Depth first search algorithm for box pleated origami design

Benjamin Wettle

Abstract—This algorithm optimized the placement of nodes in a box pleated origami design. The algorithm searches with a branch and bound depth first search with complete branching. The placement of the nodes is optimized, and a box pleated crease pattern is generated. This algorithm is implemented in a Java application, called Blue Phoenix Folding.

Keywords—Box pleating, Branch and bound, Crease pattern, Depth First Search, Origami .

I. Introduction

ORIGAMI is the ancient art of folding paper to represent various forms. Traditional artists developed step by step instructions to fold the base, and then added details. These designs were found by experimentation, and modification of existing designs. More modern artists have developed algorithms to generalize this process to work for arbitrary bases. This application designs bases in a box pleated style. This style gives great versatility and fold-ability, with minimal wasted space.

A. Definitions

In order to design an origami base, we first have to define a notation to describe it. The paper used is assumed to be a rectangle of negligible thickness. Though no such perfect paper exists in reality, origami paper is thin enough to ignore for most designs. Most origami artists use a square paper for designs, but the definitions and algorithm are the same for the more general case of a rectangle. The definitions used here are modified slightly from those used for box pleating by Robert Lang [1].

For a box pleated design, the paper is folded into a grid pattern, with equally spaced lines running vertically and horizontally. In a final design, some of these lines will not be folded, as they can clutter the artistic style of the design. They are useful in the description and optimization, as they define the placement and areas of the nodes.

For this paper, a simple design will be used to demonstrate the algorithm. The final design, a Blue Phoenix, can be found at the end. To describe the design, the user starts by drawing a tree plan. This shows the nodes, their sizes, and their connections. The nodes consist of two types:

a) : A leaf node is the simplest element of a base. This represents a loose flap of paper, like the example shown in Figure 1. These flaps can be easily folded to represent objects like arms, fingers, wings or even ears. This versatility allows wildly different designs to be made from the same simple structure of paper.

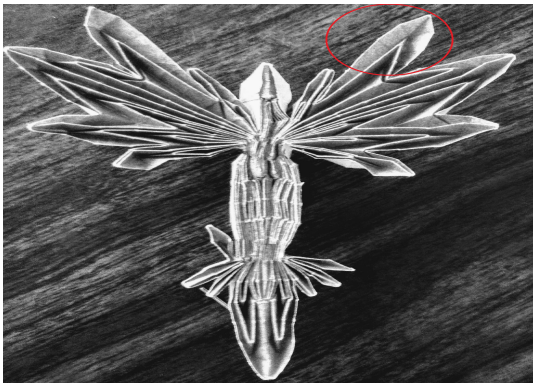

Fig. 1. Leaf Node example

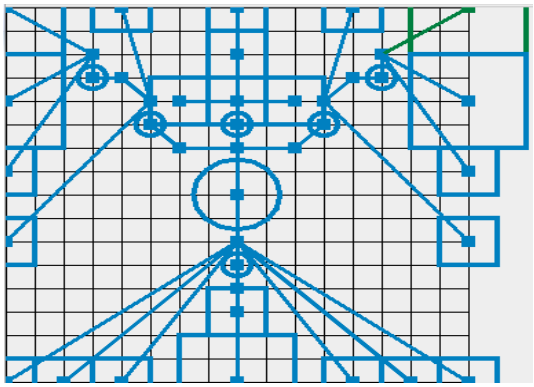

Fig. 2. Plan view of Blue Phoenix, with leaf node selected

Figure 2 shows the same design in a plan view. The boxed area shows the paper used to make each node. The minimum paper required (in a box pleated design) is a square centered on the tip of the leaf node. The square must be twice the size of the flap, on each side. The square can be partially off the paper, as in figure 2, but the center must be on the paper to be folded. This area cannot overlap with the area used for other nodes, or the paper would not fold into a loose flap. The green square in the top right shows the same leaf node as in figure 1. This node is size 2, so it must have a 4 by 4 square surrounding the center.

b) : The second type of node is a river node. Figure 3 shows a river node, separating two branches of the tree. These branches can be a set of other nodes, including other sub-branches. A river node separates the two branches by it's size, forcing them apart. This can be used to represent an arm, torso, or part of a wing, for example. Figure 4 shows the position of the same river node. In the tree plan, the river

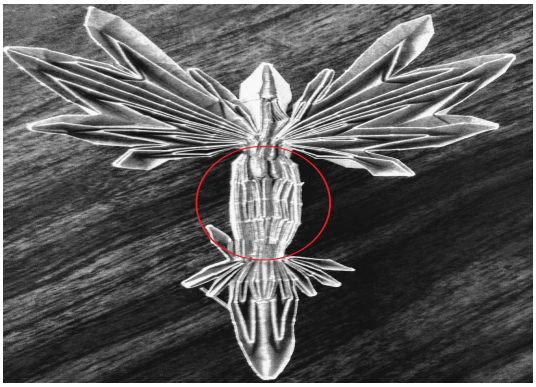

Fig. 3. River node example

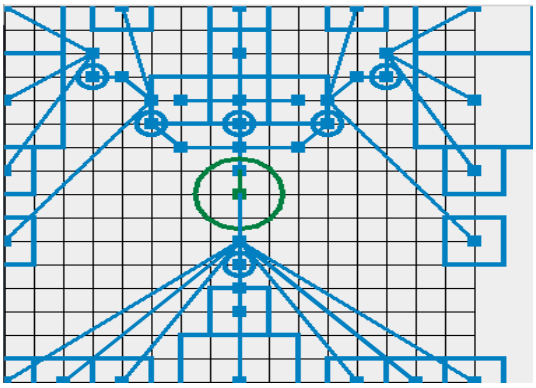

Fig. 4. Plan View of Blue Phoenix, with river node selected.

nodes are represented as circles. The locations of river nodes are not important in the tree plan, as they will be positioned during crease generation.

II. Design

Figure 5 shows the finished tree plan of the blue phoenix. Each line indicates a connection, and each filled square is a node. The outlined squares show the positions and sizes of the leaf nodes. The circles show the positions and sizes of the river nodes. If this design overlaps any two nodes, it will not fold into the proper base. To determine if two nodes overlap, we must first find the minimum distance. This is the distance that must be traveled to go from one node to the other on the tree plan.

The minimum distance does not depend on the locations of the nodes on the paper, just the location on the tree plan. For example, consider the two Nodes marked A and B the upper right corner of Figure 5. . To travel from A to B requires a distance of 4- 2 to move to the river node(i), 1 to move across the river(ii) and 1 to move to B. From this, a simple rule can be used to find if two nodes overlap. Either the difference in X or Y coordinates on the paper must be equal (or greater) than this minimum distance. This is a modified version of the rules from Robert Lang's work with circle packing[2]. These modifications are necessary to account for the differing geometry of squares.

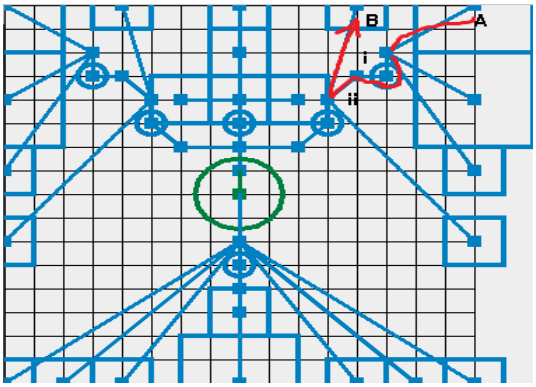

Fig. 5. Plan View of Blue Phoenix, with example distance

III. Approach

The goal of the optimization algorithm is to place each leaf nodes in the best locations. These locations should fix into the smallest square (or rectangle), to minimize wasted space. By minimizing wasted space, the final model will be as large as possible for the given paper size.

The GUI allows the user to enter in a tree plan, like the one in Figure [2], and any constraints needed. The optimization algorithm needs (at it's simplest) two pieces of information. The first is a list of leaf nodes. River node's locations are determined in the crease generation step, so they will not be optimized. The second piece of information is the minimum distances from each leaf node to the others. This is found beforehand, using a simple depth first search across the tree plan.

A. Symmetry

Constraints can also be placed on nodes positions. If two nodes are paired about one axis, then one is added to a list of dependant nodes. When it's partner is optimized, the dependant node is also added, in a mirrored position. Nodes can also be set on an edge, or on the line of symmetry. This line is set as either the X axis or the Y axis.

For this design, most of the nodes are paired with a partner. This enforces symmetry about the y axis. The other nodes are set to be on the y axis. This ensures that the design is symmetrical and will properly represent the Blue Phoenix.

B. Optimization Parameters

Additionally, there are several pieces of information that modify the optimization algorithm. A buffer can be added the minimum distance for all nodes, to force the algorithm to search a larger area. This gives many more candidates, most with larger gaps. This slows the speed of the optimization significantly, so it is set to 0 by default. For some designs, with many conditions, this may be necessary.

If the design needs to be optimized to a rectangle of certain proportions, then the algorithm can be set to weight different axis accordingly. The maximum number of nodes searched can also be changed.

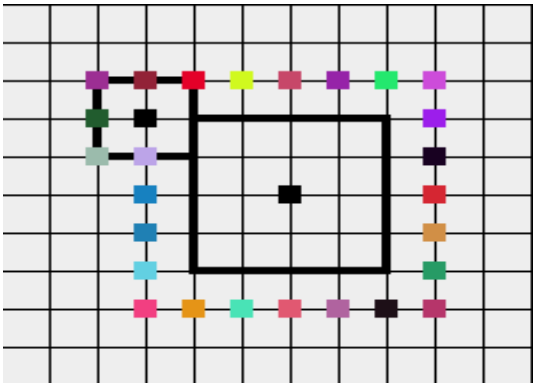

Fig. 6. Visualization of candidate placements

C. Exhaustive search

Then comes the core question of the optimization- where is the best place for these nodes to go? The first node is easy to place, since all distances are relative. It can be placed at any point, such as (0,0). Since optimizing a single node is trivial, we must also account for the rest of the nodes.

The second node could go many places, as long as it meets the minimum required distance. At this point, we don't know were the best spot is, so we will place it in several spots, for different potential designs. A square with sides of $2 * (minimum\ distance)$ is searched for potential locations. This square is searched around each of the nodes already in the design.

Of these spaces, those overlapping the other nodes can be automatically eliminated. this gives a set of placements to check. Though the number may vary from node to node, it increases roughly linearly with the number of nodes already added. Figure 6 shows the candidate locations for the third node, marked as colored squares. The third node has a minimum distance of 1 and 3 from the nodes on the right and left respectively. Note- The size of the black squares represent the size of the nodes, not the minimum distance.

For each of the candidate placements left, the algorithm checks all possible placements for the next node, eliminates those with overlap, and repeats. Once all the nodes are added, the best design is selected, based on the minimum size. This method of search is quite slow, as a huge number of candidates must be searched.

For a design with n (independent) leaf nodes, each new node added makes on the order of n new designs. Since there are n nodes to add, the full number of nodes to check is on the order of $(n)^n$.This is too slow to handle large numbers of nodes. Fortunately, it can be improved greatly.

IV. Improvements

Instead of searching exhaustively, the algorithm searches with a branch and bound depth first search with complete branching. This lets the candidates be searched intelligently, at the cost of memory [3].

By tracking the best final design found, and it size, many incomplete candidates can be pruned from the search. Only if the size is smaller should the candidate be considered any further. Candidates of equal size are stored, and can be checked for minor improvements at the end.

Additionally, the candidates can be scored as they are built. The score is simply the sum of all the gaps in the design. If a candidate already has more gaps than the best, it can be pruned from the search. Designs can also be eliminated by size and score before being checked for overlap, to reduce the wasted time. This sort of pruning works best when the candidates are examined from best to worst, which is exactly what the sorting step does.

All of the candidates are sorted by size and score as they are found. Thus, the best scoring node will be examined before any others. This means that many nodes will only be examined after the best design is found. To control this, a limit is added. After any design is found, only the next 50,000 designs will be examined. This reduces the time wasted by checking worse solutions. This count resets after a better node is found, so that a unlucky search ordering does not impact the final size of the design.

V. Crease Generation

Once the best design is found, the crease pattern can be generated. If the design has no overlaps, this pattern will fold into the base. First, a grid of mountain and valley creases is made. The valley creases follow the grid lines of the square. In between these run the mountain folds, to form the box pleated structure. In practice, some of these line are unnecessary. For display purposes, it is useful to have all of them for guidance on the diagonal creases.

For these creases, the areas allocated for each node is found. Leaf nodes are found by forming squares, as in the definitions section. River nodes are made by adding borders to the nodes they enclose. A gap of a constant width must surround all the nodes inside the river. The size of this gap is the same size as the river node. These rules are based on Robert Lang's rule for box pleating creases [1].

Once the areas are complete, there may be some unused space left over. This is saved in a excess node, which is not part of the design. For all the areas, the angle bisectors are found. The bisectors are moved sequentially inwards, until they meet another bisector or run out of space in their node's areas. Figure 7 shows the final crease pattern for the blue phoenix design. Each node is colored a random color, so that the areas and creases are clearer. In the application, only the diagonal creases would be shown, in thin black lines. There is no guarantee that this crease pattern is fold-able in separate steps. Instead, it is designed to be pre-folded along the indicated creases, and collapsed in one step. To check if the crease pattern is flat fold-able would be NP hard [4]. However,the folding of the horizontal and vertical lines is left to the user. In each case, if these lines are folded of not is determined by Kawasaki's Theorem [5]. In practice, all of the grid lines will be folded as reference points, and the choice of folding of including them in the finished design is determined naturally during the collapsing stage.

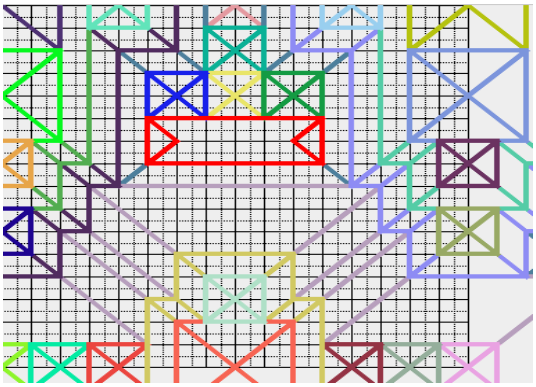

Fig. 7. Crease Pattern

Fig. 8. Blue phoenix design

VI. Evaluation

There is no such thing as a typical origami design. Each design is unique, and the exact optimization is slightly different for each. To account for this, ten sample designs were optimized and timed. Though this does not account for every design, it provides a reasonable estimate of time. These optimizations were run on a ASUStek computer with Intel(R) Core™ i7-7700HQ CPU @ 2.8 GHz. The sample designs can be found in the examples folder with the Java application. The Java application is open-source and can be found at https://github.com/bmwettle/Blue-Phoenix-Folding .

Number	Leaf Nodes	Total Nodes	Time(ms)	Size
1	5	8	5	5
2	10	19	1,386	8
3	19	46	914	21
4	22	52	1,055	23
5	26	59	1,729	23
6	29	64	2,482	21
7	36	86	13,713	32
8	44	100	29,537	35
9	53	116	71,480	37
10	64	132	100,909	41

When these designs were tested in Robert Lang's Tree-maker[6], the optimization was much faster. Though exact timing was not collected, all of the designs tested took less than 10 seconds to optimize. Though this is a drawback, it's practice implications are small.

It is unlikely that a design will have many more nodes than those tested. Many designs have symmetry, which roughly halves the effective number of nodes. In addition, folding more complex designs takes much more time than optimizing. Even simple designs can take hours to pre-fold and fold. This time overshadows the optimization times for large designs.

VII. Related Work

To the best of our knowledge, this is the first publicly available implementation of a optimization and crease generation algorithm using box pleating. The most similar work is Robert Lang's Tree-maker[6]. That program generates optimized designs and crease patterns for circle packed designs. It is significantly faster, as discussed in the evaluation section. However, it has two main flaws, both stemming from circle packing itself.

The first is the locations of nodes and points. Though they are optimized, they are often in arbitrary locations, which require additional measurement and folds to locate. In box pleating, all nodes line on an easily folded grid. All other creases are folded at $45\,\mathrm{deg}$ angles from this grid. This simplifies the folding process significantly. The second problem with circle packing also relates to crease patterns. For many designs, the crease pattern will not generate without modification. This ends up wasting time and adding extraneous nodes to the design. The simpler rules of box pleating allow for automatic filling of unused spaces, and will generate crease patterns for any given design. sting time and adding extraneous nodes to the design.

VIII. Conclusion

We present a optimization and crease generation algorithm for box pleated origami designs. The candidates are pre-sorted, and pruned based off the current best. This allows for fairly rapid optimization of origami designs. The crease pattern is then generated, and can be folded into the base. Figure 8 shows the final Blue Phoenix design, with added details.

ACKNOWLEDGMENT

I would like to thank Michelle Cheatham, for help in making the GUI, and in reviewing this paper.

REFERENCES

[1] R. J. Lang, Origami design secrets: mathematical methods for an ancient art. Boca Raton, New York: CRC Press, 2017.

[2] R. J. Lang, "A computational algorithm for origami design," Proceedings of the twelfth annual symposium on Computational geometry - SCG '96, 1996.

[3] D. R. Morrison, S. H. Jacobson, J. J. Sauppe, and E. C. Sewell, "Branch-and-bound algorithms: A survey of recent advances in searching, branching, and pruning," Discrete Optimization, vol. 19, pp. 79–102, 2016.

[4] H. A. Akitaya, K. C. Cheung, E. D. Demaine, T. Horiyama, T. C. Hull, J. S. Ku, T. Tachi, and R. Uehara, "Box Pleating is Hard," Lecture Notes in Computer Science Discrete and Computational Geometry and Graphs, pp. 167–179, 2016.

[5] Barile, Margherita, and Margherita Barile. "Kawasaki's Theorem." From MathWorld–A Wolfram Web Resource, created by Eric W. Weisstein. http://mathworld.wolfram.com/KawasakisTheorem.html (accessed Aug 30, 2009).

[6] R. Lang, "TreeMaker," 19-Sep-2015. [Online]. Available: https://langorigami.com/article/treemaker/. [Accessed: 31-Aug-2020].

Benjamin Wettle Mr. Wettle is a senior in the Electrical Engineering Department at Wright State University at 3640 Colonel Glenn Hwy., Dayton, OH 45435 USA. He is interested in researching signal processing as well as computational origami. He has been folding original origami designs since he was 12. Email- bmwettle@gmail.com