ANNALS

OF THE

UNIVERSITY OF CRAIOVA

Series: AUTOMATION, COMPUTERS, ELECTRONICS and MECHATRONICS

Vol. 11 (38), No. 1, 2014

ISSN 1841-0626

EDITURA UNIVERSITARIA

ANNALS OF THE UNIVERSITY OF CRAIOVA

Series: AUTOMATION, COMPUTERS, ELECTRONICS AND MECHATRONICS

Vol. 11 (38), No. 1, 2014 **ISSN 1841-0626**

***Note*:** The **"Automation, Computers, Electronics and Mechatronics Series"** emerged from **"Electrical Engineering Series" (ISSN 1223-530X)** in 2004.

Address for correspondence:

Vladimir RĂSVAN
University of Craiova
Faculty of Automation, Computers and Electronics
Al.I. Cuza Street, No. 13
RO-200585, Craiova, Romania
Phone: +40-251-438198, Fax: +40-251-438198
Email: vrasvan@automation.ucv.ro

This issue has been published under the responsibility of Emil PETRE.

We exchange publications with similar institutions from country and from abroad

Computing Minimal Separating Polygons by Convexifying Non-Self-Intersecting Spanning Trees

Eliana-Dina Andreica*, Mugurel Ionut Andreica**

**Computer Science Department, Politehnica University of Bucharest*
Splaiul Independentei 313, 060042, sector 6, Bucharest, Romania
(email: eliana.andreica@cs.pub.ro)
***Computer Science Department, Politehnica University of Bucharest*
Splaiul Independentei 313, 060042, sector 6, Bucharest, Romania
(email: mugurel.andreica@cs.pub.ro)

Abstract: In this paper we present a novel approach for the problem of computing a small perimeter simple polygon which separates a set of *M* red points from a set of *N* blue points in the plane (i.e. all the points of the same color are inside or on the border of the polygon and all the points of the other color are outside or on the border of the polygon). Our approach is based on generating non-self-intersecting spanning trees of the points located inside the polygon and then convexifying these spanning trees in order to obtain a minimal separating polygon (i.e. a polygon whose perimeter cannot be decreased further by convexification operations and which completely contains the original spanning tree). By generating multiple spanning trees we are able to obtain separating polygons with various perimeter lengths. We consider both the case when the separating polygon may only contain the original points as vertices, as well as the case when new points can be used as vertices of the polygon. In the second case each spanning tree is convexified to (approximately) the same polygon irrespective of the order in which the convexification operations are performed.

Keywords: Adaptive algorithms, Polygons, Trees.

1. INTRODUCTION

In this paper we consider the well-studied problem of computing a minimum perimeter simple polygon which separates a set of *M* red points from a set of *N* blue points in the plane (i.e. all the points of one color are inside the polygon or on its border and all the points of the other color are outside the polygon or on its border). We consider that the *M+N* points are in general position (i.e. no 3 points are collinear) and the distance between two points is the usual Euclidean distance.

We propose a novel approach for addressing this problem which consists of generating non-self-intersecting spanning trees and then convexifying them in order to obtain various separating polygons. Then we can adapt almost any generic optimization algorithm for this problem in order to consider the spanning tree rather than the solution polygon. This provides several advantages: it is very easy to generate spanning trees which lead to very different separating polygons, while generating the separating polygons directly would be a more cumbersome task.

The rest of this paper is structured as follows. In Section 2 we discuss related work. In Section 3 we discuss the issue of generating non-self-intersecting spanning trees of all the points of a given color. In Section 4 we present our convexification algorithm, considering two cases: (1) when the vertices of the separating polygon must consist of only the given points; (2) when we can add new points as vertices to the separating polygon. In Section 5 we show how we can include our spanning tree generation and convexification algorithm into several generic optimization methods for finding the minimum perimeter separating polygon. In Section 6 we present experimental results. In Section 7 we conclude and discuss future work.

2. RELATED WORK

The problem of computing the minimum perimeter polygon which separates a set of *M* red points from a set of *N* blue points has been well studied in the scientific literature. The problem is known to be NP-hard (Eades and Rappaport, 1993). An approximation algorithm with an $O(log(M+N))$ approximation ratio was presented in (Mata and Mitchell, 1995). A polynomial-time approximation scheme (PTAS) for this problem was presented in (Arora and Chang, 2003). Polygonizations of points of a given color which exclude the points of the other color were studied in (Fulek et al., 2010).

We are not aware of any previous publications which report attempts to compute minimum separating polygons by using generic optimization algorithms. In theory this should be possible by considering candidate separating

polygons and by defining operations which transform a separating polygon into another separating polygon which is "close" to the original one. However, defining such operations is not easy. Moreover, providing guarantees that the optimal separating polygon can be obtained by a sequence of these operations is difficult. On the other hand, it is easy to define such operations and provide such guarantees for our spanning tree-based approach.

Other related problems regarding constrained minimum perimeter enclosing polygons (Mitchell and Winters, 1991) and minimum perimeter polygon bipartitions (Provencal and Lachaud, 2009) were considered in the literature.

3. GENERATING NON-SELF-INTERSECTING SPANNING TREES

Let's consider the problem of generating non-self-intersecting spanning trees of all the points of the same color. Without loss of generality, we will consider that we need to generate non-self-intersecting spanning trees of the M red points. A spanning tree is defined to be non-self-intersecting if no two non-adjacent edges intersect. Two edges are adjacent if they share a common endpoint.

First of all, it is obvious that a minimum spanning tree of the M red points is non-self-intersecting (if two edges (u,v) and (w,t) of the minimum spanning tree intersect, then we could swap their endpoints and, thus, replace them with the edges (u,w) and (v,t), in order to obtain a spanning tree with a shorter total length, which would imply that the original spanning tree was not a minimum spanning tree). This leads us to a first approach for generating non-self-intersecting spanning trees: trying to generate spanning trees which are "close" to the minimum spanning tree, such that they do not contain self-intersections with a high probability. We considered a modified Prim's algorithm (Sedgewick and Wayne, 2011) for generating random spanning trees which are "close" to the minimum spanning tree. At each step of Prim's algorithm we have a set of points which are part of the spanning tree while the others are still outside of the spanning tree. For each point x outside of the spanning tree we have a potential parent y, which, in the standard version of Prim's algorithm, is the closest point to x which belongs to the spanning tree. We introduce the following two changes. At each step of the algorithm we first find the minimum distance $Dmin$ between a point outside the spanning tree and its potential parent. Then, all the points outside the tree whose distance to their potential parents is at most $C+Dmin$ will be considered candidates. C is a distance equal to a percentage CP of the average distance between all the pairs of red points. One of the candidates will be randomly chosen and added to the tree (by connecting it to its potential parent). Let the selected candidate be p. After this we consider all the points q which are still outside the spanning tree and compute their distance to p. If $distance(p,q)+C$ is smaller than the distance from q to its potential parent then we set p to be q's new potential parent. Otherwise, if $distance(p,q)$ is smaller than the distance from q to its potential parent we set p to be q's potential parent with a fixed probability $prob$.

A second possible approach is to generate any random spanning tree and then transform it into a spanning tree without self-intersections. The transformation can be performed by repeatedly applying *2-opt moves*, as was proposed in (Zhu et al., 1996) for transforming non-simple polygons into simple polygons on the same set of vertices.

In Fig. 1 we present a set of red and blue points and in Fig. 2, 3 and 4 we present three different non-self-intersecting spanning trees of the red points. We will use these three spanning trees in order to illustrate the convexification algorithm presented in the next section.

4. CONVEXIFYING NON-SELF-INTERSECTING SPANNING TREES

Let's consider that we have a non-self-intersecting spanning tree of the red points. It is easy to transform this tree into a non-simple separating polygon which contains all the red points on its border. Let's consider an arbitrary red point q and let's sort all of its tree neighbors r in ascending order according to the angle the segment r-q makes with the OX axis (from 0 to 2π). Let the order of the $K(q)$ tree neighbors of the red point q be $r(q,0), r(q,1), \ldots, r(q,K(q)-1)$. We will now perform an Euler tour (Tarjan and Vishkin, 1984) of the spanning tree starting from an arbitrary point u. The first edge to be traversed will be $(u, r(u,0))$. Then, let's assume that the most recent edge traversed was $(r(v,i), v)$. The next traversed edge will be $(v, r(v, (i+1) \text{ modulo } K(v)))$. The traversal will stop when the edge $(r(u, K(u)-1), u)$ is traversed. The initial polygon P will consist of all the traversed edges, in the order in which they are traversed. Note that P is not a simple polygon. Each point q will occur as a vertex of P a number of times equal to its degree $K(q)$ in the spanning tree.

After obtaining the initial polygon P corresponding to the spanning tree we will apply the following convexification algorithm:

1) Find three consecutive vertices u, v, w, on the border of P, such that v is a red point and the corner (u,v,w) can be convexified.

2) If no three consecutive vertices (u,v,w) such that the corner (u,v,w) can be convexified were found then stop. Otherwise convexify the corner (u,v,w) and then go back to step 1.

We will now discuss conditions for being able to convexify a corner (u,v,w) of P. First of all, the corner (u,v,w) must be a concave corner. The second condition depends on whether we can use new points as vertices of the separating polygon or only the original $M+N$ points can be used as vertices. We will obtain the set $S(u,v,w)$ of all the points located inside the triangle (u,v,w): points already located on the polygon border and blue points which are not among the polygon's vertices. We will compute the convex hull of the set

$$S'(u,v,w) = S(u,v,w) \cup \{u,w\} \qquad (1)$$

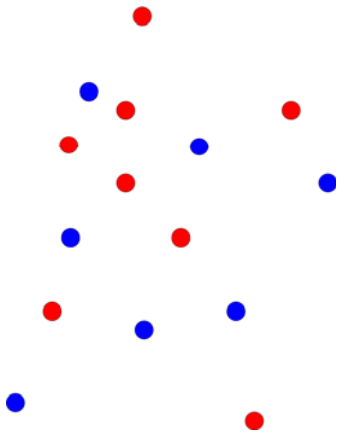

Fig. 1. A set of M=8 red and N=7 blue points.

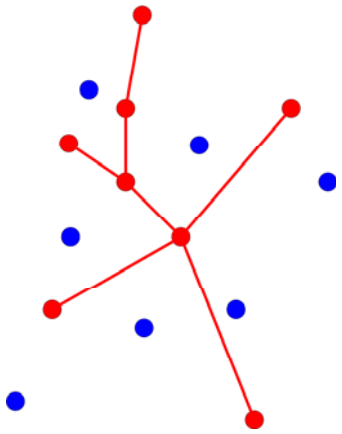

Fig. 2. Non-self-intersecting spanning tree 1 of the red points from Fig. 1.

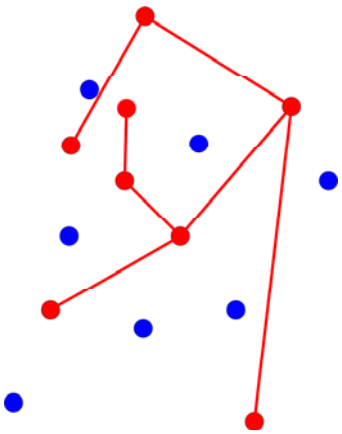

Fig. 3. Non-self-intersecting spanning tree 2 of the red points from Fig. 1.

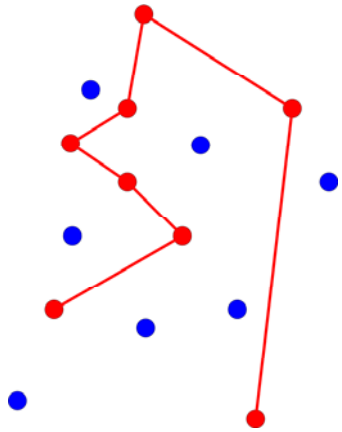

Fig. 4. Non-self-intersecting spanning tree 3 of the red points from Fig. 1.

The convex hull will consist of the segment *u-w* and a chain *C(u,v,w)* of points going from *u* to *w* (when *S(u,v,w)* is empty this chain is also the segment *u-w*). When new points cannot be used (case 1), then the second condition states that no other point from the polygon's border (except *u* and *w*) should be located on *C(u,v,w)*. When new points can be used (case 2) then there is no extra condition besides the corner *(u,v,w)* being a concave corner.

In case 1 we will replace the corner *(u,v,w)* by the chain *C(u,v,w)* on the border of the polygon *P*. In case 2 we will compute *C'(u,v,w)* by replacing each point *q* from *C(u,v,w)* which is already located on *P*'s border (except for *u* and *w*) by another point *q'* which is very close to *q*, but inside the polygon defined by *C(u,v,w)* and the segments *u-v* and *v-w*. We will consider these new points *q'* to be of a third color (e.g. violet). Then we will replace the corner *(u,v,w)* by the chain *C'(u,v,w)* on the border of *P*. Note that after this convexification operation the length of *P*'s perimeter decreases (because the length of *C(u,v,w)* or *C'(u,v,w)* is smaller than the sum of the lengths of the segments *u-v* and *v-w*).

An example of a successful corner convexification (case 1) is presented in Fig. 5 and an example of an impossible corner convexification (case 1) is presented in Fig. 6. Note how in Fig. 5 we can covexify the corner *(u,v,w)*, because *C(u,v,w)* consists of the sequence of points *u-x-y-w* which does not contain any other points from the polygon's border except *u* and *w* (despite the fact that *S(u,v,w)* contains the point *z* which belongs to the polygon's border). In Fig. 6 the corner *(u,v,w)* cannot be convexified because *C(u,v,w)* contains the point *x* which belongs to the polygon's border. When we switch from case 1 to case 2, the convexification of the corner *(u,v,w)* becomes possible by creating a new point *x'* very close to the point *x* (and inside the polygon *u-x-y-w-v*) and adding *x'* to the border of the polygon, as depicted in Fig. 7.

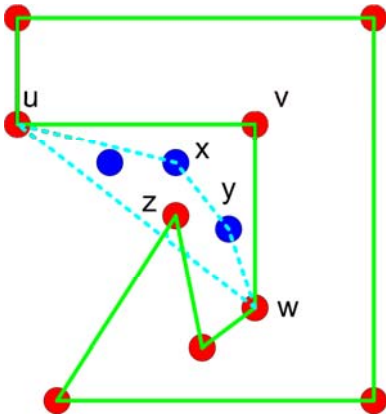

Fig. 5. Successful corner convexification (case 1).

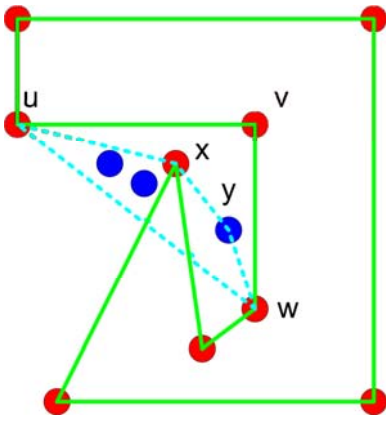

Fig. 6. Impossible corner convexification (case 1).

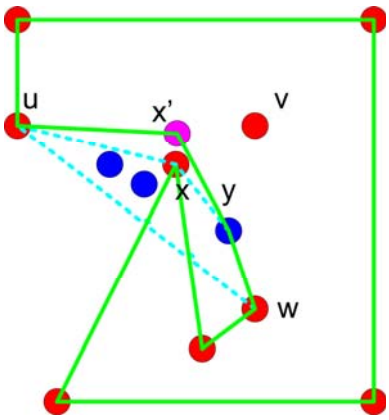

Fig. 7. Successful corner convexification (case 2).

In Fig. 8 we present the end result of the convexification algorithm (case 1) when applied to the spanning tree (and set of points) presented in Fig. 2. Solid red edges are spanning tree edges. Dashed greed edges are edges on the border of the separating polygon.

In Fig. 9 we present the result of the convexification algorithm (case 1) when applied on the spanning tree (and set of points) presented in Fig. 3. Solid red edges are spanning tree edges located inside the separating polygon, solid green edges are spanning tree edges located on the boundary of the separating polygon and dashed green edges are polygon edges not part of the spanning tree. Note how, in the end, the resulting polygon is not simple, because the concave corner *(u,v,w)* could not be convexified (because point *x* is part of *C(u,v,w)*) and, thus, points *v* and *w* occur twice on the boundary of the polygon. It is not easy to know before-hand (before starting the convexification procedure) if the convexification algorithm will end with a simple polygon *P* or not. However, this problem can be solved when considering case 2 (i.e. when we are allowed to add new points as vertices of the separating polygon), as depicted in Fig. 10. Note how, by adding a new point *x'* very close to point *x*, we are able to convexify the corner *(u,v,w)* (and then the corner *(x',w,y)*) and no point occurs multiple times on the boundary of the polygon. In fact, in case 2, we can start from the very beginning with a simple polygon. Let's consider the polygon *P* obtained from the Euler tour of the spanning tree. We will construct a new polygon *P'* as follows. For each corner *(r(q,i), q, r(q, (i+1) modulo K(q)))* the point *q* will be replaced by a copy *q(i)* of it which is located very close to *q*, in the wedge defined by its two neighbors *r(q,i)* and *r(q, (i+1) modulo K(q))*. For instance, a good candidate for placing *q(i)* is on the bisector of the angle formed by *r(q,i)* and *r(q, (i+1) modulo K(q))* with *q*, but very close to *q*. When *q* has only one spanning tree neighbor then *r(q,i)* = *r(q,(i+1) modulo K(q))*. In this case we consider the wedge to be the "exterior" wedge (whose angle is equal to 2π). The bisector of this angle goes in the opposite direction of the segment *(q,r(q,0))*. Fig. 11 shows the modified polygon *P'* for the spanning tree presented in Fig. 3, where the new points are the violet points. Then, the convexification algorithm will start from the modified polygon *P'* which is already a simple separating polygon.

The end result of the convexification algorithm applied to the spanning tree from Fig. 4 (considering case 1) is presented in Fig. 12. As in Fig. 9, solid red edges are spanning tree edges located inside the separating polygon, solid green edges are spanning tree edges located on the boundary of the separating polygon and dashed green edges are polygon edges not part of the spanning tree.

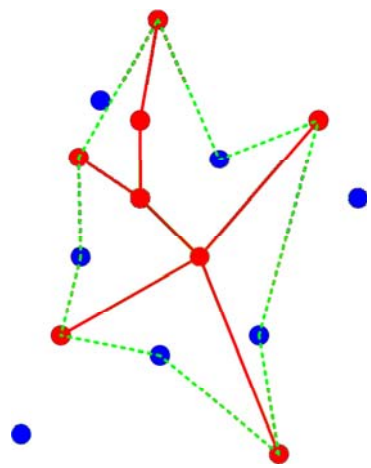

Fig. 8. End result of the convexification algorithm applied on the spanning tree and set of points from Fig. 2 (case 1).

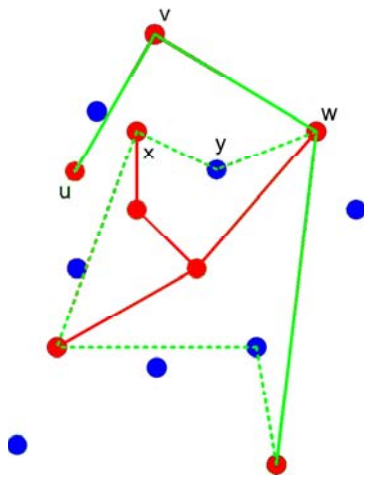

Fig. 9. End result of the convexification algorithm applied on the spanning tree and set of points from Fig. 3 (case 1).

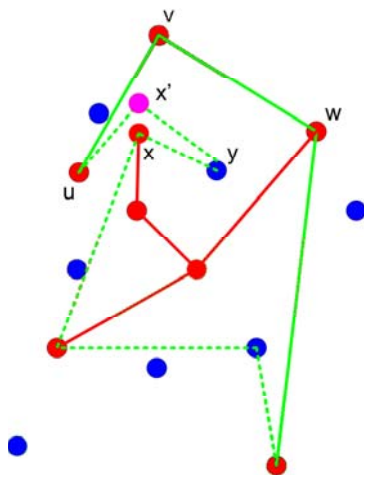

Fig. 10. End result of the convexification algorithm applied on the spanning tree and set of points from Fig. 3 (case 2).

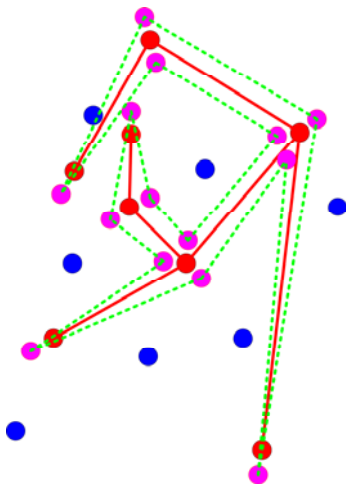

Fig. 11. Modified initial separating polygon (case 2).

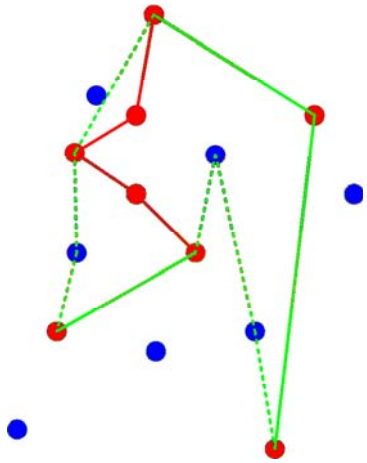

Fig. 12. End result of the convexification algorithm applied on the spanning tree and set of points from Fig. 4 (case 1).

We will now discuss about several implementation issues of the convexification algorithm presented in this section. First of all, in step 1, the algorithm does not specify which corner should be selected for convexification, in case there are multiple candidates. In general, the easiest approach would be to use a pointer *u* which tries to convexify the corner *(u, next(u), next(next(u)))* (we denoted by *next(v)* the next point on the polygon boundary after *v*). If the corner *(u, next(u), next(next(u)))* could be convexified, then we do not update the pointer (because, perhaps, another convexification operation could be performed here); otherwise, we advance the pointer *u* to *next(u)*. After the pointer was moved completely along the polygon boundary without performing any more convexification operations (i.e. since the last convexification operation or since the beginning if no such operation has been performed) we can stop. Of course, other approaches could be used – for instance, at each step, all the corners could be evaluated and the corner to be convexified could be the one which reduces the length of the polygon's perimeter the most.

A question which arises is if the order in which we perform the convexification operations matters for the end result. In case 1 (when no extra points can be used as polygon vertices), this order indeed matters. In Fig. 13 we show an example in which three corners, *(t,u,v)*, *(u,v,w)* and *(v,w,x)* can be convexified. If we convexify *(t,u,v)* or *(v,w,x)* first the final polygon will be the one depicted in Fig. 14. If we convexify *(u,v,w)* first the final polygon will be the one shown in Fig. 15.

In case 2 the order of the convexification operations does not matter and the end result will always be approximately the same (we say approximately because the locations of the new points will not necessarily be exactly the same, but they will be very close to some of the original points). It is easy to see that in case 2 the polygons from Fig. 14 and Fig. 15 can be convexified further to approximately the same final polygon (shown in Fig. 16).

For step 2 we need to find all the points located in a given triangle *(u,v,w)*. An easy $O(M+N)$ approach is to independently test each point on the polygon boundary and each blue point outside of the polygon. However, more efficient approaches can be used. For case 1 we can preprocess the points into a static data structure which can efficiently answer triangle range reporting queries. Data structures with $O(M+N)$ preprocessing and $O((M+N)^{0.5}+F)$ query time exist, as well as data structures with $O((M+N)^2)$ preprocessing and $O(log^2(M+N)+F)$ query time or $O((M+N)^{2+\varepsilon})$ preprocessing and $O(log(M+N)+F)$ query time (Matousek, 1994), (Chazelle et al., 1992), (Erickson, 2000), where F is the number of points located inside the query triangle. Computing the convex hull of the points from $S'(u,v,w)$ can be easily performed in $O(|S'(u,v,w)|\cdot log(|S'(u,v,w)|))$ time (Dave and Dave, 2008) or better if the points inside the triangle *(u,v,w)* are reported in sorted order.

In case 2, when new points can be created as vertices of the polygon, we may need to use dynamic data structures. A simple alternative is to use dynamic data structures for orthogonal range search (e.g. 2D range trees, quad-trees, kd-trees (de Berg et al., 2008), (Andreica and Tapus, 2010). Then, when performing a triangle query, we will first find all the points located in the minimum bounding box of the triangle (the sides of the bounding box are parallel to the OX and OY axes) and then we will keep only those which are also located inside the triangle.

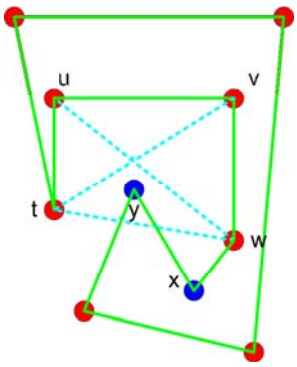

Fig. 13. Three corners can be convexified: *(t,u,v)*, *(u,v,w)* and *(v,w,x)*.

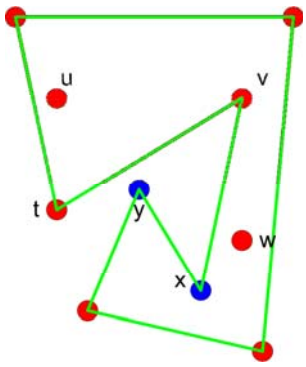

Fig. 14. Final polygon if we convexify the corners *(t,u,v)* or *(v,w,x)* first in Fig. 13 (considering case 1).

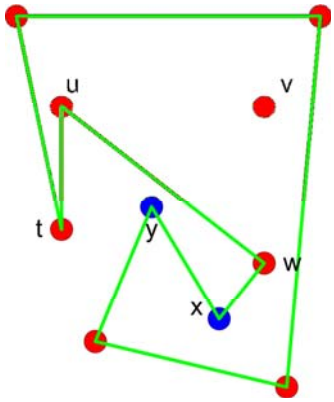

Fig. 15. Final polygon if we convexify the corner *(u,v,w)* first in Fig. 13 (considering case 1).

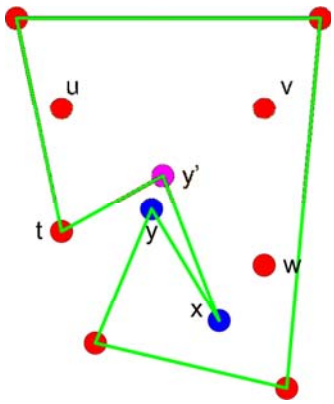

Fig. 16. Final polygon if we convexify the polygon depicted in Fig. 13 (considering case 2). The order of convexification operations does not matter.

5. FINDING A MINIMUM PERIMETER SEPARATING POLYGON

In this section we will assume that we want to compute a minimum perimeter separating polygon such that all the red points are inside or on the border of the polygon and all the blue points are outside or on the border of the polygon. Obviously, a minimum perimeter separating polygon may have either the red or the blue points inside. So, in order to try to find a minimum perimeter separating polygon we will need to run the algorithms presented below twice (once considering that the red points are kept inside the polygon and again considering that the blue points are kept inside the polygon). We will assume that there is a time limit for which the algorithms are allowed to run.

Algorithm 1 is an extremely simple algorithm. It will keep generating random non-intersecting spanning trees, convexify them and keep the best polygon found (*Popt*).

```
Algorithm 1.
(1)Popt = empty
(2)while the time limit was not exceeded do
```