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.

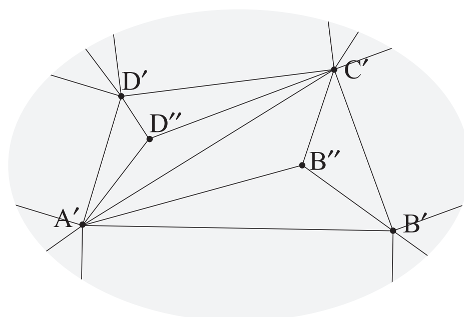


Figure 15. On the inner quadrilateral, construct the bisectors of each triangle to find points B'' and D''.

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.

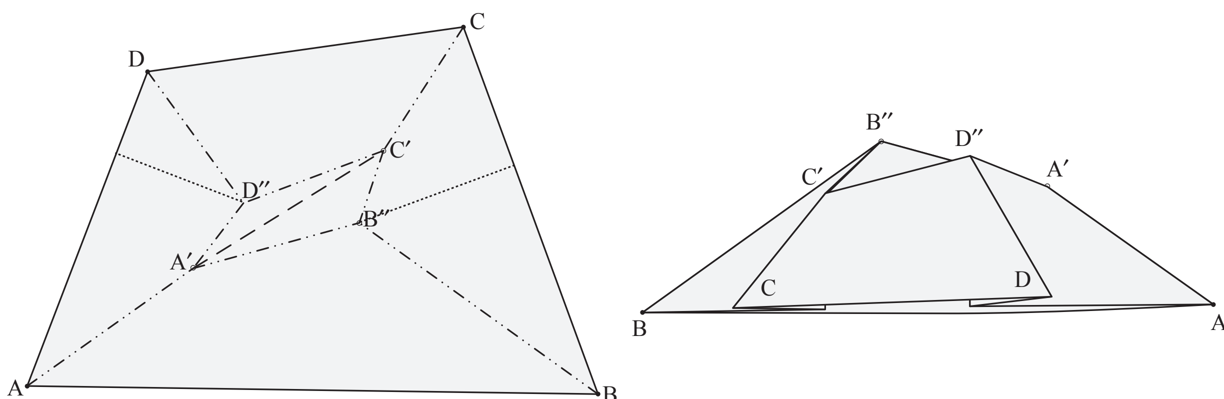


Figure 5.16. (Left) The mountain and valley folds of the gusset quad. Two tri-state creases are shown. (Right) The folded version of the gusset quad.

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.

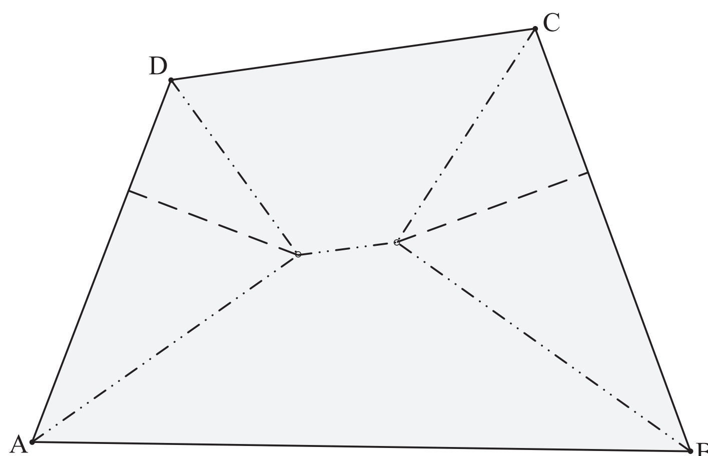


Figure 5.17. The Maekawa crease pattern.

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.

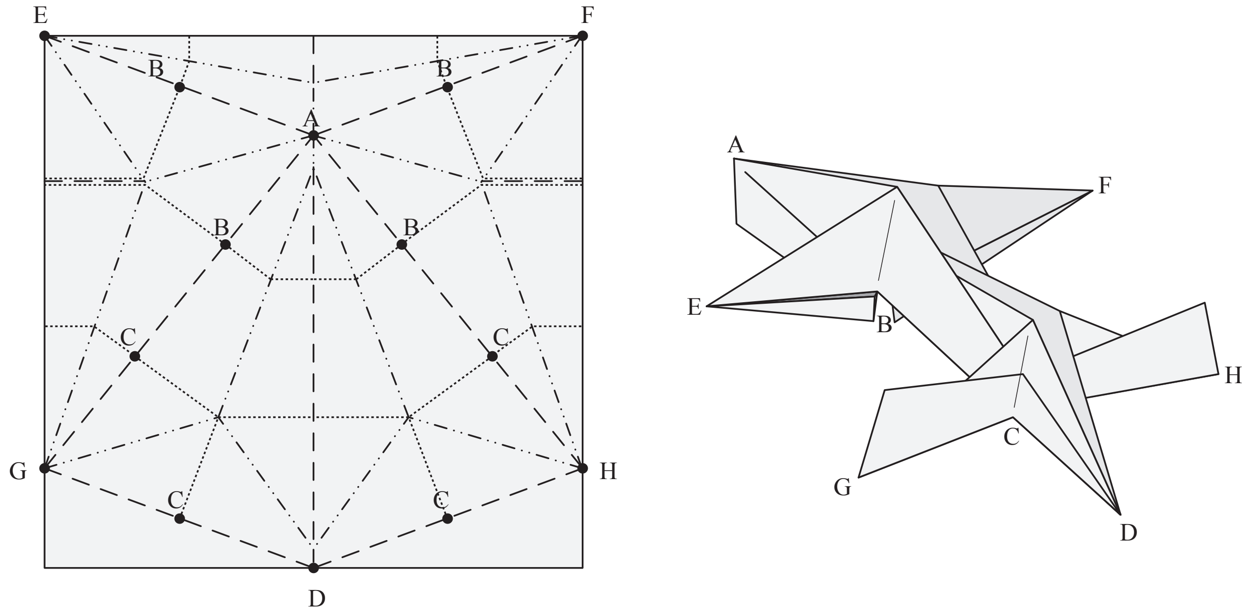


Figure 5.18. Full crease pattern and six-legged base.

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.

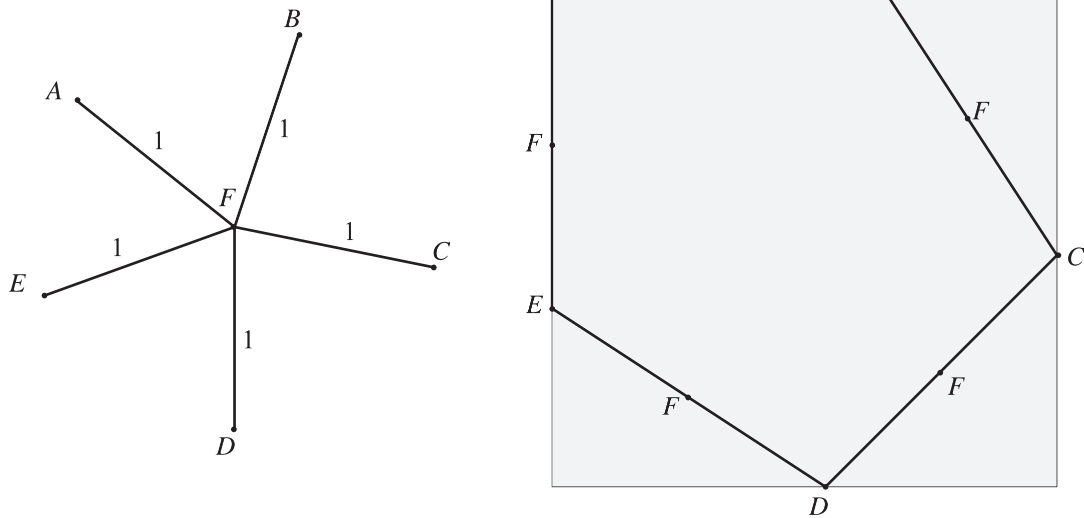


Figure 5.19. (Left) Tree for a base with 5 equal flaps. (Right) Pattern of terminal nodes and active paths corresponding to this tree.

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.

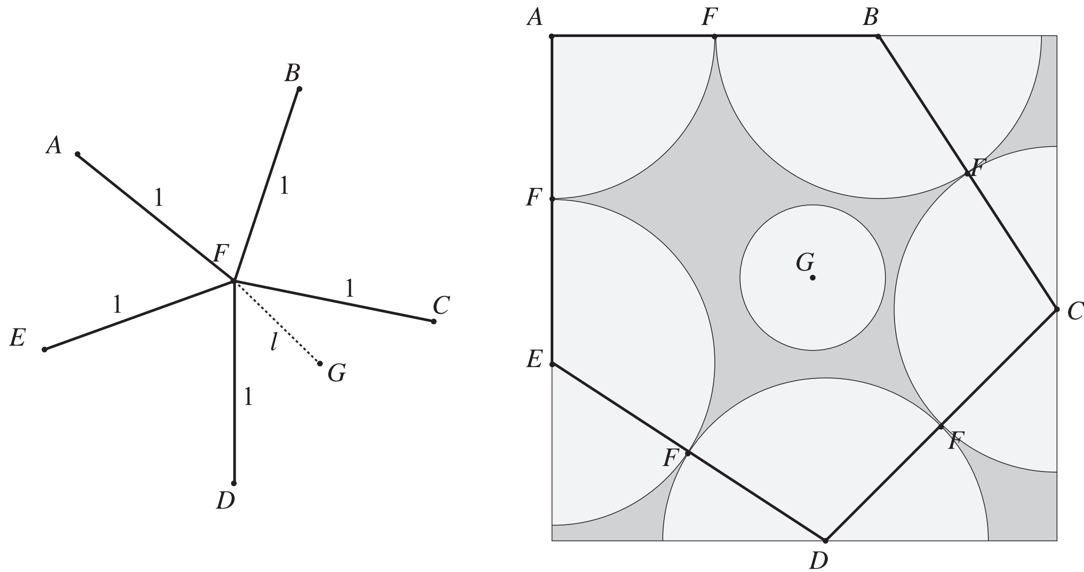


Figure 5.20. (Left) Tree with a new terminal node. (Right) One possible position for node G that satisfies the tree theorem.

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.

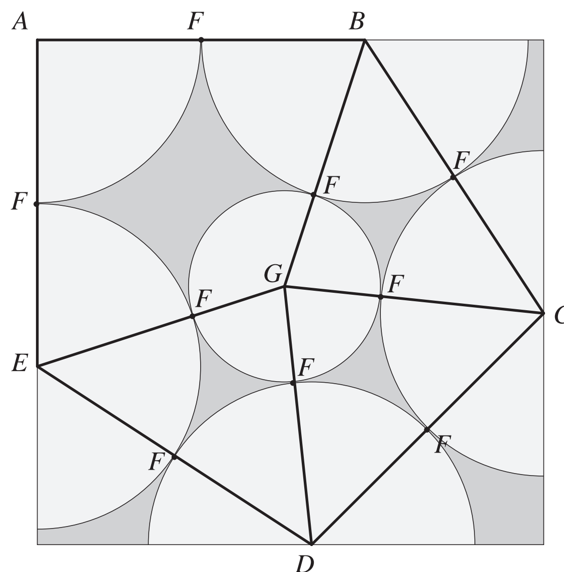


Figure 5.21. Terminal node pattern with the largest possible circle around node G.

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.

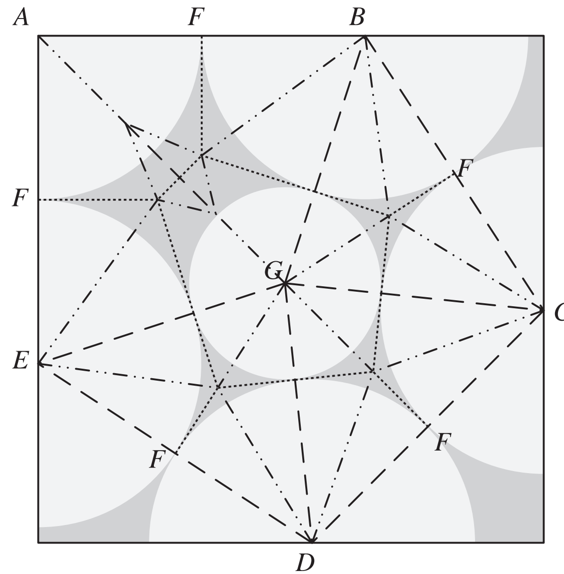


Figure 5.22. Crease pattern for the 5-pointed base.

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.

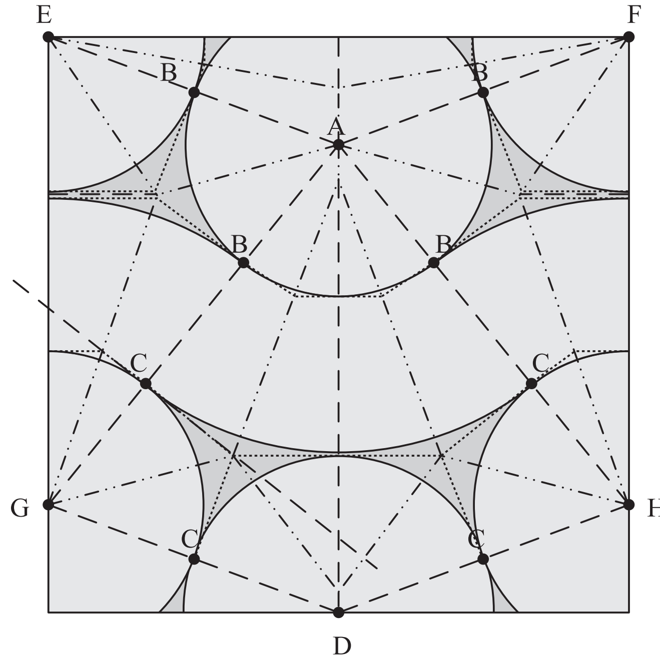


Figure 5.23. Circles and contours for the six-pointed lizard base.

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 $A_1, A_2, \ldots$ then we will label the inset vertices $A_1', A_2', \ldots$ as we did for the gusset quad construction. I will call the inset polygon a **reduced polygon** of the original polygon.

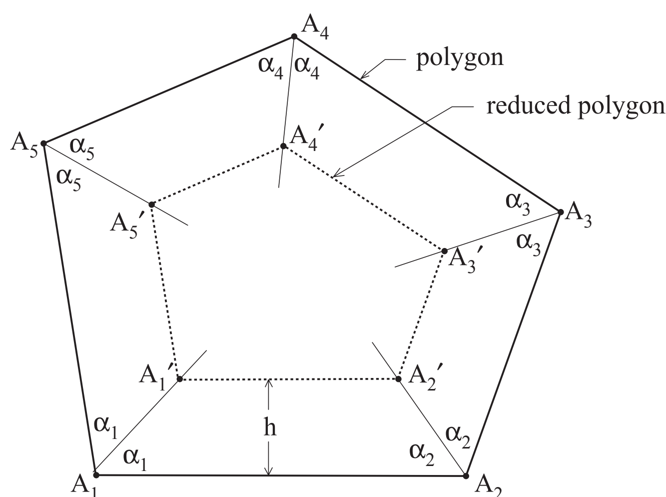


Figure 5.24. A reduced polygon is inset a distance h inside of an active polygon. The inset corners lie on the angle bisectors (dotted lines) emanating from each corner.

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.

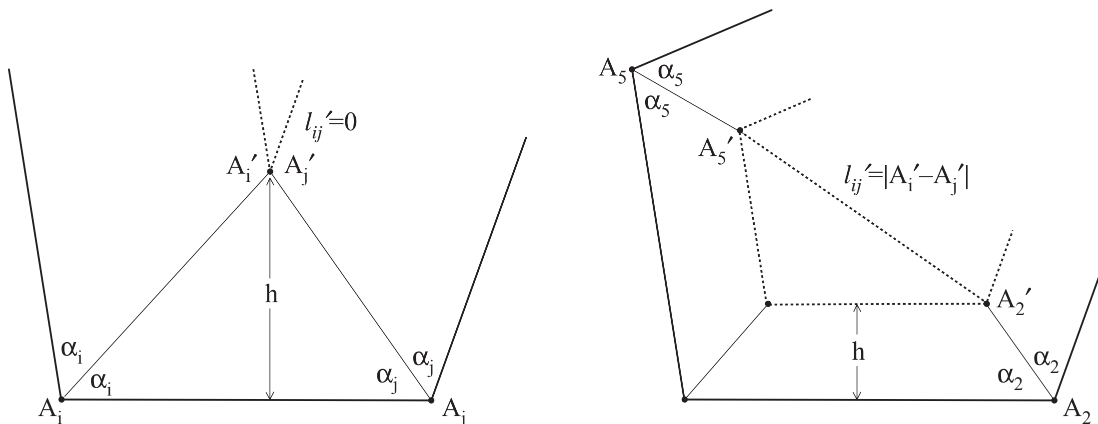


Figure 5.25. (Left) Two corners are inset to the same point, which is the intersection of the angle bisectors. (Right) Two nonadjacent corners inset to the point where the reduced path between the inset corners becomes active.

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.

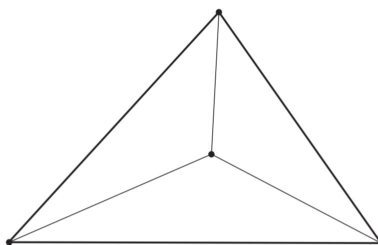


Figure 5.26. In a triangle, all three corners are inset to the same point, which is the intersection of the angle bisectors. This gives the rabbit ear molecule.

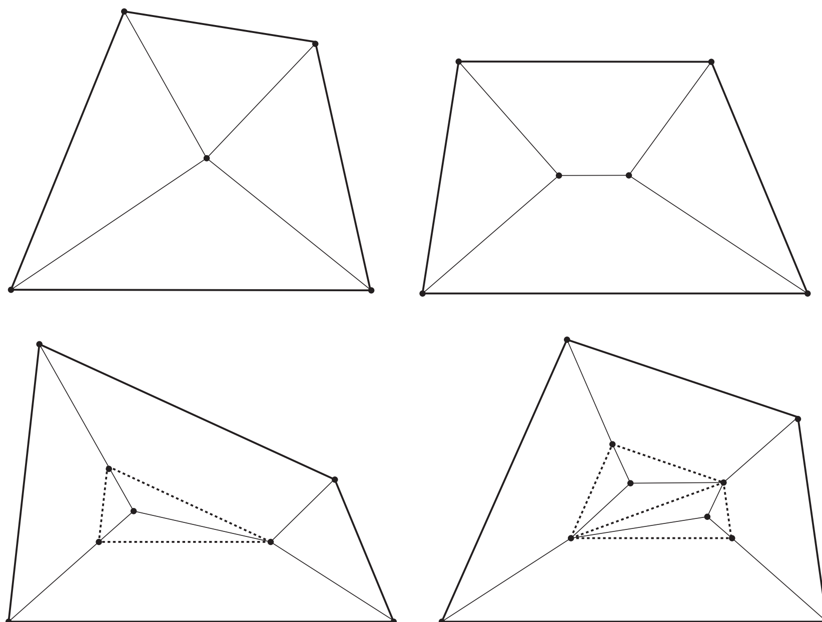


Figure 5.27. The four possible universal molecules for a quad. If all four corners are inset to the same point, the result is the Husimi molecule (top left). If adjacent pairs of corners are inset to two points, the Maekawa molecule is obtained (top right). If the inset polygon is a triangle, it is filled 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.

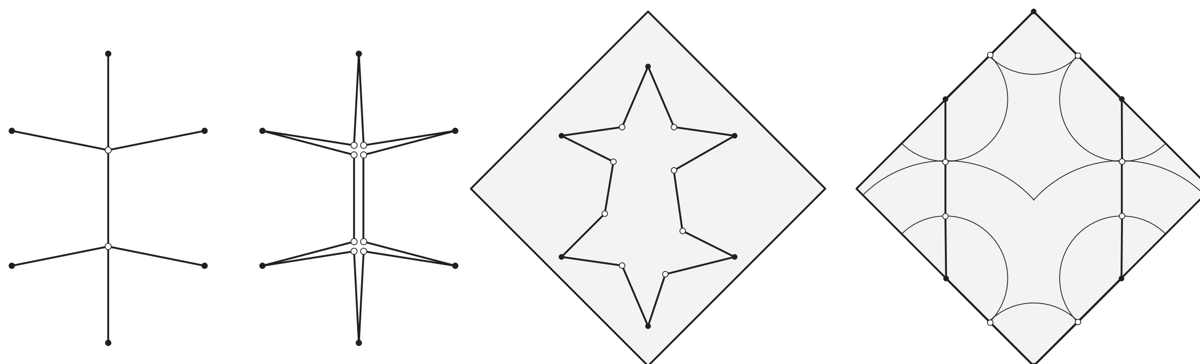


Figure 5.28. The string-of-beads design method. The tree graph is turned into a closed polygon, which is then "inflated" inside of a square with straight lines between the terminal nodes. The result is a large polygon inside the square that is collapsed into the base.

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.

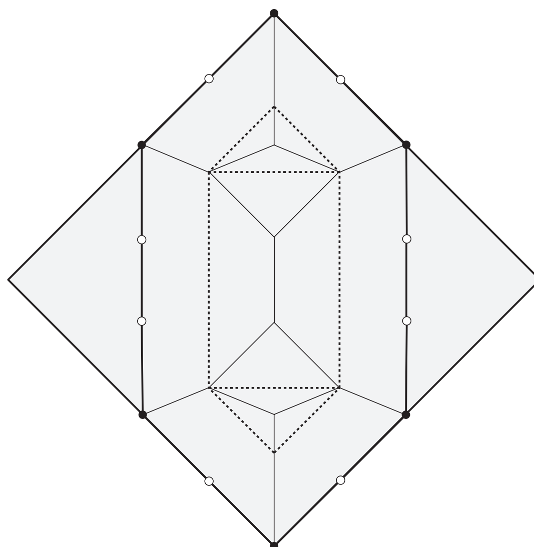


Figure 5.29. Construction of the universal molecule for the polygon shown in figure 28.

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.)

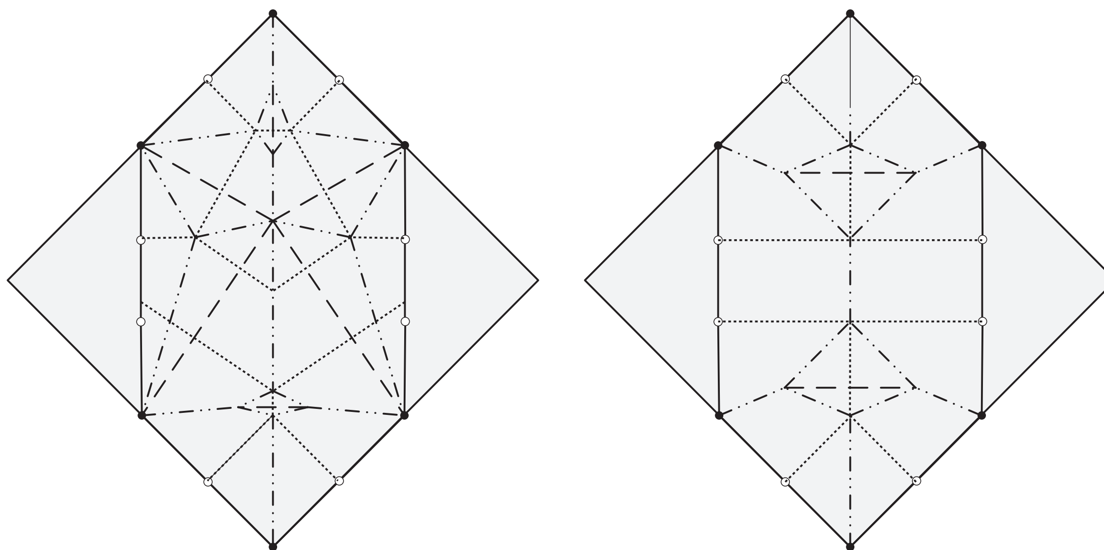


Figure 5.30. (Left) Crease pattern obtained by adding an additional node to the hexagonal active polygon. (Right) Crease pattern obtained by using the universal molecule.

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.

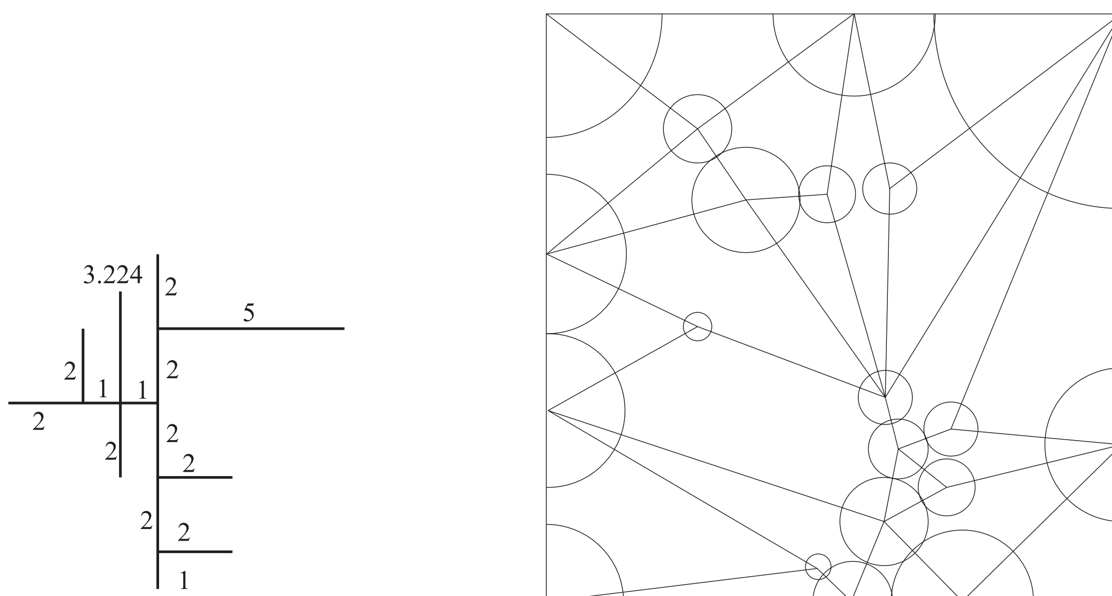


Figure 5.31. (Left) tree and (right) crease pattern from *Oru*

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.

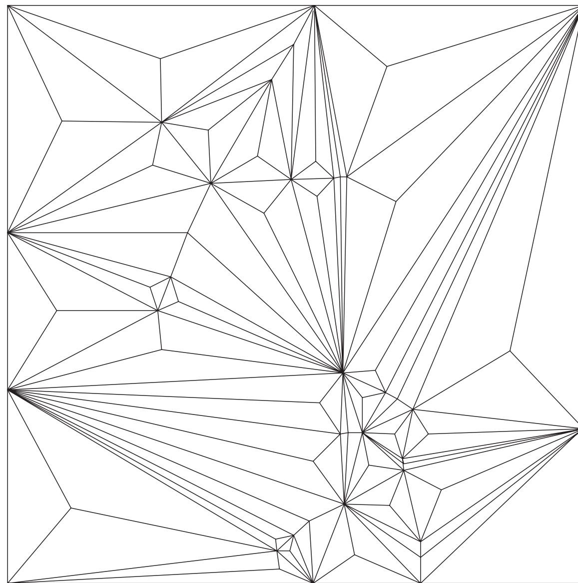
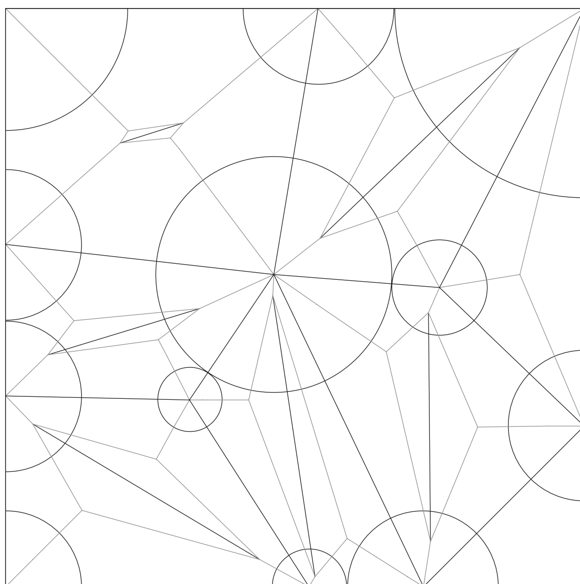


Figure 5.32. Crease pattern from Oru using arrowhead quad and rabbbit-ear molecules.

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.



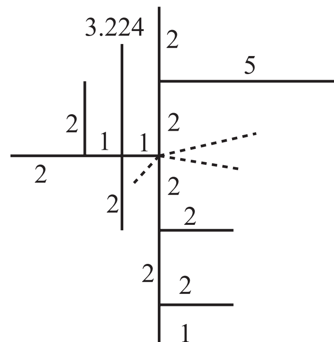


Figure 5.33. Tree and crease pattern for the *Oru* tree with three additional interior circles and filled with triangle and gusset quads.

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.

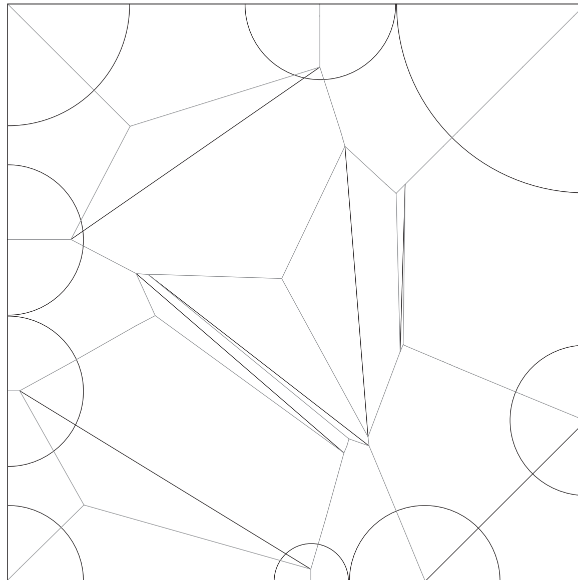


Figure 5.34. Universal molecule version of the *Oru* tree

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.

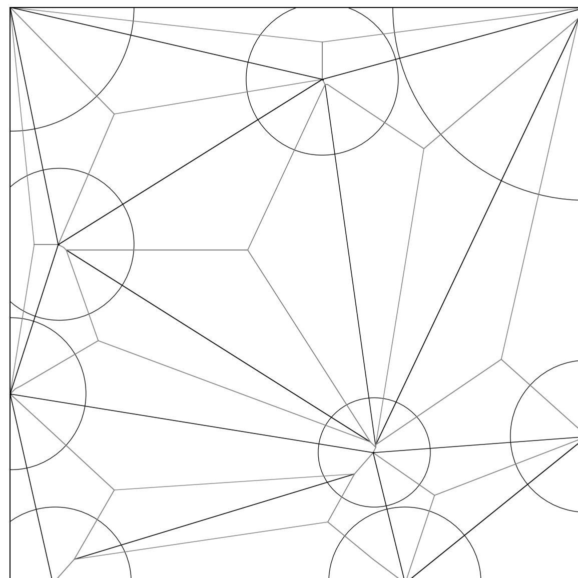


Figure 5.35. *Oru* tree crease pattern with middle points.

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

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

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

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 $\{\sigma_i\big|_{e_i \in E^s}, \mathbf{u}_j \in U^s\}$ 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) $m\left[\sum_{\substack{e_k \in p_{ij} \\ e_k \in E^s}}\left[(1+\sigma_k)l_k\right] + \sum_{\substack{e_k \in p_{ij} \\ e_k \notin E^s}}\left[(1+\sigma_k)l_k\right]\right] - \sqrt{\left(u_{i,x}-u_{j,x}\right)^2 + \left(u_{i,y}-u_{j,y}\right)^2} \le 0$ for all $p_{ij}, \in P^t$

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

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

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

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

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

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

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

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

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

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 $\delta\alpha \equiv \frac{180°}{N}$ 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

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

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:

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

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.

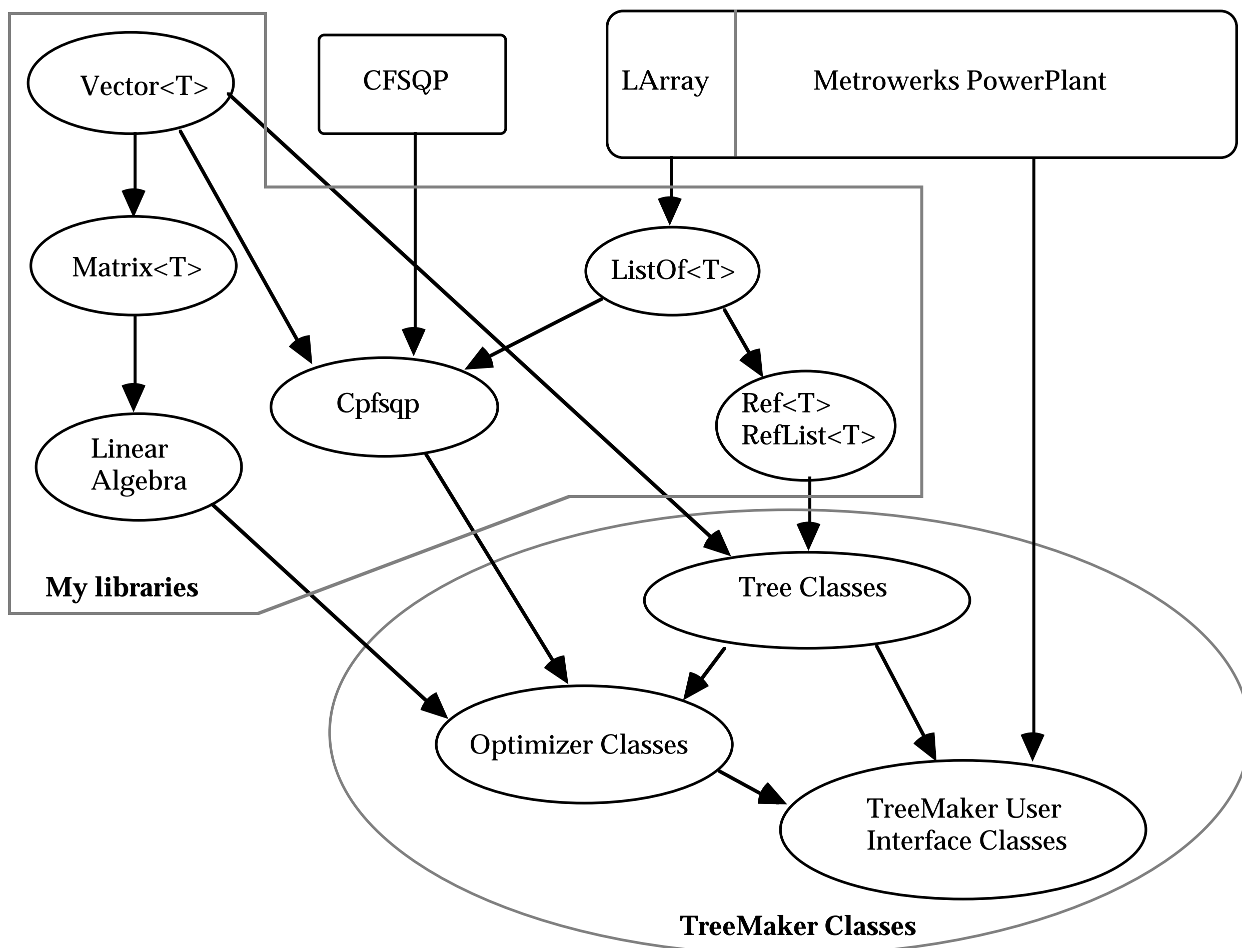
TreeMaker is strongly object-oriented. Early versions of TreeMaker were written in Object Pascal, which was a step up from its predecessors (I actually carried out some of the earliest work in Fortran), but beginning with version 3.0, I switched to C++. for its power and extensibility. Because of this, 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:

```
                    Conditions for External Use


   1. The CFSQP routines may not be distributed to third parties.
      Interested  parties  should contact  the authors directly.
   2. If modifications  are  performed  on  the  routines, these
      modifications  shall be  communicated to the authors.  The
      modified  routines  will  remain the sole  property of the
      authors.
   3. 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.
   4. The CFSQP  routines  may  only  be  used  for research and
      development, unless it has been agreed  otherwise with the
      authors in writing.
```


```
Copyright (c) 1993-1997 by Craig T. Lawrence, Jian L. Zhou, and
                        Andre L. Tits
All rights Reserved.
```



```
   Enquiries should be directed to

           Prof. André 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*.)

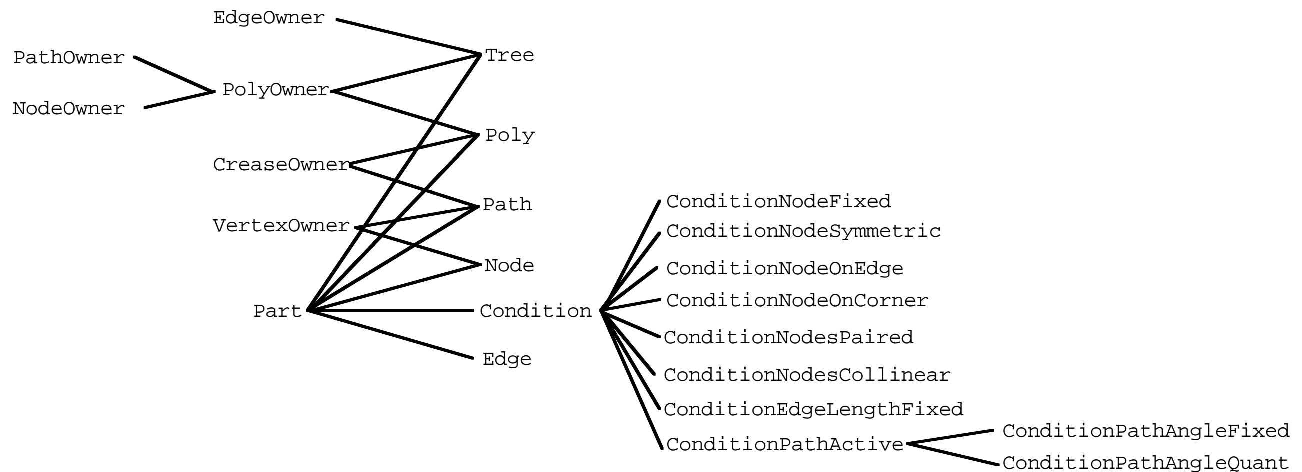
TreeMaker Classes

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.

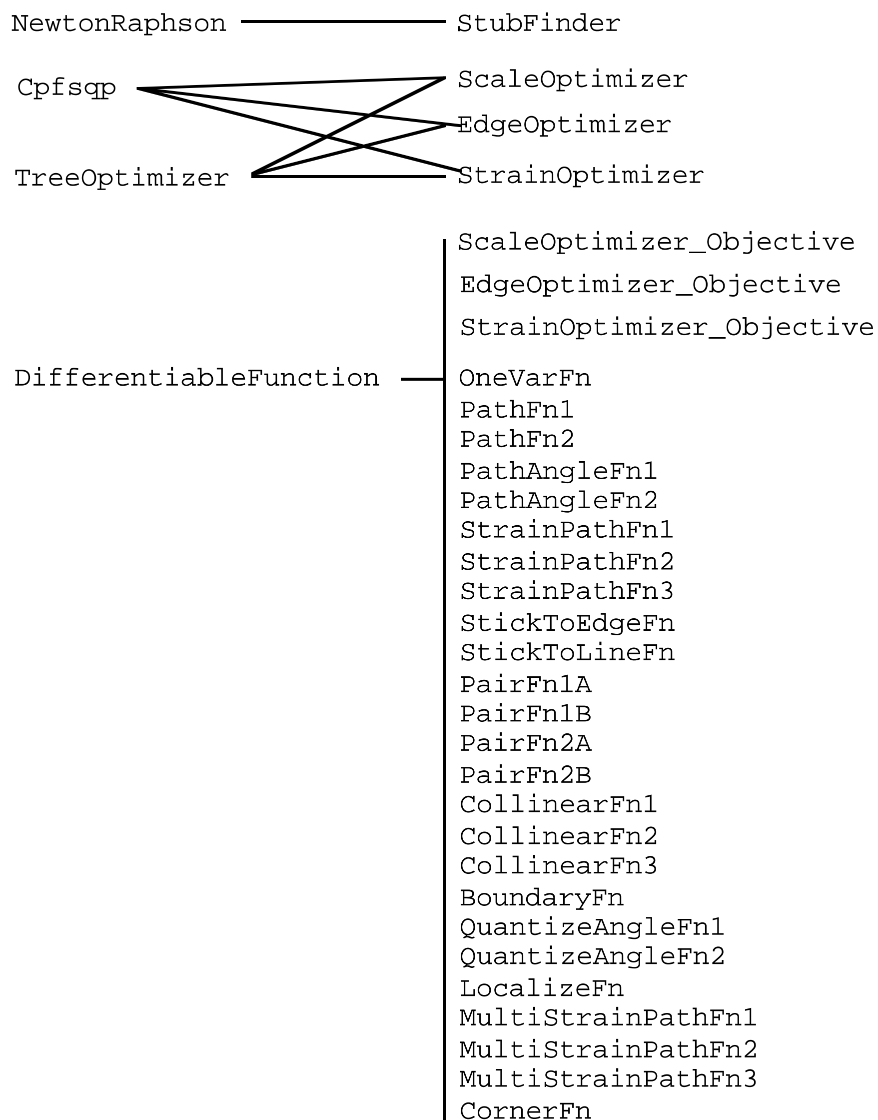
`StrainOptimizer` is used in heavily constrained situations; it finds a configuration that minimizes the strain on the edges.

`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 `Conditions` for `Parts` and lists of `Parts`.

PartEditorMixin

LModalDialog — ToggleSizeDialog

TreeEditor
NodeEditor
EdgeEditor
PathEditor
PolyEditor
CreaseEditor
VertexEditor
ConditionEditor

TreeViewEditor
SelectPartEditor
SizeEditor
SplitEdgeEditor
StubEditor
NodeFixedEditor
EdgeLengthEditor
QuantizationEditor

LView

TreeView
ConditionEditorView
ConditionView

ConditionNodeFixedView
ConditionNodeOnCornerView
ConditionNodeOnEdgeView
ConditionNodesCollinearView
ConditionNodesPairedView
ConditionNodeSymmetricView
ConditionEdgeLengthFixedView
ConditionPathActiveView
ConditionPathAngleFixedView
ConditionPathAngleQuantView

ConditionRegistry
ConditionTool<P,C>

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.

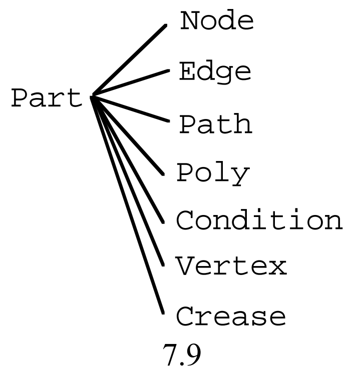
Part-ness

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 Parts with names like mNodes, mEdges, mPaths, etc. This provides flat access to the hierarchical data structure; it simplifies searching over the Parts and lets us assign the Part index according to its position within the master list.

Ownership

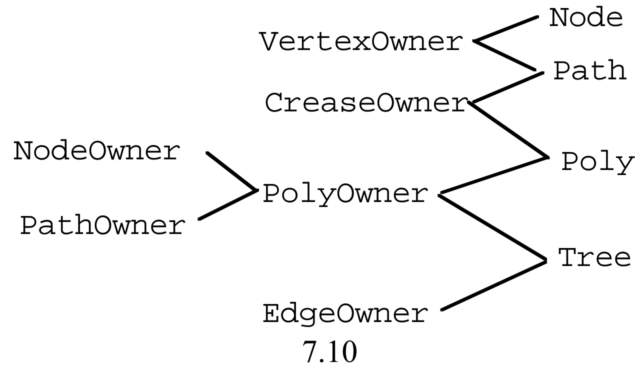
The data structures in *TreeMaker* are tied together by various relations. One important set of relations is the concept of "ownership." Every object is owned by another object. The chain of ownership is a simple rooted tree, that is, every object has a unique owner, which in turn has a unique owner, 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 Polys, which are called subPolys.

2. Both Poly and Tree own Polys and hence (through inheritance) also own Nodes and Paths. A Poly owns all Nodes that are fully enclosed by the Poly and owns all Paths that connect those Nodes.

Both Poly and Path can own Creases. However, only the Tree can own Edges.

3. Both Nodes and Paths 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 Paths own Creases: 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 Creases that connect to its reduced polygons and/or any ridge Crease. A Poly also owns Creases that run between internal vertices. A Poly also owns its sub-polys.

A Tree owns its Nodes, Edges, Paths, and Polys.

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.

Conditions, 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), Conditions are created on the fly through the ConditionRegistry object.

References: Ref<T> and RefList<T>

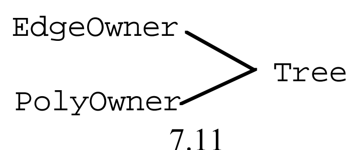
Because of the complexity of the data structure — most parts maintain several lists of related parts (e.g., a Node has a list of all Paths that begin or end on it) — when we create or delete parts, there are a lot of references to clean up, particularly when objects are deleted. Earlier versions of *TreeMaker* 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.

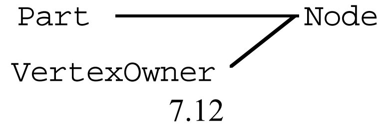
Object	*References*	*What*
`Tree`	**`mOwnedNodes`**	All nodes in the primary tree (i.e., not including nodes in the interior of a poly, which are owned by the poly itself)
	`mOwnedEdges`	All edges in the primary tree
	`mOwnedPaths`	All paths in the primary tree (i.e., not including reduced paths inside 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.



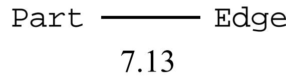
7.12

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.

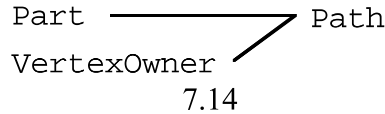
Object	*References*	*What*
`Edge`	`mNodes`	2 nodes at each end of this edge
	`mEdgeOwner`	Tree that owns this edge

`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.



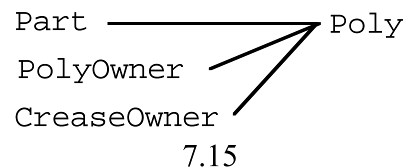
7.14

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

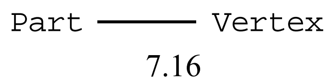
The references contained by a `Poly` are summarized below. Names in bold refer to inherited member variables.

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

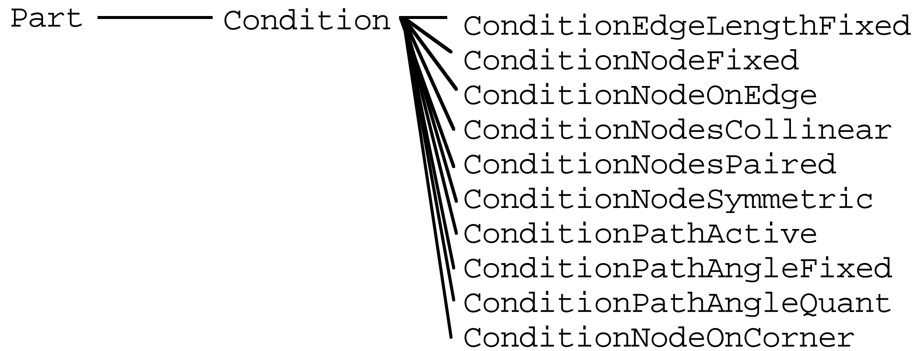
A Crease is a line in the crease pattern. There are three types of Creases: ValleyCrease, MountainCrease, and TristateCrease. The Crease inherits only from Part

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