On Unfolding Lattice Polygons/Trees and Diameter-4 Trees*

Sheung-Hung Poon

Department of Mathematics and Computer Science, TU Eindhoven,
5600 MB, Eindhoven, the Netherlands.
spoon@win.tue.nl

Abstract. We consider the problems of straightening polygonal trees and convexifying polygons by continuous motions such that rigid edges can rotate around vertex joints and no edge crossings are allowed. A tree can be *straightened* if all its edges can be aligned along a common straight line such that each edge points "away" from a designated leaf node. A polygon can be *convexified* if it can be reconfigured to a convex polygon. A *lattice tree* (resp. *polygon*) is a tree (resp. polygon) containing only edges from a square or cubic lattice. We first show that a 2D lattice chain or a 3D lattice tree can be straightened efficiently in $O(n)$ moves and time, where n is the number of tree edges. We then show that a 2D lattice tree can be straightened efficiently in $O(n^2)$ moves and time. Furthermore, we prove that a 2D lattice polygon or a 3D lattice polygon with simple shadow can be convexified efficiently in $O(n^2)$ moves and time. Finally, we show that two special classes of diameter-4 trees in two dimensions can always be straightened.

Keywords. Comput. geom., unfolding, straightening, convexifying.

1 Introduction

Graph reconfiguration problems have wide applications in contexts including robotics, molecular conformation, animation, wire bending, rigidity and knot theory. The motivation for reconfiguration problems of lattice graphs arises in applications in molecular biology and robotics. For instance, the bonding-lengths in molecules are often similar [6, 11, 12], as are the segments of robot arms.

A *unit tree* (resp. *unit polygon*) is a tree (resp. polygon) containing only edges of unit length. An *orthogonal tree* (resp. *orthogonal polygon*) is a tree (resp. polygon) containing only edges parallel to coordinate-axes. A *lattice tree* (resp. *lattice polygon*) is a tree (resp. polygon) containing only edges from a square or cubic lattice. Note that a lattice tree or polygon is basically a unit orthogonal tree or polygon. A graph is *simple* if non-adjacent edges do not intersect. We consider the problem about the reconfiguration of a simple chain, polygon, or

* This research was supported by the Netherlands' Organisation for Scientific Research (NWO) under project no. 612.065.307.

tree through a series of continuous motions such that the lengths of all graph edges are preserved and no edge crossings are allowed. A tree can be *straightened* or *flattened* if all its edges can be aligned along a common straight line such that each edge points "away" from a designated leaf node. In particular, a chain can be straightened if it can be stretched out to lie on a straight line. A polygon can be *convexfied* if it can be reconfigured to a convex polygon. We say a chain or tree is *locked* if it cannot be straightened. We say a polygon is *locked* if it cannot be convexified. We consider one *move* in the reconfiguation as a continuous monotonic change for the joint angle at some vertex.

In four dimensions or higher, a polygonal tree can always be straightened, and a polygon can always be convexified [7]. In two dimensions, a polygonal chain can always be straightened and a polygon can always be convexified [9, 14, 5]. However, there are some trees in two dimensions that can lock [3, 8, 13]. In three dimensions, even a 5-chain can lock [4]. Alt et al. [2] showed that deciding the reconfigurability for trees in two dimensions and for chains in three dimensions is PSPACE-complete. However the problem of deciding straightenability for trees in two dimensions and for chains in three dimensions remains open. Due to the complexity of the problems in two and three dimensions, some special classes of trees and polygons have been considered. Poon [13] showed that a unit tree of diameter 4 in two dimensions can always be straightened. In their paper, they posed a challenging open question whether a unit tree in either two or three dimensions can always be straightened.

The rest of this paper is organized as follows. We define some technical terms used in our paper in Section 2. We present efficient algorithms to straighten lattice chains and trees and to convexify lattice polygons in both two and three dimensions, respectively, in Sections from 3 to 6. In Section 7, we show that two special classes of diameter-4 trees in two dimensions can always be straightened. Finally, we conclude with some conjectures in Section 8.

2 Definitions

Let P be unit tree or polygon in two or three dimensions. Define a small value $\epsilon = \frac{1}{100n}$, where n is the number of edges in P. We call point q is *convergent* to p if q is within distance ϵ from p. A unit edge is called *convergent* to a lattice edge if any point on the edge is within distance ϵ from a particular lattice edge. Such a unit edge is called a *near-lattice edge*, and the particular lattice edge is called its *core edge*. A em core vertex is a vertex of some core edge. A *near-lattice tree* (resp. *near-lattice polygon*) is a tree (resp. polygon) that contains only near-lattice edges. Suppose P is a near-lattice tree or polygon. The *core* of P, denoted by $K(P)$, is the union of core edges for all edges in P. A *spring* in P is the set of edges in P converging to a common lattice edge. A spring with only one edge is called a *singleton*. A *leaf spring* is a spring with its core edge possessing a leaf vertex in the core of P.

A near-lattice tree is called *folded* if its core contains a single lattice edge. A near-lattice polygon is called *nearly folded* if its core is a lattice rectangle

of unit width. Remark that the definition of a *nearly-folded* polygon is due to our unfolding algorithm, in which we convert a given lattice polygon to such a polygon, which can then be convexified straightforwardly.

3 2D lattice chains and 3D lattice trees

2D lattice chains. Given a simple 2D lattice chain $P = p_0 p_1 \dots p_n$. Starting from an end edge, say $p_0 p_1$, we fold up the whole chain, edge by edge. At step i, a spring S_i containing a zig-zag path from p_0 to p_{i+1} is formed such that $p_i p_{i+1}$ is a lattice edge staying at its original position, and $\angle p_j = \epsilon^2$ for $1 \leq j \leq i$. Vertices $p_{i-1}, p_{i-3}, \dots$ all lie in a lattice cell, say σ. Step $i+1$ of the algorithm tries to combine the spring S_i and the edge $e = p_{i+1} p_{i+2}$ to form a new spring S_{i+1}. We need to consider several cases depending on the position of e. If p_{i+1} is non-straight and e does not lie on σ, we rotate S_i around vertex p_{i+1}, away from σ, until $p_i p_{i+1}$ makes an angle ϵ^2 with e, and we are done. See Figure 1(a) for illustration. Otherwise, we first rotate e around p_{i+2} to the side containing σ until

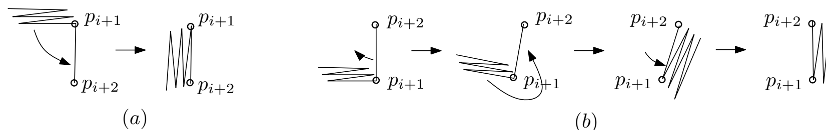

Fig. 1. Folding 2D chains.

e makes an angle of $\pi/12$ with its original position, and it is safe now to rotate S_i around p_{i+1} by sweeping through the side not containing S_i until $\angle p_i p_{i+1} p_{i+2} = \epsilon^2$. Then we move e back to its original position. See Figure 1(b) for illustration. It is clear that it takes only constant number of moves to construct S_{i+1} from S_i. Thus the whole chain can be folded up, in $O(n)$ moves and time, into a zig-zag path, which can then be straightened straightforwardly.

Theorem 1. *A 2D lattice chain can be straightened in $O(n)$ moves and time.*

3D lattice trees. As in the previous section, we can unfold 3D lattice chains in the same manner. In fact, we can even unfold 3D lattice trees in a similar fashion. we do this in a bottom-up fashion according to the given tree structure. The folding process starts from the leaves of the given tree. In each step, we fold up each set of all leaf springs incident to a common internal core vertex v to the internal core edge incident to v. Each time when we fold up a spring towards an edge, we keep the "tail" of the spring away from its moving direction. It is clear that folding each leaf spring takes constant number of moves. Thus we obtain the following theorem.

Theorem 2. *A 3D lattice tree can be straightened in $O(n)$ moves and time.*

4 2D lattice trees

Given a 2D lattice tree P. We consider a leftmost vertex r of P as its root. We consider the parent of the root r as the lattice point to the left of r. Our algorithm proceeds by pulling P to the left successively until the whole tree is straightened. Each pulling step moves each vertex along its edge connecting to its parent until it is within distance ϵ^2 to its parent in the previous step. This step is repeated n times so that, finally, P is straightened. Figure 2 shows the execution of the algorithm on a small tree. Step i generates a new polygon P_i. We assume $P = P_0$. First, we can show the following lemma.

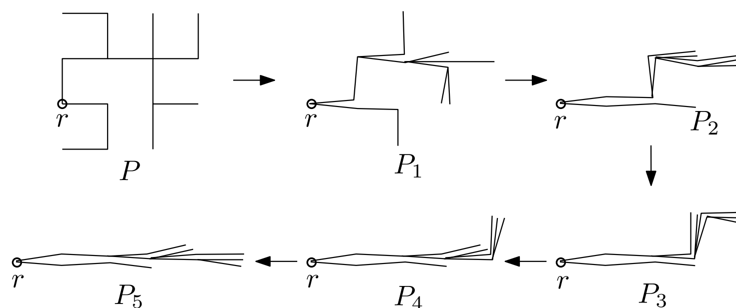

Fig. 2. Straightening a lattice tree by pulling it to the left successively.

Lemma 1. *During step i of the algorithm, suppose v is moving on the edge $v_{i-1}u_{i-1}$, where v_{i-1} and u_{i-1} are the positions of v and its parent just after step $i-1$. Then*

(i) Each vertex of P_i is within distance $i\epsilon^2$ to a lattice vertex.
(ii) Each vertex v is within distance $i\epsilon^2$ to the core lattice edge of e.
(iii) No edge crossings can occur.

Proof. Suppose, just after step i, v stops at position v_i. Note that $d(v_i, u_{i-1}) \leq \epsilon^2$ due to the algorithm.

(i) It is clear that any vertex of P_0 is a lattice point. Assume any vertex of P_{i-1} is within distance $(i-1)\epsilon^2$ to a lattice vertex U. Consider any vertex v_i of P_i. We know that $d(v_i, u_{i-1}) \leq \epsilon^2$. As $d(u_{i-1}, U) \leq (i-1)\epsilon^2$, we have $d(v_i, U) \leq i\epsilon^2$.

(ii) Suppose the core edge of $u_{i-1}v_{i-1}$ is UV, where u_{i-1} and v_{i-1} converge to U and V, respectively. Since $d(v_i, U) \leq i\epsilon^2$ and $d(v_{i-1}, V) \leq (i-1)\epsilon^2$, we have $d(v, UV) \leq i\epsilon^2$ as v moves along segment $v_{i-1}v_i$.

(iii) (*Sketch*) Consider a moving edge $e = uv$, where u is the parent of v. We show that e cannot cross other moving edges. Let the parent of u_{i-1} in P_{i-1} be t_{i-1}, which converges to lattice point T. By part (ii), we have $d(u, TU) \leq i\epsilon^2$. We consider two cases depending on whether T, U and V are collinear. If they

are, then any point p on e is within distance $i\epsilon^2$ from TUV. Otherwise, T, U and V are not collinear. Then any point p on e either lies in the lattice cell with T, U and V as its vertices or is within distance $i\epsilon^2$ from TUV. Also $d(e, TUV) \leq 1/\sqrt{2} + 2i\epsilon^2 \leq 2/3 < 1 - 2\epsilon$. Then it is not hard to see that e cannot cross other moving edges. The details are omitted in this abstract. □

As each pulling step takes $O(n)$ moves and time, the whole algorithm takes $O(n^2)$ moves and time.

Theorem 3. *A 2D lattice tree can be straightened in $O(n^2)$ moves and time.*

5 2D lattice polygons

Given a simple 2D lattice polygon P. Start with any lattice edge e of P. We label the edges of P in counter-clockwise order with consecutive numbers by starting with labeling edge e with the number 1. An edge of P is called an *odd edge* if its label number is odd; otherwise, it is called an *even edge*. Note that a 2D lattice polygon has even number of edges. We suppose that, throughout the entire motion, the parity of each edge remains fixed. A vertex is called *straight* if it is collinear with both its preceding and following vertices on the polygon. Otherwise, it is called *non-straight*.

A *block* of a lattice polygon is a rectangle of width one such that its left, top and bottom sides coincides with the edges of the given polygon. A *collapsible block* of a lattice polygon is a block such that its right side complementing the given polygon is a single segment. Such a segment is called the *opening segment*, or simply the *opening*, of the corresponding collapsible block. And the two endpoints of an opening segment are called *opening vertices*. The path between its two opening vertices on a collapsible block is called *a collapsible path*. A block or path of a near-lattice polygon is *collapsible* if the corresponding block or path of its core lattice polygon is collapsible.

The parity of a spring is defined as the parity of its two end edges. Suppose we walk along the edges of a near-lattice polygon in anti-clockwise order. We call a non-singleton spring *left-twisted* if its edges run to the left; otherwise it is called *right-twisted*. See Figure 3(a) for examples. A near-lattice polygon is

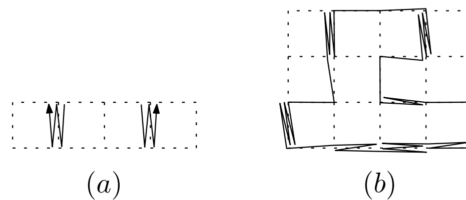

Fig. 3. (a) A left-twisted spring and a right-twisted spring, respectively. (b) A consistently twisted near-lattice polygon.

called *consistently twisted* if odd and even springs have opposite directions of twisting. See Figure 3(b) for an example of a consistently-twisted near-lattice polygon. We first need the following two lemmas.

Lemma 2. *In a non-nearly-folded near-lattice polygon, there is a collapsible block; more precisely, the block with the smallest height is a collapsible block.*

Proof. Suppose to the contrary that the block B with the smallest height is not collapsible. Then its opening contains at least two segments. Thus there is some block B' with its left side between these two segments. Obviously the height of B' is shorter than that of B. This contradicts our assumption. □

Lemma 3. *In a consistently-twisted near-lattice polygon, consider a collapsible block B such that all its non-singleton springs lie on its left side. Then*

(i) All its non-singleton springs can be transformed into one non-singleton spring on any edge on the left side of the block with consistent twisting.
(ii) Its corresponding collapsible path can be transformed to a consistently-twisted near-lattice path convergent to the opening segment of B.

Proof. (*Sketch*)
(i) Note that the non-singleton springs lie only on the left side of B. A non-singleton spring can be transformed into a singleton by moving its remaining edges to the adjacent spring. Repeating this process results in only one non-singleton spring on the left side of B. See Figure 4(a) for illustration.

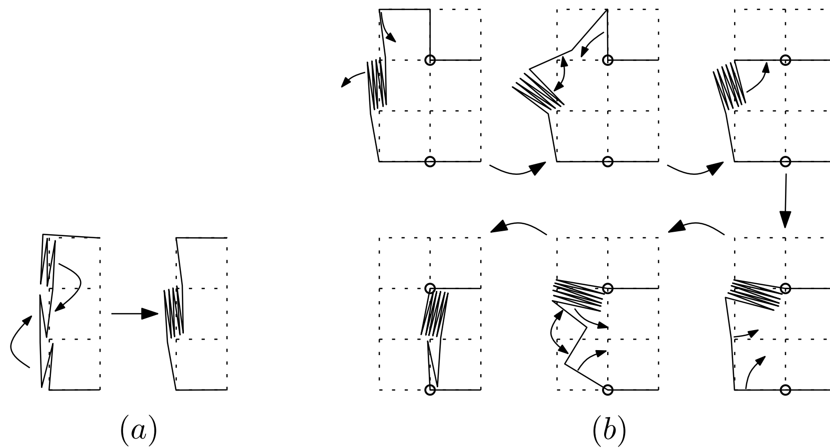

Fig. 4. (a) Transform several springs into a single non-singleton spring on the left side of B. (b) Collapse a collapsible block.

(ii) We need to consider several cases depending on different directions of the incident edges of its two opening vertices. Note that according to part (i),

we can assume the left side of B contains only one non-singleton spring, which reduces a lot of cases we need to consider. Figure 4(b) shows the collapse of a specific collapsible block. Other cases can be handled in a similar way, whose details are omitted in this abstract. ◻

Lemma 2 says that a near-lattice polygon which is not nearly folded always contains a collapsible block, which can then be collapsed using Lemma 3 in $O(n)$ moves and time. Hence, each step of our algorithm is to select the block with smallest height in the polygon to collapse. Note that all the blocks of the polygon can be maintained in a priority queue with their heights as keys. After $O(n)$ collapsing steps, we end up with a nearly folded polygon, which clearly can be convexified in $O(n)$ moves and time.

Theorem 4. *A 2D lattice polygon can be straightened in $O(n^2)$ moves and time.*

6 3D orthogonal chains and polygons

A 3D 5-chain can lock [4]. We can simulate this chain by an orthogonal 9-chain as shown in Figure 5(a), where each of its two end edges is longer than the union of its internal edges. Furthermore, by doubling this 9-chain, we obtain a 3D locked orthogonal unknotted 20-gon as shown in Figure 5(b).

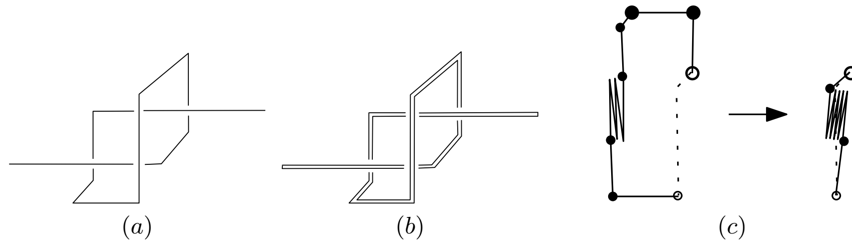

Fig. 5. (a) A 3D orthogonal locked 9-chain. (b) A 3D orthogonal locked 20-gon. (c) Collapse a 3D collapsible block.

Theorem 2 implies that a 3D lattice chain can be straightened in $O(n)$ moves. We present below an algorithm to straighten a class of unknotted lattice polygons in 3D.

A *3D polygon with simple shadow* is a simple 3D polygon whose shadow is a simple polygon when it is projected orthogonally onto some plane, which we assume to be xy-plane. A *3D polygon with simple projection* is a 3D polygon with simple shadow such that any line parallel to z-axis intersects the polygon with at most one single connected component. Alberto-Calvo et al. [1] showed that a 3D polygon with simple projection can be convexified in $O(n+T)$ time, where T is the running time of an algorithm to convexify the planar projection. There are

several algorithms to convexify a planar polygon [5, 9, 14]. The best bound for T is $O(n^{79})$ due to the algorithm by Cantarella et al. [5], where the constant is a polynomial in the ratio between the maximum edge length and initial minimum distance between a vertex and an edge. In this section, we present an efficient algorithm to convexify a 3D lattice polygon with simple shadow in $O(n^2)$ moves and time. In this abstract, we only sketch the main idea.

Suppose P is the given 3D lattice polygon with simple shadow. Consider any vertical lattice plane π parallel to z-axis. Let P_π be the intersection of π and P. Our algorithm starts by collapsing the blocks in each P_π. This step is similar to what we do for the 2D case. The difference is that we need to consider more types of blocks. We define a *vertical block* as a block of width one with its opening on its left or right side, and a *horizontal block* as a block of height one with its opening on its top or bottom side. Note that a block in 2D case is basically a vertical back with its opening on its right side. A *collapsible block* is defined similarly as the 2D case. This step of our algorithm searches for any collapsible vertical or horizontal block to collapse for each P_π. It takes $O(n^2)$ moves and time, and ends up with a near-lattice polygon P' whose core is a polygon with simple projection.

In order to collapse the edges of P', we need to define the three dimensional version of blocks. A *3D block* of P' is a 3D box B with width one along x-axis or y-axis such that the intersection of B and the core of P' projects orthogonally to a two dimensional block on xy-plane. A *collapsible 3D block* is a 3D block whose orthogonal projection is a collapsible 2D block. Figure 5(c) shows the result of collapsing a 3D collapsible block, which can be done in $O(n)$ time. Let lattice polygon Q be projection of the core of P' onto xy-plane. Each 2D block in Q corresponds to a 3D block in P'. In this step, our algorithm finds each 3D collapsible block by identifying its corresponding 2D collapsible block in its orthogonal projection. As there are at most $O(n)$ 3D collapsible blocks to consider, the running time for this step is again $O(n^2)$.

Theorem 5. *A 3D lattice polygon with simple shadow can be straightened in $O(n^2)$ moves and time.*

7 Diameter-4 trees in 2D

Let T be a polygonal tree of diameter 4 in the plane, with o as its central node. We call the edges incident to o *back edges*, and the rest *front edges*. A *UB-tree* is a tree of diameter 4 with all its back edges of unit length. An *SB-tree* is a tree of diameter 4 with all its back edges not longer than their corresponding front edges. In this section, we show that a UB-tree or SB-tree in two dimensions can always be straightened.

We first to define some technical terms. A *branch* is a path from o to a leaf of T. We define $E(B)$ to be the extension ray of the back edge of B. The *direct straightening* of branch $B = ouv$ means to rotate the front edge uv around the back vertex u until it aligns along $E(B)$ by sweeping through the smaller angle.

We denote $S(B)$ to be the swept region for directly straightening B. The *direct collapse* of branch $B = ouv$ means to rotate the front edge uv around the back vertex u by sweeping through the smaller angle until it makes an arbitrarily small angle with ou. We say B' *follows* B if B' intersects $E(B)$, and B' and B are branches of the same turn. We say B' *directly covers* B if B' intersects $E(B) \cup S(B)$, and B and B' are branches of opposite turns. We then define B' *covers* B if there exists some branch B'' following B or simply being B such that B' directly covers B''. We define a branch B' *mutually covers* another branch B if B' covers B, and vice versa. Branch B' *non-mutually covers* branch B if B' covers B, but B does not cover B'.

We categorize the branches into three groups:

- (Group I) branches falling on a maximal following cycle,
- (Group II) those falling on non-mutual covering sequences, and
- (Group III) those including mutually covered branches and branches covered by some straightened branches.

See Figure 6 for an example of different groups of branches in a UB-tree. Group I branches are those with all of their vertices dotted, Group III branches are bold, and the remained branches are Group II branches.

Our algorithm consists of three phases, in which we straighten the three groups of branches in the above order respectively. In the following cycle (if there is), the front edge of some of its branches B can be swept through $S(B)$ by passing through an angle of $\Omega(\frac{1}{n})$. Thus Phase I needs $O(n^2)$ moves to straighten such a following cycle. In Phase II, we observe that the last branch (including its following branches) of a maximal non-mutual covering sequence can be straightened directly. We repeat this peeling process to straighten all Group II branches. Finally, we consider the final phase of our algorithm. Group III branches have an important property that their front edges are shorter than their corresponding back edges. This implies that all branches with a common back edge can all be collapsed directly altogether. Then all the collapsed branches can be rotated to pack inside a small wedge, say a quadrant. Now the collapsed branches can be drawn out of the quadrant one by one to be straightened directly. This completes our algorithm. The algorithm can be implemented to take $O(n^2)$ moves and $O(n^3 \log n)$ time. We summarize in the following theorem.

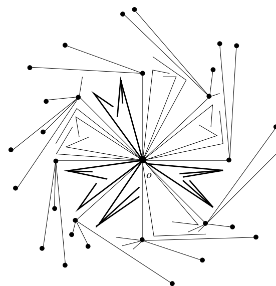

Fig. 6. A UB-tree.

Theorem 6. *A UB-tree or SB-tree of diameter 4 in two dimensions can be straightened in $O(n^2)$ moves and in $O(n^3 \log n)$ time.*

8 Conclusion

We present efficient algorithms to straighten lattice chains and trees and to convexify lattice polygons in both two and three dimensions, respectively. In par-

ticular, we only manage to show that a special class of lattice polygons in three dimensions can be convexified. We believe that any unknotted lattice polygon in three dimensions can always be convexified. We are currently investigate in this direction. It is also open whether an unknotted unit polygon in three dimensions can always be straightened. Furthermore, it is open whether a unit tree in two or three dimensions can always be straightened [13]. In Section 6, we show that an orthogonal chain in three dimensions can lock. However, it is unknown whether an orthogonal tree in two dimension can lock. In fact, we conjecture that it can always be straightened.

References

1. J. Alberto-Calvo, D. Krizanc, P. Morin, M. Soss, G. Toussaint. Convexifying polygons with simple projections. *Information Processing Letters*, 80(2):81-86, 2001.
2. H. Alt, C. Knauer, G. Rote, and S. Whitesides. The Complexity of (Un)folding. In *Proc. 19th ACM Symposium on Computational Geometry (SOCG)*, 164–170, 2003.
3. T. Biedl, E. Demaine, M. Demaine, S. Lazard, A. Lubiw, J. O'Rourke, S. Robbins, I. Streinu, G. Toussaint, and S. Whitesides. A Note on Reconfiguring Tree Linkages: Trees can Lock. *Discrete Applied Mathematics*, volume 117, number 1-3, pages 293-297, 2002.
4. T. Biedl, E. Demaine, M. Demaine, S. Lazard, A. Lubiw, J. O'Rourke, M. Overmars, S. Robbins, I. Streinu, G. Toussaint, and S. Whitesides. Locked and Unlocked Polygonal Chains in Three Dimensions. *Discrete & Computational Geometry*, volume 26, number 3, pages 269-281, October 2001.
5. J. Cantarella, E.D. Demaine, H. Iben, and J. O'Brien. An Energy-Driven Approach to Linkage Unfolding. In *Proceedings of the 20th Annual ACM Symposium on Computational Geometry (SoCG 2004)*, 134–143, 2004.
6. H.S. Chan and K.A. Dill. The protein folding problem. *Physics Today*, pages 24–32, February 1993.
7. R. Cocan and J. O'Rourke. Polygonal Chains Cannot Lock in 4D. *Computational Geometry: Theory & Applications*, 20, 105–129, 2001.
8. R. Connelly, E. Demaine, and G. Rote. Infinitesimally Locked Self-Touching Linkages with Applications to Locked Trees. In *Physical Knots: Knotting, Linking, and Folding of Geometric Objects in* R^3, edited by J. Calvo, K. Millett, and E. Rawdon (editors), 2002, pages 287–311, American Mathematical Society.
9. R. Connelly, E.D. Demaine, and G. Rote. Straightening Polygonal Arcs and Convexifying Polygonal Cycles. *Discrete & Computational Geometry*, volume 30, number 2, 205–239, 2003.
10. E. D. Demaine, S. Langerman, J. O'Rourke, and J. Snoeyink. Interlocked Open Linkages with Few Joints. In *Proc. 18th Annual ACM Symposium on Computational Geometry (SoCG)*, Barcelona, Spain, June 5-7, pages 189–198, 2002.
11. K.A. Dill. Dominant forces in protein folding. *Biochemistry*, 29(31), 7133–7155, August 1990.
12. B. Hayes. Prototeins. *American Scientist*, 86, 216–221, 1998.
13. S.-H. Poon. On Straightening Low-Diameter Unit Trees. In *Proc. 13th International Symposium on Graph Drawing*, 519–521,2005.
14. I. Streinu. A combinatorial approach for planar non-colliding robot arm motion planning. In *Proc. 41st ACM Annual Symposium on Foundations of Computer Science (FOCS)*, 443–453, 2000.