A Fixed Parameter Algorithm for Minimum Weight Triangulation: Analysis and Experiments

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. We show that with our algorithm the minimum weight triangulation can be found in time at most $O(n^4 4^k k)$, and thus in time polynomial in n if $k \leq O(\log n)$. 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 [10], image processing [9], database systems [7], and data compression [8].

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, 6].

Published as Technical Report LU-CS-TR:2005-234 on April 20, 2005
ISSN 1650-1276 Report 154, Lund University, Sweden 2005

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 the time complexity is then 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. A first algorithm based on such an approach was presented in [5] and analyzed to run in $O(n^5 \log(n)\, 6^k)$ time. In this paper we describe a fixed parameter algorithm that is inspired by the basic idea of this algorithm, but deviates in some respects. Due to improvements in both the algorithm and its analysis, we are able to show that the time needed to find a MWT of a polygon with holes is at most $O(n^4 4^k k)$. In addition, we implemented our algorithm in Java and performed 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 $V_p = \{v_1, v_2, \ldots, v_{n-k}\}$, assuming that they are numbered in counterclockwise order starting at an arbitrary vertex. The set of hole vertices we denote by $V_h = \{v_{n-k+1}, v_{n-k+2}, \ldots, v_n\}$. The set of all vertices is denoted by $V = V_p \cup V_h$, the pointgon formed by them is denoted by G.

Definition 1. *A vertex $u \in V$ is said to be* **lexicographically smaller** *than a vertex $v \in V$, written $u \prec v$, iff one of the following conditions is satisfied:*

1. *The x-coordinate of u is smaller than the x-coordinate of v.*
2. *The x-coordinate of u is equal to the x-coordinate of v,*
   but the y-coordinate of u is smaller than the y-coordinate of v.

W.l.o.g. we assume that the hole vertices (i.e. the vertices in V_h) are in lexicographical order, that is, we assume that $\forall i; n-k < i < n : v_i \prec v_{i+1}$. (Otherwise we can sort and renumber them, incurring negligible computation costs.)

Definition 2. *A path in a pointgon G, defined by a sequence of vertices from V, is called* **lexi-monotone** *iff the sequence of vertices is either lexicographically increasing or lexicographically decreasing. A* **separating lexi-monotone path** *(or simply a "separating path") of a pointgon G is a lexi-monotone path with start and end vertices on the perimeter of G (i.e. vertices in V_p) and a (possibly empty) sequence of hole vertices (i.e. vertices in V_h) in the middle, which does not intersect the perimeter of G (start and end vertex do not count as intersections).*

With these definitions, the core idea of our algorithm (as well as the core idea of the algorithm in [5]) is based on the following simple observation:

Let $v \in V_p$ be an arbitrary vertex on the perimeter of a pointgon G. Then in every triangulation T of G there exists: **either** *a separating path π starting at v* **or** *two perimeter vertices v_c and v_{cc} that are adjacent to v and that together with v form a triangle without any hole vertices in its interior.*

As a consequence, we can try to find the MWT of a given pointgon G, which is not a triangle without hole vertices, with the following recursion that considers possible splits of G into two sub-pointgons: In the first place we consider all separating paths starting at an arbitrarily chosen vertex $v \in V_p$. Each such path splits the pointgon into two sub-pointgons, one to the left and one to the right of the separating path. Secondly, we consider the special path that connects the two perimeter vertices that are adjacent to v and thus "cuts off" v from the rest of the pointgon, provided that the triangle formed by v and its adjacent perimeter vertices does not contain any hole vertices. In any case we have two sub-pointgons, which can be processed recursively. The minimum weight triangulation is then obtained as the minimum over all these splits [5].

Formally, we can describe the solution procedure as follows: Let G be a given pointgon and $v \in V_p$ an arbitrary vertex on the perimeter of G. Let $\Pi(G, v)$ be the set of all separating paths of G starting at v. If $\pi \in \Pi(G, v)$ is a separating path, let $|\pi|$ be the length of π (that is, the sum of its edge lengths) and $L(G, \pi)$ and $R(G, \pi)$ the sub-pointgons to the left and to the right of π, respectively. Furthermore, let v_c and v_{cc} be the perimeter vertices that are adjacent to v in clockwise and counterclockwise direction, respectively. Then the weight of a MWT of G can be computed recursively as

$$\begin{aligned}\mathrm{MWT}(G) = \min\Big\{ &\min_{\pi \in \Pi(G,v)} \{\mathrm{MWT}(L(G,\pi)) + \mathrm{MWT}(R(G,\pi)) - |\pi|\}, \\ &\mathrm{MWT}(R(G,(v_{cc}, v_c))) + |(v, v_{cc})| + |(v, v_c)|\Big\}.\end{aligned}$$

The first term in the outer minimum considers all splits by separating lexi-monotone paths. Note that we have to subtract the length of the path π from the sum of the minimum triangulation weights of the two sub-pointgons, because this path is part of both sub-pointgons. The second term in the outer minimum refers to the special path (v_{cc}, v_c) that "cuts off" v from the rest of the pointgon. Note that in this second term only one recursion is necessary, because we consider it only if the triangle formed by v, v_{cc}, and v_c does not contain any hole vertices. Obviously, for such a triangle no recursive processing is necessary. Note also that the length of the third edge (v_{cc}, v_c) of the triangle is contained in $\mathrm{MWT}(R(G,(v_{cc}, v_c)))$, so that it need not be added.

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.

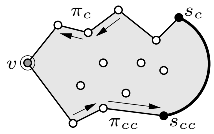

Fig. 1. A sub-pointgon is represented by a counterclockwise walk round its perimeter, following the at most two lexi-monotone bounding paths. The encircled vertex is the anchor, the thick line represents a coherent perimeter piece.

3 Dynamic Programming

The basic idea, as it was outlined in the preceding section, solves the MWT problem 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 different subproblems that we meet in the recursion, and we have to find a representation for them. The core idea here is the following: if in the recursion we prefer to use the same vertex v for attaching separating paths as in the preceding split, every subproblem we encounter can be described by one or two lexi-monotone paths that start at the same vertex v (which we call the *anchor* of the subproblem) and a coherent piece of the perimeter of the input pointgon (see Figure 1). A detailed analysis of this statement, providing a proof, will be given later.

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, π_{cc}, π_c) and describes a counterclockwise walk round the perimeter of the subproblem. The first element is the anchor v, which may be either a perimeter vertex or a hole vertex of the input pointgon and thus can have n possible values. π_{cc} and π_c describe the sequences of hole vertices of the input pointgon that are on the separating paths. All elements of π_{cc} and π_c are in V_h—with the possible exception of the last elements, which may be perimeter vertices s_{cc} and s_c, respectively. Note that either π_{cc} or π_c or both may be empty. Note also that in the case where the separating paths bounding a subproblem end at a hole vertex of the input pointgon, or at the same perimeter vertex, this end vertex is contained only in π_{cc} to avoid duplicate entries.[3] Finally, note that the vertices in a coherent perimeter piece between the end vertices s_{cc} and s_c (if it exists) are not part of the subproblem representation, but are left implicit.

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

[3] It is, of course, an arbitrary choice to include it only in π_{cc}. One may just as well decide to include it only in π_c.

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 to 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, ⊥);
           return wgt;
    fi;
    min = +∞; best = ⊥;
    for all paths π ∈ Π(polygon(key), key.v) ∪ {(v_cc(key.v), v_c(key.v)} do
        wgt = MWT(L(key, π)) + MWT(R(key, π)) − |π|;
        if     (wgt < min)
        then min = wgt; best = π; fi;
    done;
    trie.add(key, min, best);
    return min;
end;
```

In this pseudocode the symbol $\perp$ is used to indicate that there is no separating path. To collect the edges of the solution, the following function is used:

```
function collect (word key) : set of edges
begin
    π = trie.getpath(key);
    if π = ⊥ then return ∅; fi;
    return collect(L(key, π)) ∪ collect(R(key, π)) ∪ edges(π);
end;
```

Before we proceed with a detailed analysis of the different types of pointgons and how they are handled in our algorithm, it is worthwhile to make a few remarks about how to code index words representing sub-pointgons in a computer. The simplest way would be to use lists of vertices for π_{cc} and π_c. However, from a theoretical point of view, this has the disadvantage that in this case each element needs $\log k$ space, leading to a worst case space complexity of $O(k \log k)$. A better way would be to use bit vectors of length k, each bit of which corresponds to

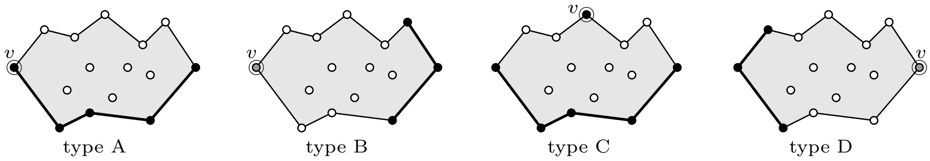

Fig. 2. The four types of pointgons we encounter. A black point indicates a vertex belonging to V_p, a white point a vertex belonging to V_h, and a gray point a vertex belonging to $V = V_p \cup V_h$. The encircled point is the anchor, thick lines indicate pieces of the perimeter of the input pointgon.

one hole vertex. If a bit is set, the corresponding hole is on the path, otherwise it is not. In this case the worst case space complexity is only $O(k)$. For an implementation, however, such a way of coding the vertex sequences π_{cc} and π_c is, at best, inconvenient. Therefore we use the simple list representation, relying on the fact that a real-world computer has limited memory and thus in practice every vertex is coded with a fixed amount of memory even in this case.

4 Types of Pointgons

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

A Sub-pointgons of this type have only one separating path starting at the anchor v, which must be on the perimeter of the input pointgon. The vertices on the path are lexicographically increasing. In addition, there is a coherent perimeter piece of the input pointgon.

B Sub-pointgons of this type are bounded by two separating paths starting at the anchor v, which may be either a perimeter vertex or a hole vertex of the input pointgon. The vertices on both paths are lexicographically increasing. There may or may not be a coherent perimeter piece of the input pointgon. (Note that it may consist of only a single perimeter vertex as a special case.)

C Sub-pointgons of this type are bounded by two separating paths starting at the anchor v, which must be a perimeter vertex of the input pointgon. One of the two paths is lexicographically increasing, the other decreasing. As a consequence there must be a perimeter piece of the input pointgon.

D Sub-pointgons of this type are bounded by two separating paths starting at the anchor v, which may be either a perimeter vertex or a hole vertex of the input pointgon. The vertices on both paths are lexicographically decreasing. There must be a perimeter piece of the input pointgon, which contains at least two vertices (with only one vertex it could be turned into type B).

The general principle of the choice of the anchor is that it is the leftmost vertex on the separating path if there is just one path, and the vertex that is on both paths if there are two separating paths. If there are two vertices that are on both paths (because they share both start and end vertex), we choose the leftmost one.

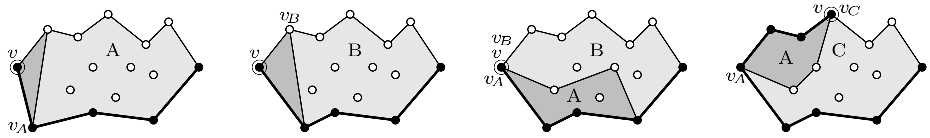

Fig. 3. Behavior of Type A pointgons in the recursion (possible splits). v denotes the original anchor and v_*, $* \in \{A, B, C\}$, denotes the anchor of a sub-pointgon of type $*$ due to the split.

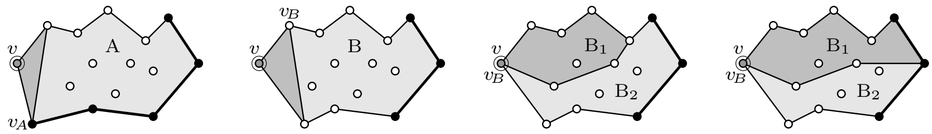

Fig. 4. Behavior of Type B pointgons in the recursion (possible splits). v denotes the original anchor and v_*, $* \in \{A, B\}$, denotes the anchor of a sub-pointgon of type $*$ due to the split.

For the input pointgon, we choose the lexicographically smallest perimeter vertex as the anchor (just for convenience). Regardless of the path we use to split this pointgon (whether it starts at the anchor or cuts off the anchor), we obtain a sub-pointgon of type A for both subproblems. The other types of pointgons can only be created in deeper levels of the recursion. In the following we consider how these types of pointgons are treated in the recursion in our algorithm and thus also prove that these are the only types that occur.

4.1 Type A Pointgons

The different splits of a type A pointgon are sketched in Figure 3. On the very left a path "cutting off" the anchor, which is seen as leading from the counterclockwise neighbor of v to its clockwise neighbor, can be merged with the existing separating path to give a new type A pointgon. Otherwise, we obtain a type B pointgon (second sketch). For a path starting at the anchor, we distinguish whether it is lexicographically increasing (third sketch) or decreasing (fourth sketch, note the different anchor). In the former case, we obtain one type A and one type B pointgon, which receive the same anchor as the original pointgon. In the latter case, we obtain one type A pointgon, with its anchor at the end of the new separating path, and one type C pointgon, with its anchor equal to that of the original pointgon. Note that all cases may also occur mirrored at a horizontal axis, which should also be kept in mind for the other types.

4.2 Type B Pointgons

Type B pointgons behave similar to type A pointgons (see Figure 4). Again we have to check whether a type A pointgon can result (note that in this case one separating path must consist of only one edge, see leftmost sketch). Otherwise we get a type B pointgon with an anchor that is one end of the cutting path (second

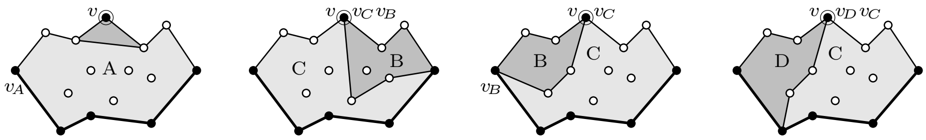

Fig. 5. Behavior of Type C pointgons in the recursion (possible splits). v denotes the original anchor and v_*, $* \in \{A, B, C, D\}$, denotes the anchor of a sub-pointgon of type $*$ due to the split.

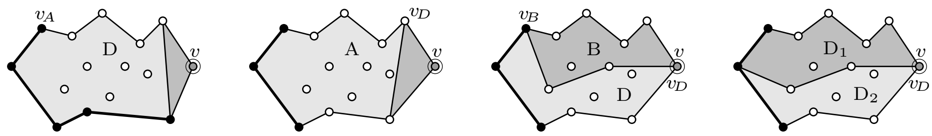

Fig. 6. Behavior of Type D pointgons in the recursion (possible splits). v denotes the original anchor and v_*, $* \in \{A, B, D\}$, denotes the anchor of a sub-pointgon of type $*$ due to the split.

sketch). For separating paths starting at the anchor only type B pointgons can result (third and fourth sketch), because there is not the possibility of a path leading to the left (as there was for type A pointgons), since both bounding separating paths are lexicographically increasing.

4.3 Type C Pointgons

Type C pointgons are the most complicated case (see Figure 5). If the anchor is "cut off", we only have one separating path, so the anchor is set to its starting vertex and thus we obtain a type A pointgon (leftmost sketch). If a new separating path is attached to the anchor, we have to distinguish whether it is lexicographically increasing or decreasing. Increasing paths are simpler, leading to a split into one type C and one type B pointgon (second sketch). If the path is lexicographically decreasing, we have to check whether there is a perimeter piece of the input pointgon with at least two vertices. If there is not, we obtain one type B pointgon, with its anchor at its leftmost vertex, and one type C pointgon, which maintains the anchor of the original pointgon (third sketch). Otherwise we obtain one type D and one type C pointgon, both of which receive the anchor of the original pointgon (rightmost sketch).

4.4 Type D Pointgons

Type D pointgons behave symmetrically to type B pointgons. When the anchor is "cut off" we also have to check whether a type A pointgon results (note that in this case one separating path must consist of only one edge, see leftmost sketch). Otherwise we get a type D pointgon with an anchor that is one end of the cutting path (second sketch). For separating paths starting at the anchor either one type B and one type D pointgon (third sketch), namely if one perimeter piece is empty, or two type D pointgons result (rightmost sketch).

5 Analysis

To estimate the time complexity of our algorithm, we proceed generally in the same way as the authors of [5], that is, by multiplying 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 splits (paths) of a sub-pointgon multiplied with the (worst case) time for processing one split. However, one of our refinements consists in grouping the subproblems and analyzing the groups separately. The groups we consider are defined by the number of perimeter vertices of the input pointgon a sub-pointgon has on its perimeter.

So consider the number of subproblems with l, $0 \le l \le k$, hole vertices on the perimeter. The worst case is that we have three perimeter vertices of the input pointgon, namely the anchor and the two ends of the separating paths. This gives us a factor of n^3. Next we have to choose l of the k input hole vertices, for which we have $\binom{k}{l}$ possibilities, and then we have to distribute the chosen hole vertices on the two paths, for which there are 2^l possibilities. As a consequence we have in the worst case $O(n^3\binom{k}{l}2^l)$ possible sub-pointgons with l holes on the perimeter. To check the consistency of this number with the total number of $O(n^3 3^k)$ subproblems in the worst case (where the 3^k stems from the possible ways of distributing the k holes onto the tree states "on counter-clockwise separating path", "on clockwise path", and "on neither path"), we sum the numbers of subproblems for all different values of l. That is, we compute $\sum_{l=0}^{k} n^3\binom{k}{l}2^l = n^3\sum_{l=0}^{k}\binom{k}{l}2^l = n^3 3^k$, where the last step follows from Newton's binomial series $(1+x)^k = \sum_{l=0}^{k}\binom{k}{l}x^l$ with $x = 2$.

Given a sub-pointgon with l holes on the perimeter, there are at most $k-l$ holes left to form a separating path and at most n end points. This gives us a maximum of $n2^{k-l}$ possible paths. For each path, we have to check whether it 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 separating path 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. With this table we can check in $O(k-l)$ whether a given separating path intersects a (possibly existing) perimeter piece. In addition, we have to check for an intersection with the at most two already existing separating paths, which contain at most $l+2$ edges. By exploiting that all paths are lexi-monotone, this check can be carried out in $O(k)$. Once a path is found to be valid, the sub-pointgons have to be constructed by collecting their at most $k+3$ defining vertices, and their solutions have to be looked up. Both operations take $O(k)$ time. Finally the length of the path has to be computed, which takes $O(k-l)$ time. Therefore processing one path takes in all $O(k)$ time.

As a consequence the overall time complexity is

$$O\left(\sum_{l=0}^{k} \underbrace{n^3\binom{k}{l}2^l}_{\text{subproblems}} \cdot \underbrace{n2^{k-l}}_{\text{paths}} \cdot \underbrace{k}_{\substack{\text{time}\\ \text{/path}}}\right) = O\left(n^4 2^k k \underbrace{\sum_{l=0}^{k}\binom{k}{l}}_{=2^k}\right) = O(n^4 4^k k).$$

$n-k$	k	time in seconds	time/$n^4 4^k k$
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	$3.235\cdot10^{-11}$

$n-k$	k	time in seconds	time/$n^5 4^k k$
3	1	0.010± 0.000	$2.441\cdot10^{-6}$
6	1	0.011± 0.000	$1.636\cdot10^{-7}$
9	2	0.048± 0.004	$9.314\cdot10^{-9}$
12	3	0.075± 0.006	$5.144\cdot10^{-10}$
15	4	0.144± 0.027	$5.679\cdot10^{-11}$
18	5	0.490± 0.129	$1.487\cdot10^{-11}$
21	6	1.844± 0.763	$5.229\cdot10^{-12}$
24	7	7.437± 3.217	$2.265\cdot10^{-12}$
27	8	34.617±14.639	$1.257\cdot10^{-12}$
30	9	131.192±52.538	$6.163\cdot10^{-13}$

Table 1. Results obtained with our Java implementation of the described MWT algorithm and a modification. All results are averages over 20 runs, with randomly generated convex pointgons. The left table refers to the normal algorithm, the right to the alternative version with bookkeeping of the hole vertices.

6 Implementation

As already pointed out above, we implemented our algorithm in Java. In addition to the standard algorithm as it was described above, this implementation offers an alternative version of the algorithm: it can optionally keep track of the hole vertices that are inside a sub-pointgon, thus reducing the number of candidate separating paths that are considered in the recursion. Theoretically such bookkeeping worsens the (worst case) time complexity to $O(n^5 4^k k)$ due to the test which of the two sub-pointgons contains a hole. Such a test, which is carried out by counting how many times a half-line from a hole cuts the perimeter of a sub-pointgon, has time complexity of $O(n)$ for each hole. Hence the checks for all holes incur costs in the order of $O(nk)$, which are then the costs for processing one path (instead of $O(k)$). In practice, however, activating this option resulted in shorter processing times in all tests we did. The reason seems to be that the reduction of the number of holes in the recursion leads to bigger gains than the losses incurred by the bookkeeping w.r.t. the holes.

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 9.2 and Sun Java 1.4.2_05. 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 candidate paths with the perimeter, are usually processed faster. The left table shows the results for the normal algorithm as it was described in the preceding sections, while the right table shows the results for the modification in which it is determined which hole vertices are inside a subproblem (see above). As these tables show, it pays in practice to invest the additional costs of this bookkeeping, since the execution times in the right table are generally lower than those in the left.

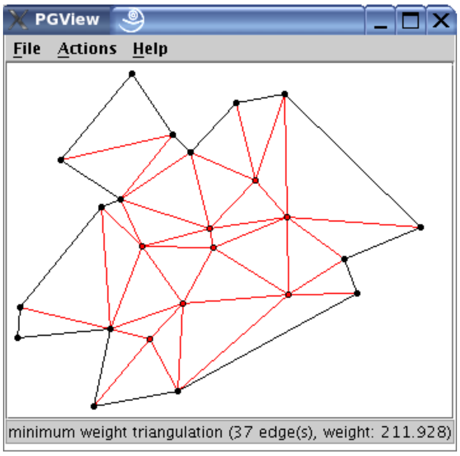

Fig. 7. A screenshot of the graphical user interface to our implementation of the described algorithm. The program allows for loading pointgons from a text file, generating random pointgons, finding their minimum weight triangulation, and modifying a triangulation as well as recomputing its weight in 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 screenshot 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 2.5 seconds (on a Intel Pentium 4C@2.6GHz system with 1GB of main memory running S.u.S.E. Linux 9.2).

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 versions of our algorithm 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. We even conjecture that there is still room for improvement of the analysis (basically w.r.t. the dependence on k), and continue working on it.

To give an impression of the graphical user interface (GUI) of the program, Figure 7 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`.

7 Conclusions

We described a fixed parameter algorithm for computing the minimum weight triangulation of a simple polygon with hole vertices, which is inspired by the algorithm in [5]. Due to improvements of both the algorithm as well as its analysis, we were able to show that its time complexity is $O(n^4 4^k k)$. Thus we provided a considerable improvement over the result of [5], who gave a time complexity of $O(n^5 \log(n)\, 6^k)$. 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 even be considerably better than the result of our theoretical analysis.

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. 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
6. G.T. Klincsek. Minimal Triangulations of Polygonal Domains. *Annals of Discrete Mathematics* 9:121–123. ACM Press, New York, NY, USA 1980
7. E. Lodi, F. Luccio, C. Mugnai, and L. Pagli. On Two-Dimensional Data Organization, Part I. *Fundamenta Informaticae* 2:211–226. Polish Mathematical Society, Warsaw, Poland 1979
8. 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
9. D. Moitra. Finding a Minimum Cover for Binary Images: An Optimal Parallel Algorithm. *Algorithmica* 6:624–657. Springer-Verlag, Heidelberg, Germany 1991
10. D. Plaisted and J. Hong. A Heuristic Triangulation Algorithm. *Journal of Algorithms* 8:405–437 Academic Press, San Diego, CA, USA 1987