Minimum Weight Triangulation by Cutting Out Triangles

Magdalene Grantson[1], Christian Borgelt[2], and Christos Levcopoulos[1]

[1] Department of Computer Science
Lund University, Box 118, 221 Lund, Sweden
{magdalene,christos}@cs.lth.se

[2] Department of Knowledge Processing and Language Engineering
University of Magdeburg, Universitätsplatz 2, 39106 Magdeburg, Germany
borgelt@iws.cs.uni-magdeburg.de

Abstract. We describe a fixed parameter algorithm for computing the minimum weight triangulation (MWT) of a simple polygon with $(n-k)$ vertices on the perimeter and k hole vertices in the interior, that is, for a total of n vertices. Our algorithm is based on cutting out empty triangles (that is, triangles not containing any holes) from the polygon and processing the parts or the rest of the polygon recursively. We show that with our algorithm a minimum weight triangulation can be found in time at most $O(n^3 k!\, k)$, and thus in $O(n^3)$ if k is constant. We also note that $k!$ can actually be replaced by b^k for some constant b. We implemented our algorithm in Java and report experiments backing our analysis.

1 Introduction

A *triangulation* of a set S of n points in the plane is a maximal set of non-intersecting edges connecting the points in S. A *minimum weight triangulation* (MWT) of S is a triangulation of minimum total edge length. (Note that a MWT need not be unique.) Although it is unknown whether the problem of computing a MWT of a set S of points is NP-complete or solvable in polynomial time [3], it is surely a non-trivial problem, for which no efficient (i.e. polynomial time) algorithm is known. The MWT problem has been studied extensively in computational geometry and has applications in computer graphics [11], image processing [10], database systems [8], and data compression [9].

In this paper we consider the slightly more general problem of finding a MWT of a simple polygon with $(n-k)$ vertices on the perimeter and k hole vertices in the interior, that is, for a total of n vertices. In this case a triangulation is a maximal set of non-intersecting edges (in addition to the perimeter edges), all of which lie inside the polygon. Note that the problem of finding the MWT of a set S of points can be reduced to this problem by finding the convex hull of S, which is then treated as a (convex) polygon, while all vertices not on the convex hull are treated as holes. Here, however, we do *not* require the polygon to be convex and thus solve a more general problem. Note also that for $k = 0$ (no holes) a MWT can be found by dynamic programming in time $O(n^3)$ [4, 7].

Recent attempts to give exact algorithms for computing a MWT in the general case exploit the idea of parameterization. The basis of such approaches is the notion of a so-called *fixed parameter algorithm*. Generally, such an algorithm has a time complexity of $O(n^c \cdot f(k))$, where n is the input size, k is a (constrained) parameter, c is a constant independent of k, and f is an arbitrary function [2]. The idea is that an algorithm with such a time complexity can be tractable if the parameter k is constrained. For example, for constant k the problem becomes efficiently tractable, because then the time complexity is polynomial in n.

W.r.t. a MWT of a simple polygon with holes the total number n of vertices is the size of the input and we may choose the number k of hole vertices as the constrained parameter. An algorithm based on such an approach was presented in [6] and analyzed to run in $O(n^5 \log(n)\, 6^k)$ time. In [5] we presented an improved algorithm, which we analyzed to run in $O(n^4 4^k k)$ time. We implemented our algorithm in Java and performed experiments backing our analysis.

In this paper we also present a fixed parameter algorithm for the MWT problem, which, however, is based on a different idea. The algorithm developed in [5] repeatedly splits a pointgon with lexi-monotone paths and processes the parts recursively. In contrast to this, our new algorithm repeatedly cuts empty triangles (that is, triangles not containing any holes) from the polygon and processes the parts or the remaining polygon recursively. We analyze that our algorithm runs in at most $O(n^3 k!\, k)$ time, and thus in $O(n^3)$ time if k is constant. We also note that $k!$ can actually be replaced by b^k for some constant b. Again we implemented our algorithm in Java and report experiments backing our analysis.

2 Preliminaries and Basic Idea

As already pointed out, we consider as input a simple polygon with $(n - k)$ vertices on the perimeter and k hole vertices, thus a total of n input vertices. Following [1], we call such a polygon with holes a *pointgon* for short. We denote the set of perimeter vertices by $V_p = \{v_0, v_1, \ldots, v_{n-k-1}\}$, assuming that they are numbered in counterclockwise order starting at an arbitrary vertex. The set of hole vertices we denote by $V_h = \{v_{n-k}, v_{n-k+1}, \ldots, v_{n-1}\}$. The set of all vertices is denoted by $V = V_p \cup V_h$, the pointgon formed by them is denoted by G.

The core idea of our algorithm is based on the following simple observation:

Observation 1 *With the definition $s(i) = (i+1) \bmod (n-k)$, let $e_i = (v_i, v_{s(i)})$, $1 \le i < n - k$, be an arbitrary edge on the perimeter of a pointgon G. Then in every triangulation T of G there exists a vertex $v \in V \backslash \{v_i, v_{s(i))}\}$, adjacent to v_i and $v_{s(i)}$ that together with v_i and $v_{s(i)}$ forms an empty triangle, that is, a triangle without any hole vertices in its interior.*

As a consequence of this observation, we can try to find the MWT of a given pointgon G—which is not a triangle without hole vertices itself—by checking all possible empty triangles that can be built over an arbitrarily chosen perimeter edge $(v_i, v_{s(i)})$. Each such empty triangle splits the pointgon into at most two sub-pointgons (see Figure 1), which are then processed recursively.

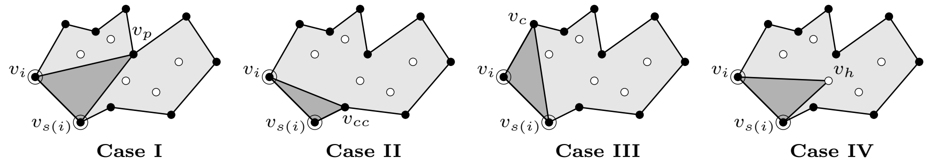

Fig. 1. The four possible ways of forming a triangle from a perimeter edge $(v_i, v_{s(i)})$ (encircled points). A black point indicates a vertex belonging to V_p, a white point a vertex belonging to V_h.

We consider four cases I, II, III, IV in our recursion, as shown in Figure 1. Note that cases II and III can be included as special cases into case I. In the following description, v_c is the vertex adjacent to v_i in clockwise direction, and v_{cc} is the vertex adjacent to $v_{s(i)}$ in counterclockwise direction.

I In this case we consider all empty triangles formed by v_i, $v_{s(i)}$, and a perimeter vertex $v_p \in V_p \backslash \{v_i, v_{s(i)}, v_c, v_{cc}\}$. Each such triangle $(v_i, v_{s(i)}, v_p)$ splits the pointgon G into two sub-pointgons, one to the left and one to the right of the triangle. We denote the set of all such triangles by $\Theta_p(G, v_i)$, where the second argument v_i indicates the base edge $e_i = (v_i, v_{s(i)})$ of the triangle. For each $\theta \in \Theta_p(G, v_i)$, we denote by $L(G, \theta)$ and $R(G, \theta)$ the sub-pointgons to the left and to the right of θ, respectively.

II In this case we consider the triangle $(v_i, v_{s(i)}, v_{cc})$, provided that it is empty. Cutting such a triangle from G yields the sub-pointgon $L(G, (v_i, v_{s(i)}, v_{cc}))$ to the left of the triangle. (There is no sub-pointgon to the right.)

III In this case we consider the triangle $(v_i, v_{s(i)}, v_c)$, provided that it is empty. Cutting such a triangle from G yields the sub-pointgon $R(G, (v_i, v_{s(i)}, v_c))$ to the right of the triangle. (There is no sub-pointgon to the left.)

IV In this case we consider all empty triangles formed by v_i, $v_{s(i)}$, and any hole vertex $v_h \in V_h$. Cutting any such triangle from the pointgon G leaves a sub-pointgon. We denote the set of all such triangles by $\Theta_h(G, v_i)$, where the second argument v_i indicates the base edge $e_i = (v_i, v_{s(i)})$ of the triangle. For each $\theta \in \Theta_h(G, v_i)$, we denote by $L(G, \theta)$ the remaining sub-pointgon.

Given this, the weight of a MWT of G can be computed recursively as

$$\begin{aligned} \mathrm{MWT}(G) = \min \Big\{ & \min_{\theta \in \Theta(G, v_i)} \left\{ \mathrm{MWT}(L(G,\theta)) + \mathrm{MWT}(R(G,\theta)) + |(v_i, v_{s(i)}| \right\}, \\ & \mathrm{MWT}(L(G, (v_i, v_{s(i)}, v_{cc}))) + |(v_i, v_{s(i)})| + |(v_{s(i)}, v_{cc})|, \\ & \mathrm{MWT}(R(G, (v_i, v_{s(i)}, v_c))) + |(v_i, v_{s(i)})| + |(v_i, v_c)|, \\ & \min_{\theta_h \in \theta(G, v_h)} \left\{ \mathrm{MWT}(L(G, \theta_h)) + |(v_i, v_{s(i)})| \right\} \Big\}, \end{aligned}$$

where i is an arbitrary integer number in $\{1, \ldots, n-k-1\}$. The four terms in the outer minimum refer to the cases I, II, III, and IV, respectively.

Although the above recursive formula only computes the weight of a MWT, it is easy to see how it can be extended to yield the edges of a MWT. For this, each recursive call also has to return the set of edges that is added in order to achieve a triangulation. The union of these sets of edges for the term that yields the minimum weight is a MWT for the original pointgon G.

Technically, one selects an arbitrary initial perimeter edge $(v_i, v_{s(i)})$ of the pointgon G to be triangulated. Then all triangles listed in the above four cases I, II, III, and IV are generated. Each of these is cut out of the pointgon and the parts or the rest is processed recursively. The minimum weight triangulation is obtained as the minimum over all resulting triangulations.

3 Dynamic Programming

The basic idea, as it was outlined in the preceding section, solves the MWT problem of an input pointgon G in a recursive manner. However, it is immediately clear that it should not be implemented in this way, because several branches of the tree recursion lead to the same sub-pointgon. Hence a direct implementation would lead to considerable redundant computations. A better approach consists in using dynamic programming, which ensures that each possible subproblem is solved at most once, thus rendering the computation much more efficient.

To apply dynamic programming, we have to identify the classes of subproblems that we encounter in the recursion, and we have to find a representation for them. The basic idea is to exploit the fact that we can choose freely, which perimeter edge of a sub-pointgon we want to use as the base of the triangles. By always choosing an edge that was a side of the triangle chosen in the preceding recursion step (preferring the left side if both are in the new subproblem), we can achieve that the sides of the triangles chosen in consecutive recursion steps connect into a path through the holes (of the input pointgon). As a consequence, every subproblem we encounter can be described by a path π through zero or more hole vertices and a possible coherent perimeter piece δ_p of G (see Figure 2). We show later that we actually encounter only subproblems of this type.

We represent a subproblem by an index word over an alphabet with n characters, which uniquely identifies each subproblem. This index word has the general form (v_i, π) and describes a counterclockwise walk round the perimeter of the subproblem. The first element is the so-called *anchor* v_i of the subproblem, which may be a perimeter vertex or a hole vertex of the input pointgon. π describes a sequence of hole vertices of the input pointgon, i.e., all elements of π are in V_h, with the possible exception of the last element, which may be a perimeter vertex $s_\pi \in V_p$ of the input pointgon. Note that in this case a possible coherent perimeter piece between the anchor v_i and the end vertex s_π (if it exists) is not part of the subproblem representation, but is left implicit. Note that in the case where the path π bounding a subproblem loops back to the anchor, this end vertex is not contained in π to avoid duplicate entries. As a consequence we can have at most $O(n^2 k!)$ index words, where the n^2 comes from the possible choices of the anchor and the end vertex, and the $k!$ from the possible sequences of holes.

The general idea of using such an index word is the following: in order to avoid redundant computations, we have to be able to efficiently store and retrieve the solutions of already solved subproblems. For the problem at hand it is most convenient to use a trie structure, which is accessed through the index word representing a subproblem. That is, for our implementation, we do not use the standard, table-based form of dynamic programming, but a version of implementing the recursion outlined above, which is sometimes called "memorized". In each recursive call, we first access the trie structure (using the index word of a sub-pointgon) in order to find out whether the solution of the current subproblem is already known. If it is, we simply retrieve and return the solution. Otherwise we actually carry out the split computations and in the end store the found solution in the trie. Although this approach is slightly less efficient than a true table-based version (since there are superfluous accesses to the trie, namely the unsuccessful ones), its additional costs do not worsen the asymptotic time complexity. The following pseudocode describes our algorithm:

function MWT (word *key*) : real
begin
  if *key* is in *trie*
  then return *trie*.getweight(*key*); **fi**;
  if polygon(*key*) is a triangle
  and polygon(*key*) contains no holes
  then *wgt* = perimeter_length(polygon(*key*));
    trie.add(*key*, *wgt*, $\perp$);
    return *wgt*;
  fi;
  $min = +\infty$; $best = \perp$;
  for all triangles $\theta \in \Theta_p(\text{polygon}(key), key.v_i)$ (∗ case I ∗)
    $\cup \{(key.v_i, key.v_{s(i)}, v_c(key.v_i))\}$ (∗ case II ∗)
    $\cup \{(key.v_i, key.v_{s(i)}, v_{cc}(key.v_i))\}$ (∗ case III ∗)
    $\cup\, \Theta_h(\text{polygon}(key), key.v_i)$ **do** (∗ case IV ∗)
    $wgt = \text{MWT}(\text{L}(key,\theta)) + \text{MWT}(\text{R}(key,\theta)) + |v_i, v_{s(i)}|$;
    if $(wgt < min)$
    then $min = wgt$; $best = \theta$; **fi**;
  done;
  trie.add(*key*, *min*, *best*);
  return *min*;
end;

In this pseudocode the symbol $\perp$ is used to indicate that there is no triangle θ that can be cut out. (This is, of course, only the case if the sub-pointgon is itself a triangle without holes.) $key.v_i$ and $key.v_{s(i)}$ are the anchor and the first vertex on the path representing a subproblem. v_c and v_{cc} are to be seen as functions yielding the clockwise or the counterclockwise neighbor on the perimeter of the subproblem. Note that if we are in one of the cases II, III, or IV, one of $\text{L}(key,\theta)$ or $\text{R}(key,\theta)$ is empty. It should be clear that in this case the corresponding term is dropped from the sum computing the triangulation weight.

Fig. 2. The two types of pointgons we encounter. A black point indicates a vertex belonging to V_p, a white point a vertex belonging to V_h, a grey point a vertex either in V_p or in V_h. The encircled and labeled points are the anchors. Thick lines indicate pieces of the perimeter of the input pointgon.

To collect the edges of the solution, the following function is used:

function collect (word *key*) : set of edges
begin
  $\theta = \textit{trie}.\text{gettriangle}(\textit{key})$;
  if $\theta = \perp$ **then return** $\emptyset$; **fi**;
  return $\text{collect}(\text{L}(\textit{key}, \theta)) \cup \text{collect}(\text{R}(\textit{key}, \theta)) \cup \text{edges}(\theta)$;
end;

This function is called with the index word describing the input pointgon, that is, an index word consisting only of the initial anchor vertex v_i.

3.1 Types of Pointgons

Apart from the input pointgon, which is of neither of these types, we encounter two types of sub-pointgons (see Figure 2 for sketches):

- **A** Sub-pointgons of this type are bounded by a coherent perimeter piece of the original pointgon and a path π through zero or more hole vertices. Its anchor is the vertex at the clockwise end of the path (perimeter vertex).
- **B** Sub-pointgons of this type are bounded by a path π through hole vertices and zero or one perimeter vertex $v_p \in V_p$ of the input pointgon. Its anchor is the lexicographically smallest vertex (hole or perimeter vertex).

We show now that we only encounter these two types of pointgons in the recursion. For this, recall that in each step of the recursion, one of the four cases I, II, III, and IV is used to cut a triangle from a pointgon (see Section 2).

Type A Pointgons (Note that the input pointgon, although strictly speaking it is not of this type, behaves exactly in the same way.) The different splits of type A pointgons are sketched in Figure 3. Note that the triangle is always formed over the edge at the clockwise end of the path through hole vertices. Therefore, if we have a triangle of case I, we obtain either two sub-pointgons of type A or one sub-pointgon of type A and B each. If the triangle is of case II, then depending on the number of vertices in the coherent perimeter section the result is either a sub-pointgon of type A or of type B. Type B results if there are only two perimeter vertices in the section, type A otherwise. If the triangle is of case III or case IV, the result is necessarily of type A again. Note from the sketches in Figure 3 how the anchors of the sub-pointgons are selected.

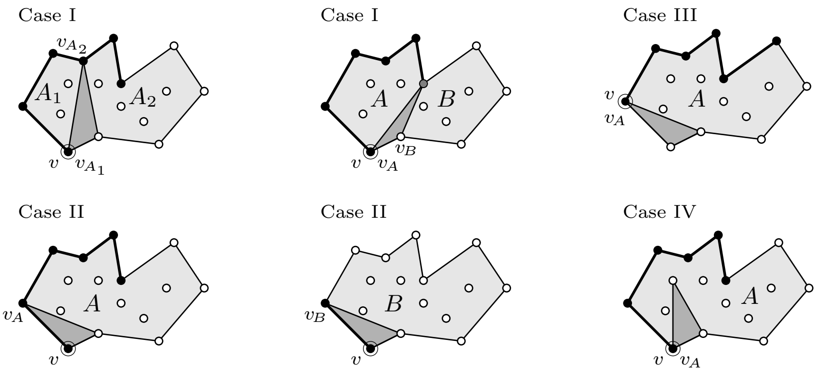

Fig. 3. Behavior of Type A pointgons in the recursion (possible splits). v denotes the original anchor and v_*, $* \in \{A, B\}$, denotes the anchor of a sub-pointgon of type $*$ due to the split.

Type B Pointgons Since type B pointgons do not have a consecutive perimeter piece of the input pointgon (they may contain at most one perimeter vertex), it is immediately clear that regardless of the case we consider, the resulting sub-pointgon(s) must be of type B again (cf. Figure 1, which shows the four cases). It is only important to notice that in a sub-pointgon to the right of the triangle, the anchor has to be set to the lexicographically smallest vertex on the perimeter of the sub-pointgon. (Note that in a sub-pointgon to the left of the triangle the old anchor must still be the lexicographically smallest vertex.)

4 Analysis

In order to estimate the time complexity of our algorithm, we multiply the (worst case) number of subproblems by the (worst case) time it takes to process one subproblem. This time is computed as the (worst case) number of possible triangles that can be built over a given perimeter edge multiplied with the (worst case) time for processing one such triangle.

As discussed in Section 3, there are at most $O(n^2k!)$ possible subproblems if we proceed in the described manner. The factor $k!$ is actually an overestimate as the path through the holes must not intersect itself. Drawing on bounds for planar configurations of a set of points [12], it can be shown that the actual number of valid paths is only $O(b^k)$ for some constant b. However, the best known asymptotic bounds on the constant b are fairly large, so that for practical purposes, for which k has to be very small, $O(k!)$ is a better estimate.

Given a subproblem and a perimeter edge of this subproblem, there are at most $n-2$ possible triangles that can be built over the given edge, as there are at most $n-2$ possible choices for the tip of the triangle.

For each triangle, we have to check whether its two additional sides (the base of the triangle, of course, need not be checked) intersects the perimeter of the sub-pointgon. This check can be made very efficient by a preprocessing step in which we determine for each edge that could be part of a triangle whether it intersects the perimeter of the input pointgon or not. The resulting table has a size of at most n^2, which is negligible compared to the number of subproblems, and its computation takes at most $O(n^3)$ time, because each of the edges has to be checked against the $n - k$ perimeter edges. With this table we can check in $O(1)$ whether a given triangle intersects a (possibly existing) perimeter piece.

In addition, we have to check for an intersection of the two added triangle sides with the path through hole vertices, which contains at most $k + 1$ edges. Since we have to check for each of the edges on the path whether it intersects with one of the two triangle sides, this check can be carried out in $O(k)$.

Finally we have to check whether the triangle is empty (i.e., contains no hole vertices). This also takes at most $O(k)$ time, because we have to check the location of at most k hole vertices w.r.t. the triangle sides.

Once a triangle is found to be valid, the sub-pointgons have to be constructed by collecting their at most $k+2$ defining vertices, and their solutions have to be looked up using the describing index words. Both operations obviously take $O(k)$ time. Finally the length of the base of the triangle and maybe also the length of one of the added sides have to be computed, which takes constant time. As a consequence processing one triangle takes in all $O(k)$ time.

As a consequence the overall time complexity is at most

$$O(\underbrace{n^2 k!}_{\text{subproblems}} \cdot \underbrace{n}_{\text{triangles}} \cdot \underbrace{k}_{\text{time/triangle}}) = O(n^3 k!\, k).$$

5 Implementation

As already pointed out above, we implemented our algorithm in Java. Example results for different numbers of holes and perimeter vertices are shown in Table 1. The test system was an Intel Pentium 4C@2.6GHz with 1GB of main memory running S.u.S.E. Linux 9.3 and Sun Java 1.5.0_01. All execution times are averages of 20 runs, carried out on randomly generated convex pointgons. We used convex pointgons, because they seem to represent the worst case. Non-convex pointgons, due to intersections of paths through holes with the perimeter, are usually processed faster (there are fewer possible sub-pointgons). The left table shows the results for the algorithm as it was described in the preceding sections, while the right table shows the results for the algorithm we developed in [5]. As these tables show, our new algorithm beats the previous one by an order of magnitude, making it the method of choice for few hole vertices.

To check our theoretical result about the time complexities, we computed the ratios of the measured execution times to the theoretical values (see last columns of both tables; note that the two algorithms have different theoretical time complexities). As can be seen, these ratios are decreasing for increasing values of n and k, indicating that the theoretical time complexity is actually a worst case, while average results in practice are considerably better. As pointed out above, it can even be shown that the dependence on k is actually only $O(b^k)$ for some constant b, and thus asymptotically better than $O(k!) = O(k^k)$.

$n-k$	k	time in seconds	time/$n^3k!\,k$
3	1	0.008± 0.000	$1.250\cdot10^{-4}$
6	1	0.009± 0.000	$2.624\cdot10^{-5}$
9	2	0.014± 0.001	$2.630\cdot10^{-6}$
12	3	0.039± 0.005	$4.280\cdot10^{-7}$
15	4	0.060± 0.004	$3.417\cdot10^{-8}$
18	5	0.092± 0.017	$2.420\cdot10^{-9}$
21	6	0.272± 0.102	$2.962\cdot10^{-10}$
24	7	0.885± 0.285	$3.607\cdot10^{-11}$
27	8	2.849± 0.818	$3.961\cdot10^{-12}$
30	9	12.791± 5.480	$5.566\cdot10^{-13}$

$n-k$	k	time in seconds	time/n^44^kk
3	1	0.010± 0.000	$9.766\cdot10^{-6}$
6	1	0.011± 0.000	$1.145\cdot10^{-6}$
9	2	0.049± 0.004	$1.046\cdot10^{-7}$
12	3	0.076± 0.006	$7.819\cdot10^{-9}$
15	4	0.151± 0.029	$1.132\cdot10^{-9}$
18	5	0.535± 0.144	$3.734\cdot10^{-10}$
21	6	2.115± 0.869	$1.619\cdot10^{-10}$
24	7	9.142± 3.982	$8.631\cdot10^{-11}$
27	8	43.548±19.198	$5.535\cdot10^{-11}$
30	9	176.588±71.473	$3.235\cdot10^{-11}$

Table 1. Results obtained with our Java implementations of the described MWT algorithm (left) and the path-based algorithm of [5] (right, version without hole distribution). All results are averages over 20 runs, with randomly generated convex pointgons.

To give an impression of the graphical user interface (GUI) of the program, Figure 4 shows a screen shot of the main window. With this user interface it is possible to load pointgons from text files, to generate random pointgons, to find their minimum weight triangulation, and to modify a triangulation as well as to recompute its weight in order to check whether it is actually minimal. Apart from the GUI version, the program can be invoked on the command line, a feature we exploited to script the test runs reported above. The Java source code of our implementation as well as an executable Java archive (jar) can be downloaded free of charge at `http://fuzzy.cs.uni-magdeburg.de/~borgelt/pointgon.html`.

6 Conclusions

We described a fixed parameter algorithm for computing the minimum weight triangulation of a simple polygon with hole vertices, which is based on recursively cutting out empty triangles from the input pointgon. As we showed in our analysis, the time complexity of our algorithm is $O(n^3k!\,k)$. (Note that we use $k!$ instead of b^k, for some constant b, because in known upper bounds on the number of crossing-free paths of a set of points the constant b is so large that b^k is worse than $k!$ for the small values of k that are practically relevant.) W.r.t. the total number of vertices it is therefore much better than the algorithm we developed in [5] (time complexity $O(n^44^kk)$). An important advantage of our algorithm is that for a constant number of holes (and in particular for no holes, i.e. for $k = 0$) it achieves the best known $O(n^3)$ time bound for the minimum weight triangulation of a simple polygon. In addition, we presented a Java implementation of our algorithm, and reported experiments that were carried out with this implementation. These experiments indicate that the actual time complexity may be considerably better than the result of our theoretical analysis (presumably due to an overestimate of the number of subproblems).

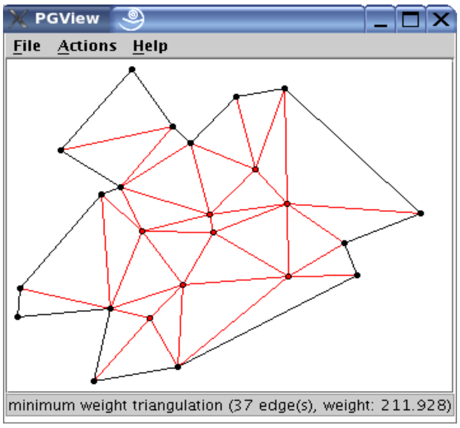

Fig. 4. A screen shot of the graphical user interface to our implementation of the described algorithm. The program allows for loading pointgons from a text file, generating random pointgons, finding their minimum weight triangulation, and modifying a triangulation as well as recomputing its weight in order to check whether it is actually minimal. When a MWT is computed, search statistics are printed about the number of triangles, subproblems, and separating paths considered. This screen shot shows the MWT of a randomly generated (star-like) pointgon with 16 vertices on the perimeter and 8 holes, which has 37 edges (excluding the perimeter edges). It was computed in about 0.3 seconds (on an Intel Pentium 4C@2.6GHz system with 1GB of main memory running S.u.S.E. Linux 9.3 and Sun Java 1.5.0_01).

References

1. O. Aichholzer, G. Rote, B. Speckmann, and I. Streinu. The Zigzag Path of a Pseudo-Triangulation. *Proc. 8th Workshop on Algorithms and Data Structures (WADS 2003), LNCS 2748*, 377–388. Springer-Verlag, Berlin, Germany 2003
2. R. Downey and M. Fellows. *Parameterized Complexity.* Springer-Verlag, New York, NY, USA 1999
3. M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to Theory of NP-Completeness.* Freeman, New York, NY, USA 1979
4. P.D. Gilbert. New Results in Planar Triangulations. *Report R-850.* University of Illinois, Coordinated Science Lab, 1979
5. M. Grantson, C. Borgelt and C. Levcopoulos. A Fixed Parameter Algorithm for Minimum Weight Triangulation: Analysis and Experiments. Technical Report LU-CS-TR:2005-234, ISSN 1650-1276 Report 154. Lund University, Sweden 2005
6. M. Hoffmann and Y. Okamoto. The Minimum Triangulation Problem with Few Inner Points. *Proc. 1st Int. Workshop on Parameterized and Exact Computation (IWPEC 2004), LNCS 3162*, 200–212. Springer-Verlag, Berlin, Germany 2004
7. G.T. Klincsek. Minimal Triangulations of Polygonal Domains. *Annals of Discrete Mathematics* 9:121–123. ACM Press, New York, NY, USA 1980
8. E. Lodi, F. Luccio, C. Mugnai, and L. Pagli. On Two-Dimensional Data Organization, Part I. *Fundaments Informaticae* 2:211–226. Polish Mathematical Society, Warsaw, Poland 1979
9. A. Lubiw. The Boolean Basis Problem and How to Cover Some Polygons by Rectangles. *SIAM Journal on Discrete Mathematics* 3:98–115. Society of Industrial and Applied Mathematics, Philadelphia, PA, USA 1990
10. D. Moitra. Finding a Minimum Cover for Binary Images: An Optimal Parallel Algorithm. *Algorithmica* 6:624–657. Springer-Verlag, Heidelberg, Germany 1991
11. D. Plaisted and J. Hong. A Heuristic Triangulation Algorithm. *Journal of Algorithms* 8:405–437 Academic Press, San Diego, CA, USA 1987
12. M. Sharir and E. Welzl. On the Number of Crossing-Free Matchings (Cycles and Partitions), *Proc. 17th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2006)*, to appear.