Deforming Curves in the Plane for Tethered-Robot Motion Planning

(Extended Abstract)[*]

Susan Hert[†] Vladimir Lumelsky[‡]

Abstract

We present an algorithm that deforms a given set of polygonal lines (*polylines*) defined on a set of $2n$ vertices into the new set of polylines that results from one of the vertices moving along a straight line to a new point in the plane, creating bends in the polylines intersected by the line of motion. The algorithm has applications in robotics, where the moving point is one of n robots in the plane and the polylines are representations of taut tethers attached to the robots. The algorithm runs in $O(kn^2)$ time, where k is the maximum number of line segments a polyline may have. The algorithm makes use of work presented in [8], which requires that a triangulation of the set of polyline vertices be maintained. When the polyline vertices change, so must the triangulation. Also presented here is a new triangulation algorithm and data structure that allows for easy insertion and deletion of triangle vertices. The data structure requires $O(n)$ space and allows vertices to be inserted in $O(n)$ time in the worst case and deleted in $O(n^2)$ time. The expected time for insertion or deletion of a vertex is $O(\log n)$.

1 Introduction

We consider the following problem. A set of $2n$ distinct points, $S_i, T_i, i = 1, \ldots, n$, in the plane is given. The points $S = \{S_i\}$ lie on the boundary of a convex polygon P with vertices V; the points $T = \{T_i\}$ lie in its interior. Each pair of points (S_i, T_i) is connected by a polygonal line (*polyline*, for short) PL_i that begins at S_i (the *initial vertex*) and ends at T_i (the *final vertex*). In addition, each polyline may have any number of *internal vertices*. These vertices are members of the set T and are not necessarily distinct (*i.e.*, a single vertex may be repeated any number of times, although not in sequence) (Figure 1(a)). Each polyline PL_i may be seen as a discrete representation of a continuous, taut curve C_i in the plane that bends around small circular objects located at the points T_i (Figure 1(b)). The polylines do not intersect (or self-intersect), by which we mean that the continuous curves they represent do not intersect. Note that, by this definition, intersecting as well as nonintersecting polylines may share vertices and edges (Figure 1).

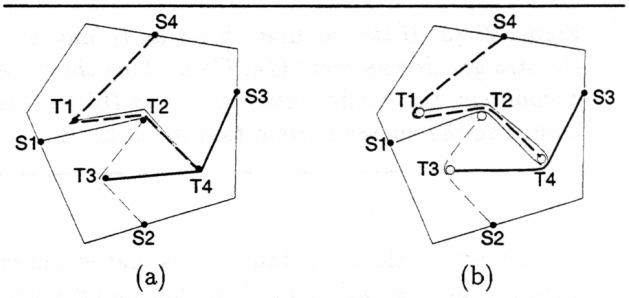

Figure 1: *(a) A set of $n = 4$ nonintersecting polylines is shown. Polyline PL_1 contains a repeated edge (T_2, T_4). The edges (T_1, T_2) and (T_2, T_4) are shared between PL_1 and PL_4 but these polylines do not intersect since the set of curves corresponding to these polylines, shown in (b), do not intersect. In (b), the points $T_1, \ldots T_4$ have been replaced by small circular objects around which the curves bend.*

A particular arrangement of polylines is known as a *configuration*. The problem considered here is to determine how a configuration changes when one of the points T_i moves along a straight line in the plane to a new location T_i'. The new configuration that results

*This work is supported in part by the National Science Foundation Grant IRI-9220782 and DOE (Sandia Labs) Grant 18-4379C and the Sea Grant Program (U.S. Dept. of Commerce, grant NA46RG048).

[†]University of Wisconsin – Madison, Computer Sciences Department, 1210 West Dayton St., Madison, WI, 53706, USA., E-mail: hert@cs.wisc.edu

[‡]University of Wisconsin – Madison, Robotics Laboratory, 1513 University Ave. Madison, WI 53706, USA. E-mail: lumelsky@engr.wisc.edu

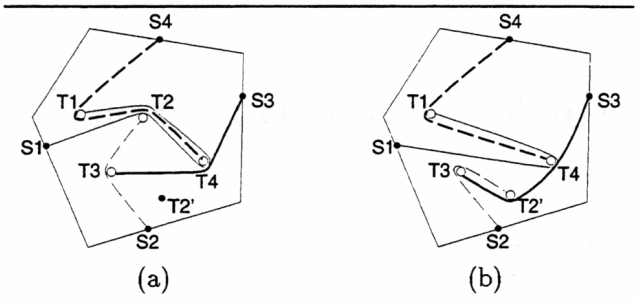

Figure 2: *(a) If the circular object at T_2 moves along the straight line segment (T_2, T_2') pushing the curves it encounters, the configuration shown in (b) will result, assuming the curves remain taut at all times.*

corresponds to the set of taut curves that would result from the circular object at T_i moving along a straight line in the plane to T_i', pulling with it the curve C_i ending at T_i and pushing the other curves it encounters out of its way (Figure 2).

This problem appears in robotics in the context of motion planning for multiple tethered robots in the plane [10, 11, 13], and has applications to commercial systems such as RobotWorld [12]. The moving bodies represent small, disc-like robots; the deforming curves are their flexible cables. The cables provide resources necessary for the robots to perform their tasks. When planning the motion for these robots, the configurations of their cables that result from the motion must be taken into account. We assume the cables may be pushed and bent by other robots that come in contact with them but remain taut at all times. An algorithm for determining a taut configuration of the cables given the points S and T is presented in [5]. Algorithms for planning the motion of a set of robots based on such a configuration have also been developed [4, 6]. Presented here is an algorithm for determining the configuration of the cables after a certain motion plan has been executed.

The algorithm we present makes use of the work of Leiserson and Maley [8] for routing wires in a planar VLSI design around certain features (points) in the plane. They present an algorithm for producing from a given set of curves in the plane a topologically equivalent set of polylines (the *rubberband equivalents* of the set of curves), the vertices of which are members of a given set of points in the plane. For the given set of curves, the rubberband equivalents are the shortest set of topologically equivalent curves. The algorithms of [8] do not provide a way of altering the rubberband equivalents when one of the points in the plane changes. Here we extend the work of Leiserson and Maley to provide an efficient way of changing from one rubberband equivalent to another when the set of potential vertices changes.

To create the rubberband equivalents for a given set of curves, a triangulation of the given set of potential vertices is necessary. For the algorithm presented here, the set of potential vertices changes. Thus, a dynamic triangulation method, that allows vertices to be added and deleted easily without a complete retriangulation, is necessary. Though there are several partially dynamic triangulations, that allow for easy insertion of a new vertex but not deletion (*e.g*, [3, 7]), only two other fully dynamic triangulation algorithms have been presented in the literature, each of which is designed to maintain a Delaunay triangulation on a set of points [1, 9]. To compute the rubberband equivalents, any triangulation of the potential vertices will suffice; in particular, the triangulation need not be Delaunay. We present here a new, fully dynamic method for maintaining a non-Delaunay triangulation. By removing the constraint that the triangulation be Delaunay, we are able to construct a dynamic triangulation method that is arguably simpler than the Delaunay methods. This triangulation method also provides a simple way to maintain the data structure that represents the polylines and their relationship to the changing triangulation.

The dynamic Delaunay triangulation method of Palacios-Velez and Cuevas-Renaud [9] uses an $O(n)$ size data structure for a triangulation with $O(n)$ vertices that supports $O(n)$ vertex insertion and deletion procedures. Our data structure, the *Triangulation Tree*, also requires $O(n)$ space. Insertion of a new vertex using the Triangulation Tree can be done in $O(n)$ time and deletion in $O(n^2)$. Devillers, Meiser, and Teillaud [2] show that with their data structure, the *Delaunay Tree*, the expected time for inserting or deleting a vertex is $O(\log n)$. Using the framework developed by Boissonnat, *et al.* for randomized analysis of geometric algorithms [1], we arrive at the same expected time of $O(\log n)$ for insertions and deletions using the Triangulation Tree.

The Triangulation Tree and polyline data structures are described in Section 2 along with the procedures for modifying them. Using these procedures, the algorithm $NewConfig$ for producing the new configuration of polylines that results from the movement of the point from T_i to T_i' is presented and analyzed in Sec-

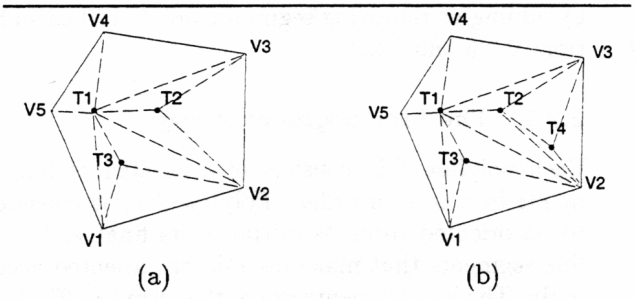

Figure 3: *(a) The triangulation Δ_3. (b) When T_4 is added as a triangulation vertex, triangle $V_2V_3T_2$ is subdivided into three smaller triangles.*

tion 3, followed by concluding remarks in Section 4.

2 Data Structures

2.1 The Triangulation Tree

The *Triangulation Tree*, first introduced in [6] as a data structure that easily incorporates the insertion of a new triangulation vertex, is modified here to allow for efficient deletions of vertices as well. Let Tr_i represent the Triangulation Tree associated with the triangulation Δ_i of the vertices $\{T_1, \ldots, T_i\} \cup V$. The tree Tr_i is unbalanced and hierarchical, containing one leaf node per triangle and one nonleaf node per vertex T_i. The root of the tree represents the polygon P. All other nonleaf nodes of Tr_i represent triangles of previous triangulations that have been subdivided due to the insertion of one or more points.

The hierarchical structure of the tree is based on the following observation: to create Δ_{i+1} from Δ_i when the point T_{i+1} is added to the set of triangulation vertices, one needs only find the triangle of Δ_i containing T_{i+1} and subdivide it into three smaller triangles (Figure 3). If TN is the leaf node corresponding to the triangle of Δ_i containing T_{i+1}, tree Tr_{i+1} is created from Tr_i by adding three leaf nodes as children of TN[1].

For each node TN, which corresponds to a unique triangle τ in either the current or a previous triangulation, the following information is stored:

Parent – a pointer to its parent in the tree;

Vertices – the counterclockwise list of vertices of the triangle τ (for the root node, this is the list of vertices of polygon P);

PointNum – the number i of the point T_i that subdivides τ (for leaf nodes, $PointNum = NoPoint$);

Children – an ordered list of children;

Neighbors – a pointer to the nodes representing the neighbors of τ across each of its edges (for the root node, this list is empty).

The root of the tree has $v = |V|$ children; all other nonleaf nodes have three children. We use the notation $FieldName(TN)$ (as in $Parent(TN)$) to indicate the data associated with a particular node TN.

The data for each node in the tree requires $O(1)$ space and there are $O(|T|) = O(n)$ nodes in the tree. The Triangulation Tree is thus an $O(n)$ data structure.

2.1.1 Adding a Triangulation Vertex

As discussed above, to add a new vertex to the triangulation one needs only find the proper triangle in the current triangulation to subdivide and add children to the corresponding node in the Triangulation Tree.

To find the leaf node corresponding to the triangle containing a given point, a walk from the root of the tree down one branch is done by testing for inclusion of the point in successively smaller triangles. In the worst case, this requires $O(n)$ time for a tree with $O(n)$ nodes, but, on average, it requires $O(\log n)$ time [1]. All information about a child can be easily determined from its parent's information in $O(1)$ time. Thus, adding a new triangulation vertex requires at most $O(n)$ time and, on average, $O(\log n)$ time.

2.1.2 Deleting a Triangulation Vertex

To delete a vertex T_i from the triangulation while maintaining the hierarchical structure of the tree, some nodes must be removed from the tree and then reinserted. Let TN be the node in tree Tr with $PointNum(TN) = i$ and let τ be the triangle TN represents. Node TN and its descendants must then be adjusted to reflect the deletion of T_i. The adjustment can be done by reinserting all vertices contained in τ. Note, however, that all nodes reinserted will

[1] We assume here, for ease of explanation, that the point T_{i+1} does not lie on an edge of Δ_i. When this is not the case, insertion of T_{i+1} causes two triangles to be divided into two triangles each and, correspondingly, the addition of two children to each of two nodes. The same theoretical results presented here apply in this case but with minor modifications to the algorithms.

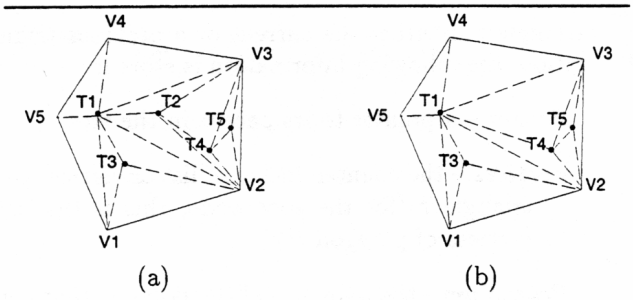

Figure 4: *If vertex T_2 is deleted from the triangulation shown in (a), T_4 and T_5, the other two points contained in triangle $V_2V_3T_1$ are candidates for reinsertion. When vertex T_4 is reinserted (b), the triangle $V_2V_3T_4$ is stable and vertex T_5 need not be reinserted.*

be descendants of TN so it may be assumed to be the root of the tree for the vertex insertion procedure. Further note that some triangles (called *stable triangles*) that were created by vertices contained in τ will not be affected by the deletion of T_i (Figure 4); their corresponding nodes (*stable nodes*) and descendants may be simply copied to the appropriate positions in the modified tree. Which triangles are stable depends on the order of reinsertion of the vertices in τ. The vertices are reinserted in the same order they were initially inserted (*i.e*, from node TN down) to achieve the most stability.

In the worst case there will be $O(n)$ nodes to reinsert after the deletion of T_i, which implies an $O(n^2)$ worst-case running time for the vertex deletion procedure. On average, however, only a constant number of nodes nodes will have to be reinserted, which implies an $O(\log n)$ expected running time for the procedure.

2.2 Polylines

Our algorithm for deforming one set of polylines into another when one of the polyline vertices moves requires that the relationship between the polylines and the triangulation of the polyline vertices be maintained as the triangulation is updated to incorporate the changed vertex. Section 2.2.1 describes the data structure used to represent the polylines. To maintain this data structure, an additional data field is also required for each node in the Triangulation Tree; this is described in Section 2.2.2 along with the necessary modifications to the vertex insertion and deletion procedure. Procedures for changing the polylines by adding or removing segments are described in Sections 2.2.3 and 2.2.4.

2.2.1 The Line Segment Array

Each polyline PL_i consists of a number of line segments in a certain order. Polyline PL_i is considered to be oriented from its initial to its final vertex; the line segments that make up PL_i are oriented accordingly. The line segments are of the form (S_i, T_j) (j not necessarily different from i) or $(T_j, T_k), j \neq k$. Associated with each polyline is a set of *corridor vertices*, which, for a given triangulation, is the ordered list of vertices of the triangles encountered when moving from the initial to the final vertex of PL_i together with an indication of on which side of PL_i each vertex lies [8]. Note that, for the triangle containing the initial vertex, there are two corridor vertices; for all other triangles, there is one, which is the vertex not shared by the previous triangle.

Though the polylines may be arbitrarily large with any number of segments, the set of unique line segments present in any configuration of nonintersecting polylines forms a planar graph on the points S and T. Therefore, there are $O(|S| + |T|) = O(n)$ distinct line segments. The data for these line segments are stored in the *Line Segment Array*, LS.

For each line segment $L \in LS$, the following data are stored:

Start, *End* – the endpoints of L;

TriList – the list of the Triangulation Tree nodes corresponding to the triangles intersected by L, ordered from $Start(L)$ to $End(L)$;

CV – the corridor vertices for L, ordered from $Start(L)$ to $End(L)$;

DeformPLine – a temporary list of line segments created for each line segment altered by the movement of a polyline vertex.

The $TriList$ for each segment L begins with a pointer to the node corresponding to the triangle subdivided by $Start(L)$, if $Start(L) = T_j$ for some j, and ends with the analogous node for $End(L)$. When L is an edge of the triangulation, these are the only two elements of $TriList(L)$. For every node in $TriList(L)$, except the first and last, there is an entry in $CV(L)$.

Each polyline is represented as a list of pointers to the appropriate entries in LS together with an indication of the orientation of each segment. The corridor vertices of each polyline can be easily determined from

the corridor vertices $CV(L)$ for each segment L of the polyline and the initial and final nodes of $TriList(L)$. The initial and final nodes of $TriList(L)$ are used to determine the corridor vertices contributed by every internal vertex of a polyline that is also a triangulation vertex.

The maximum size of $TriList(L)$, for any segment L is $O(n)$ since the number of triangles generated by $O(n)$ vertices is $O(n)$ and any segment may pass through all but a constant number of triangles in a given triangulation. Thus there may also be $O(n)$ corridor vertices. $DeformPLine(L)$ is a polyline of at most four segments (Section 3). Thus the Line Segment Array requires $O(n^2)$ storage.

2.2.2 Triangulation Tree with Polylines

When vertices are added to or removed from the triangulation, the list of corridor vertices and intersected triangles must be updated for certain segments in LS. To keep track of which segments are affected by such changes, an additional data field is added to each node TN in the Triangulation Tree:

SegList – an unordered list of pointers to segments $L \in LS$ that intersect the triangle τ that TN represents, together with an indication of TN's location in $TriList(L)$.

Since each triangle may be intersected by $O(n)$ segments from LS, the addition of this data field makes the Triangulation Tree an $O(n^2)$ data structure.

The procedures for inserting and deleting triangulation vertices must also be changed to incorporate the maintenance work on the polyline data structure that is necessary for each change in the triangulation. The worst-case complexity of these procedures is not affected by these changes, but average-case complexity for each is increased to $O(n)$.

Though the addition of the *SegList* data field causes an increase in the space required for the Triangulation Tree and average-time complexity of its procedures, this data field provides an easy and efficient way to update the list of corridor vertices for each segment affected by a change in the triangulation. Without this data field, it would be necessary to consider each line segment's list of corridor vertices and tree nodes as a possible candidate for change for every addition or deletion of a triangle vertex.

2.2.3 Inserting a Line Segment

When a line segment L is added to the Line Segment Array, it is necessary to trace it through the current triangulation to determine its set of corridor vertices $CV(L)$ and the corresponding tree nodes $TriList(L)$. The segment L may pass through $O(n)$ triangles and thus $CV(L)$ and $TriList(L)$ may be created in $O(n)$ time. Also, for each node TN corresponding to a triangle L intersects, $SegList(TN)$ must be updated. Each update requires $O(1)$ time and there may be $O(n)$ nodes to update. Thus inserting a new segment in the Line Segment Array requires $O(n)$ time in the worst case.

2.2.4 Deleting a Line Segment

Deletion of a segment L from LS can be done by simply removing L from $SegList(TN)$ for each $TN \in TriList(L)$. This operation is bounded by the number of triangles through which a single segment may pass, which is $O(n)$.

3 The *NewConfig* Algorithm

Having described the data structures used by our algorithm and the procedures for updating these data structures, we are now prepared to describe the complete algorithm, called *NewConfig*, that produces the new configuration of polylines that results when a polyline vertex moves to a new location in the plane along a straight line. The input to the algorithm is the initial set of polylines $PL = \{PL_i\}$, the Triangulation Tree Tr for the points $T \cup V$, the index i of the vertex T_i that is moving, and the new location T_i' of that vertex. Let M be the line segment (T_i, T_i'). To produce the new configuration of polylines PL' that results from the motion of the point along M, the point T_i is removed from the triangulation and T_i' is inserted. Then, for each line segment $L \in LS$ that is intersected by M at a point p in its interior, $DeformPLine(L)$ is created as a polyline of five vertices: $Start(L), p, T_i', p, End(L)$. For each line segment L with an endpoint on M, $DeformPLine(L)$ is a polyline of two segments: L and M. From the set of original and deformed line segments, the set of corridor vertices for each polyline affected by the move along M can be easily determined.

The corridor vertices of each polyline are used as input to algorithm W of [8] to produce the new set of polylines PL'. Finally, the array LS is updated by removing segments that are not part of the polylines PL' and adding any new segments.

The algorithm of [8] runs in time linear in the number of corridor vertices. If k is the maximum number

of segments in a polyline, the number of corridor vertices is $O(kn)$ for each polyline. It can be shown, based on the previous analysis of the data structure procedures, that the other steps of the algorithm require no more than $O(kn^2)$ steps in total. Thus the algorithm requires no more than $O(kn^2)$ time in the worst case.

4 Conclusion

We have presented an algorithm, $NewConfig$, for altering a given set of nonintersecting polylines that correspond to taut curves in the plane, to reflect the changes brought about by one of the polyline vertices moving along a straight line in the plane. We have also presented the data structures used by the $NewConfig$ algorithm, one of which supports a new dynamic triangulation method. This triangulation method is comparable in space complexity and worst-case and average-case time complexity to existing fully dynamic triangulation methods, but also supports easy maintenance of the data structure that represents the polylines.

The $NewConfig$ algorithm has potential applications in any problem setting that involves curves bending around polygonal or circular objects in the plane, such as in planar VLSI design and robotics. In particular, $NewConfig$ has been used as a basis for a motion planning algorithm for a set of tethered robots in the plane. The polylines represent the taut tethers of the robots. When the path of each robot is a polyline itself, the $NewConfig$ algorithm may be used repeatedly for each segment of the path to produce the configuration of cables that results when each robot has moved to its final destination. The resulting configuration may then be used to determine future motion plans for the robots.

References

[1] J. Boissonnat, O. Devillers, R. Schott, M. Teillaud, and M. Yvinec. Applications of random sampling to on-line algorithms in compuational geometry. *Disc. Comput. Geom.*, 8:51–71, 1992.

[2] O. Devillers, S. Meiser, and M. Teillaud. Fully dynamic Delaunay triangulation in logarithmic expected time per operation. In *Proc. 2nd Workshop Alg. Data Struct.*, volume 519 of *Lecture Notes in Computer Science*, pages 42–53. Springer-Verlag, 1991.

[3] L. Guibas, D. Knuth, and M. Sharir. Randomized incremental construction of Delaunay and Voronoi diagrams. In *Int. Colloq. Automata, Lang. and Prog.*, pages 414–431, Warwick University, England, July 1990. Springer-Verlag.

[4] S. Hert and V. Lumelsky. Moving multiple tethered robots between arbitrary configurations. In *Proc. 1995 IEEE/RSJ Int. Conf. Intelligent Robots and Systems*, pages 280–285, August 1995.

[5] S. Hert and V. Lumelsky. Planar curve routing for tethered-robot motion planning. *Int. J. Comput. Geom. & Appl.*, to appear 1996.

[6] S. Hert and V. Lumelsky. The ties that bind: Motion planning for multiple tethered robots. *Robotics and Autonomous Systems*, to appear 1996.

[7] D. T. Lee and B. Schachter. Two algorithms for constructing Delaunay triangulations. *Int. J. Comp. Info. Sci.*, 9(3):219–242, 1980.

[8] C. E. Leiserson and F. M. Maley. Algorithms for routing and testing routability of planar VLSI layouts. In *Proc. 17th Ann. Symp. on Theory of Comput.*, pages 69–78, 1982.

[9] O. Palacios-Velez and B. Cuevas-Renaud. Dynamic hierarchical subdivision algorithm for computing Delaunay triangulations and other closest-point problems. *ACM Trans. Math. Softw.*, 16(3):275–292, Sept. 1990.

[10] G. Pardo-Castellote and H. Martins. Real-time motion scheduling for a SMALL workcell. In *Proc. 1991 IEEE Conf. on Rob. and Aut.*, pages 810–817, April 1991.

[11] D. Parsons and J. Canny. A motion planner for multiple mobile robots. In *Proc. 1990 IEEE Conf. on Rob. and Aut.*, pages 8–13, May 1990.

[12] V. Scheinman. RobotWorld: A multiple robot vision guided assembly system. In *Proc. 4th Int. Symp. on Robotics Research*, pages 23–27, 1987.

[13] F. Sinden. The tethered robot problem. *Int. J. Robotics Research*, 9(1):122–133, February 1990.