder, but in fact we have actually simplified things. One of the things that makes origami design so difficult is the interrelatedness of all the parts of the base; if you try to change the dimensions of one part of the model, you have to take into account how that part interacts with the rest of the paper. But it turns out that when you have a square broken up into active polygons, you can separate the design into independent pieces. Recall that the tree is the projection of the base, which is folded from the complete square. Each polygon on the square corresponds to a portion of the overall base and if you collapse any polygon into a section of the base — which I call a **subbase** — the projection of the subbase is itself a portion of the projection of the complete base, i.e., a portion of the original tree. For example, figure 5.8 shows the polygons for our six-legged base and the corresponding trees for each subbase. Note that since all of the corners of an active polygon must be terminal nodes, the triangles at the bottom corners of the square are *not* active polygons and in fact do not contribute to the base at all.

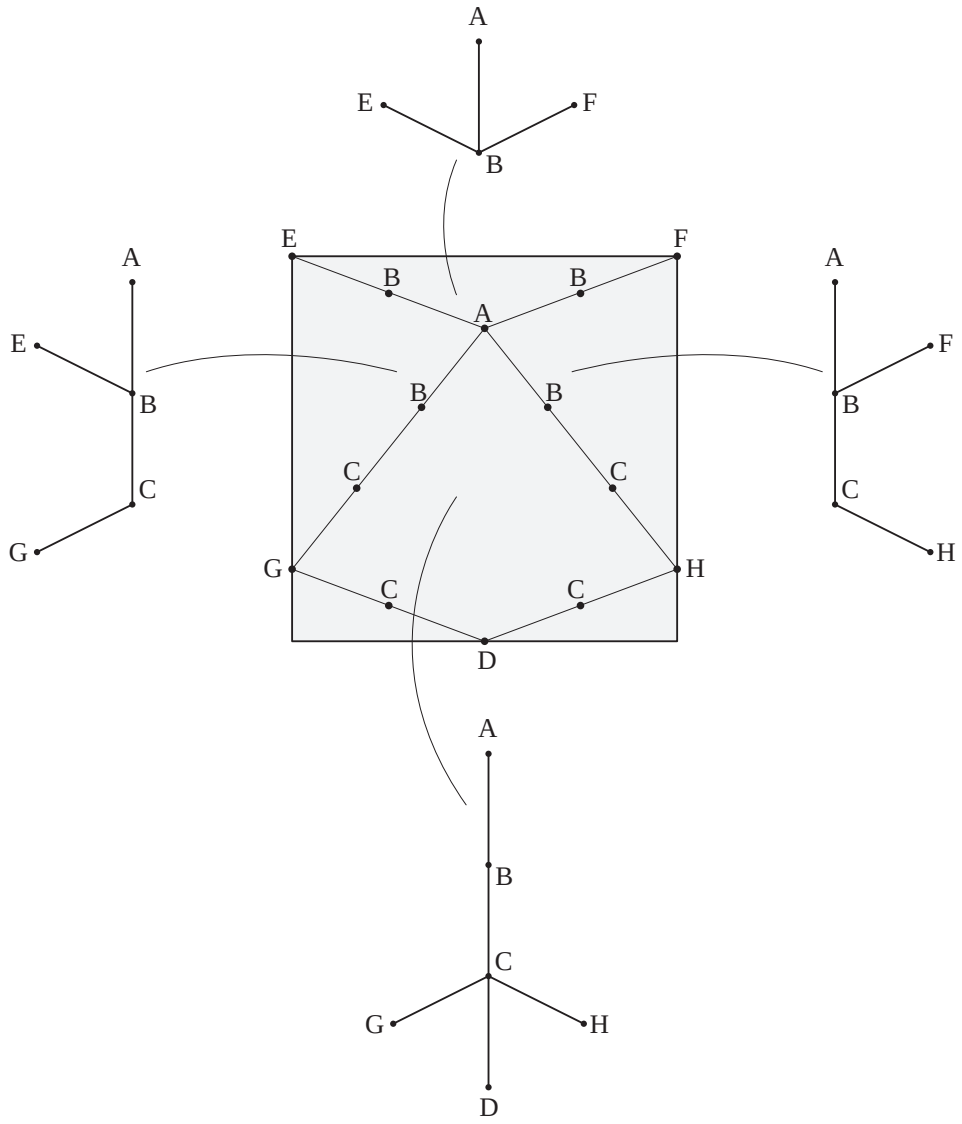


Figure 5.8. The four active polygons for the six-legged base and the trees corresponding to each subbase.

An incredibly useful property of active polygons is that although two polygons may share a common side, if that side is an active path, any crease pattern that collapses one polygon into a subbase is *always* compatible with a crease pattern that collapses the adjacent polygon into its subbase! If you like, for example, you could cut up the square into separate polygons, fold each polygon into a subbase, and then glue matching edges of the subbases back together and be assured that they would match up. This matching property is unique to active paths, so any two polygons that share a side that is an active path will be guaranteed to match up. The practical upshot of this is that in order to find the creases that collapse the entire base, we can find crease patterns that collapse each polygon into a section of the overall base, taking the polygons one at a time. So here is a way to break up the design of a complicated base into a bunch of much simpler bases. To collapse the entire square into a uniaxial base, it suffices to find a way to collapse each polygon individually into the appropriate subbase.

Now, there is conceivably an infinite number of polygons and a polygon can have any number of sides, so we're not out of the woods yet. We need to find a way to collapse an arbitrary polygon into its corresponding subbase. Let's look at a few simple cases.

We'll start with triangles. A triangle has three sides and three corners. Recall that the corners of active polygons are terminal nodes. Thus, the tree of a triangle must be a tree with three terminal nodes and vice-versa. There is only one graph with three terminal nodes, which is shown in figure 5.9. This tree corresponds to a uniaxial base with three flaps, whose lengths I have dubbed *a*, *b*, and *c*.

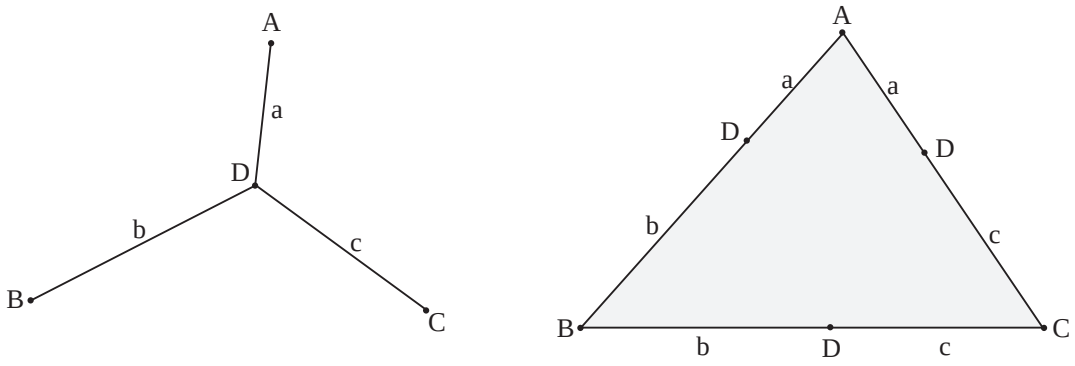


Figure 5.9. (Left) The tree with three terminal nodes, corresponding to a subbase with flaps of length *a*, *b*, and *c*. (Right) The corresponding triangle polygon.

Now, one of the assumptions we've made is that the sides of the active polygons are active paths. For a triangle whose three sides are active paths, that means we already know the lengths of the sides: they are, respectively, $(a+b)$, $(b+c)$, and $(c+a)$. A triangle is uniquely specified by its sides, so if we are given the tree we know what the triangle is and vice versa. Given a triangle with sides defined above, is there a crease pattern that collapses this triangle into a subbase with flaps of the appropriate length?

There is such a crease pattern and it is one familiar to every paper folder: the humble rabbit ear. It's been known since Euclid that the three bisectors of a triangle intersect at a point, which enables one to form a rabbit ear from any triangle with all of the edges lying in the same plane (or if you flatten the rabbit ear, the edges all lie on a line). One can also show the converse; if three sides of a triangle are $(a+b)$, $(b+c)$, and $(c+a)$, then the length of the flaps of the resulting rabbit ear are *a*, *b*, and *c*. So any triangle bounded by three active paths can be filled in with the creases of a rabbit ear, as shown in figure 5.10.

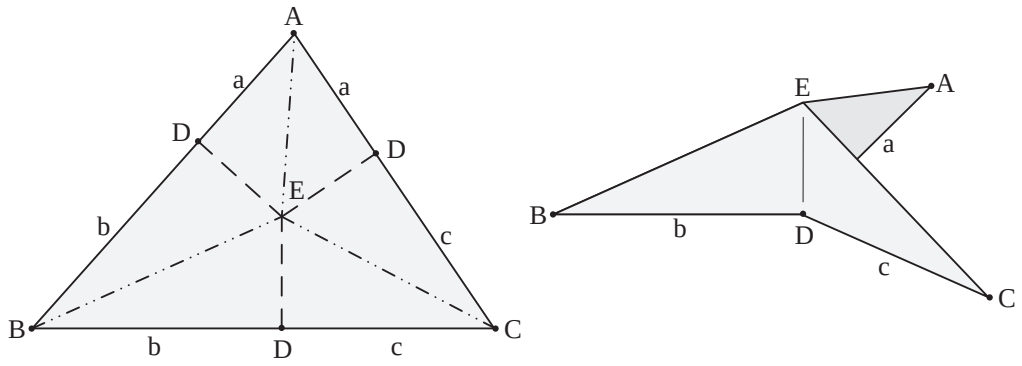


Figure 5.10. Crease pattern for a rabbit ear and resulting subbase. Note that the three flaps of the subbase have the same lengths as the three edges of the tree.

If you are actually folding the active polygon, you can find the intersection of the angle bisectors — point *E* in the figure — just by pinching each corner in half along the bisector and finding the point where all the creases come together. If you are calculating the crease locations numerically, there is a beautiful formula for the location of the intersection of the angle bisectors of an arbitrary triangle. If $\mathbf{p}_A$, $\mathbf{p}_B$, and $\mathbf{p}_C$ are the vector coordinates of corners *A*, *B*, and *C* and $\mathbf{p}_E$ is the coordinate of the bisector intersection, then $\mathbf{p}_E$ is given by the simple formula

$$\mathbf{p}_E = \frac{\mathbf{p}_A(b+c) + \mathbf{p}_B(c+a) + \mathbf{p}_C(a+b)}{2(a+b+c)}.$$

That is, the location of the bisector intersection is simply the average of the coordinates of the three corners with each corner weighted by the length of the opposite side.

What happens when one of the sides of the triangle is not an active path? This can happen, for example, when one of the sides of the triangle lies along an edge of the square; all of the triangles in figure 5.8 are of this type. Since the distance between any two terminal nodes must be greater than or equal to the minimum path length, the side that isn't an active path must be slightly too long to be an active path rather than too short. Fortunately, only a slight modification of the rabbit ear is necessary to address this situation. Figure 5.11 shows the crease pattern and subbase when side BC is slightly too long.

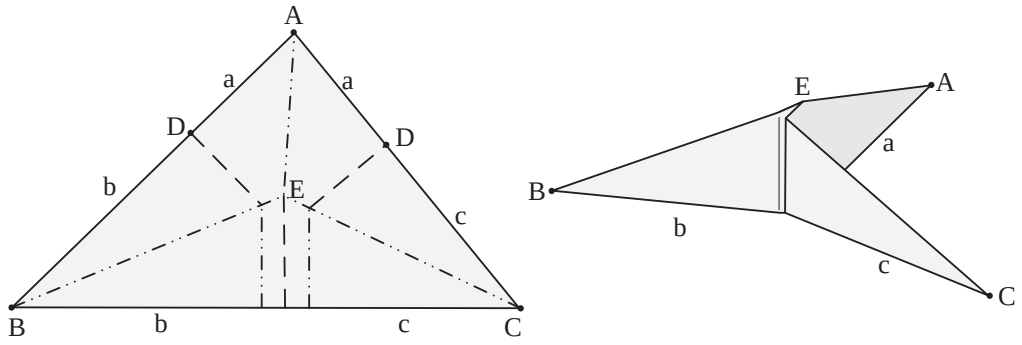


Figure 5.11. (Left) Crease pattern for a triangle when side BC is not an active path. (Right) Resulting subbase.

If the triangle has two sides that aren't active paths, a similar modification will still collapse it appropriately.

Another case that we should consider is a triangle graph that has one or more internal nodes along its edges, which may or may not be due to a kink in the subgraph. For example, the two side subgraphs in figure 5.8 each have three terminal nodes, but in each graph, one of the edges has an internal node because the subgraph has a kink at that point. We can still use the rabbit ear construction to find most of the creases, but wherever we have an internal node along an active path, we need a crease radiating inward from the internal node to the mountain fold that forms the "spine" of the subbase.

We might wonder which creases are valley and mountain folds. All of the creases that form the spine of the subbase are always mountain folds. Most of the creases corresponding to active paths are valley folds; however, for terminal nodes that fall in the interior of the paper, there is a mathematical requirement that the number of mountain and valley folds that come together at a point differ by 2; consequently, for any terminal node that lies in the interior of the square, one of the active paths must be converted to a mountain fold.

The other type of creases we will encounter are those creases that radiate inward from internal nodes; they can be mountain folds, valley folds, or no folds at all, depending on which way we lay the flaps when we flatten the uniaxial base. Since any such fold has three possible states, I call them **tri-state** folds. To give some indication which way the folds go, I'll draw all of the active paths as valley folds (even though a few of them may actually turn out to be mountain folds) and draw all of the tri-state folds as x-ray lines (since they can go either way).

Using the rabbit-ear crease pattern, we can fill in three of the four active polygons of the six-legged base as shown in figure 5.12. Using similar patterns, we can fill in the crease patterns for any triangle in any network of active polygons. Let us now turn out attention to the next case: four-sided active polygons.

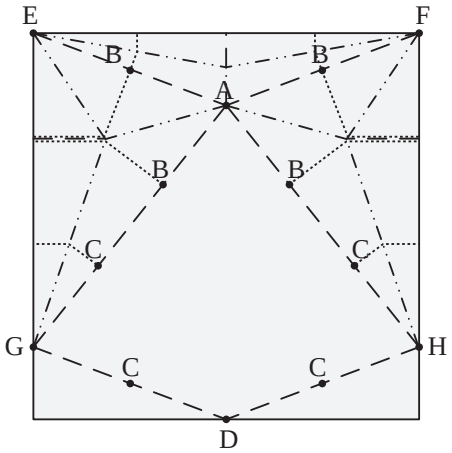


Figure 5.12. Crease pattern for the six-legged base with triangles filled in.

Just as a triangular polygon corresponded to a tree with three terminal nodes, a four-sided polygon, or quadrilateral, has a tree with four terminal nodes. There are two distinct types of trees with four terminal nodes, shown in figure 5.13. I call these two graphs the "4-star" and the

"sawhorse." If *a*, *b*, *c*, and *d* are the lengths of the four terminal edges in either graph and *e* is the length of the "body" in the sawhorse, then a quadrilateral whose sides are all active paths will have sides of length (*a*+*b*), (*b*+*c*), (*c*+*d*), and (*d*+*a*) for the 4-star and (*a*+*b*), (*b*+*c*+*e*), (*c*+*d*), and (*d*+*a*+*e*) for the sawhorse. There is an ambiguity associated with quadrilateral polygons; with the triangle, if we are given the lengths of the three sides, then the lengths of the edges of the tree are uniquely determined but with a quadrilateral, knowledge of just the lengths of the sides is insufficient to fully specify the dimensions of the tree. We must have the full tree to know whether the graph is a 4-star or a sawhorse and in the latter case, to know the length of the body.

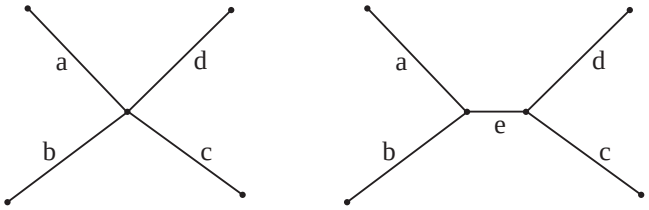


Figure 5.13. The two trees with four terminal nodes. (Left) The 4-star. (Right) The sawhorse.

Finding crease patterns for quadrilaterals is more complex than finding patterns triangles because there is more than one crease pattern that will collapse a given quadrilateral into the appropriate subbase. In fact, there can be infinitely many different crease patterns for a given quad. However, there is one pattern that has the desirable property that it works for quadrilaterals corresponding to either type of tree. I call it "the gusset quad" crease pattern because it usually has a gusset across its top. Its construction is rather complex, but I will describe it briefly here.

Given a quadrilateral *ABCD* as shown in figure 5.14. Inscribe a smaller quadrilateral inside whose sides are parallel to the sides of the original quadrilateral but are shifted inward a distance *h* (the value of *h* is not yet determined). Denote the corners of the new quadrilateral by *A*′, *B*′, *C*′, and *D*′. Drop perpendiculars from these four corners to the sides of the original quadrilateral. Label their points of intersection A_{AB} where the line from *A*′ hits side *AB*, B_{AB} where the line from *B*′ hits *AB*, and so forth.

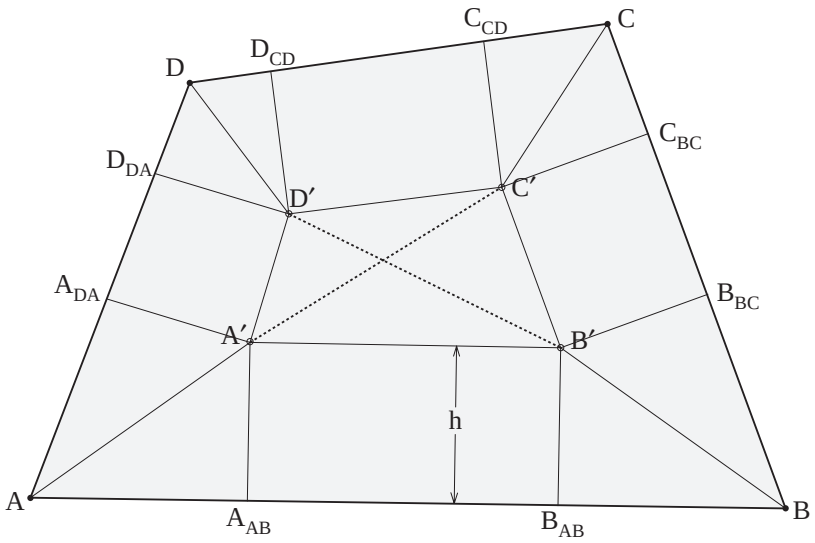


Figure 5.14. Construction of the gusset quad for a quadrilateral *ABCD*. Inset the quadrilateral a distance *h*; then drop perpendiculars from the new corners to the original sides.

Now we need some distances from the tree. Let l_{AC} be the distance from node *A* to node *C* on the tree and l_{BD} be the distance from node *B* to node *D*. In *most* cases (see below for the exceptions), there is a unique solution for the distance *h* for which one of these two equations holds:

$$AA_{AB} + A'C' + CC_{BC} = l_{AC}, \text{ or}$$

$$BB_{BC} + B'D' + DD_{AD} = l_{BD}.$$

Let us suppose we found a solution for the first equation. The diagonal $A'C'$ divides the inner quadrilateral into two triangles. Find the intersections of the bisectors of each triangle and call them B'' and D''. (If the second equation gave the solution, you'd use the opposite diagonal of the inner quadrilateral and find bisector intersections A'' and C''.) The points A', B'', C', and D'' are used to construct the complete crease pattern.

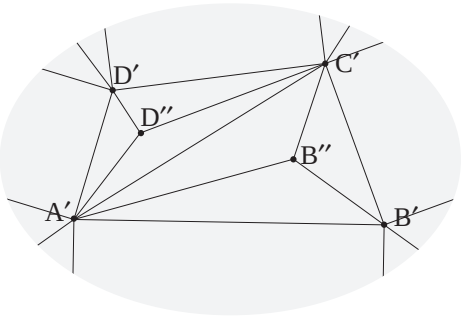


Figure 15. On the inner quadrilateral, construct the bisectors of each triangle to find points B'' and D''.

You've now found all of the points necessary to construct the crease pattern; connect them with creases as shown in figure 16 to produce the mountain folds that form the spine and the one valley fold that forms the gusset. You will also have to construct tri-state creases from each internal node along the sides of the quad to complete the crease pattern. Figure 5.16 also shows the folded gusset quad and its tree.

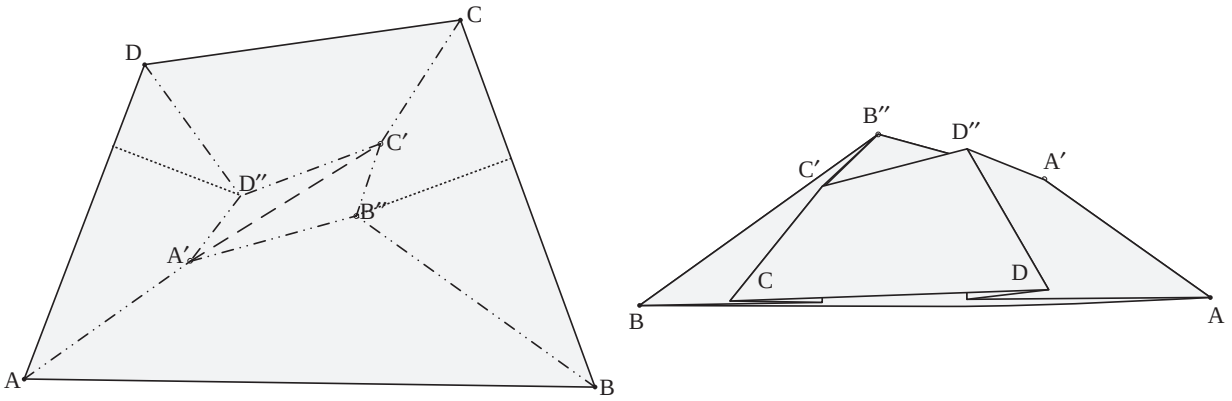


Figure 5.16. (Left) The mountain and valley folds of the gusset quad. Two tri-state creases are shown. (Right) The folded version of the gusset quad.

You can construct an equation for the distance h in terms of the coordinates of the four corners and the distances; it's a rather involved quadratic equation, but can be solved using a pocket calculator and high-school algebra. For arbitrary quadrilaterals, there is not a simple method to find the crease pattern by folding, but symmetric quadrilaterals (such as the one in the middle of figure 5.8) can be found by folding alone.

I alluded to exceptions above; there are quadrilaterals for which the points A', B', C', and D' all fall on a line. In these special cases, you don't get an inner quadrilateral; all of the inner creases collapse onto a line (or sometimes a point) and you get the simplified crease pattern shown in figure 5.17. Jun Maekawa has proven a theorem — called, appropriately enough, the Maekawa theorem — that says, in essence, any quadrilateral can have its edges collapsed onto a line using a crease pattern similar to this one. However, only in a relatively small number of cases will the lengths of the resulting flaps match the desired tree.

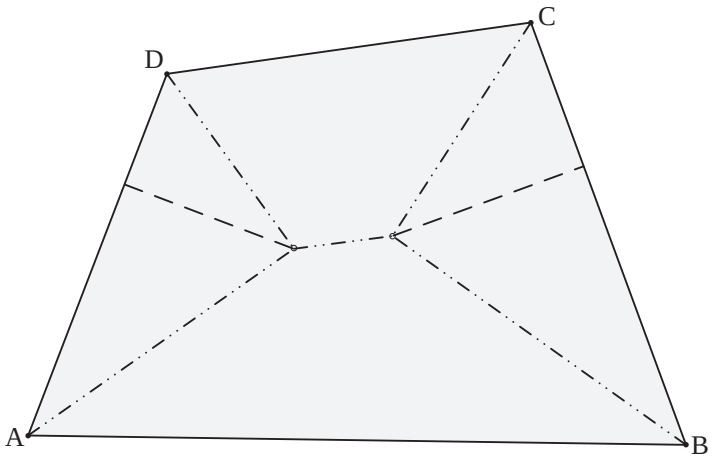


Figure 5.17. The Maekawa crease pattern.

The Japanese folder Toshiyuki Meguro has extensively explored crease patterns that collapse polygons onto lines and has coined the name "bun-shi," or "molecules," to describe such patterns. Just as individual molecules fit together to make a larger biological structure, so too do origami molecules fit together to make an origamical structure: the base.

Using the rabbit ear crease pattern for triangles and the gusset quad crease pattern for quadrilaterals, you can fill in any active polygon network that consists of triangles and quadrilaterals to get the complete crease pattern for the base. Such a polygon network is the one for the six-legged base considered earlier in this article. Figure 5.18 shows the full crease pattern for the six-legged base and the resulting base. Unlike figure 5.1, this base is a real, foldable base; you can easily verify the crease pattern by cutting it out and folding it on the lines. As you can see, the projection of the base into the plane is indeed the tree, and all of the flaps have their proper length.

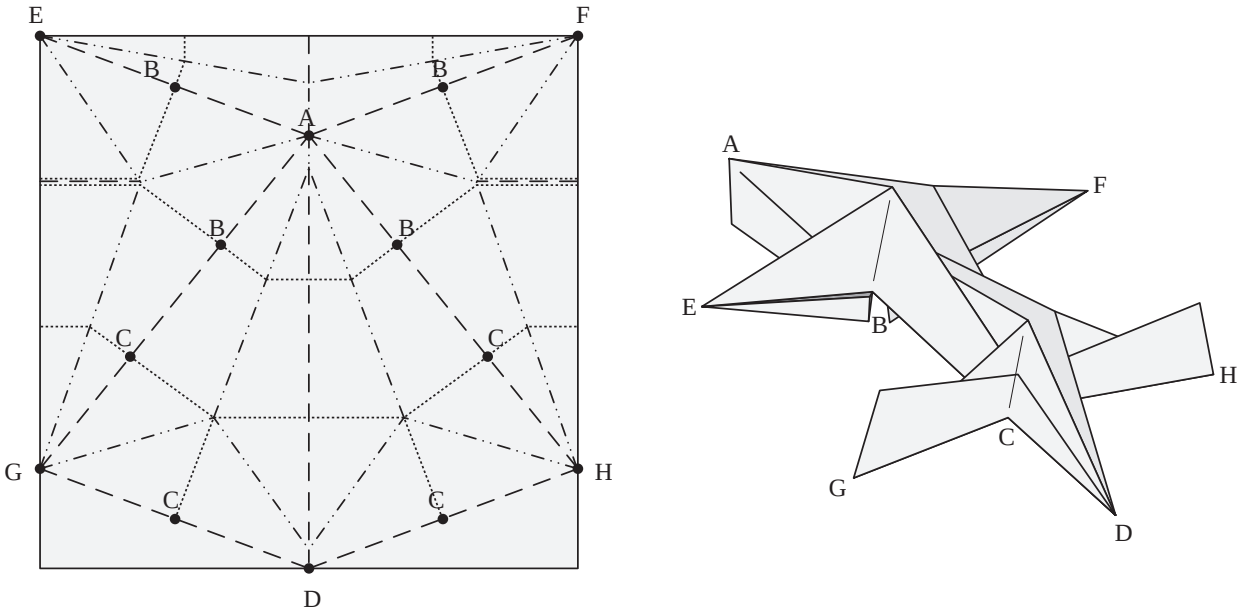


Figure 5.18. Full crease pattern and six-legged base.

If you try to collapse figure 5.18 into the base, you will have to flatten some valley folds and turn several tri-state creases into valley or mountain folds, depending on how you stack the layers and arrange the points. Although Tom Hull, Toshikazu Kawasaki, and others have identified several rules for assigning mountain and valley folds to a crease pattern that will allow it to be folded flat, I haven't yet identified an algorithm to assign mountain and valley folds to a tree method crease pattern — in fact, as you can tell by folding up a base, there is always more than one distribution of mountain and valley folds for a given crease pattern. In any event, when you collapse the base, all of the points will be free and unattached from the others and each segment of the base is precisely the same length as corresponding segment on the tree. You can thin the points further and add reverse folds, et cetera, to turn the base into a subject.

Much the same procedure can be used for any network of active polygons. However, what happens if there are polygons with five, six, or more sides? You saw the jump in complexity going from three to four corners was considerable. Although there was only one type of tree for a triangle and two for a quadrilateral, for a five-sided polygon there are three possible types of trees and the number rises quickly beyond that. So there are many more possibilities to enumerate. In addition, computation of the crease pattern for higher-order polygons gets very complicated very quickly, and you can imagine the difficulties as the number of points increases. For a nineteen-pointed insect, the network of active polygons could conceivably consist of a single nineteen-sided polygon! How would we ever collapse such a beast?

I recently discovered one solution: there is a generalization of the gusset quad that produces a crease pattern for any active polygon. However, its construction is even more complex than the construction of the gusset quad was. It turns out, though, that we need no more than the gusset quad and the rabbit ear to fold a base for any tree.

Again, a paradox: we'll simplify the design problem by making the tree more complicated. Consider the tree in figure 5.19, which is a 5-pointed star. This graph leads to a set of active paths that comprise a single five-sided polygon.

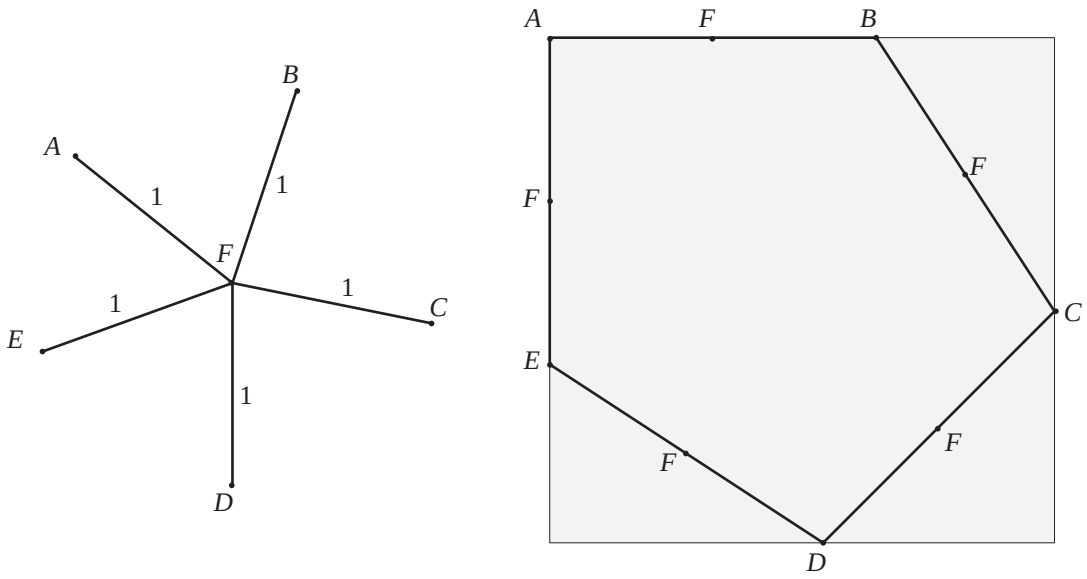


Figure 5.19. (Left) Tree for a base with 5 equal flaps. (Right) Pattern of terminal nodes and active paths corresponding to this tree.

Now suppose we wanted to add one more point to the star. That would entail adding one more terminal node — node *G* — to the active polygon network. On the tree graph, the new node would be connected to node F as shown in figure 5.20. (If we connected it to one of the other nodes that wouldn't add another point; it would just lengthen an existing point.) Suppose the point has a length *l*. Then since all the other edges are 1 unit long, the tree theorem tells us that the new terminal node must be separated from each of the other terminal nodes by a distance $(1+l)$.

We can visualize these constraints by imagining that each terminal node is surrounded by a circle whose radius is equal to the length of the edge attached to that node in the tree. That is, nodes *A*–*E* are surrounded by circles of unit radius, while node *G* is surrounded by a circle of radius *l*. The requirement that *G* be separated from the other nodes by at least a distance $(1+l)$ is equivalent to the requirement that circle G not overlap with any other circle.

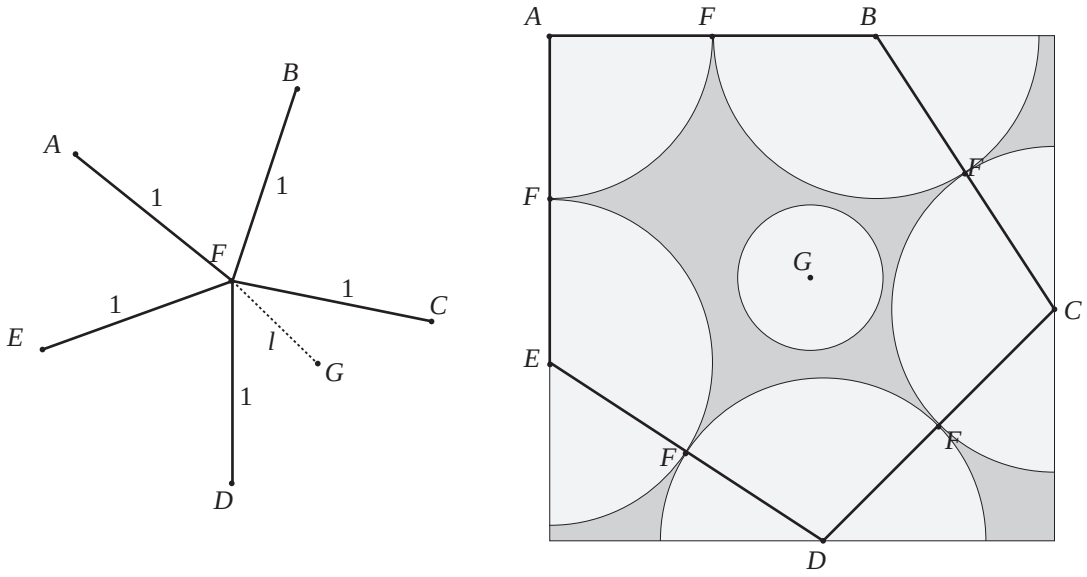


Figure 5.20. (Left) Tree with a new terminal node. (Right) One possible position for node G that satisfies the tree theorem.

Figure 5.20 shows the circles around each node. As long as the circles do not overlap, the tree theorem is satisfied. As *l*, the length of the new edge, is increased, the size of the circle around node *G* must be increased as well. Eventually, circle *G* will swell until it is touching at least three other circles, and at that point, shown in figure 5.21, the new point is as large as it can possibly be.

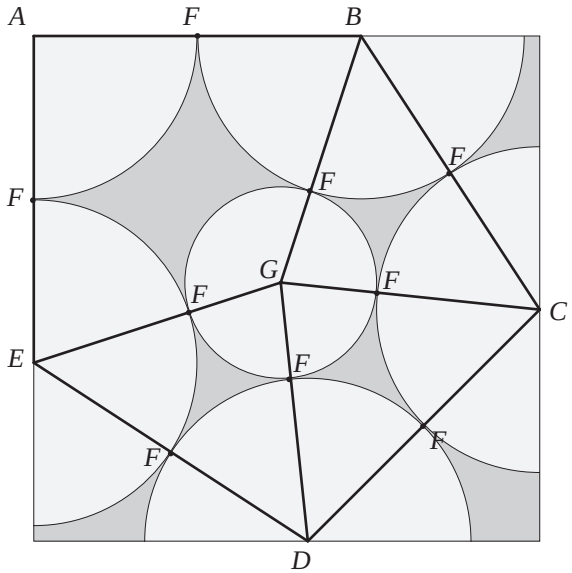


Figure 5.21. Terminal node pattern with the largest possible circle around node *G*.

Wherever circle *G* touches another circle, the two terminal nodes are spaced at their minimum separation. Consequently, the paths between the nodes of touching circles are active paths.

Whenever we add a new node inside a polygon, if we make the corresponding flap sufficiently large, it forms several new active paths with the vertices of the old active polygon. In the process, it breaks up the polygon into smaller polygons *with fewer sides than the original polygon*. In figure 5.21, adding the new point has broken the five-sided polygon into one quadrilateral and three triangles. But we already know crease patterns for quadrilaterals and triangles! Filling in the crease patterns in each polygon gives a crease pattern for the full base, shown in figure 5.22.

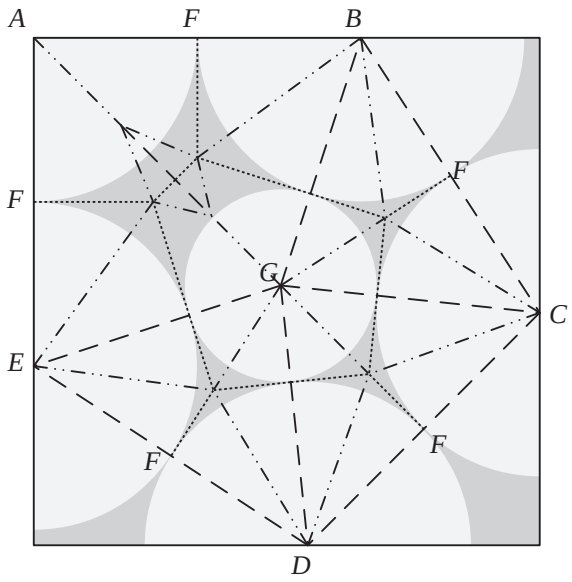


Figure 5.22. Crease pattern for the 5-pointed base.

Imagining terminal nodes as being surrounded by circles makes it easier to visualize the path constraints of the tree theorem. Although strictly speaking, the circle analogy only holds for terminal nodes with exactly one internal node between them, one can devise a similar condition for more widely separated nodes. If you draw in circular arcs around each terminal node tangent to the tri-state creases, you get a set of contours on both the crease pattern and the base, in which edges attached to terminal nodes are represented by circles and edges attached only to internal nodes are represented by contours that snake through the crease pattern as in figure 5.23. For numerical computation, I find that it is simpler to do all calculations in terms of paths and path lengths, but for intuition and visualization, the circles and contours constitute an equivalent and more easily visualized tool for devising new crease patterns.

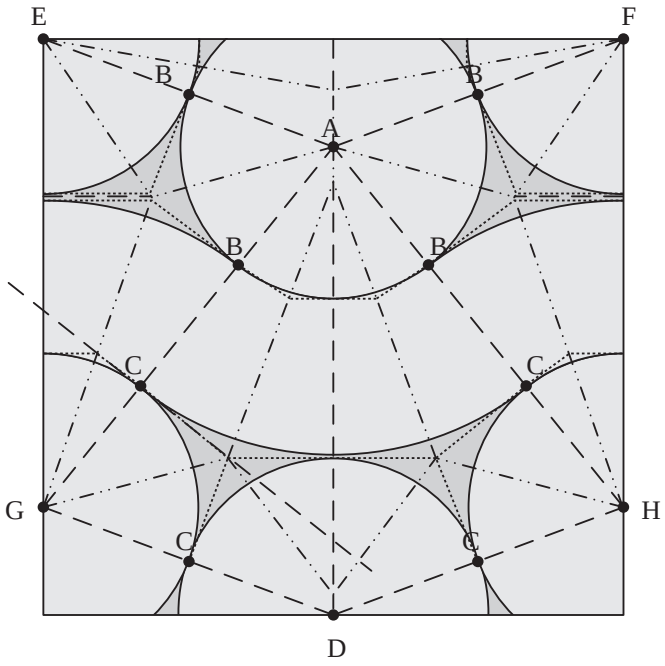


Figure 5.23. Circles and contours for the six-pointed lizard base.

If you've stuck with me so far, you're now at the goal: an algorithm for finding a crease pattern to fold a base with any number of points from a square, rectangle, or any other shape paper. (In fact, this algorithm even works for circular paper!) To summarize the algorithm:

1. Draw a tree, or stick figure, of the base, labeling each edge with its desired length.

2. Find a pattern of terminal nodes on the square that satisfies the tree theorem, namely, that the distance between any two nodes on the square is greater than or equal to their separation on the tree.

3. Mark all of the active paths, paths whose actual length is equal to their minimum length. Identify the active polygons.

4. For any active polygon with five or more sides, add a node and edge to the tree attached at an internal node of the polygon and make the edge as large as possible, thereby breaking up the active polygon into quads and triangles.

5. Fill in each active polygon with quad and triangle crease patterns.

The good news is that using the algorithm described above, a base can be constructed for *any* tree — in fact, there are usually *many* distinctly different solutions for a single tree. The bad news is, as you might suspect from some of the above, constructing the crease pattern can be computationally intensive.

(The worse news is that even when you have the crease pattern, folding it up into a base can be infuriatingly difficult; there is rarely a step-by-step folding sequence. More often than not, you need to precrease everything, then collapse the base all at once.)

Since every polygon network can be broken up into triangles and quads by the addition of extra circles, the triangle and quad molecules are by themselves sufficient for filling in the crease pattern for any tree. However, there are many other possible molecules, including molecules that can be used for higher-order polygons. It turns out that the gusset quad is just a special case of a more general construction that is applicable to any higher-order polygon. I call this construction the **universal molecule**. In fact, many of the known molecules — the Maekawa and Meguro molecules, the rabbit ear, and so forth (but not the arrowhead quad, as it turns out) are special cases of the universal molecule. The rest of this article describes the construction of this molecule for an arbitrary polygon.

Consider a general polygon that satisfies the tree theorem, i.e., any two vertices are separated by a distance greater than or equal to their separation on the tree graph. Since we are considering a single active polygon, we know that of the paths between nonadjacent vertices, none are at their minimum length (otherwise it would be an active path and the polygon would have been split).

Suppose we inset the boundary of the polygon by a distance h, as shown in figure 5.24. If the original vertices of the polygon were A_1, A_2,… then we will label the inset vertices A_1', A_2',… as we did for the gusset quad construction. I will call the inset polygon a **reduced polygon** of the original polygon.

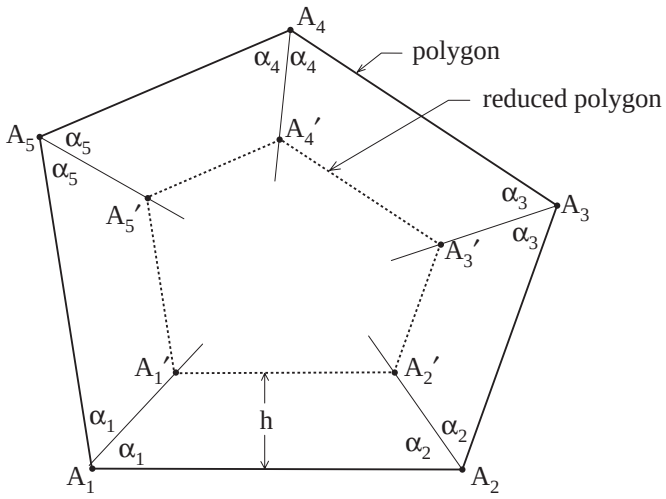


Figure 5.24. A reduced polygon is inset a distance h inside of an active polygon. The inset corners lie on the angle bisectors (dotted lines) emanating from each corner.

Note that the points A_i' lie on the bisectors emanating from the points A_i for any h. Consider first a reduced polygon that is inset by an infinitesimally small amount. In the folded base, the sides of the reduced polygon all lie in a common plane, just as the sides of the original active polygon all lie in a common plane; however, the plane of the sides of the reduced polygon is offset from the plane of the sides of the active polygon by a perpendicular distance h. As we increase h, we shrink the size of the reduced polygon. Is there a limit to the shrinkage? Yes, there is, and this limit is the key to the universal molecule. Recall that for any polygon that satisfies the tree theorem, the path between any two vertices satisfies a path length constraint

$$\left|A_i - A_j\right| \geq l_{ij}, \tag{1}$$

where l_{ij} is the path length between nodes i and j measured along the tree graph. There is an analogous condition for reduced polygons; any two vertices of a reduced polygon must satisfy the condition

$$\left|A_i' - A_j'\right| \geq l_{ij}', \tag{2}$$

where l_{ij}' is a **reduced path length** given by

$$l_{ij}' = l_{ij} - h\left(\cot\alpha_i + \cot\alpha_j\right) \tag{3}$$

and α_i is the angle between the bisector of corner i and the adjacent side. I call equation (2) the **reduced path constraint** for a reduced polygon of inset distance h. Any path for which the reduced path constraint becomes an equality is, in analogy with active paths between nodes, called an **active reduced path**.

So for any distance h, we have a unique reduced polygon and a set of reduced path constraints, each of which corresponds to one of the original path constraints. We have already assumed that all of the original path constraints are met; thus, we know that all of the reduced path constraints are met for the h=0 case (no inset distance). It can also be shown that there is always some positive nonzero value of h for which the reduced path constraints hold. On the other hand, as we increase the inset distance, there comes a point beyond which one or more of the reduced path constraints is violated. Suppose we increase h to the largest possible value for which every reduced path constraint remains true. At the maximum value of h, one or both of the following conditions will hold:

(1) For two adjacent corners, the reduced path length has fallen to zero and the two inset corners are degenerate; or

(2) For two nonadjacent corners, a path between inset corners has become an active reduced path.

These two situations are illustrated in figure 5.25.

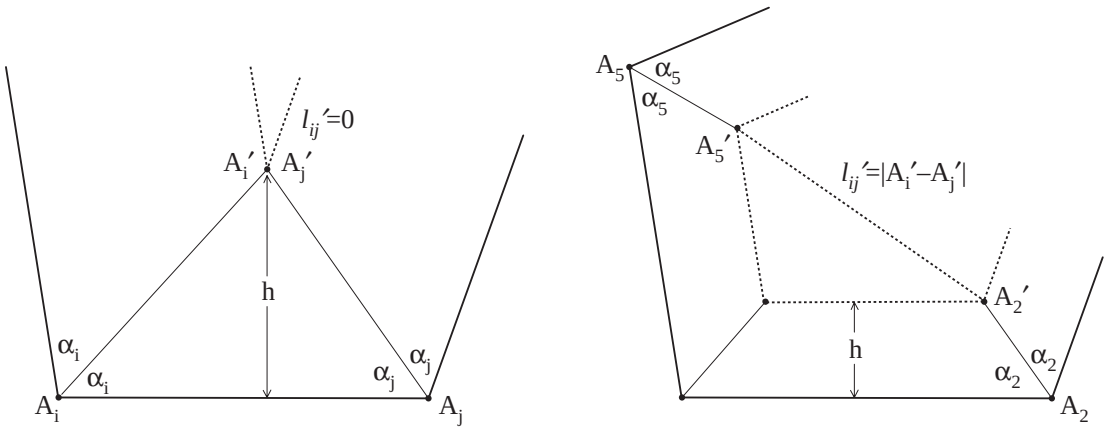


Figure 5.25. (Left) Two corners are inset to the same point, which is the intersection of the angle bisectors. (Right) Two nonadjacent corners inset to the point where the reduced path between the inset corners becomes active.

As I said, one or the other or both of these situations must apply; it *is* possible that path corresponding to both adjacent and nonadjacent corners have become active simultaneously or for multiple reduced paths to become active for the same value of *h* (this happens surprisingly often). In either case, the reduced polygon can be simplified, thus reducing the complexity of the problem.

In a reduced polygon, if two or more adjacent corners have coalesced into a single point, then the reduced polygon has fewer sides (and paths) than the original active polygon. And if a path between nonadjacent corners has become active, then the reduced polygon can be split into separate polygons along the active reduced paths, each with fewer sides than the original polygon had (just as in the polygon network, an active path across an active polygon splits it into two smaller polygons). (In the gusset quad, for example, the reduced quad is inset until one of its diagonals becomes an active path; the reduced quad is then split along the diagonal into two triangles.) In either situation, you are left with one or more polygons that have fewer sides than the original. The process of insetting and subdivision is then applied to each of the interior polygons anew, and the process repeated as necessary.

If a polygon (active or reduced) has three sides, then there are no nonadjacent reduced paths. The three bisectors intersect at a point, and the polygon's reduced polygon evaporates to a point, leaving a rabbit ear molecule behind composed of the bisectors.

Four-sided polygons can have the four corners inset to a single point or to a line, in which case no further insetting is required, or to one or two triangles, which are then inset to a point. Higher-order polygons are subdivided into lower-order ones in direct analogy.

Since each stage of the process absolutely reduces the number of sides of the reduced polygons created (although possibly at the expense of creating more of them), the process must necessarily terminate. Since each polygon (a) can fold flat, and (b) satisfies the tree theorem, then the entire collection of nested polygons must also satisfy the tree condition. Consequently, *any* active polygon that satisfies the tree theorem — no matter how many sides — can be filled with a crease pattern using the procedure outlined above and collapsed into a base on the resulting creases.

So what are the creases of the universal molecule? Each polygon is divided into two parts: the **core** is the reduced polygon (which may be crossed by active reduced paths); the border around the core is the **ring**. The angle bisectors that cross the ring are mountain folds. Internal nodes along active paths propagate inward across the ring forming tri-state folds. Active reduced paths that cross the core are valley folds. The boundary of reduced polygons can also be tri-state folds, as one may or may not fold layers along them. The same assignment of crease applies to each level of the recursive universal molecule construction.

A remarkable feature of the universal molecule is that many of the molecular crease patterns that have been previously enumerated are just special cases of it, including the rabbit ear molecule, the gusset quad, and both Maekawa and Meguro quads. Figures 5.26 and 5.27 illustrates these special cases.

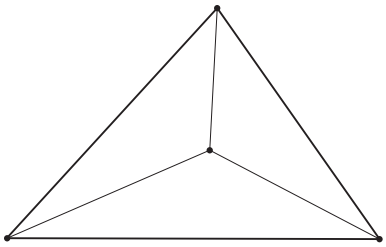


Figure 5.26. In a triangle, all three corners are inset to the same point, which is the intersection of the angle bisectors. This gives the rabbit ear molecule.

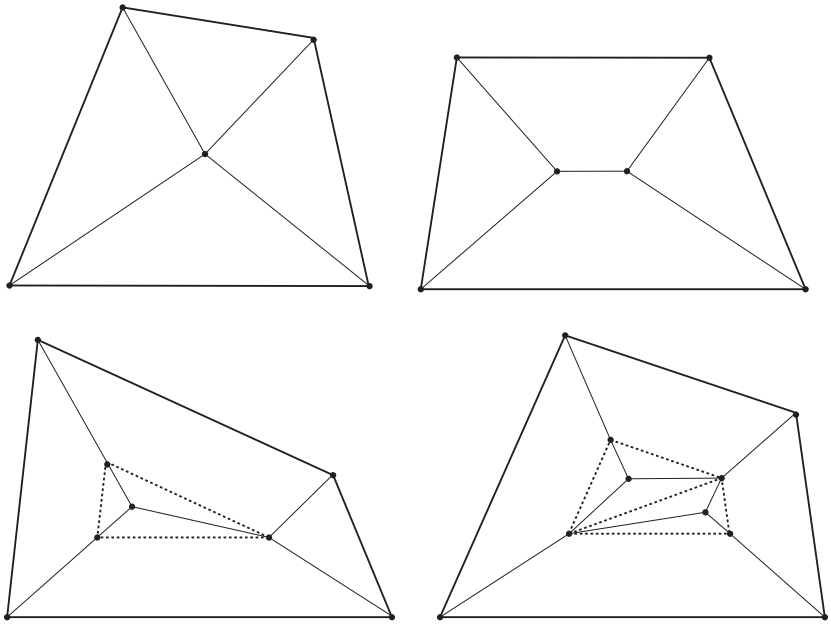


Figure 5.27. The four possible universal molecules for a quad. If all four corners are inset to the same point, the result is the Husimi molecule (top left). If adjacent pairs of corners are inset to two points, the Maekawa molecule is obtained (top right). If the inset polygon is a triangle, it is filled in with a rabbit ear molecule, which also results in a Maekawa molecule (bottom left). Finally, if the inset polygon is a quad crossed by an active reduced path, the result is the gusset quad.

These examples are just the tip of the iceberg; beyond four sides, the possibilities rapidly explode. But this explosion doesn't matter; there is a unique universal molecule for *every possible* active polygon that satisfies the tree theorem.

An alternative design approach that blends aspects of the circle method and tree methods has been described by Kawahata[2] and Maekawa[3]. It has been called this the "string-of-beads"

[2]Fumiaki Kawahata, *Fantasy Origami*, pub. by Gallery Origami House, Tokyo, Japan, 1995 [in Japanese].

[3]Jun Maekawa, *Oru* magazine, also in Proceedings of the Second International Conference on Origami Science and Technology, 1994 [in Japanese]

approach to design. As in the tree method, you begin with a tree graph of the model to be folded. Each line of the graph is doubled and the graph is expanded to fill a square, with the nodes of the graph spaced around the edges of the square like beads on a string. The process is illustrated for a six-flap based in figure 5.28.

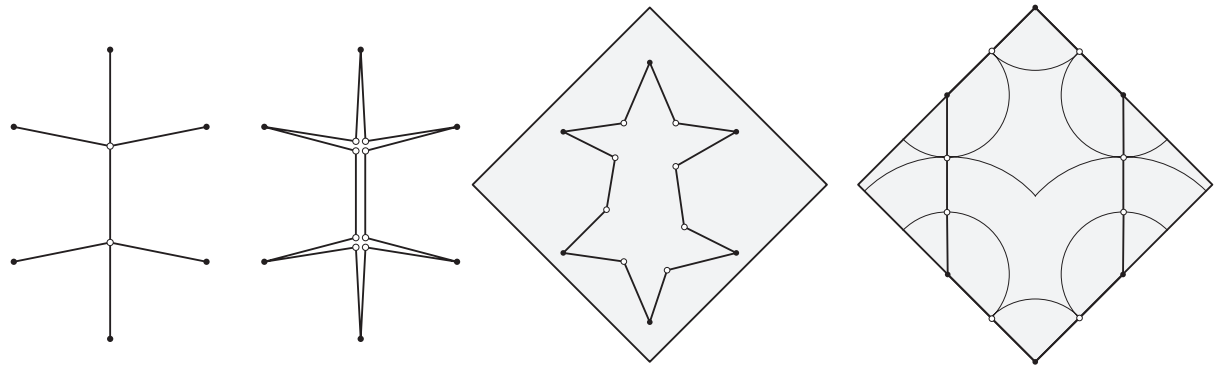


Figure 5.28. The string-of-beads design method. The tree graph is turned into a closed polygon, which is then "inflated" inside of a square with straight lines between the terminal nodes. The result is a large polygon inside the square that is collapsed into the base.

In the string-of-beads method, the tree graph is converted into a large polygon in which each corner is one of the terminal nodes of the tree and each side is as long as the path between adjacent terminal nodes. It is clear that this distribution of terminal nodes is just a special case of the tree method in which we have constrained all of the nodes to lie on the edge of the square; it avoids having middle points, but at the expense of possibly reduced efficiency.

The string-of-beads approach produces a large polygon that must be collapsed into the base, and the techniques described by Maekawa involve placing tangent circles in the contours shown in the last step of figure 5.28 (which is analogous to our use of additional circles to break down active polygons into smaller polygons in the tree method; Kawahata's algorithm projects hyperbolas in from the edges to locate reference points for molecular patterns.) However, one can also apply the universal molecule directly to the string-of-beads polygon, achieving another efficient crease pattern that collapses it into a base.

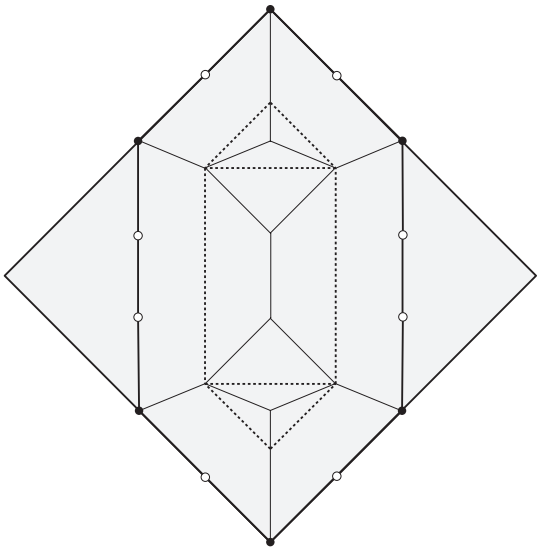


Figure 5.29. Construction of the universal molecule for the polygon shown in figure 28.

Figure 5.29 shows the universal molecule. The initial hexagon is inset to the point that the two horizontal reduced paths become active, and the hexagon is split into two triangles and rectangle. The triangles are filled with rabbit ear creases; the rectangle is further inset, forming a Maekawa molecule.

Figure 5.30 compares the crease pattern obtained from this polygon by adding an additional node to the tree pattern (i.e., adding a middle flap) and that obtained with the universal molecule. (Tri-state creases are shown as dotted lines.)

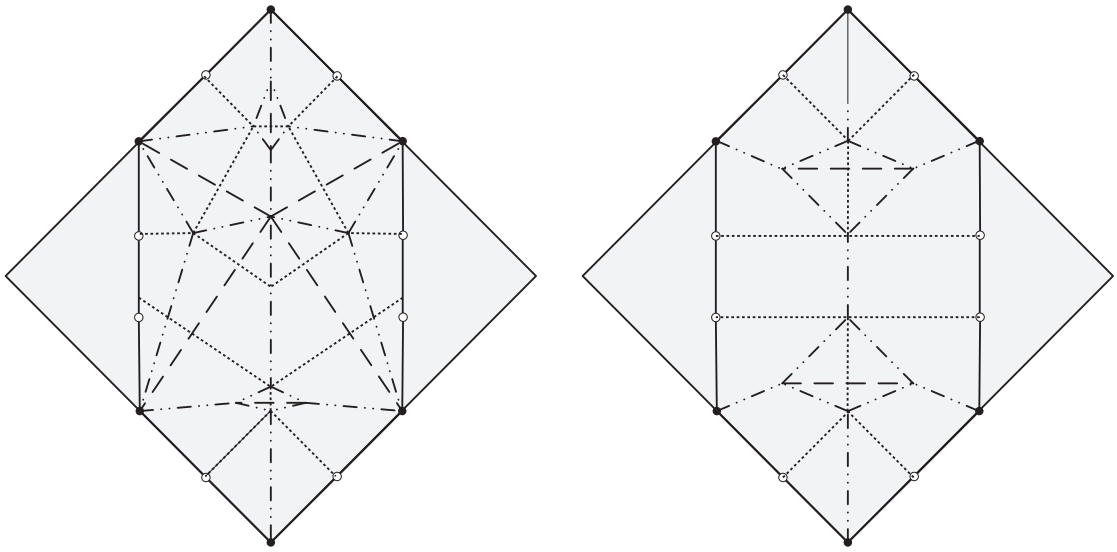


Figure 5.30. (Left) Crease pattern obtained by adding an additional node to the hexagonal active polygon. (Right) Crease pattern obtained by using the universal molecule.

A nice feature of the universal molecule is that it is very frugal with creases. A tree filled in with universal molecules tends to have relatively few creases and large, wide flaps (which can, of

course, be subsequently narrowed arbitrarily as desired). In fact, I conjecture the following: for any active polygon, the universal molecule is the crease pattern with the shortest total length of creases that collapses that polygon to the uniaxial base. Few creases translates into relatively few layers in the base (at least, until you start sinking edges to narrow them). And because you don't have to arbitrarily add circles (and hence points) to a crease pattern to knock polygons down to quads and triangles (as you do using the classical tree method algorithm), bases made with the universal molecule tend to have less bunching of paper and fewer layers near joints of the base, resulting in cleaner and (sometimes) easier-to-fold models.

As another example of the utility of the universal molecule, an article by Maekawa in *Oru* magazine illustrated the design of a tree structure using the string-of-beads/contour algorithm. In this algorithm, all terminal nodes are arranged around the outside of the square to form one large active polygon. Then fixed-size circles are added to the interior to break up the active polygon into quads and triangles. Figure 5.31 shows the tree and the circle pattern derived thereby.

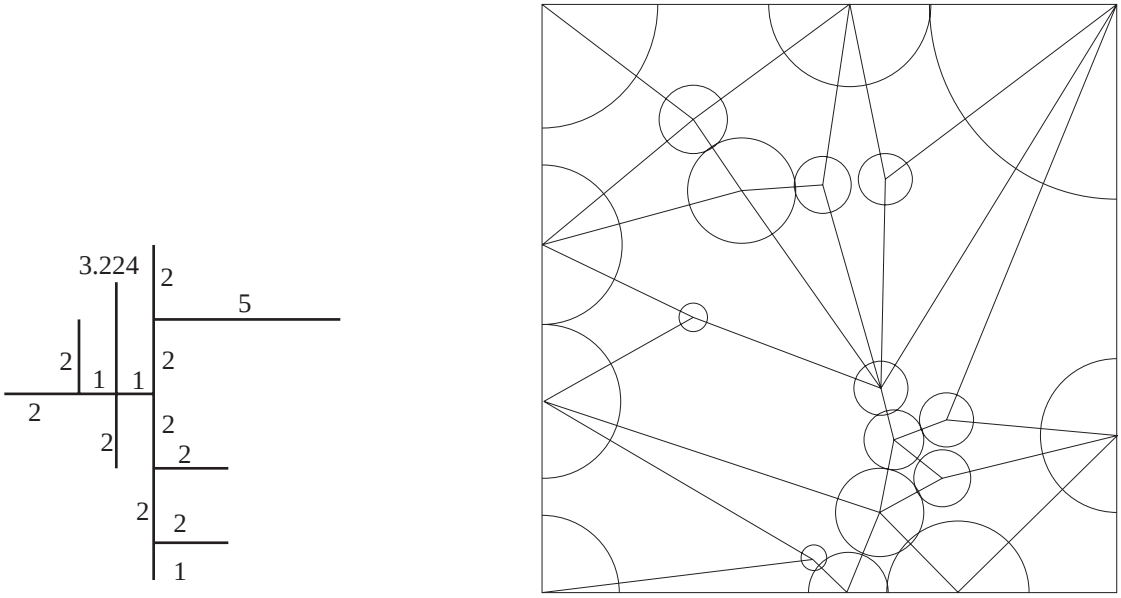


Figure 5.31. (Left) tree and (right) crease pattern from *Oru*

Maekawa's construction used rabbit ear molecules to fill in the triangles and a different, but similarly versatile construction to fill in quads, called the arrowhead quad. The resulting crease pattern is shown in figure 5.32. Because of the large number of interior circles, the crease pattern is quite complex and is quite difficult to fold flat.

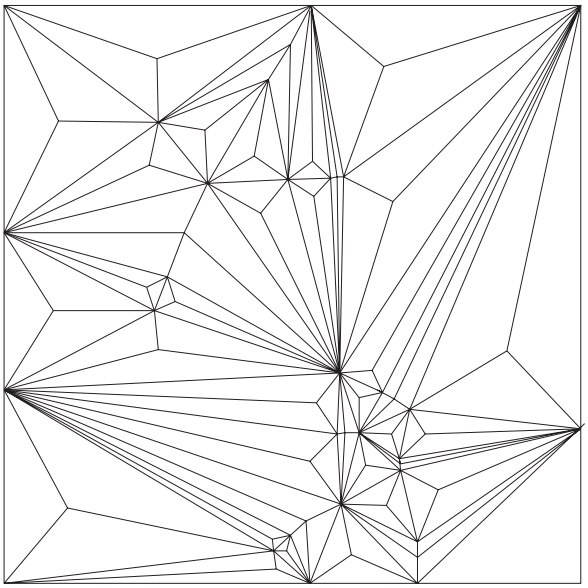


Figure 5.32. Crease pattern from Oru using arrowhead quad and rabbbit-ear molecules.

We can simplify the crease pattern obtained for the same terminal node positions by adding a smaller number of interior circles and inflating each new circle to its maximum size before adding the next, as described above. It turns out that we need add only three nodes to insure that all active polygons have three or four sides. The resulting modified tree and crease pattern, shown in figure 5.33, is reasonably simple to fold, and utilizes only rabbit-ear and gusset quad molecules.

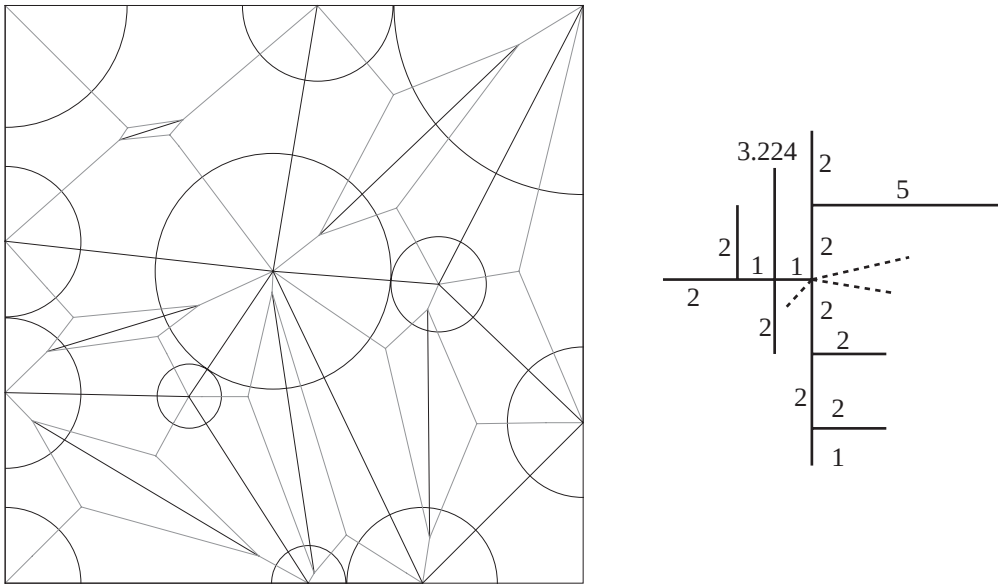


Figure 5.33. Tree and crease pattern for the *Oru* tree with three additional interior circles and filled with triangle and gusset quads.

By applying the universal molecule to the original polygon, we can get a still simpler crease pattern, which is shown in figure 5.34. This pattern has the additional perk that it is yet easier to

fold up in the base (I encourage you to try). The flaps are, as you might expect, much wider than they are long. However, by sinking the top of the base in and each, all of the flaps can be thinned to arbitrarily large aspect ratio while maintaining their relative lengths.

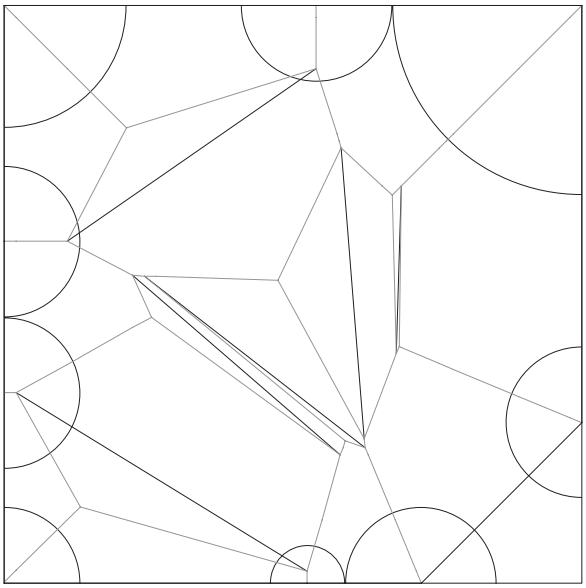


Figure 5.34. Universal molecule version of the *Oru* tree

All of the crease patterns in figures 5.31–5.34 share the property that all points lie on the edge of the square. If we relax this constraint and allow middle points, then we can achieve a slightly larger and even simpler pattern (with a scale of 0.067 as compared to 0.065 for the previous pattern), shown in figure 5.35.

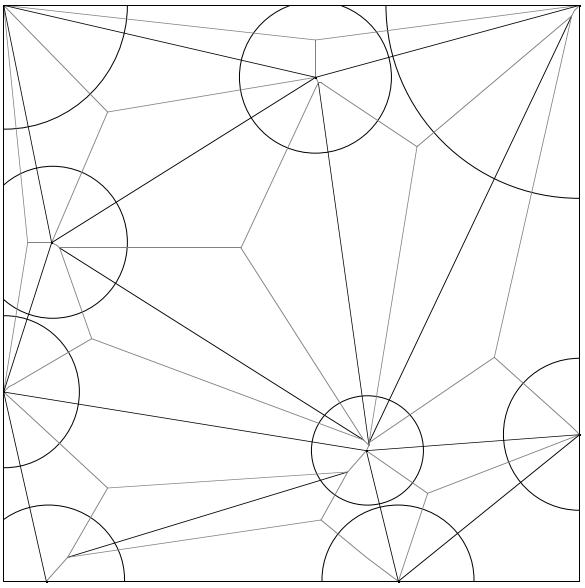


Figure 5.35. *Oru* tree crease pattern with middle points.

Without the universal molecule one must arbitrarily add circles to the network of active paths until every polygon is of order 4 or smaller. Since there is some choice about where circles are added, the tree method solution for a base is not necessarily unique. The universal molecule, however, is entirely determined. By applying the recursive universal molecule construction, any network of active paths can be filled in with creases and collapsed into a uniaxial base. Thus, the progession from tree graph to full crease pattern may be accomplished using a single optimization.

There is another, subtler advantage to the universal molecule: it tends to shift paper away from the plane of projection of the base, which simplifies collapsing the base and reduces the buildup of layers where flaps join together. Although the flaps tend to be wider than those produced with other algorithms, one can easily make points narrower by repeatedly sinking the corners of collapsed polygons to narrow the flaps. This, of course, builds up layers of paper in the base, which is to some degree unavoidable. The most equitable distribution of these extra layers comes when the points are sunk along lines parallel to the plane of projection. Since the edges of the inset polygons are parallel to the plane of projection in the folded base, these lines naturally form lines along which to sink.

There are other polygon-filling algorithms beyond the universal molecule described above, and I will mention two of them here. Toshiyuki Meguro — who coined the term "molecule" (*bun-shi*, in Japanese) has developed a technique that makes use of overlapping circles, which in effect, allows one to add new branches to the tree by adding nodes to the middle of existing edges. This technique allows each new circle to touch four, rather than three, existing circles, and cuts down on the number of circles that need to be added. I have extended and generalized Meguro's concept to apply to arbitrary polygons by introducing the concept of a "stub." Also Fumiaki Kawahata has developed a technique for filling in polygons that involves projecting hyperbolas, rather than straight lines, inward from the edges of an active polygon, which results is bases whose points can be narrower and more regular (if desired). No doubt there are yet other algorithms lurking out there in the mathematical wilds.

Although the mathematical algorithms for origami design are rigorously defined, the actual location of the nodes and creases can be computationally intensive. Computationally intensive problems are best handled by computer and indeed, the procedures described above can be cast in the mathematical and logical terms that lend themselves to computer modeling. The computer program *TreeMaker* implements these algorithms. Using *TreeMaker*, I've solved for bases for a number of subjects whose solutions have eluded me over the years — deer with varying sizes and types of antlers, 32-legged centipedes, flying insects, and more. Using a computer program accelerates the development of a model by orders of magnitude; from the tree to the full crease pattern takes less than five minutes (although folding the crease pattern into a base may take two to three hours after that!) Not only does *TreeMaker* come up with the base initially, but it lets one incrementally iterate the design of the model, shifting paper from one part of the model to another to lengthen some points and shorten others, all the while keeping the entire model maximally efficient.

One need not program a computer, of course, to use the techniques I've described to design origami — origami designers have used similar techniques for years. But I foresee a dramatic shift in the art as these techniques — what one might call "algorithmic" origami design — become more widespread. For years, technical folders concentrated on getting the right number

and lengths of points to the near-exclusion of other folding considerations such as line, form, and character. With algorithmic origami design, point count comes automatically and is no longer the overwhelming consideration in technical origami design. In the past, origami art and origami science have often been at odds, but with algorithmic origami design, the technical designer, freed from the need to expend his or her energies on the number of appendages, can focus on the art of folding, secure in the knowledge that the science will take care of itself.

6.0 TreeMaker Algorithms

This section describes the internal mathematical algorithms of *TreeMaker*. You don't need to know any of this to use the program, but if you're curious about what's going on inside your machine, you might find this section interesting. If you come up with any mathematical/algorithmic ideas of your own, send them to me and I'll put them into a future version.

6.1 Mathematical Model

TreeMaker finds crease patterns by performing several different types of nonlinear constrained optimization. The quantity to maximize is the scale, which is the size of a 1-unit flap compared to the size of the square. There are two families of constraints that must be satisfied for *any* valid crease pattern:

— The coordinates of every node must lie within the square

— The separation between any two nodes on the square must be at least as large as the scaled length of the path between the two nodes as measured along the tree.

If there are N nodes in a figure, the first condition sets $4N$ linear inequality constraints while the second sets $N(N-1)/2$ quadratic inequality constraints. In addition to these constraints, the user can set a number of other constraints:

— A node can have its position set to a fixed value

— A node can be constrained to lie on a line of bilateral symmetry

— Two nodes can be constrained to lie symmetrically about a line of symmetry

— Three nodes can be constrained to be collinear

— An edge can be constrained to a fixed length

— Two edges can be constrained to have the same strain

— A path can be forced to be active

— A path can have its angle set to a fixed value

— A path can have its angle quantized to a multiple of a given angle

The problem is solved by converting the path conditions of the Tree Theorem and any constraints into mathematical equations, specifically, a constrained nonlinear optimization. This document decribes the equations that define each type of optimization.

6.2 Definitions and Notation

Define U to be the set of all *node* coordinates $\mathbf{u}_i$, $i \in I^n$, where I^n is a set of node indices: $I^n = \{1..n_n\}$. Each node $\mathbf{u}_i$ has coordinate variables $u_{i,x}$ and $u_{i,y}$.

Define E to be the set of all *edges* e_i, $i \in I^e$, where I^e is a set of edge indices $I^e = \{1..n_e\}$. Each edge contains exactly two nodes $n_i, n_j \in e_k$. Each edge e_i has a length l_i and a strain σ_i.

Define U^t to be the set of *terminal nodes*, which are those nodes connected to exactly one edge. Define I^{tn} to be the set of terminal node indices. Clearly, $U^t \subseteq U$ and $I^{tn} \subseteq I^n$.

Define P to be the set of all *paths*, p_{ij}, $i, j \in I^n$, $i \neq j$. Each path is identified by the indices of the nodes at each end of the path. Each path has a length, l_{ij}, which is given by the sum of the strained lengths of the edges in the path; that is,

$$l_{ij} \equiv \sum_{e_k \in p_{ij}} (1 + \sigma_k) l_k$$

Define P^t to be the set of *terminal paths*, which are those paths that connect two terminal nodes

Define m to be the overall *scale* of the tree.

Define w, h to be the *width* and *height* of the paper. The paper is a rectangle whose lower left corner is the origin $(0,0)$ and whose upper right corner is the point (w,h).

6.3 Scale Optimization

The most basic optimization is the optimization of the positions of all terminal nodes and the scale of the design. This is equivalent to solving the problem

minimize (–m) over $\{m, \mathbf{u}_i \in U^t\}$ s.t.

(1) $0 \le u_{i,x} \le w$ for all $\mathbf{u}_i \in U^t$

(2) $0 \le u_{i,y} \le h$ for all $\mathbf{u}_i \in U^t$

(3) $m \sum_{e_k \in p_{ij}} [(1 + \sigma_k) l_k] - \sqrt{\left(u_{i,x} - u_{j,x}\right)^2 + \left(u_{i,y} - u_{j,y}\right)^2} \le 0$ for all $p_{ij}, \in P^t$

6.4 Edge Optimization

Edge optimization is used to selectively lengthen points by the same relative amount to "fill out" a crease pattern. The scale m is fixed, and a subset of the the edges E^s is subjected to the same variable strain s. A subset of the terminal nodes U^s is allowed to move. The edge optimizer solves the problem:

minimize $-\sigma$ over $\{\sigma, \mathbf{u}_i \in U^s\}$ s.t.

(1) $0 \le u_{i,x} \le w$ for all $\mathbf{u}_i \in U^s$

(2) $0 \le u_{i,y} \le h$ for all $\mathbf{u}_i \in U^s$

(3) $$m\left[\sum_{\substack{e_k \in p_{ij} \\ e_k \in E^s}}\left[(1+\sigma)l_k\right] + \sum_{\substack{e_k \in p_{ij} \\ e_k \notin E^s}}\left[(1+\sigma_k)l_k\right]\right] - \sqrt{\left(u_{i,x}-u_{j,x}\right)^2 + \left(u_{i,y}-u_{j,y}\right)^2} \le 0 \text{ for all } p_{ij}, \in P^t$$

Equation (3) can be broken into a fixed and variable part:

$$\underbrace{\sigma}_{\text{variable}} \cdot \underbrace{m \sum_{\substack{e_k \in p_{ij} \\ e_k \in E^s}}\left[l_k\right]}_{\text{fixed}} + \underbrace{m \sum_{\substack{e_k \in p_{ij} \\ e_k \notin E^s}}\left[(1+\sigma_k)l_k\right]}_{\text{fixed}} - \underbrace{\sqrt{\left(u_{i,x}-u_{j,x}\right)^2 + \left(u_{i,y}-u_{j,y}\right)^2}}_{\substack{\text{fixed or} \\ \text{variable}}} \le 0$$

Note, however, that the last term may or may not be a mixture of fixed and variable parts depending upon which nodes are moving.

6.5 Strain Optimization

Strain optimization is used to distort the edges of the tree minimally in order to impose other global constraints, e.g., symmetry, particular angles, etc, on the overall crease pattern. As with edge optimization, the overall scale is fixed, but in this case, rather than *maximizing* the same strain for all affected edges, the strain optimizer *minimizes* the RMS strain for a large set of edges with each edge potentially having a different strain. As with the strain optimizer, there is a set of strainable edges E^s and a set of moving nodes U^s. The strain optimizer solves the problem:

minimize $\sum_{e_i \in E^s} \sigma_i^2$ over $\{\sigma_i|_{e_i \in E^s}, \mathbf{u}_j \in U^s\}$ s.t.

(1) $0 \le u_{i,x} \le w$ for all $\mathbf{u}_i \in U^s$

(2) $0 \le u_{i,y} \le h$ for all $\mathbf{u}_i \in U^s$

(3) $$m\left[\sum_{\substack{e_k \in p_{ij} \\ e_k \in E^s}}\left[(1+\sigma_k)l_k\right] + \sum_{\substack{e_k \in p_{ij} \\ e_k \notin E^s}}\left[(1+\sigma_k)l_k\right]\right] - \sqrt{\left(u_{i,x}-u_{j,x}\right)^2 + \left(u_{i,y}-u_{j,y}\right)^2} \le 0 \text{ for all } p_{ij}, \in P^t$$

Equation 3 can also be broken into a fixed and variable part:

$$\sum_{\substack{e_k \in p_{ij} \\ e_k \in E^s}} \left[\underbrace{\sigma_k}_{\text{variable}} \underbrace{ml_k}_{\text{fixed}} \right] + \underbrace{m \left[\sum_{\substack{e_k \in p_{ij} \\ e_k \in E^s}} [l_k] + \sum_{\substack{e_k \in p_{ij} \\ e_k \notin E^s}} [(1+\sigma_k) l_k] \right]}_{\text{fixed}} - \underbrace{\sqrt{\left(u_{i,x} - u_{j,x}\right)^2 + \left(u_{i,y} - u_{j,y}\right)^2}}_{\substack{\text{fixed or} \\ \text{variable}}} \leq 0$$

6.6 Conditions

In addition to the conditions specified above, which are always required to be satisfied, one can optionally impose additional conditions on the crease pattern that are typically strict equalities. These are easily incorporated into the nonlinear constrained optimization machinery. The conditions and related equations currently implemented in TreeMaker are listed below.

Node position fixed

A node $\mathbf{u}_i$ that has one or both of its coordinates fixed to a point **a** must satisfy one or both of the equations

$$u_{i,x} - a_{i,x} = 0$$

$$u_{i,y} - a_{i,y} = 0$$

Node fixed to corner of paper

A node $\mathbf{u}_i$ that is constrained to lie on a corner of the paper must satisfy both equations

$$\left(u_{i,x} - w\right) \cdot u_{i,x} = 0$$

$$\left(u_{i,y} - h\right) \cdot u_{i,y} = 0$$

Node fixed to edge of paper

A node $\mathbf{u}_i$ that is fixed to lie on the edge of the paper must satisfy the equation

$$\left(u_{i,x} - w\right) \cdot u_{i,x} \cdot \left(u_{i,y} - h\right) \cdot u_{i,y} = 0$$

Node fixed to line

A node $\mathbf{u}_i$ that is constrained to lie on a line through point **a** running at angle α, such as a line of symmetry, must satisfy the equation

$$\left(u_{i,x} - a_x\right) \cos\alpha - \left(u_{i,y} - a_y\right) \sin\alpha = 0 .$$

Two nodes paired about a line

Two nodes $\mathbf{u}_i$ and $\mathbf{u}_j$ that are constrained to be mirror-symmetric about a line through point **a** running at angle α, such as a line of symmetry, must satisfy the two equations

$$\left(u_{i,x}-u_{j,x}\right)\cos\alpha-\left(u_{i,y}-u_{j,y}\right)\sin\alpha=0$$

$$\left(u_{i,x}+u_{j,x}-2a_{i,x}\right)\sin\alpha-\left(u_{i,y}+u_{j,y}-2a_{i,y}\right)\cos\alpha=0$$

Three nodes collinear

Three nodes $\mathbf{u}_i$, $\mathbf{u}_j$, and $\mathbf{u}_k$ that are constrained to be collinear must satisfy the equation

$$\left(u_{j,y}-u_{i,y}\right)\left(u_{k,x}-u_{j,x}\right)-\left(u_{k,y}-u_{j,y}\right)\left(u_{j,x}-u_{i,x}\right)=0$$

Edge length fixed

An edge e_k whose length is fixed must have its strain satisfy the equation

$$\sigma_k=0$$

Edges same strain

Two edge e_j and e_k that have the same strain must satisfy the equation

$$\sigma_j-\sigma_k=0$$

Path active

A path p_{ij} between two nodes $\mathbf{u}_i$ and $\mathbf{u}_j$ that is constrained to be active must satisfy the equation

$$m\sum_{e_k\in p_{ij}}\left[(1+\sigma_k)l_k\right]-\sqrt{\left(u_{i,x}-u_{j,x}\right)^2+\left(u_{i,y}-u_{j,y}\right)^2}=0$$

Path angle fixed

A path p_{ij} between two nodes $\mathbf{u}_i$ and $\mathbf{u}_j$ that is constrained to lie at an angle α must satisfy the equation

$$\left(u_{i,x}-u_{j,x}\right)\cos\alpha-\left(u_{i,y}-u_{j,y}\right)\sin\alpha=0.$$

Path angle quantized

A path p_{ij} between two nodes $\mathbf{u}_i$ and $\mathbf{u}_j$ that is constrained to lie at an angle of the form $\alpha_k=\alpha_0+k\cdot\delta\alpha$. where $\delta\alpha\equiv\frac{180^\circ}{N}$ must satisfy the equation

$$2^{N-1}\cdot\left[\left[\left(u_{i,x}-u_{j,x}\right)^2+\left(u_{i,y}-u_{j,y}\right)^2\right]^{-N/2}\right]\cdot\left[\prod_{k=0}^{N-1}\left[\left(u_{i,x}-u_{j,x}\right)\sin\alpha_k-\left(u_{i,y}-u_{j,y}\right)\cos\alpha_k\right]\right]=0$$

This function has the property that it goes to zero when the path is quantized, goes to ±1 in between quantized paths, and has no gradient component in the direction of shortening the path (which can cause problems when there are many such constraints). However, the code for the gradient of this function is rather complicated.

6.7 Meguro Stubs

A Meguro stub is a terminal node and edge added to the tree — usually emanating from the middle of an existing edge — such that the new terminal node creates exactly 4 active paths to other nodes in the tree.

When a Meguro stub is added inside of an existing *N*-sided polygon it breaks the polygon into 4 new polygons that all have fewer than *N* sides. Thus, by addition of Meguro stubs, high-order polygons are converted to lower-order polygons, until eventually all polygons in the crease patterns are triangles and can be filled with rabbit-ear molecules. This process is called "triangulation" of a crease pattern. By repeatedly adding Meguro stubs any crease pattern can be fully triangulated.

I introduce a few more definitions:

A polygon Q is defined by a set of terminal nodes U^{tQ} that form the vertices of the polygon and a set of terminal paths P^Q, which are all paths that span U^Q.

Define the set of nodes U^Q and edges E^Q as all nodes and edges that are contained within one or more of the paths P^Q; U^Q and E^Q constitute the *subtree* of polygon Q. Define U^{tQ} as the nodes in U^Q that are also terminal nodes, i.e., $U^{tQ} = U^Q \cap U^{tn}$.

Let e_{ab} be an edge of the subtree with $\mathbf{u}_a$ the node at one end of e_{ab} and $\mathbf{u}_b$ the node at the other end.

In general, for every edge e_{ab} and set of four distinct nodes $\mathbf{u}_i$, $\mathbf{u}_j$, $\mathbf{u}_k$, $\mathbf{u}_l$, there is a Meguro stub that terminates on a new node $\mathbf{u}_m$; the stub is defined by four quantities:

The node coordinates $u_{m,x}$ and $u_{m,y}$;

The distance d_m from node $\mathbf{u}_a$ at which the new stub emanates from edge e_{ab};

The length l_m of the new stub.

These four variables are found by solving the four simultaneous equalities

$$m\left[\sum_{e_k \in p_{ia}} l(e_k) + \begin{Bmatrix} d_m & \text{if } e_{ab} \in p_{ia} \\ -d_m & \text{if } e_{ab} \notin p_{ia} \end{Bmatrix} + l_m\right] - \sqrt{\left(u_{i,x} - u_{m,x}\right)^2 + \left(u_{i,y} - u_{m,y}\right)^2} = 0$$

for each of the four nodes $\mathbf{u}_i$, $\mathbf{u}_j$, $\mathbf{u}_k$, $\mathbf{u}_l$. Although a solution can potentially be found for any four nodes, not all are valid; the only valid combinations of nodes $\mathbf{u}_i$, $\mathbf{u}_j$, $\mathbf{u}_k$, $\mathbf{u}_l$ and edges e_{ab}

are those for which (1) the four nodes are distinct, and (2) *both* signs of d_m are represented among the four equations. In addition, solutions for d_m that are negative or greater than the length of e_{ab} are non-physical and must be discarded.

Note that for these equations to be used as written above, there can be no unrelieved strain in the system. They can clearly be modified to include strain.

6.8 Universal Molecule

The universal molecule is a crease pattern that is constructed by a series of repeated reductions of polygons. The construction is carried out by insetting the polygons and constructing reduced paths and fracturing the resulting network into still smaller polygons of lower order. As with triangulation, the process is guaranteed to terminate.

We use the same polygon definitions as were used in the description of Meguro stubs. In addition:

Assume the nodes $\mathbf{u}_i \in U^Q$ are ordered by their index, i.e., the *N*-sided polygon contains the indices $\mathbf{u}_1, \mathbf{u}_2, \ldots \mathbf{u}_N$, order as you travel clockwise around the polygon. Construct the following:

α_i is half of the interior angle at node $\mathbf{u}_i$. We define the quantity $\varsigma_i \equiv \cot \alpha_i$.

Define R_{90} as the operator that rotates a vector clockwise by 90°, i.e., $R_{90} \circ (u_x, u_y) = (u_y, -u_x)$

Define N as the operator that normalizes a vector, i.e., $N \circ \mathbf{u} = \dfrac{\mathbf{u}}{|\mathbf{u}|}$

$\mathbf{r}_i$ is the scaled bisector of the angle formed at $\mathbf{u}_i$, pointing toward the interior of the polygon with magnitude $\csc \alpha_i$. The vector $\mathbf{r}_i$ as well as ς_i can be constructed according to the following prescription:

$$\mathbf{r}' \equiv N \circ (\mathbf{r}_{i-1} - \mathbf{r}_i)$$
$$\mathbf{r}'' \equiv N \circ (\mathbf{r}_{i+1} - \mathbf{r}_i)$$
$$\mathbf{r}''' \equiv N \circ R_{90} \circ (\mathbf{r}'' - \mathbf{r}')$$
$$\mathbf{r}_i = \frac{\mathbf{r}'''}{\mathbf{r}''' \cdot \left[R_{90} \circ \mathbf{r}''' \right]}$$
$$\varsigma_i = \mathbf{r}_i \cdot \mathbf{r}'$$

The inset distance h is the largest value such that

(1) for every path p_{ij} of length l_{ij} between non-adjacent nodes $\mathbf{u}_i$ and $\mathbf{u}_j$,

$$\sqrt{\left(u_x + hr_x\right)^2 + \left(u_y + hr_y\right)^2} \le m\left[l_{ij} - h\left(\varsigma_i + \varsigma_j\right)\right]$$

(2) for every path p_{ij} between adjacent nodes $\mathbf{u}_i$ and $\mathbf{u}_j$,

$$h \le \frac{\mathbf{u} \cdot \mathbf{u}}{\mathbf{u} \cdot \mathbf{r}}$$

where

$$\mathbf{u} \equiv \mathbf{u}_i - \mathbf{u}_j$$

$$\mathbf{r} \equiv \mathbf{r}_i - \mathbf{r}_j$$

Although this problem is a nonlinear constrained optimization, since there is only one variable, it can be solved directly by simply solving for each equality for all possible paths p_{ij} and taking the smallest positive value of *h* found. The second relation (2) gives the value for *h* simply by replacing the inequality by equality. The solution for equality for relation (1), which is simply a quadratic equation in *h*, is given by the following sequential substitutions:

$$w = \varsigma_i + \varsigma_j$$
$$a = \mathbf{r} \cdot \mathbf{r} - w^2$$
$$b = \mathbf{u} \cdot \mathbf{r} + l_{ij} w$$
$$c = \mathbf{u} \cdot \mathbf{u} - l_{ij}^2$$
$$h = \frac{-b + \sqrt{b^2 - ac}}{a}$$

Obviously, negative or complex values of *h* should be ignored.

Once a solution is found, we create a set of new reduced nodes $\mathbf{u}_i'$ and reduced paths p_{ij}' of length l_{ij}'

$$\mathbf{u}_i' = \mathbf{u}_i + h\mathbf{r}_i$$

$$l_{ij}' = l_{ij} - h\left(\varsigma_i + \varsigma_j\right)$$

The reduced nodes and reduced paths are checked for active status and then subdivided into polygons, and the process is repeated.

7.0 Software Model

This document describes the overall structure of the *TreeMaker* software model, the data structures, and how they correspond to the mathematical structures the make up the Tree Method of origami design. *TreeMaker* is written in C++ and elements of the crease pattern are represented by C++ objects, which are data structures and functions that act upon these data structures.

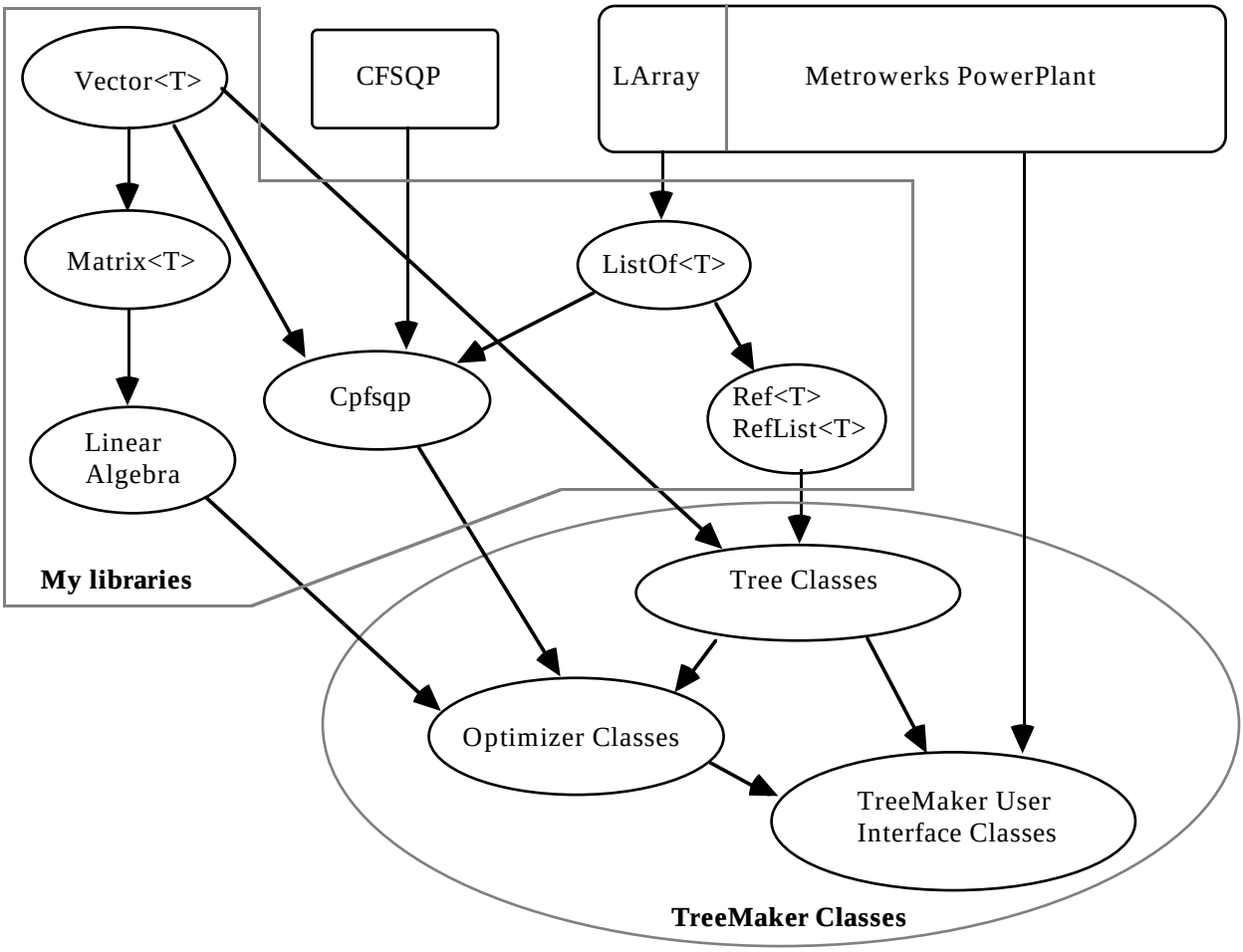
TreeMaker is strongly object-oriented. Early versions of TreeMaker were written in Object Pascal, which was a step up from its predecessors (I actually carried out some of the earliest work in Fortran), but beginning with version 3.0, I switched to C++. for its power and extensibility. Because of this, as I've added to the mathematical algorithms, I've tended to put them in a very "C++-like" form. (To a man with a hammer, everything looks like a nail.)

Whether or not C++ is the most "natural" language to do origami design modeling, I make use of many of the unique properties of the language in *TreeMaker*, including polymorphism, multiple inheritance, operator overloading, template classes and functions, run-time type identification, and exception classes. If you want to get understand the software model of *TreeMaker*, you should know C++.

A comment on terminology: there are two things called "objects" in this document: mathematical/geometrical objects (which are part of the mathematical algorithm) and software objects (C++ data structures). For simplicity, I've given the C++ objects names appropriate to their mathematical analogous structure. To avoid confusion, I'll refer to mathematical objects with lower-case names, while all C++ objects have upper-case names and will appear in `Courier font`. So, for example, a node is a part of the mathematical model; a `Node` is a C++ object that represents the mathematical object.

7.1 Overview

The code for *TreeMaker* consists of a bunch of classes that interact with each other. *TreeMaker*'s classes are grouped into modules; classes within a module are related to each other in some way. Some modules depend on or are derived from other modules. A top-level view of all the modules in the program are shown below in figure 7.1.



7.1

There are four main groups of modules. Two — PowerPlant and CFSQP — are libraries written by other guys. One — "My libraries" — is written by me. The fourth group —"TreeMaker Classes" — is composed of the classes unique to *TreeMaker*. A brief discussion of these four groups follows.

Metrowerks PowerPlant

PowerPlant is a Macintosh class library supplied with the Metrowerks CodeWarrior family of compilers. It provides a complete set of classes for building a Macintosh user interface, as well as a number of useful utility classes. The user interface of *TreeMaker* is built from the PowerPlant class library.

PowerPlant is Mac-specific. This makes porting TreeMaker to any other platform a bit of an undertaking. PowerPlant is a very complex library and is the subject of a 600-page manual (*The PowerPlant Book*), so I won't bother trying to describe any more here.

PowerPlant is itself composed of several independent modules, some of which can stand alone. One of them, the `LArray` class, as shown above, serves as the base for a number of my own class libraries.

CFSQP

CFSQP is a general-purpose nonlinear constrained optimization code, written in C, developed by the research group of Professor Andre Tits at the University of Maryland. It forms the core of the numerical optimization routines (replacing a home-rolled implementation of the Augmented Lagrangian Multiplier method used in earlier versions). A description of the routines may be found on the FSQP Home Page, located at

http://www.isr.umd.edu/Labs/CACSE/FSQP/fsqp.html

CFSQP is used with permission according to the following conditions of use:

Prof. Andre L. Tits
Electrical Engineering Dept.
and Institute for Systems Research
University of Maryland
College Park, Md 20742
U. S. A.

Phone: 301-405-3669
Fax: 301-405-6707
E-mail: andre@eng.umd.edu

My libraries

TreeMaker uses a number of general-purpose class libraries that I've written. They are:

Vector<T>

`Vector<T>` is a template class for vectors (not the same as the STL `vector<T>` class) that supports arithmetic operations and various vector operations.

Matrix<T>

`Matrix<T>` is a template class for matrices that supports arithmetic operations and various matrix operations.

LinearAlgebra

`LinearAlgebra` contains a class, `Ludecomp<T>`, that perform various linear algebra operations, including LU decomposition, matrix inversion, and solution of linear systems. It also includes a class `NewtonRaphson<T>` that solves systems of nonlinear equations.

Cpfsqp

`Cpfsqp` is a C++ interface to the CFSQP library for performing nonlinear constrained optimization. It is particularly suited for embedded applications where constraints are regularly added or removed or where the number of constraints are not known beforehand. `Cpfsqp` contains two classes, `Cpfsqp`, which takes over some of the housekeeping for problems with a large number of constraints, and `DifferentiableFunction`, which is a base class for a function of a vector that has a defined gradient function.

ListOf<T>

`ListOf<T>` is a template class based on the (stand-alone) `LArray` class in PowerPlant. It provides type-safe arrays, support for PowerPlant's `LArrayIterator` class for iterating through mutable arrays, and supports two independent notations for accessing arrays: a Pascal-style (1-based) notation, and a C-style (0-based) notation. It also adds routines for treating arrays as sets, supporting intersection and union of lists. (It's my "general-purpose list" class.)

Ref<T>, RefList<T>

The Reference classes are classes that provide the equivalent of a "smart" pointer-to-T; when the object of a `Ref<T>` is deleted, the `Ref<T>` sets itself to zero no matter where it is. This eliminates dangling pointers (or at least transforms them into NULL pointers, which are easier to catch). In addition, it's possible to configure the objects being referenced in such a way that they delete themselves when the number of references drops below a preset value (automatic garbage collection). (I don't use this latter capability in *TreeMaker*.)

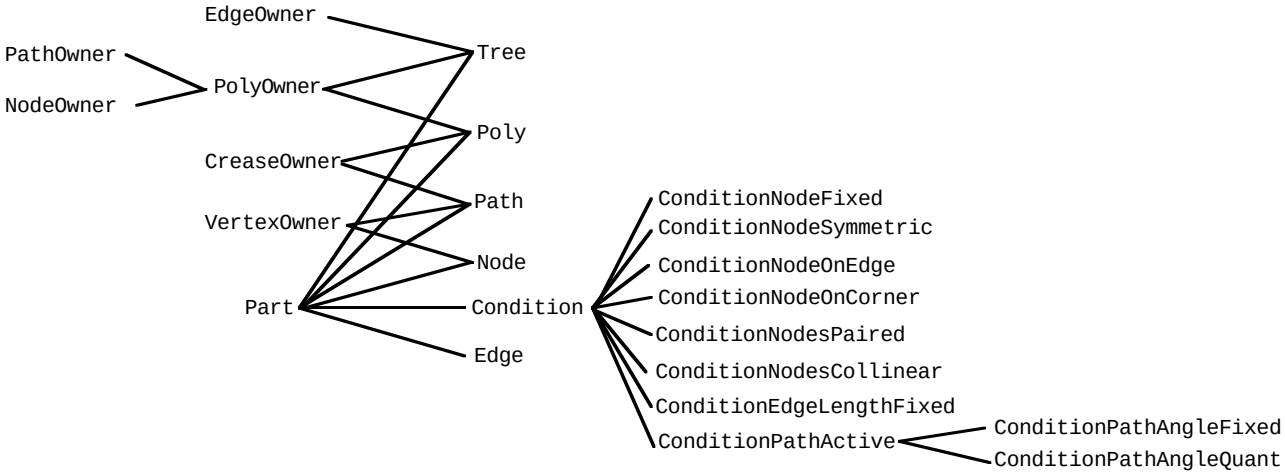
TreeMaker Classes

TreeMaker classes are the classes unique to the *TreeMaker* application. They are grouped in 3 sets: Tree Classes, which support and maintain the internal structure of the Tree; they are nearly platform-independent. Optimizer Classes perform calculations and implement algorithms that act upon the Tree. They, too, are nearly platform-independent. User Interface Classes provide the user interface, support display and editing of Tree objects, interaction with the OS and the file

system. The UI classes are heavily platform-dependent; they are built from PowerPlant, which is a Macintosh-specific class library.

Tree Classes

The Tree Classes are the classes that support a unified mathematical model of the tree (the stick figure); the crease pattern; and the relationships between the various objects. The figure below shows the different classes and their inheritance hierarchy.



7.2

The primary Tree Classes are:

`Tree`

`Node`

`Edge`

`Path`

`Poly`

`Vertex`

`Crease`

`Condition`

All primary classes inherit similar behavior from the `Part` base class. Some classes have the capability of owning instances of other classes. `Condition` is a base class for conditions imposed on the crease pattern (e.g., bilateral symmetry). A more detailed description of the Tree classes is given later.

Optimizer Classes

Optimizer classes perform mathematical operations on the Tree that generate a valid crease pattern. Optimizer classes are At present, there are four optimizer classes:

`ScaleOptimizer`

`EdgeOptimizer`

`StrainOptimizer`

`StubFinder`

These four classes do the mathematical “heavy lifting”; they actually solve for particular creases patterns. They are derived from pure numerical classes (`NewtonRaphson` and `Cpfsqp`).

`ScaleOptimizer` finds the arrangement of nodes that gives the largest possible crease pattern.

`EdgeOptimizer` selectively increases particular edges to their largest possible size.

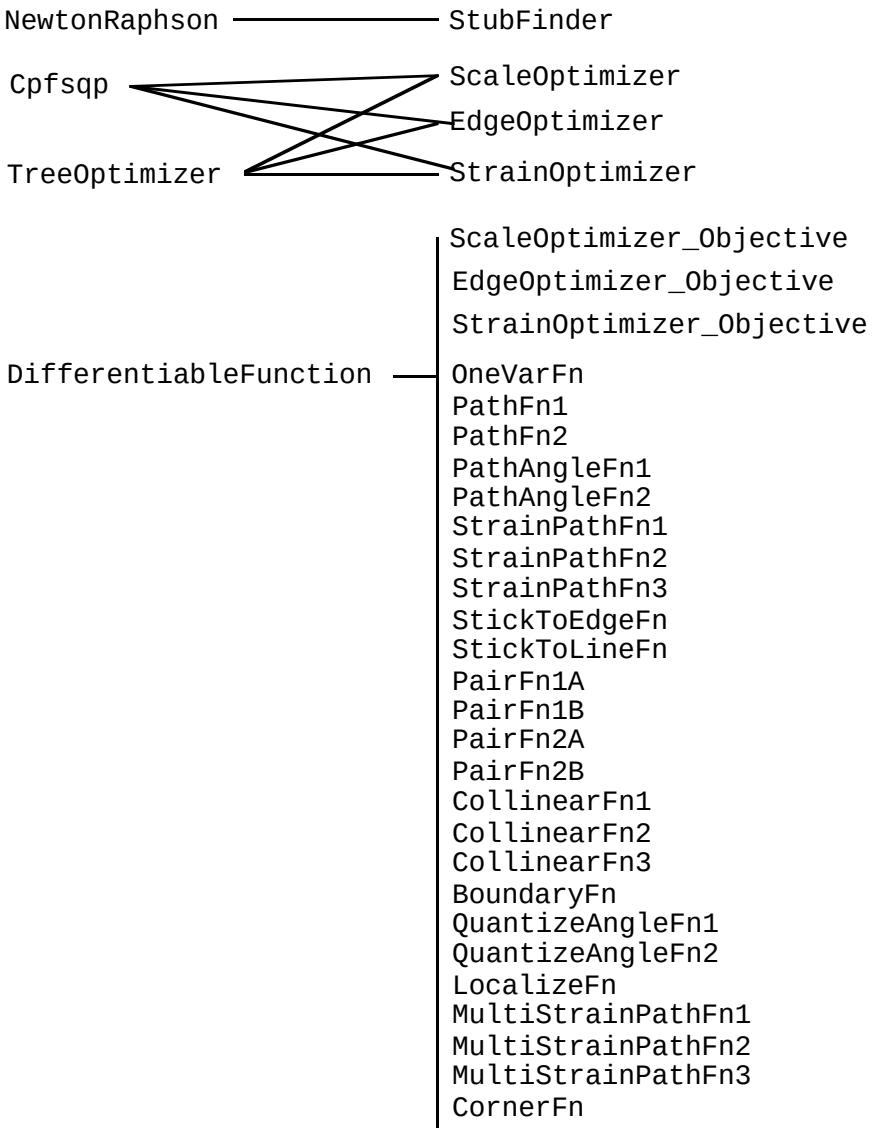
`StrainOptimizer` is used in heavily constrained situations; it finds a configuration that minimizes the strain on the edges.

`StubFinder` is a simple solver that adds stubs to a tree to break high-order polygons into lower-order polygons.

The three optimizer classes inherit from the `TreeOptimizer` class, which embodies platform-dependent behaviors: cancellation, error messages, and showing of progress.

To support the optimizers, there are a number of `DifferentiableFunction` objects that are used for constraints and objective functions for the optimizers.

The inheritance relationships between the optimizer classes are shown below.



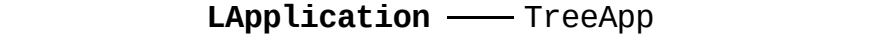
7.3

User Interface Classes

The user interface of TreeMaker is implemented by a collection of PowerPlant object descendants. In the following description, PowerPlant classes are in boldface type.

Application

The `TreeApp` is the application object that maintains interaction with the operating system, puts up and controls the menu bar, and dispatches commands and clicks to control objects.



7.4

Document

`TreeDoc` is the document object that is "command central" — menu selections, keystrokes, and mouse actions are routed to the `TreeDoc`, which places the appropriate calls to the `Tree` object (which is a member variable of the `TreeDoc`).

`LSingleDoc` ——— `TreeDoc`

7.5

Views

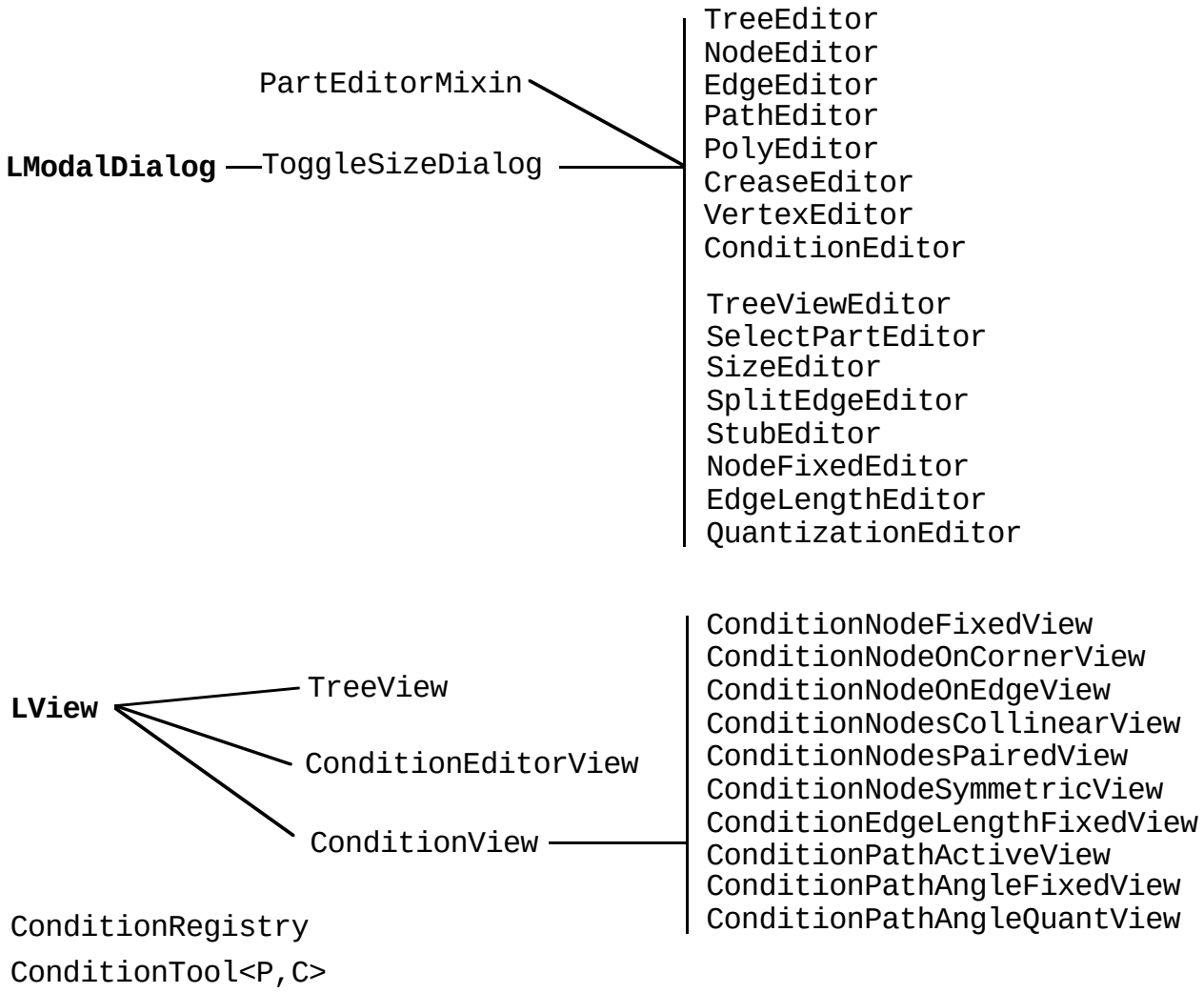
The bulk of TreeMaker consists of various **`LView`** subclasses that let the user see and manipulate the tree object. The main view is the `TreeView` object, which displays the tree, the various parts, and the crease pattern. By selectively choosing which parts to display, the user can choose to view the tree, the crease pattern, or a combination of the two.

There is also a group of editors which allow you to examine or manipulate the fields of individual objects. These are all inherited from the PowerPlant class **`LModalDialog`** and a utility class, `PartEditorMixin`, which supplies useful routines for displaying object addresses as indices.

A second group of editors perform small manipulations on the tree, e.g., changing the paper size, setting the lengths of a group of edges, and so forth.

The `ConditionRegistry` class provides a binding between `Condition` objects and the views used to reference them.

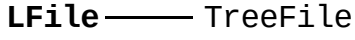
The `ConditionTool<P, C>` class provides tools for creating and filtering `Condition`s for `Part`s and lists of `Part`s.



7.6

File I/O

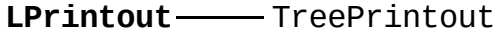
File I/O is provided by the `TreeFile` class, which implements storage in a platform-independent text-based format.



7.7

Printouts

The crease pattern can be printed out using the standard Apple print protocols. Printing is handled by a slightly modified subclass of the PowerPlant `LPrintout` object.



7.8

Miscellaneous

I also found it necessary to write a number of small utility packages or modified classes. They are:

`Alerts` - a wrapper to handle simple alerts

`AnimatedCursor` - displays a cursor animation during calculations

`LCropMark` - a pane that displays crop marks in a printout

`LFloatEditField` - an edit field that displays a floating-point value

`LGACaptEditField` - an edit field that turns into a `LCaption` when it is disabled

`LGADeadCheckbox` (obsolete) - a checkbox that doesn't respond to clicks. It's a convenient tool for displaying binary information

`LGAModalDialogBox` - a dialog box (derived from `LGADialogBox`) that displays true modal dialog behavior (disabling all commands when it's in the foreground).

`LGATargetAnnouncingEditField` - an edit field that broadcasts a message when it becomes the target.

`QDUtils` - routines for doing arithmetic with QuickDraw `Point` records

`ToggleSizeDialog` - a dialog box that has two sizes you can toggle between. I use it to hide object data fields that aren't used directly but keeps them accessible.

`UParamText` - a utility that does for PowerPlant views what the system routine `::ParamText()` does for resource-based dialogs and alerts.

`UPlaceDialog` - a utility that places dialogs in the right place on the main screen

`Sort` - a template class for sorting an array of numbers.

7.2 Tree Classes: Details

In this section the fundamental data structures are discussed along with their inheritance hierarchy. Many objects inherit from a "Owner" class; as discussed below, an "ObjectOwner" maintains a list of references to "Object" and when the "Owner" is deleted, it kills all of its owned objects.

The relationships between objects are maintained by references and lists of references. These are summarized in the table below. Bolded names are references that are inherited from subclasses.

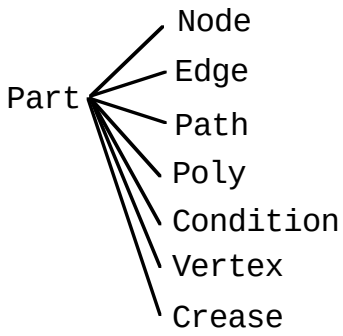
Part-ness

Most objects inherit from a base class called `Part` which has two member variables: an index, which is set by the owner of the part, and a pointer to the top-level `Tree` structure.

```
class Part
```

```
{
   public:
      short mIndex;
      Tree* mTree;
}
```

References to objects are made visible to the user by the `mIndex`, the value of which is set by the owner of the part (see below). Given a reference to any `Part`, the `mTree` reference gives access to the overall `Tree`. All TreeMaker objects inherit from `Part` as shown below:



7.9

Conversely, the `Tree` has a set of lists of references to all `Part`s with names like `mNodes`, `mEdges`, `mPaths`, etc. This provides flat access to the hierarchical data structure; it simplifies searching over the `Part`s and lets us assign the `Part` index according to its position within the master list.

Ownership

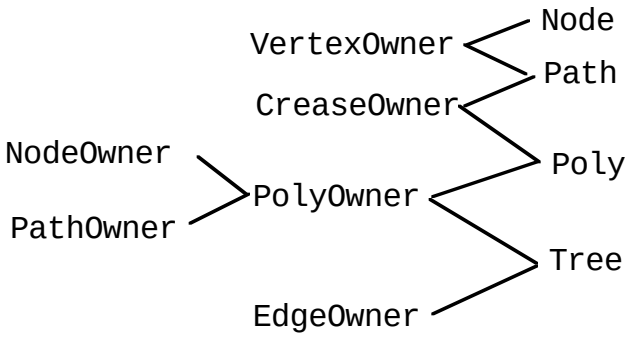
The data structures in *TreeMaker* are tied together by various relations. One important set of relations is the concept of "ownership." Every object is owned by another object. The chain of ownership is a simple rooted tree, that is, every object has a unique owner, which in turn has a unique owner, all the way up to the `Tree` at the top of the hierarchy. (It's similar to a "chain of command.")

Ownership confers two important properties:

When an owner object is deleted, it deletes all of the objects that it owns

An owner object can be interrogated about objects that it owns, for example, to return an object that satisfies certain criteria.

Ownership ability is conferred through inheritance. The chain of ownership is shown in the figure below. Objects to the right inherit from objects to the left.



7.10

There are a couple of interesting aspects of these relationships that could be confusing:

1. A `Poly` can own other `Poly`s, which are called subPolys.

2. Both `Poly` and `Tree` own `Poly`s and hence (through inheritance) also own `Node`s and `Path`s. A `Poly` owns all `Node`s that are fully enclosed by the `Poly` and owns all `Path`s that connect those `Node`s.

Both `Poly` and `Path` can own `Creases`. However, only the `Tree` can own `Edge`s.

3. Both `Node`s and `Path`s can own `Vertices`.

The owner for an object has a member variable that points back to the owner; e.g., the `Node` object has a member variable, `mOwnedNodes`, that points to the `NodeOwner`.

Because the behavior of different types of Owners is similar, there is some logic to embedding ownership behavior into a template class (e.g., `Owner<Vertex>`, `Owner<Node>`, etc.). However, I found that it is easier to follow the code if I give each owner a unique name ("`mNodeOwner`") and use unique names for owned objects ("`mOwnedNodes`"). Also, as you will see, there are some member functions unique to particular Owner classes.

Here are the types of objects that other objects own:

A terminal `Node` or ring `Node` may own a single `Vertex` that coincides with the position of the `Node`.

The following types of `Path`s own `Crease`s: active paths, spoke paths, ridge paths, and optionally, ring paths.

Active paths, ring paths, and ridge paths can own internal vertices as well that lie along the paths.

A `Poly` owns radial `Crease`s that connect to its reduced polygons and/or any ridge `Crease`. A `Poly` also owns `Crease`s that run between internal vertices. A `Poly` also owns its sub-polys.

A `Tree` owns its `Node`s, `Edge`s, `Path`s, and `Poly`s.

Persistence

In memory, references to objects are made by pointers. To the user, and in storage, references are maintained by the Part index. The Tree contains master lists of all Parts which are used solely for storage and indexing. This lets us follow this model for reading in the structure:

1. Create the appropriate number of blank, uninitialized Parts

2. Read in each part, and "on the fly" convert all indices to Part references.

Every Part constructor supplies the `Tree*` variable separately from the owner variable.

`Condition`s, because they are polymorphic, are handled slightly differently, since we don't know a priori which type of blank `Condition` to create. Therefore, as described further in the source code (TreeFile.cp), `Condition`s are created on the fly through the `ConditionRegistry` object.

References: `Ref<T>` and `RefList<T>`

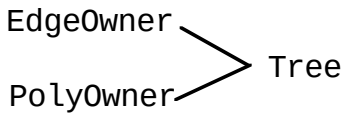
Because of the complexity of the data structure — most parts maintain several lists of related parts (e.g., a `Node` has a list of all `Path`s that begin or end on it) — when we create or delete parts, there are a lot of references to clean up, particularly when objects are deleted. Earlier versions of *TreeMaker* maintained these references by pointers and lists of pointers. Unfortunately, the program was frequently plagued by dangling pointers created when I failed to clear all pointers scattered throughout the data structure that referenced a deleted part.

In *TreeMaker* 4.0, references and lists of references are maintained through the `Ref<T>` and `RefList<T>` template classes, which are essentially "smart pointer" classes. A `Ref<T>` behaves like a `T*` (you can dereference it both directly and indirectly and assign it to `T*`) but if you ever delete an object of type T, every `Ref<T>` that referred to it clears itself and subsequent attempts to dererence return `NULL`. This simplifies the programming, since I don't need to clear such references explicitly, and it also eliminates a great many bugs since attempts to dererence a pointer to a deleted object (which snuck through in earlier versions) are converted to attempts to dereference a void*, which is more easily caught and eliminated.

A `RefList<T>` is similar to a `Ref<T>`. It behaves like a `ListOf<T*>`, except when an object from the list is deleted, its reference is completely removed from the list.

`Tree`

A `Tree` is the top-level data structure that contains and ties together the stick figure we are trying to represent and the crease pattern for the corresponding base. There is one unique `Tree` for each document.



7.11

The `Tree` class has two friend classes, `TreeCleaner` and `TreeFile`, which are utilized for housecleaning and persistence, respectively.

The references contained by a `Tree` are summarized below. Names in bold refer to inherited member variables.

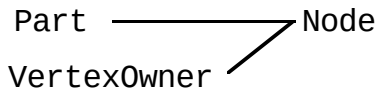
Object	*References*	*What*
`Tree`	**`mOwnedNodes`**	All nodes in the primary tree (i.e., not including nodes in the interior of a poly, which are owned by the poly itself)
	`mOwnedEdges`	All edges in the primary tree
	`mOwnedPaths`	All paths in the primary tree (i.e., not including reduced paths inside a poly, which are owned by the poly)
	`mOwnedPolys`	All top-level polygons (i.e., not including polys that are subpolys)
	`mNodes`	All nodes
	`mEdges`	All edges
	`mPaths`	All paths
	`mPolys`	All polygons
	`mConditions`	All conditions
	`mVertices`	All vertices
	`mCreases`	All creases

`Node`

A `Node` is the data structure that represents a node of a planar graph. There are several types of planar graph utilized in *TreeMaker*; there is the top-level graph, which is the target of the modeling; but there are also sub-graphs that are used in the process of insetting polygons.

There are two types of `Node`. A “tree node” is a node that is part of the tree and is owned by the tree. Tree nodes come in two flavors: terminal nodes, which have exactly one edge, and internal nodes, which represent nodes where two or more edges come together.

A “poly node” is a node that is owned by a poly. Poly nodes are the reduced images of tree nodes; so all poly nodes are also reduced nodes.



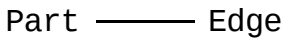
7.12

The references contained by a `Node` are summarized below. Names in bold refer to inherited member variables.

Object	*References*	*What*
`Node`	**`mOwnedVertices`**	The vertex associated with this node
	`mEdges`	Edges connected to this node
	`mPaths`	Paths that terminate on this node (*not* paths that contain this node)
	`mNodeOwner`	Poly or tree that owns this node

`Edge`

A `Edge` is the data structure that represents an edge of the top-level planar graph. An `Edge` connects two `Node`s. Each `Edge` corresponds to a flap (or segment) of a base.



7.13

The references contained by an `Edge` are summarized below. Names in bold refer to inherited member variables.

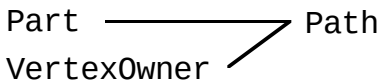
Object	*References*	*What*
`Edge`	`mNodes`	2 nodes at each end of this edge
	`mEdgeOwner`	Tree that owns this edge

`Path`

A `Path` is an object that describes a particular relationship between two `Node`s. Terminal paths connect terminal nodes. A `Path` has associated with it a length, which is the minimum length between two nodes of the graph.

A “tree path” is a path owned by the `Tree`. Tree paths have a list of edges and internal nodes that represents the path along the tree from which its length is derived.

A “poly path” is a path owned by a `Poly`. Poly paths have their lengths computed by reducing the length of the original path from which it is derived.



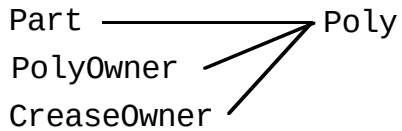
7.14

The references contained by a `Path` are summarized below. Names in bold refer to inherited member variables.

Object	*References*	*What*
`Path`	**`mOwnedVertices`**	All vertices that occur along this path
	`mNodes`	Ordered list of nodes from one end of path to the other
	`mEdges`	Ordered list of edges from one end of path to the other
	`mFwdPoly`	The polygon that this path belongs to, enumerated in the forward direction
	`mBkdPoly`	The polygon that this path belongs to, enumerated in the backward direction
	`mPathOwner`	poly or tree that owns this path

`Poly`

A `Poly` is a polygon of the crease pattern. The paper is divided into one or more polygons; each polygon may be further subdivided into one or more reduced polygons. Each polygon owns reduced nodes, reduced paths, and creases.



7.15

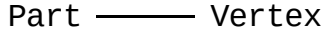
The references contained by a `Poly` are summarized below. Names in bold refer to inherited member variables.

Object	*References*	*What*
`Poly`	**`mOwnedNodes`**	All nodes owned by this poly; this is the distinct subset of `mInsetNodes`; i.e., every node in `mInsetNodes` appears exactly once in this list.

mOwnedPaths	All paths owned by this poly, which are one of the following: (1) paths that connect two owned nodes, i.e., ring or cross paths or the ridge path (2) paths from the outer poly to owned nodes, i.e., spoke paths
mOwnedPolys	All subpolys of this poly
mOwnedCreases	All creases that connect the edges of this poly to the edges of its subpolys
mRingNodes	The nodes that form the vertices of this polygon
mRingPaths	The paths that form the edges of this polygon, i.e., an ordered list of paths that connect consecutive vertex nodes.
mCrossPaths	All other paths that connect the mRingNodes, i.e., those paths connecting non-consecutive mRingNodes.
mInsetNodes	Mapping from ring nodes to inset nodes. Note that several ring nodes may map to the same inset node, i.e., this list may have duplicate entries.
mSpokePaths	Paths that connect ring nodes to inset nodes for this poly.
mRidgePath	If the poly has exactly two distinct inset nodes, this is the path that connects them. We need this path because some creases may terminate on it.
mPolyOwner	Poly or tree that owns this poly

Vertex

A Vertex is a point where two or more creases come together. The Vertex inherits only from Part.



7.16

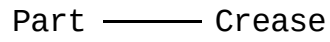
The references contained by a Vertex are summarized below. Names in bold refer to inherited member variables.

Object	*References*	*What*

Vertex	mCreases	All creases that terminate on this vertex
	mVertexOwner	node or path that owns this vertex

Crease

A Crease is a line in the crease pattern. There are three types of Creases: ValleyCrease, MountainCrease, and TristateCrease. The Crease inherits only from Part

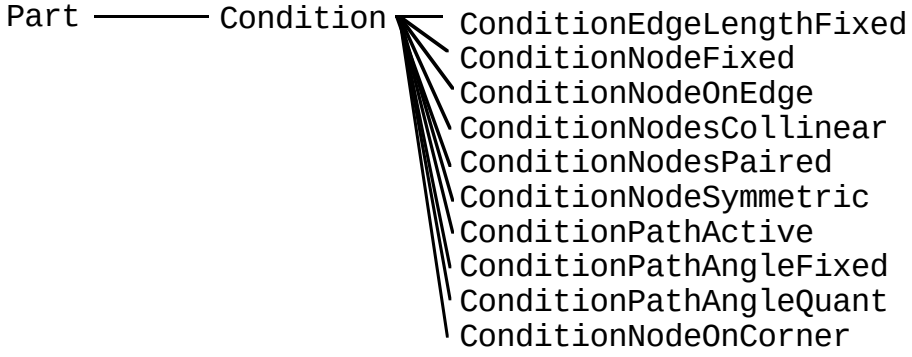


The references contained by a Crease are summarized below. Names in bold refer to inherited member variables.

Object	*References*	*What*
Crease	mVertices	2 vertices at each end of the crease
	mCreaseOwner	poly or path that owns this crease

Condition

A Condition is a relationship between portions of the tree to implement symmetry, fixed angles, and other constraints on the crease pattern.



7.17

Condition is a base class that defines the behavior for all possible conditions, which may establish and enforce relationships any other part of the tree. Before an optimization is performed, all Conditions associated with a Tree are polled and are allowed to apply mathematical constraints (equalities and inequalities) that enforce the condition.

At present, there are 10 types of Condition on Nodes, Edges, and Paths.

8.0 Final Stuff

8.1 Comments and Caveats

While *TreeMaker* is a very powerful tool for origami design (if I do say so), it is important to be aware of its limitations, so you don't try to use it for something it's not suited for. There are also several subtleties to *TreeMaker* design that, when properly accounted for, will further improve your productivity.

Keep in mind that *TreeMaker* is basically a dumb optimizer; it does not seek out symmetries that lead to particularly elegant folding sequences, nor does it recognize when a slight modification of the lengths of branches might allow a more elegant or symmetric folding sequence. *TreeMaker* always takes the brute-force approach, seeking the absolute optimum, even when a slight deviation from the optimum would make a much more elegant model. However, you can recognize when a particularly symmetric configuration is close at hand and can set conditions to force a symmetric and/or more elegant solution.

Also, *TreeMaker* finds local optima, not global optima. If you see a lot of unused space in the crease pattern, you might consider rearranging a few circles and restarting the search routine. You may find a better solution with a different initial trial solution. In general, efficient solutions have many 3- and 4-sided polygons after the first optimization. If you find that you get several polygons with a large number of sides, try dragging one or more nodes into the center of the big polygon and re-optimizing.

Most importantly, some subjects just don't lend themselves to design using tree theory. Since tree theory abstracts the model as a stick figure, for subjects that can be approximated by a stick figure, *TreeMaker* works very well; but for subjects that are not easily turned into a stick figure, *TreeMaker* is the wrong tool for the job. For example, for chunky models or models that require large flat regions (such as a hippo or butterfly) other design algorithms are likely to do a better job than *TreeMaker*. Also, since the tree itself is inherently one-dimensional, you can't specify colors directly for models that show both sides of the paper; while you can force flaps to be edge or corner flaps (and thus are easily reversible), you can't easily design complex two-toned models such as zebras or tigers.

Finally, keep in mind that for all its power, *TreeMaker* only provides a base. It's still up to you to thin the flaps and convert the base into the final model, and there is a lot of room for personalizing, customizing, and putting your own stamp of artistry onto your design.

TreeMaker is written in C++ using Metrowerks's CodeWarrior development environment. Versions prior to 4.0 of *TreeMaker* may be freely distributed provided that you include all documentation and the copyright notice unmodified and that you do not charge anything for it. Version 4.0 may not be freely distributed due to copyright restrictions on some of the code. See the cover page of this manual for license terms.

TreeMaker is constantly evolving, and my priorities are to fix known bugs and add features but not to deal with obscure incompatibilities. It ain't bulletproof, and while I currently know of no life-threatening bugs and will try to fix any that I hear about, it still may exhibit unexpected behavior (*i.e.*, crashes). Have fun playing around, but use it at your own risk.

If you have any suggestions or comments, please send them to me at the address given at the end of this document.

8.2 Version History

(1.0) 05-01-93. Original *TreeMaker* program. Edits nodes and edges and finds optimum distribution of nodes but no creases. Written in Symantec's THINK Pascal 5.0 using the THINK Class Library, v. 1.1.

(2.0) 08-06-93. Added CPath object and an editor for same. You can now individually change the length of a path, as well as control its angle and/or its collinearity with another node. Numerous small changes in user interface. Version 2.0 documents are incompatible with version 1.0 application and vice versa.

(2.0.1) 03-11-94. Changed the scale box to display 5 digits of accuracy.

(2.0.2) 03-31-94. Added “Show All Paths” to the Action menu so you can toggle paths on and off with a keystroke.

(3.0a1) 03-01-95. Complete from-the-ground-up rewrite in C++, using Metrowerks CodeWarrior and the Metrowerks PowerPlant class library. Completely restructured mathematical model, improved scale optimization routines, added secondary optimization, many new constraints, text-based file format, computation of molecule creases, and other stuff too numerous to mention.

(3.0b1) 06-25-95. First beta version.

• Added universal molecules and removed computation of tristate creases for compatibility with universal molecules.

• Added “diag” and “book” preset buttons to Tree Editor.

(3.0) 08-18-95 released.

• Fixed bug in computation of inset creases for nearly-parallel lines.

• Increased weight on “Stick to Edge” and “Collinear Node” constraints, which improves optimizations when many nodes are present.

• Finished documentation and uploaded for distribution to origami-L archives.

(3.5) 12-22-95 released.

• Generally improved consistency of dialogs and validation:

• In all dialogs, controls and edit fields are disabled and blanked if they’re not appropriate (e.g., symmetry-related controls are disabled if the tree doesn’t have a symmetry line defined).

• All editors now use a common syntax (“FillDialog”) for initializing the dialog.

• Dialogs no longer alter parts directly; instead, they issue commands to the TreeDoc together with a list of the new settings. (This is in preparation for implementing Undo/Redo by bottlenecking all editing actions through the TreeDoc.)

• In dialogs, eliminated unnecessary ctor/dtor method declarations and inlined from-LStream ctors.

• Private data shown in part editors was made less obtrusive (smaller type)

• All part indices are now fully validated in dialogs

• Validation alerts are provided by a new superclass for all dialogs, class TreeDialog. Text in validation alerts is now stored in an 'STR#' resource (per Mac UI guidelines).

• Fixed bug in which menus were not updated properly after a Tab or Delete keystroke.

• *Optimize Selected Nodes…* command and StretchyEdgeOptimizer have been modified so that the only nodes and edges considered for scaling are those in the current selection.

• Many new commands and optimizers added to Action Menu:

• Added *Split Edge…* command which lets you add a node to the middle of an edge

• Added *Absorb Node* command which lets you remove a redundant node from the middle of an edge

• Added *Absorb Redundant Nodes* command, which removes all redundant nodes

• Added *Arrange Internal Nodes* command, which cleans up the position of internal nodes.

• Added *Fracture Poly…* command, which lets you add a node inside a polygon that forms four active paths with the polygon nodes, effectively fracturing the polygon into smaller polys.

• Added *Triangulate Tree* command, which fractures all polys in the crease pattern down to order-3 polys.

• Fixed memory leak in ConstrainedMinimax (constraints were not being destroyed)

• Restored computation of tristate creases for rabbit-ear molecules in *Build Creases*.

• Updated documentation and added tutorials to describe the new commands.

• "Wait" cursor is now in color.

• Improved *About…* box.

(3.6) 03-18-96 released

• Added "Show Creases and Circles" command, which is a useful view for printing.

• Rewrote TreeFile class to buffer file I/O, gaining over an order of magnitude in speed during Save/Open.

• Converted mathematical classes (Minimax, ConstrainedMinimax, etc.) to templates

• *About…* box is faster.

(3.7) 12-3-96

• Fixed a bug that affected the display of polygons in the main window and screwed up printing on certain types of printers; also added a background to the main window.

• Added a background to valid polygons.

• Added crop marks in multipage printouts so you can print out large patterns on multiple sheets and can accurately cut and paste them together

• Removed the unnecessary border around the square in the printout, which increases the size of the largest square you can fit onto a single page to 7.5 in.

• UI is now updated to PP Constructor 2.3 format, including use of new LPrintout.

• Added a modest speed improvement to the “Fracture Poly” algorithm by eliminating some unnecessary calculations

• Added new Path::mIsValidPath member variable, which is used in the improved polygon-finding algorithm. Also added the corresponding display in the Path Editor.

• Big improvements in the polygon-finding algorithm; border paths don’t have to be active to form polygons. Non-optimized but valid node arrangements will now form crease patterns.

• Improved the algorithm that identifies pinned nodes. Nodes are now pinned only by active paths and the edges of the paper.

• Added offscreen drawing, which improves the look and feel.

• Tweaked the color scheme slightly for better visibility and contrast.

• Added a polygon editor. There’s nothing to change, but it does let you examine some of the internal characteristics of polygons.

(4.0) 4-1-98 released

• Complete overhaul of software model. Standard objects — Node, Edge, Path, Poly, Tree — have many new fields. Phased out the Fold object; introduced the Vertex, Crease, and Condition objects.

• All objects are now descended from Part and have an index.

• Polys are now hierarchical. Objects can own one another.

• Initial suite of Conditions includes 10 different types.

• Introduced strain into edges

• Replaced ConstrainedMinimax<> optimizers with optimizers based on CFSQP 2.5. Created supporting classes for same.

• Introduced ToggleSizeDialog so that part flags are normally concealed.

• Replaced all View objects with Grayscale classes.

• Introduced strain minimization. Rewrote EdgeOptimizer to utilize strain as well.

• Polys are no longer automatically built, but must be explicitly called.

• Introduced TreeCleaner class to insure automatic cleanup after editing.

• Added Condition menu to hold related commands.

• Rearranged command-key equivalents in menus.

• Introduced new file structure for future compatibility.

8.3 Sources

For comments, suggestions, attaboys and whaps, please write to me at:

Robert J. Lang
7580 Olive Drive
Pleasanton, CA 94588
rjlang@aol.com

For more information on CFSQP, check out the CFSQP web site at:

http://www.isr.umd.edu/Labs/CACSE/FSQP/fsqp.html