Fixed Parameter Algorithms for the Minimum Weight Triangulation Problem

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 discuss and compare four fixed parameter algorithms for finding the minimum weight triangulation of a simple polygon with $(n-k)$ vertices on the perimeter and k vertices in the interior (hole vertices), that is, for a total of n vertices. All four algorithms rely on the same abstract divide-and-conquer scheme, which is made efficient by a variant of dynamic programming. They are essentially based on two simple observations about triangulations, which give rise to triangle splits and paths splits. While each of the first two algorithms uses only one of these split types, the last two algorithms combine them in order to achieve certain improvements and thus to reduce the time complexity. By discussing this sequence of four algorithms we try to bring out the core ideas as clearly as possible and thus strive to achieve a deeper understanding as well as a simpler specification of these approaches. In addition, we implemented all four algorithms in Java and report results of experiments we carried out with this implementation.

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.) The MWT problem has been studied extensively in computational geometry and has applications in computer graphics [29], image processing [27], database systems [24], and data compression [25].

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 vertices in the interior (hole vertices), 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.

The original version of this paper was published as
Technical Report LU-CS-TR:2005-238 on February 13, 2006,
ISSN 1650-1276 Report 158, Lund University, Sweden 2006.
This is an extended version published on February 20, 2006.

1.1 Related Work

The time complexity of finding the MWT of a general planar point set has been open since 1975, when it was included in a list of problems neither known to be NP-complete nor known to be solvable in polynomial time [10]. It was known, though, that if it is slightly generalized, so that the weight of an edge is an arbitrary function unrelated to its (Euclidean) length, the MWT problem becomes NP-complete [23]. Recently, however, it was reported that the standard MWT problem (that is, with Euclidean lengths as edge weights) is also NP-hard [28]. Several earlier attempts to prove that the problem is NP-hard lead to related NP-completeness results, which are listed in [14].

On the other hand, for certain *restricted problem instances* a MWT can be computed in polynomial time. For example, if S is a set of points in convex position (which is equivalent to a convex polygon without holes) or even a simple polygon without holes, the MWT can be computed in $O(n^3)$ time using a dynamic programming algorithm as was shown independently in [11] and [17]. Another example are the four algorithms discussed in this paper, all of which have polynomial time complexity if the number k of hole vertices is constant.

Due to the difficulty of solving the general problem exactly, some authors concentrated on *approximating* the MWT. For example, it was shown that any triangulation achieves total edge length $O(n)$ times the minimum [18]. Even the Delaunay triangulation, which was once claimed to be the MWT, can be as long as $\Omega(n)$ times the optimum [18, 26]. The greedy triangulation, as another approximation candidate, can yield a result as bad as $\Omega(\sqrt{n})$ [20]. Furthermore, several other heuristics have been proposed to solve the MWT problem (see, for instance, [22, 29, 14]). Among these the quasi-greedy approach in [21] is worth noting as it yields an approximation within a constant factor of the optimum.

In order to find an exact algorithm, *linear programming* methods have been tried [19]. Another, fairly popular line of attack is the *subgraph approach* [4, 33]: start by finding a (suitable) subgraph U of a MWT of S. For instance, find the set of edges that do not properly intersect any other edge between two vertices in S and thus must necessarily be in any minimum weight triangulation. (Note that this set comprises the edges of the convex hull of S.) If U has k connected components, try all possibilities to add $k-1$ edges to turn it into a connected graph C. Complete each resulting graph C to a triangulation by optimally triangulating its faces (using, for example, the dynamic programming algorithm for simple polygons without holes mentioned above), and select a triangulation with minimum weight. This approach was analyzed to have a time complexity of $O(n^{k+2})$. As sophisticated choices for the subgraph U so-called LMT-skeletons [2, 3, 5, 8] and β-skeletons [6, 16, 32] have been studied. [3] showed that there exist point sets where the number of components is linear in n.

1.2 Outline of this Paper

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 [9]. 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. More general results can be derived as follows: since the problem is described by two quantities, namely the input size n and the parameter k, there are several "ways" of letting the problem size go to infinity and thus to analyze the asymptotic time complexity. For example, we may consider a sequence of problem instances for which $k \in O(\log n)$. In this case the time complexity is still polynomial in n if, for example, $f(k) \in O(2^k)$, because then $f(k) = f(\log n) \in O(n)$.

W.r.t. the 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 this idea was proposed in [15] and analyzed to run in $O(n^5 \log(n)\, 6^k)$ time. The four algorithms we discuss in this paper are all inspired by this seminal paper, although they exploit several improvements and simplifications of both the processing scheme as well as the analysis, most of which are derived from [12, 13, 31].

The purpose of this paper is to provide a deeper insight into the core ideas underlying this specific fixed parameter approach to the MWT problem by separating and clarifying the two essential observations that underlie all four algorithms as well as how they can be combined to improve the basic processing scheme. We do so by first presenting two algorithms, each of which uses only one of the two observations [12, 13], thus showing that they can also be used in isolation to solve the MWT problem. Then we proceed with two algorithms that rely on a combined approach, the first of which can be seen as a simplification of the seminal algorithm as it was proposed in [15]. By drawing on an additional insight from [31] (but maintaining the two basic operations), this algorithm can be further improved, leading to a fairly simple final algorithm that can be seen as a considerably simplified version of the algorithm proposed in [31].

The general idea of all four algorithms is a recursive partitioning scheme, in which all possible splits (of certain types) of a polygon into sub-polygons are considered. The sub-polygons are processed recursively and the result is then computed as the minimum over the results for each split. This general scheme is described generally in Section 2, which also introduces some basic notation, reviews the core ideas of using dynamic programming to make the recursion efficient, and states the part that is common to all four algorithms as pseudocode. In Section 3 we consider the two basic observations about triangulations, from which we later derive possible splits of polygons. Section 4 describes the details of the four algorithms, listing for each of them the types of subproblems that occur in the recursion and how they are processed. In Section 5 we analyze the time complexity of the algorithms, following the same basic lines for all four algorithms, but emphasizing special techniques. In Section 6 we present our Java implementation of the discussed algorithms and report experimental results. Finally, in Section 7 we draw conclusions from our discussion.

2 Basic Idea of the Algorithms

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.

2.1 Recursive Partitioning

The basic idea of all four algorithms discussed in this paper is to split an input pointgon into at most two sub-pointgons, each of which is then processed recursively. The recursive splitting proceeds until we arrive at triangles without holes (which we call *empty triangles* in the following), for which computing the MWT is trivial. The results of the recursive calls are combined and the weight of a MWT is computed as the minimum over all considered splits. The core differences between the four algorithms consist in what types of splits they use and whether it is necessary to consider all splits of a certain type or whether some can be replaced by others, which are more convenient for the processing. However, for the general description in this section it suffices to imagine splits as single edges or paths through hole vertices that lead from one perimeter vertex to another and thus split a given pointgon into two disjoint parts.

More formally, we can describe such a split-based divide-and-conquer approach as follows: let G be a pointgon defined by vertex sets V_p and V_h and let $\Theta(G)$ be the set of all splits of G that we consider for a specific algorithm. For each split $\theta \in \Theta(G)$ let $L(G,\theta)$ be the sub-pointgon to the left of the split and $R(G,\theta)$ the sub-pointgon to the right of the split. Finally, let $w(\theta)$ be the weight of the split itself. Then the weight of a MWT of G can be computed as:

$$\mathrm{MWT}(G) = \min_{\theta \in \Theta(G)} \{\mathrm{MWT}(L(G,\theta)) + \mathrm{MWT}(R(G,\theta)) + w(\theta)\}\,. \qquad (1)$$

Note that, as we will see below, some splits may lead to only one (reduced) sub-pointgon, so that no combination of results is necessary. We handle this formally be defining that one of $L(G,\theta)$ and $R(G,\theta)$ is a null pointgon for such splits (technically: a pointgon with empty sets V_p and V_h), for which the MWT has a formal weight of 0. Note also that we need the term $w(\theta)$ to account for a possible weight of the split. As a simple illustration why this is necessary, consider a split by a simple edge from one perimeter vertex to another, non-adjacent perimeter vertex. This edge will be part of both sub-pointgons $L(G,\theta)$ and $R(G,\theta)$ and thus would be counted twice if we simply added the results for these sub-pointgons. To correct this, we have to subtract the length of the splitting edge, showing that in this case $w(\theta)$ is the negative edge length. In other cases, which we will discuss in detail in Section 4, the weight may also be positive in order to account for edges that enter neither of the two sub-pointgons formed.

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 MWT of the sub-pointgons. The union of these sets of edges for the term that yields the minimum weight is a MWT for the original pointgon G.

2.2 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 can 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. The basic idea is to avoid repeating identical recursive calls by memorizing the solutions to already solved subproblems. In this way, whenever the same subproblem is encountered again, the result is already available, and no computation is needed.

There are two basic variants of dynamic programming that can be used:

1. As a direct implementation of the outlined idea, one may set up a global data structure to store all subproblems, which have already been processed, together with their solutions. In the recursive function we then precede the loop over the possible splits into subproblems by a check whether the solution of the current subproblem is already known, which accesses the global data structure. If the subproblem has been solved before, we simply retrieve and return the solution. Only if the solution is not yet known, the recursion is actually executed. This technique is known as *memoization* (no 'r'!) [7].
2. As a tree recursion defines a directed acyclic graph over the set of subproblems (which are the nodes of this graph and are connected in such a way that there is a directed edge from each subproblem to all subproblems that result directly from it in the recursion), one can go one step further. Described most generally, one finds a topological order[3] of the nodes of the subproblem graph and then processes the subproblems in reverse order. (That is, one starts with the subproblem that comes last in the topological order and works backward.) A clever choice of the topological order (which, in general, is not unique) can lead to a very simple processing scheme. Note that often only this second variant is actually called *dynamic programming.*

Of course, if possible, one should use the second variant, as it has several advantages. In the first place, it avoids all unsuccessful accesses to the global data structure, since processing the subproblems in reverse topological order ensures that all subproblems needed for a recursive solution of the next subproblem have already been solved. Secondly, in many cases it is possible to choose the

[3] A topological order of a directed acyclic graph is a numbering of the nodes such that all edges lead from a node with a lower number to a node with a higher number.

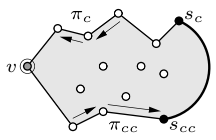


Fig. 1. A sub-pointgon is represented by a counterclockwise walk round its perimeter, following the at most two bounding paths. The encircled vertex is the anchor, the thick line represents a coherent perimeter piece.

topological order in such a way that solutions to certain subproblems may be discarded again, since at some point the solutions of all subproblems, for which they may be needed in the recursion, have already been computed. This can lead to considerable savings in the amount of memory needed.

However, for designing an algorithm, the second solution has severe disadvantages. If the structure of the recursion is complex (as it is for the algorithms discussed in this paper), it can be extremely tedious and complicated to find a (proper) topological order for the subproblems. In such cases, the first variant (that is, *memoization*) is surely preferable, as it is much easier to implement. We simply define a unique description of each subproblem and set up a data structure that allows us to map such descriptions to subproblem solutions. The additional time needed for the unsuccessful accesses of the data structure can usually be neglected as it has no influence on the asymptotic time complexity.

As a consequence of these considerations we choose a memoized processing scheme for the algorithms we discuss in this paper. The global data structure we use is a trie that maps index words, which describe subproblems, to solutions. These solutions comprise the weight of a MWT of the corresponding subproblem and a description of the split by which the corresponding MWT can be achieved.

2.3 Index Word Representation of a Subproblem

In order to be able to use an efficient trie structure to store already found subproblem solutions, we represent a subproblem by an index word over an alphabet with n characters (one for each vertex of the input pointgon), which uniquely identifies each subproblem. In all four algorithms, this index word describes a counterclockwise walk round the perimeter of the subproblem, and hence an index word can be at most n characters long. Generally, this would give rise to $n!$ possible index words (as each character can appear exactly once in each word).

However, due to the way in which subproblems are processed (we will study the details in Section 4), only a limited set of index words actually occurs, namely those that have the general form (v, π_{cc}, π_c) (see Figure 1 for a sketch). 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 paths through hole vertices, where the index c indicates that the path is to the clockwise side of the anchor, the index cc that it is to the counterclockwise side (see Figure 1). These paths either join at some hole vertex or lead to perimeter vertices of the input pointgon. That is, all elements of π_{cc} and π_c are in V_h (and thus can have k possible values), with the possible exception of the last elements, which may be perimeter vertices s_{cc} and s_c, respectively (which can thus have n possible values). In Figure 1, a hole vertex is depicted by a white point, a perimeter vertex by a black point, and a vertex that may be either a hole vertex or a perimeter vertex by a grey point. Edges of the paths are shown as thin lines, edges of the perimeter of the input pointgon as thick lines. (We will use the same color/thickness coding in all figures of this paper.)

Note that either π_{cc} or π_c may also be empty, so that there is only one path through holes. Note also that in the case where the paths π_{cc} and π_c 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.[4] In the first algorithm, however, we use only one path if all vertices in the index word, except at most one, are hole vertices. In this case the anchor, which would formally be the end vertex of the path (as it loops back to the anchor), is not contained in the path to avoid duplicate entries. 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. Hence, with these restrictions, there can already be no more than $O(n^3k!)$ possible index words.

2.4 General Structure of the Algorithms

All four algorithms we discuss in this paper have the same basic structure, which is, as already outlined above, a memoized version of the tree recursion resulting from the recursive partitioning scheme. In pseudocode, it looks like this:

```
function MWT (word key) : real
begin
    if    key is empty (i.e., has length 0)
    then return 0;
    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 splits $\theta \in \Theta$(polygon(key), key.v) do
        wgt = MWT(L(key, $\theta$)) + MWT(R(key, $\theta$)) + weight($\theta$);
        if    wgt < min
        then min = wgt; best = $\theta$; fi;
    done;
    trie.add(key, min, best);
    return min;
end;
```

[4] 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.

Here *key* is the index word that describes the subproblem to be solved. To start the recursion, we code the input pointgon as follows: we choose an arbitrary vertex as the anchor and view the edge to the counterclockwise adjacent vertex as a path edge. That is, the initial key simply consists of the characters for two adjacent perimeter vertices of the input pointgon.

The symbol $\perp$ is used to indicate that there is no split. It is stored with an empty triangle to indicate that no subproblems need to be considered. The functions L and R construct the keys of the two sub-pointgons that are to the left and to the right of the split θ. They yield empty keys (words of length 0) if the corresponding sub-pointgon does not exits (recall from Section 2 that we will also consider certain splits that lead to only one (reduced) pointgon). This explains the formal check for an empty key at the beginning of the above function, which is meant to handle such cases. The function "weight" computes the weight of the split (see Section 2), as some of its edges may be part of (1) both subproblems, (2) neither subproblem, or (3) only one subproblem. In the first case their weight has to be subtracted, in the second added to the sum of the triangulation weights of the sub-pointgons. Only in the third case no correction of the sum of the triangulation weights is necessary.

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

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

Here the function "edges" yields the edges of the split. Note that this function is meant to yield only the edges that have to be added to the input pointgon in order to triangulate it. The perimeter edges of the input pointgon are assumed to be already given (or to be added later).

3 Observations about Triangulations

In order to derive useful splits of pointgons, which up to now we treated in a completely abstract way, we study two basic observations about triangulations. The first of these two observation is very simple:

Observation 1 *With the definition* $s(i) = (i+1) \bmod (n-k)$*, let* $e_i = (v_i, v_{s(i)})$*,* $1 \leq i < n-k$*, be an arbitrary edge on the perimeter of a pointgon* G*. Then in every triangulation* T *of* G *(and thus also in the minimum weight triangulation) there exists a vertex* $v \in V \backslash \{v_i, v_{s(i))}\}$ *adjacent to both* 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 may use triangles built over an arbitrary, but fixed perimeter edge as splits: if we consider all possible empty triangles that

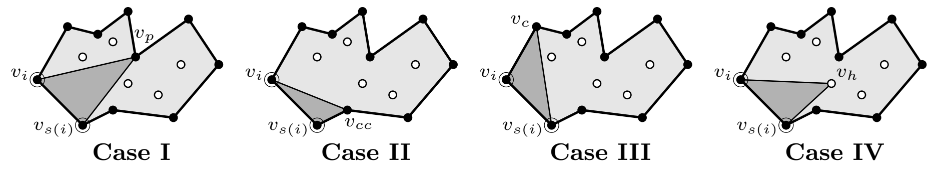


Fig. 2. The four possible ways of forming a triangle from a perimeter edge $(v_i, v_{s(i)})$ (encircled points; see Observation 1). A black point indicates a vertex belonging to V_p, a white point a vertex belonging to V_h. Note that cases II and III can be seen as special variants of case I, in which either the right or the left subproblem does not exist.

can be built over this fixed edge, we can be sure that we have considered the one yielding the minimum weight triangulation. This leads to the definition:

Definition 1. *Let $e_i = (v_i, v_{s(i)})$, $1 \leq i < n - k$, be an arbitrary perimeter edge of a pointgon G and let $v \in V \backslash \{v_i, v_{s(i)}\}$ be an arbitrary third vertex. The triangle $(v_i, v_{s(i)}, v)$ is called a* **triangle split** *of the pointgon G iff it contains no vertex $v' \in V \backslash \{v_i, v_{s(i)}, v\}$ and neither of the edges (v_i, v) and $(v_{s(i)}, v)$, except when it is a perimeter edge itself, properly intersects a perimeter edge. We call the edge $(v_i, v_{s(i)})$ the* **base** *of the triangle $(v_i, v_{s(i)}, v)$ and v its* **tip***.*

The different types of partitions that can result from such triangle splits are shown in Figure 2: in the first three cases the tip of the triangle is on the perimeter, in the fourth case it is a hole vertex. Note that cases II and III can be seen as special cases of case I in which the right or the left subproblem, respectively, are empty. Note also that in case IV there is necessarily only one sub-pointgon, which we formally define to be left of the split. The algorithm we study in Section 4.1 is based solely on this splitting scheme.

For the following observations and definitions we need to be able to compare vertices w.r.t. their location. We do so with the following standard definition:

Definition 2. *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.*

With this definition we can also characterize triangle splits more precisely, which we need for the last two algorithms that combine the two split types.

Definition 3. *Let $e_i = (v_i, v_{s(i)})$, $1 \leq i < n - k$, be an arbitrary edge on the perimeter of a pointgon G and let $v \in V \backslash \{v_i, v_{s(i)}\}$ be an arbitrary third vertex. Then the triangle $(v_i, v_{s(i)}, v)$ is called a* **left triangle** *iff $v \prec v_i$ and $v \prec v_{s(i)}$ (that is, if the tip is lexicographically smaller than the base), a* **middle triangle** *iff $v_i \prec v \prec v_{s(i)}$ or $v_{s(i)} \prec v \prec v_i$ (that is, if the tip lies lexicographically between the two base vertices), and a* **right triangle** *iff $v_i \prec v$ and $v_{s(i)} \prec v$ (that is, if the tip is lexicographically greater than the base).*

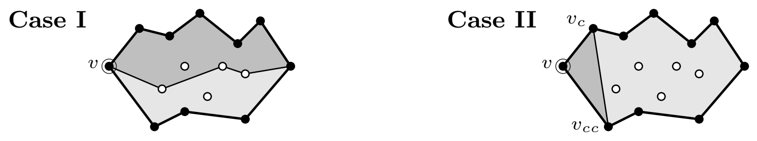


Fig. 3. The two possible ways of splitting a pointgon with a path starting at a chosen perimeter vertex (case I) or cutting off this vertex (case II; see Observation 2). A black point indicates a vertex belonging to V_p, a white point a vertex belonging to V_h.

The lexicographic relation between vertices can easily be extended to define lexi-monotone sequences or lexi-monotone paths:

Definition 4. *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.*

With this definition we can now define path splits:

Definition 5. *A* **lexi-monotone path split** *or simply a* **path split** *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 properly intersect the perimeter of G (start and end vertex do not count as intersections). In addition, no edge of the path may contain a hole vertex other than its end vertices.*

About path splits we can make the following observation:

Observation 2 *Let $v \in V_p$ be an arbitrary vertex on the perimeter of a pointgon G. Then in every triangulation T of G (and thus also in the minimum weight triangulation) there exists:* **either** *a lexi-monotone path π starting at v (where π can be increasing or decreasing)* **or** *two perimeter vertices v_c and v_{cc} that are adjacent to v and that together with v form an empty triangle, that is, a triangle without any hole vertices in its interior.*

These two possibilities are sketched in Figure 3: in case I a path is attached to the vertex v, in case II it is cut off from the rest of the pointgon with an empty triangle. As a consequence of this observation, we may use lexi-monotone paths attached to an arbitrary, but fixed perimeter vertex as splits: if we consider all possible such paths as well as the edge cutting off this vertex from the rest of the pointgon (provided that the resulting triangle is empty), we can be sure that we have considered the split yielding the minimum weight triangulation. The algorithm we study in Section 4.2 is based solely on this splitting scheme.

The remaining two algorithms, studied in Sections 4.3 and 4.4, use, in addition to the pure forms described above, a combination of such splits, namely a triangle split with a lexi-monotone path attached to its tip. Note that in such a combination we need not consider the case in which the vertex, to which the path is attached, is cut off from the rest of the pointgon. Since the triangle is concave at the tip, the tip cannot be cut off with a single edge.

4 Specific Details of the Algorithms

As already mentioned, Algorithm 1, which is presented in Section 4.1, is purely based on triangle splits, while Algorithm 2, presented in Section 4.2, is purely based on path splits. Algorithms 3 and 4 combine the two splits types (that is, they also use triangle splits with paths attached to the triangle tip). Algorithm 4 also draws on another, additional insight beyond those already presented in the preceding section. However, we present this observation only in Section 4.4, as it can be explained much more easily after Section 4.3 has been studied.

In the following sections we first discuss the specific form of the recursive processing scheme for each of the four algorithms, emphasizing the specific choices that are made in the recursion in order to keep the number of subproblem types small. Then we list the types of subproblems that are encountered in the recursion and how these subproblems are processed. The latter will also generally provide a proof that the listed subproblems are indeed the only ones that occur in the recursion, as no other types of subproblems can result from the processing.

4.1 Algorithm 1 (Purely Triangle-based Algorithm)

Algorithm 1 is purely based on Observation 1 and thus considers all triangle splits that can be formed from an arbitrarily chosen perimeter edge $(v_i, v_{s(i)})$. In the following description, v_c is the vertex adjacent to the vertex v_i in clockwise direction, and v_{cc} is the vertex adjacent to $v_{s(i)}$ in counterclockwise direction.

Recursive Processing We consider four cases I, II, III, IV in our recursion, as shown in Figure 2 (which sketches the four different cases of triangle splits).

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.

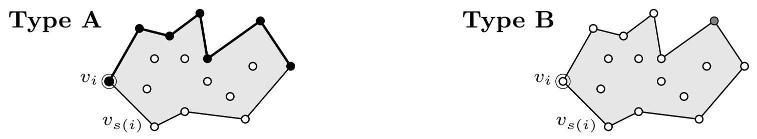


Fig. 4. The two types of pointgons we encounter in the purely triangle-based algorithm. A black point indicates a vertex belonging to V_p, a white point a vertex belonging to V_h, a grey point a vertex either in V_p or in V_h. The encircled point is the anchor. Thick lines indicate pieces of the perimeter of the input pointgon.

Given these four cases, Equation 1, which describes the general recursion to find the weight of a MWT of G, can be made more specific as:

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

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. Note that the edge lengths that are added in the individual cases are the weights of the splits, which are all positive in this algorithm.

In the recursion we 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 built 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 built 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 4).

Types of Pointgons Apart from the input pointgon (which, strictly speaking, is of neither of these types, but can be treated like a type A pointgon, see above), we encounter two types of sub-pointgons (see Figure 4 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).

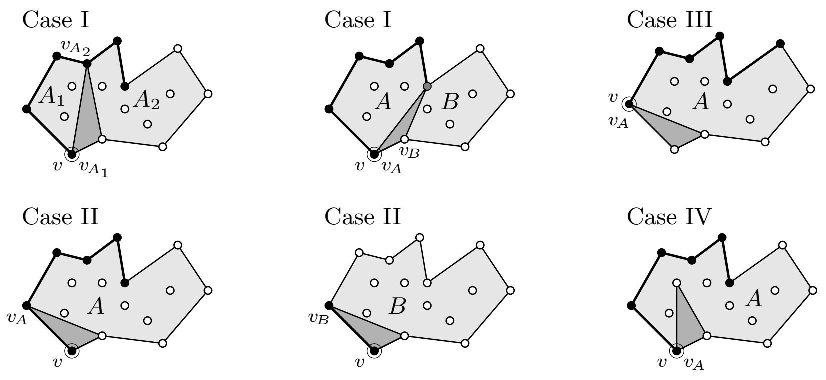


Fig. 5. Behavior of Type A pointgons in the recursion (possible splits). v denotes the original anchor and v_*, $* \in \{A, B\}$, denotes the anchor of a sub-pointgon of type $*$ due to the split. Note again that cases II and III can be seen as special variants of case I.

Processing the Types of Pointgons We show now that we actually encounter only these two types of subproblems by discussing how each of them is processed.

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 5. 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 5 how the anchors of the sub-pointgons are selected.

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 2, 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.)

4.2 Algorithm 2 (Purely Path-based Algorithm)

Algorithm 2 is purely based on Observation 2 and thus considers all lexi-monotone paths that can be attached to an arbitrarily chosen perimeter vertex as well as the possibility to cut off the perimeter vertex with an empty triangle.

Recursive Processing We consider two cases I and II in our recursion, as shown in Figure 3 (which sketches the two different cases for path splits).

I In this case we consider all lexi-monotone 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 lexi-monotone path. We denote the set of all such paths by $\Theta(G, v)$. The length $|\theta|$ of such a path $\theta \in \Theta(G, v)$ is the sum of the lengths of its edges. $L(G, \theta)$ and $R(G, \theta)$ are the sub-pointgons to the left and to the right of θ, respectively.

II In this case 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. We denote the two perimeter vertices that are adjacent to v in clockwise and counterclockwise direction by v_c and v_{cc}, respectively.

Given these two cases, Equation 1, which describes the general recursion to find the weight of a MWT of G, can be made more specific as:

$$\mathrm{MWT}(G) = \min\Big\{ \min_{\theta\in\Theta(G,v)} \{\mathrm{MWT}(L(G,\theta)) + \mathrm{MWT}(R(G,\theta)) - |\theta|\}\,, \\ \mathrm{MWT}(R(G,(v_{cc},v_c))) + |(v,v_{cc})| + |(v,v_c)|\Big\}.$$

The first term in the outer minimum refers to case I. Note that we have to subtract the length of the path θ from the sum of the MWT weights of the two sub-pointgons, because this path is part of both sub-pointgons. The second term in the outer minimum refers to case II. Obviously, for the 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 $\mathrm{MWT}(R(G,(v_{cc},v_c)))$ and thus need not be added.

In the recursion we exploit the fact that we can choose freely, which perimeter vertex of a sub-pointgon we want to use as the starting point for the paths. By preferring the same vertex v for attaching splitting paths as in the preceding split—provided it is still contained in the subproblem—and choosing the leftmost or rightmost key vertex in all other cases, every subproblem we encounter can be described by one or two lexi-monotone paths that start at the same vertex v (which we choose as the anchor of the subproblem) and a coherent piece of the perimeter of the input pointgon (see Figure 1 for a sketch). A detailed analysis of this statement, providing a proof, is given below.

Types of Pointgons Apart from the input pointgon, which, strictly speaking, is of neither of these types, we encounter four types of sub-pointgons (see Figure 6).

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

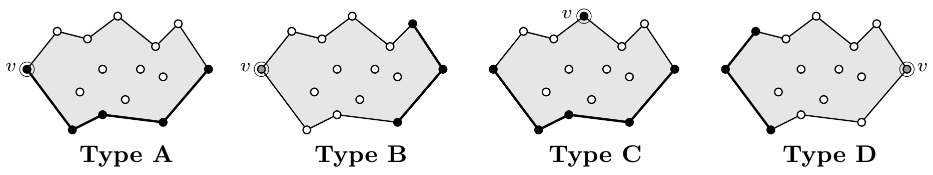


Fig. 6. The four types of pointgons we encounter in the purely path-based algorithm. A black point indicates a vertex belonging to V_p, a white point a vertex belonging to V_h, a grey point a vertex either in V_p or in V_h. The encircled point is the anchor. Thick lines indicate pieces of the perimeter of the input pointgon.

B Sub-pointgons of this type are bounded by two lexi-monotone 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 lexi-monotone 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 lexi-monotone 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 would be a type B pointgon).

The general principle of the choice of the anchor is that it is the leftmost vertex on the lexi-monotone path if there is just one path, and the vertex that is on both paths if there are two lexi-monotone paths. If there are two vertices that are on both paths (because they share both start and end vertex), we choose the leftmost one (thus, in a way, giving type B preference over type D).

Processing the Types of Pointgons We show now that we actually encounter only these types of subproblems by discussing how each of them is processed. Note that the input pointgon can be treated as a type A pointgon by seeing an arbitrary perimeter edge as a lexi-monotone path with only one edge (see above).

Type A Pointgons. The different splits of a type A pointgon are sketched in Figure 7. 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 lexi-monotone path to give a new type A pointgon. Otherwise, we obtain a type B pointgon (second sketch). For a lexi-monotone 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

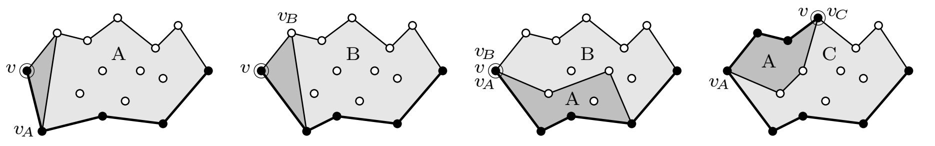


Fig. 7. Behavior of Type A pointgons in the recursion (possible splits). v denotes the original anchor and v_*, $* \in \{A, B, C\}$, denotes the anchor of a sub-pointgon of type $*$ due to the split.

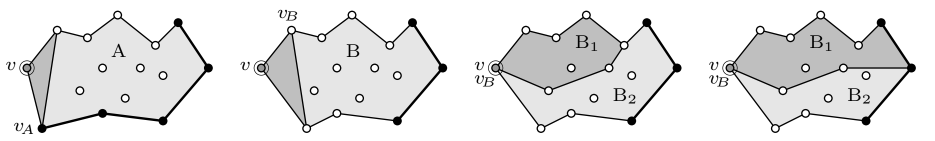


Fig. 8. Behavior of Type B pointgons in the recursion (possible splits). v denotes the original anchor and v_*, $* \in \{A, B\}$, denotes the anchor of a sub-pointgon of type $*$ due to the split.

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 lexi-monotone 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, where the same is possible.

Type B Pointgons. Type B pointgons behave similar to type A pointgons (see Figure 8). Again we have to check whether a type A pointgon can result (note that in this case one lexi-monotone 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 sketch). For lexi-monotone 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 lexi-monotone paths are lexicographically increasing.

Type C Pointgons Type C pointgons are the most complicated case (see Figure 9). If the anchor is "cut off", we only have one lexi-monotone path, so the anchor is set to its starting vertex and thus we obtain a type A pointgon (leftmost sketch). If a new lexi-monotone 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

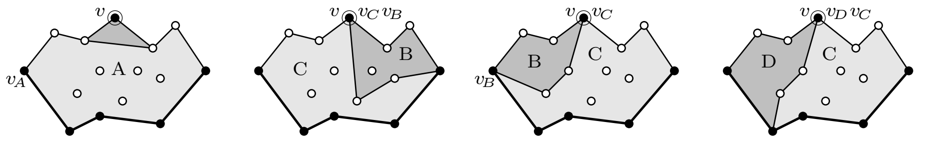


Fig. 9. Behavior of Type C pointgons in the recursion (possible splits). v denotes the original anchor and v_*, $* \in \{A, B, C, D\}$, denotes the anchor of a sub-pointgon of type $*$ due to the split.

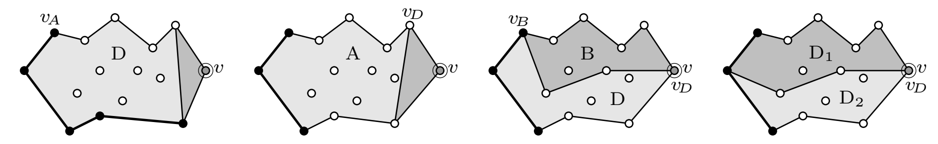


Fig. 10. Behavior of Type D pointgons in the recursion (possible splits). v denotes the original anchor and v_*, $* \in \{A, B, D\}$, denotes the anchor of a sub-pointgon of type $*$ due to the split.

(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).

Type D Pointgons. Type D pointgons behave symmetrically to type B pointgons (see Figure 10). When the anchor is "cut off" we also have to check whether a type A pointgon results (note that in this case one lexi-monotone 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 lexi-monotone 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).

4.3 Algorithm 3 (First Combined Algorithm)

In addition to pure triangle splits and pure paths splits Algorithm 3 also uses a combination of both, that is, triangle splits with a lexi-monotone path attached to the tip. This algorithm is best explained as an extension of Algorithm 1 (the purely triangle-based algorithm) as follows: a disadvantage of using pure triangle splits is that the path that describes the current subproblem can become non-lexi-monotone, namely when the triangle built over the first path edge is a left or a right triangle. (Only middle triangles maintain an existing lexicographic monotonicity of the path in this case, see the second sketch in Figure 12 for an example.) However, if in these cases we attach a lexi-monotone path to the tip (a decreasing path for a left triangle and an increasing path for a right triangle), we can avoid this drawback. The two additional types of splits are shown in the first and third sketch in Figure 12. Note, however, that the advantage of

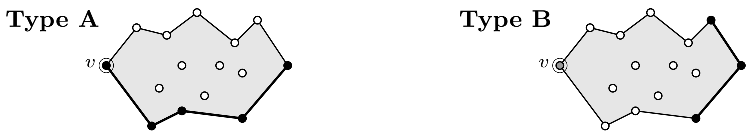


Fig. 11. The two types of pointgons we encounter in the first combined algorithm. A black point indicates a vertex belonging to V_p, a white point a vertex belonging to V_h, a grey point a vertex either in V_p or in V_h. The encircled point is the anchor. Thick lines indicate pieces of the perimeter of the input pointgon.

having only lexi-monotone paths in the subproblem descriptions comes at a price: while we needed only one (though possibly non-lexi-monotone) path in the purely triangle-based algorithm (Algorithm 1), we now obtain subproblems with two lexi-monotone paths, namely when a right triangle with an increasing path attached to its tip is built over the first edge of a path that contains more than one edge (with only one edge both subproblems must always be type A).

Recursive Processing The recursive processing is basically the same as for Algorithm 1 (purely triangle-based algorithm, see Section 4.1), only that the two special cases of a left triangle with an attached decreasing path and a right triangle with an attached increasing path have to be added (all possibilities of which have to be considered). In addition, we have to switch to the processing scheme of Algorithm 2 (see Section 4.2) as soon as we obtain a subproblem that is bounded by two lexi-monotone paths through hole vertices. Of course, we also have to switch back to the (extended) triangle-based scheme as soon as such a subproblem is reduced to one with only a single lexi-monotone path.

Types of Subproblems Apart from the input pointgon, which, strictly speaking, is of neither of these types, we encounter two types of sub-pointgons (see Figure 11). These two types are exactly the same as types A and B in Algorithm 2 (see Figure 6 in Section 4.2 and the corresponding description).

Processing the Types of Pointgons We show now that we actually encounter only these two types of subproblems by discussing how each of them is processed. Note that the input pointgon can be treated as a type A pointgon by seeing an arbitrary perimeter edge as a lexi-monotone path with only one edge (see above).

Type A Pointgons. The different splits of a type A pointgon are sketched in Figure 12. In the first sketch a decreasing lexi-monotone path is attached to a left triangle, leading to two type A subproblems. Note that the result would be the same if the tip of the triangle were a perimeter vertex and thus the path were empty. A middle triangle, shown in the second sketch, maintains the lexi-monotonicity of the path and thus does not need a path attached to its tip.

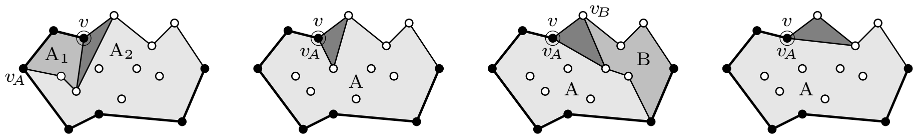


Fig. 12. Behavior of Type A pointgons in the recursion (possible splits). v denotes the original anchor and v_*, $* \in \{A, B\}$, denotes the anchor of a sub-pointgon of type $*$ due to the split. Note that the rightmost diagram shows a special case of the operation depicted in the third, which may also occur analogously for the first operation.

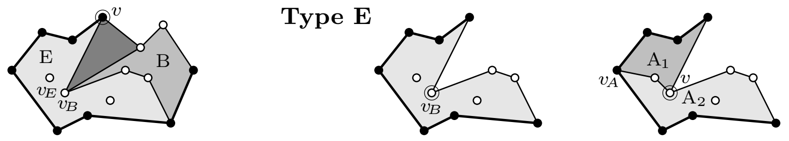


Fig. 13. A different way of processing type A pointgons in the recursion and the additional type of subproblems that results from it. v denotes the original anchor and v_*, $* \in \{B, E\}$, denotes the anchor of a sub-pointgon of type $*$ due to the split.

The result is a type A pointgon. Note, however, that if the tip of such a middle triangle is on the perimeter, we obtain *two* type A pointgons. The third sketch shows a right triangle with an increasing lexi-monotone path attached to its tip. If the triangle base is not the only edge of the path bounding the subproblem (as it is the case in the sketch), we obtain one type A and one type B subproblem. However, in the special case where the base of the triangle is the only edge of the path bounding the subproblem, we get two type A subproblems. The fourth sketch reminds us that we still have to consider the special cases that result from cases II and III of the triangle splits (see Figure 2).

Type B Pointgons. Type B pointgons are treated in exactly the same way as type B pointgons in Algorithm 2 (see Section 4.2). It was ascertained there that processing a type B subproblem can only lead to type A or type B subproblems.

Comparison to [15] Algorithm 3 is fairly similar to the algorithm proposed in the seminal paper [15]. The differences consist in the following: in [15] an increasing lexi-monotone path is attached to all triangles, regardless of whether they are left, middle, or right triangles. While this only leads to unnecessary splits for middle triangles, left triangles also cause a new subproblem to arise, as can be seen from the first sketch in Figure 13. The new type E, shown in the middle of Figure 13, is similar to a type B subproblem, as it also has two increasing lexi-monotone paths. However, in contrast to type B subproblems it is concave at the anchor, thus ruling out the possibility to cut it off from the sub-pointgon. Note also that one of its two paths always contains only one edge.

Such a type E subproblem is processed by attaching a *decreasing* lexi-monotone path to the anchor, which splits it into two type A subproblems (see the rightmost sketch in Figure 13). Note that with this processing scheme we arrive at almost the same situation as if we had attached a decreasing path to the left triangle directly. One type A subproblem is the same, the other has (unnecessarily) been split into a type A and a type B subproblem already by the increasing path that was attached to the triangle tip in the preceding processing step.

4.4 Algorithm 4 (Second Combined Algorithm)

With Algorithm 3 we tried to avoid non-lexi-monotone bounding paths for subproblems by attaching lexi-monotone paths to the tips of left and right triangles if these tips are hole vertices. However, for this improvement we had to pay the price that subproblems arose, which had two bounding lexi-monotone paths (type B subproblems), while in the purely triangle based-algorithm we had only subproblems bounded by one (though possibly non-lexi-monotone) path.

In this situation it is natural to ask whether there is the possibility of maintaining lexi-monotonicity without paying this price. This is indeed possible, as an insight from [31] shows. To understand the reasoning underlying this insight, consider the following: obviously right triangles are the problem, though only if their base is not the only edge on the path bounding a type A subproblem. If we could do without them, or rather, if we could replace them with another, more convenient split, which kept us in the realm of type A subproblems without losing the guarantee to find the minimum weight triangulation, everything would be fine. Fortunately, such a replacement is possible due to the following observation (which is considerably reformulated from the insight in [31], but in this form it becomes clearer why we do not lose optimality):

Observation 3 *Let π be a lexi-monotone path on the perimeter of a pointgon G. Then in every triangulation T of the pointgon G (and thus also in the minimum weight triangulation), at least one of the following must hold w.r.t. the path π:*

1. *The first edge of the path is the base of an empty left triangle.*
2. *There exists an edge on the path that is the base of an empty middle triangle.*
3. *There exists a vertex on the path (apart from its first and last vertex) at which the path is convex, and which forms an empty triangle with the preceding and the following vertex on the path.*
4. *The last edge of the path is the base of an empty right triangle.*

We prove this observation by contradiction: Assume that neither of the above four cases holds. Then the first edge can neither be the base of a left triangle, because then case 1 would hold, nor of a middle triangle, because then case 2 would hold. Hence the first edge must be the base of a right triangle. This right triangle cannot cut off the second vertex on the path (that is, cannot have the third vertex as its tip), because then case 3 would hold. As a consequence the *second* edge cannot be the base of a left triangle, as this would collide with the right triangle over the first edge (except if it cut off the second vertex, which we already ruled out). The second edge can also not be the base of a middle triangle, since then case 2 would hold. Therefore the second edge must be the base of a right triangle. Reasoning in exactly the same way to the third path edge and on, we traverse the whole path, always concluding that the next edge must be the base of a right triangle. Finally we reach the last edge of the path, concluding that it is the base of a right triangle. However, this contradicts the assumption that case 4 does not hold. Therefore the above observation is correct.

Recursive Processing Due to the above observation, we have to consider the following six cases in order to find a MWT (in all cases π refers to the same arbitrary, but fixed lexi-monotone path on the perimeter of the pointgon). As they exhaust all possibilities listed in the above observation, we must have considered the one obtaining in a MWT of the given pointgon G with them.

I In this case we consider all possible empty left triangles over the first edge of the path π, with the exception of the triangle that cuts off the first vertex of the path from the rest of the pointgon. If the tip of the triangle is not a perimeter vertex, we consider all possible decreasing lexi-monotone paths that can be attached to the tip of the triangle (see the first sketch in Figure 15). The set of splits we obtain in this way we denote by $\Theta_l(G, \pi)$. The length $|\theta|$ of such a split $\theta \in \Theta_l(G, \pi)$ is the sum of the lengths of its path edges (i.e., the triangle sides are excluded). $L(G, \theta)$ and $R(G, \theta)$ are the sub-pointgons to the left and the right of θ, respectively.

II In this case we consider all possible empty middle triangles over any edge of the path, with the exception of the triangles that cut off the first vertex of the path or the last vertex of the path from the rest of the pointgon (see the second sketch in Figure 15). Note that the tip of the triangle may be a hole vertex or a perimeter vertex. The set of splits we obtain in this way we denote by $\Theta_m(G, \pi)$ and for each such split $\theta \in \Theta_m(G, \pi)$ we denote the tip of the triangle by v_m and the path vertices at the end of the base of the triangle by $v_{l(m)}$ and $v_{r(m)}$. If the tip of the triangle is a perimeter vertex, $L(G, \theta)$ and $R(G, \theta)$ are the sub-pointgons to the left and the right of the triangle, respectively. If the tip is a hole vertex, $L(G, \theta)$ is the remaining sub-pointgon, while $R(G, \theta)$ is a null pointgon (subproblem does not exist).

III In this case we consider all possible empty triangles that cut off vertices, at which the path is convex, from the rest of the pointgon (see the third sketch in Figure 15). The first and the last vertex of the path are excluded. The set of splits we obtain in this way we denote by $\Theta_c(G, \pi)$. The path vertex that is cut off we denote by $v_{(c)}$, its predecessor on the path by $v_{(c-1)}$ and its successor by $v_{(c+1)}$. Note that all three vertices are on the path.

IV In this case we consider all possible empty right triangles over the last edge of the path π, with the exception of the triangle that cuts off the last vertex of the path from the rest of the pointgon. If the tip of the triangle is not a perimeter vertex, we consider all possible increasing lexi-monotone paths that can be attached to the tip of the triangle (see the fourth sketch in

Figure 15). The set of splits we obtain in this way we denote by $\Theta_r(G,\pi)$. The length $|\theta|$ of such a split $\theta \in \Theta_l(G,\pi)$ is the sum of the lengths of its path edges (i.e., the triangle sides are excluded). $L(G,\theta)$ and $R(G,\theta)$ are the sub-pointgons to the left and the right of θ, respectively.

V In this case we consider the triangle that cuts off the first vertex of the path from the rest of the pointgon, provided it is empty and either a left or a middle triangle (this case corresponds roughly to case III in Figure 2). The vertex that is the tip of this triangle (the base is the first path edge) we denote by v_l. $L(G,\theta)$ is the remaining pointgon.

VI In this case we consider the triangle that cuts off the last vertex of the path from the rest of the pointgon, provided it is empty and either a right or a middle triangle (this case corresponds roughly to case II in Figure 2). The vertex that is the tip of this triangle (the base is the last path edge) we denote by v_r. $R(G,\theta)$ is the remaining pointgon.

Given these six cases, Equation 1, which describes the general recursion to find the weight of a MWT of G, can be made more specific as follows ($v_{(1)}$ is the first, $v_{(2)}$ the second, $v_{(-2)}$ the last but one, and $v_{(-1)}$ the last vertex on the path):

$$\begin{aligned}
&\mathrm{MWT}(G)\\
&= \min\Big\{ \min_{\theta\in\Theta_l(G,\pi)} \big\{\mathrm{MWT}(L(G,\theta)) + \mathrm{MWT}(R(G,\theta)) + |(v_{(1)}, v_{(2)}| - |\theta|\big\},\\
&\qquad \min_{\theta\in\Theta_m(G,\pi)} \big\{\mathrm{MWT}(L(G,(v_{l(m)}, v_{r(m)}, v_m))) + |(v_{l(m)}, v_{r(m)})|\big\},\\
&\qquad \min_{\theta\in\Theta_c(G,\pi)} \big\{\,\mathrm{MWT}(L(G,(v_{(c-1)}, v_{(c)}, v_{(c+1)})))\\
&\qquad\qquad + |(v_{(c-1)}, v_{(c)})| + |(v_{(c)}, v_{(c+1)})|\big\},\\
&\qquad \min_{\theta\in\Theta_r(G,\pi)} \big\{\mathrm{MWT}(L(G,\theta)) + \mathrm{MWT}(R(G,\theta)) + |(v_{(-2)}, v_{(-1)}| - |\theta|\big\},\\
&\qquad \mathrm{MWT}(L(G,(v_{(1)}, v_{(2)}, v_l))) + |(v_{(1)}, v_{(2)})| + |(v_{(1)}, v_l)|,\\
&\qquad \mathrm{MWT}(R(G,(v_{(-1)}, v_{(-2)}, v_r))) + |(v_{(-1)}, v_{(-2)})| + |(v_{(-1)}, v_r)|\Big\},
\end{aligned}$$

where π is an arbitrary lexi-monotone path on the perimeter of G. The six terms in the outer minimum refer to the cases I to VI, respectively.

Although we may choose another lexi-monotone path in the recursion, it is obvious that we should maintain the one constructed in the preceding recursion step. In this way be have to consider only one type of subproblem, rendering this algorithm very simple, despite the six cases needed for processing it.

Types of Pointgons We encounter only one subproblem which is exactly the same as the type A subproblem of Algorithms 2 and 3 in Sections 4.2 and 4.3. This subproblem type is shown again in Figure 14. It consists of a single lexi-monotone path that may be clockwise or counterclockwise from the anchor.

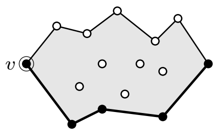


Fig. 14. The only type of subproblems we encounter in the second combined algorithm. A black point indicates a vertex belonging to V_p, a white point a vertex belonging to V_h. The encircled point is the anchor. Thick lines indicate pieces of the perimeter of the input pointgon.

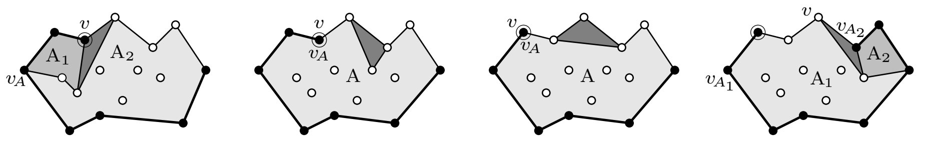


Fig. 15. Behavior of Type A pointgons in the recursion (possible splits). v denotes the original anchor and v_A denotes the anchor of a sub-pointgon of type A due to the split.

Processing the Type of Pointgons How the only type of subproblems is processed has basically been described already. The most important four cases are sketched in Figure 15. In the first sketch a left triangle is built over the first path edge and a decreasing lexi-monotone path is attached to its tip. In the second sketch a middle triangle is built over an arbitrary path edge. Note that this case can also occur with a perimeter vertex as the triangle tip. In the third sketch, a path vertex, at which the path is convex, is cut off from the pointgon and in the fourth sketch a right triangle is built over the last edge of the path and an increasing lexi-monotone path is attached to its tip. These sketches make it very clear that in all cases one or two subproblems of type A result.

Comparison to [31] Algorithm 4 shares its basic scheme with the algorithm proposed in [31], which introduced the important insight on which it is based (and which is expressed in Observation 3). However, the algorithm in [31] handles the cases of a left triangle over the first path edge and a right triangle over the last path edge differently: instead of cutting the (empty) left or right triangle immediately (as it is done in Algorithm 4, see above), only one side of the triangle, namely the one leading to the second or last but one vertex on the path, respectively, is added and extended by a lexi-monotone path. In addition, the path is built edge by edge, an idea, which we only exploit in the analysis below (as it only complicates the description of how subproblems are processed). As a consequence of these differences the algorithm in [31] leads to five types of subproblems, while our Algorithm 4 needs only one (see Figure 14).

5 Analysis

In order to estimate the time complexity of the four algorithms, we proceed in basically the same way for all of them: we multiply the (worst case) number of subproblems by the (worst case) time it takes to process one subproblem.

This processing time is estimated as the (worst case) number of possible splits of a sub-pointgon multiplied with the (worst case) time for processing one split. However, for all algorithms involving paths the analysis of the worst case time to process one subproblem has to be carried out carefully, as the large number of path splits can easily lead to an overestimate. Here we consider two technical tricks, derived from [12] and [31], to arrive at good estimates (see Section 5.2). The former groups subproblems, the second considers building paths edge by edge, formally extending the set of subproblems that can occur.

5.1 Purely Triangle-based Algorithm

Since both end vertices of the only path, which bounds a subproblem in the purely triangle-based algorithm, can be any input vertex (perimeter as well as hole vertex), while the path can contain any sequence of the hole vertices (provided each hole appears at most once), there are at most $O(n^2k!)$ possible subproblems. However, 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 [30], 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. (We also use it to evaluate our experimental results in the next section, where $k \leq 9$.)

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) intersect 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 $O(n^2)$, 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. Alternatively we may, in a preprocessing step, set up a table in which we store for each triple of input vertices whether the triangle having these vertices as corners is empty or not. The resulting table has a size of $O(n^3)$, which is often negligible compare to the number of subproblems, and its computation takes at most $O(n^3k)$ time. With such a table the check can be carried out in constant time.

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. Therefore processing one triangle takes in all $O(k)$ time.

As a consequence the overall time complexity is at most

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

5.2 Algorithms with Path Splits

In principle, algorithms with path splits or combined splits can be treated in basically the same way as the purely triangle-based algorithm. Depending on whether the set of subproblems comprises types bounded by two lexi-monotone paths or not, there are $O(n^3 3^k)$ or $O(n^2 2^k)$ possible subproblems, respectively. The factors n^3 or n^2 come from the three or two vertices that are at the ends of the bounding path(s), each of which can be any input vertex. That we only have factors of 3^k and 2^k for the possible sequences of hole vertices (compared to $k!$ for the purely triangle-based algorithm) results from the fact that all such paths have to be lexi-monotone. As a consequence we can specify the holes on the path(s) by stating whether a hole is on the clockwise path, or the counterclockwise path, or on neither of the two (three possibilities for k objects and thus 3^k possibilities for the paths) or only whether a hole is on the single bounding path or not (two possibilities for k objects and thus 2^k possibilities for the path). The order of the hole vertices on a path can be left implicit as it can be derived from the lexicographic relations of the holes that are assigned to the path.

However, multiplying this number of subproblems with the number of possible splits, which in the worst case can be as large as $O(n2^k)$, is not a such good idea in this case (even though it worked perfectly well for the purely triangle-based algorithm) as it overestimates the actual complexity. One approach to deal with this problem consists in grouping the subproblems and analyzing the groups separately [12]. The groups we consider are defined by the number of hole vertices of the input pointgon that are on the perimeter of a sub-pointgon.

So consider the number of subproblems with l, $0 \le l \le k$, hole vertices on the perimeter. We discuss only the algorithms in which subproblems can be bounded by two lexi-monotone paths and thus can contain up to three perimeter vertices of the input pointgon, namely the anchor and the two ends of the two 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 (see above), 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 splitting path and at most n and 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 efficient by the same preprocessing step we already considered for the purely triangle-based algorithm, namely setting up a table, in which we store for each edge that could be considered as part of a splitting path whether it intersects the perimeter of the input pointgon or not. With this table we can check in $O(k-l)$ whether a given splitting path intersects a (possibly existing) perimeter piece.

In addition, we have to check for an intersection with the at most two already existing splitting paths, which contain at most $l+2$ edges. By exploiting that all paths are lexi-monotone, this check can be carried out in $O(k)$. Alternatively, and more conveniently, we can construct the paths in a specific order, in which from one constructed path to the next at most two edges of the path change. These two edges have to be checked against the at most $k+2$ edges of the path(s), which also yields a time complexity of $O(k)$ for this check.

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

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

An alternative approach uses the idea to build the paths edge by edge [31]. In order to do so, we have to formally extend the set of subproblems, so that they comprise subproblems with partial paths, which start at the anchor of a subproblem or at the tip of a triangle, but do not yet split the subproblem. The only way of processing such subproblems is to extend the partial path, until the partial path reaches the perimeter of the subproblem and finally splits it. Obviously there are at most $O(n)$ such extensions of the partial path by an edge, just as they are at most $O(n)$ possible tips for a triangle. The advantage of this approach is that it is more general than the preceding one and thus can easily be applied to any path-based or combined algorithm.

For algorithms with two lexi-monotone paths bounding a subproblem, the analysis works as follows: for the extended subproblems there are at most three vertices that can be any of the $n-k$ perimeter vertices, namely the anchor and the two ends of the paths. For the holes we have to specify whether they are on the clockwise path, the counterclockwise path, the partial path, or neither of them. Hence we have $O(n^3 4^k)$ possible subproblems, each of which can be extended or split in n possible ways. Consequently, as is easy to ascertain, we get the same time complexity we already computed above.

For Algorithm 4 (second combined algorithm), in which all subproblems are bounded by only one lexi-monotone path, there are two vertices (the start and end vertex of the path) that can be any of the $n - k$ perimeter vertices. For the holes we only have to specify whether they are on a path or not. We do not even have to distinguish whether they are on the path bounding the subproblem or on the partial path, since the partial path must be completely to the left of the leftmost vertex or completely to the right of rightmost vertex of the bounding path. The reason is that the partial path is either a decreasing path attached to the tip of a left triangle over the leftmost edge or an increasing path attached to the tip of a right triangle over the rightmost edge of the bounding path. Hence the lexicographic relation of the vertices already determines which of the paths (bounding or partial) a hole vertex is on: if it is between the start and the end vertex of the bounding path (which are, of course, perimeter vertices), it is on the bounding path, otherwise it is on the partial path.

As a consequence we have at most $O(n^2 2^k)$ possible subproblems, each of which can be extended in $O(n)$ possible ways. Checking an extension and building the new subproblem key takes at most $O(k)$ time, relying on the same arguments we already studied above. Therefore the overall time complexity is

$$O\Bigg(\underbrace{n^2 2^k}_{\text{subproblems}} \cdot \underbrace{n}_{\text{extensions}} \cdot \underbrace{k}_{\text{time/extension}}\Bigg) = O(n^3 2^k k).$$

6 Implementation and Experiments

We implemented all four algorithms in Java. In this implementation we always choose the leftmost vertex of the input pointgon as the initial anchor. Of course, in principle any vertex could be chosen as the anchor. However, the leftmost vertex has the practical advantage that it allows us to determine easily whether the perimeter vertices are given in clockwise or in counterclockwise order and thus to take care of this input variant appropriately.

At least for Algorithms 2, 3, and 4 (that is, for those that have subproblems bounded solely by lexi-monotone paths) 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 arrays or lists of vertices for the bounding paths 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 $O(k \log k)$ for stating the holes on the bounding path(s). For lexi-monotone paths a better way would be to use bit vectors of length k, each bit of which corresponds to 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 the holes on the bounding path(s). For an implementation, however, such a way of coding the index words is, at best, inconvenient. Therefore we used the simple array/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.

$n-k$	k	weight	triangles	subprobs.	splits	time in seconds	time/$n^3\,k!k$
3	1	7.8745	3.0	3.0	2.0	0.008 ± 0.001	$1.172 \cdot 10^{-4}$
6	1	11.5363	29.0	19.2	32.5	0.008 ± 0.000	$2.362 \cdot 10^{-5}$
9	2	15.4201	124.2	98.0	255.8	0.013 ± 0.001	$2.372 \cdot 10^{-6}$
12	3	18.3976	309.9	395.3	1272.8	0.035 ± 0.004	$5.791 \cdot 10^{-7}$
15	4	20.8347	602.7	1560.3	5870.9	0.055 ± 0.005	$8.363 \cdot 10^{-8}$
18	5	23.0442	1013.6	5393.8	22935.9	0.078 ± 0.009	$1.074 \cdot 10^{-8}$
21	6	24.9047	1541.3	18149.0	84364.8	0.166 ± 0.043	$1.956 \cdot 10^{-9}$
24	7	26.5899	2195.9	63802.2	322239.8	0.560 ± 0.202	$5.327 \cdot 10^{-10}$
27	8	28.0598	2957.7	217918.0	1176956.8	1.981 ± 0.818	$1.432 \cdot 10^{-10}$
30	9	29.4855	3858.2	744608.9	4198128.7	7.248 ± 3.308	$3.741 \cdot 10^{-11}$

Table 1. Experimental results for the purely triangle-based algorithm.

$n-k$	k	weight	triangles	subprobs.	splits	time in seconds	time/$n^4 4^k k$
3	1	7.8745	3.0	5.1	2.2	0.009 ± 0.000	$8.682 \cdot 10^{-6}$
6	1	11.5363	29.0	59.9	75.5	0.010 ± 0.000	$1.066 \cdot 10^{-6}$
9	2	15.4201	124.2	428.6	1116.2	0.041 ± 0.004	$8.668 \cdot 10^{-8}$
12	3	18.3976	309.9	1974.8	8503.7	0.077 ± 0.004	$7.954 \cdot 10^{-9}$
15	4	20.8347	602.7	7193.9	45393.3	0.125 ± 0.016	$9.359 \cdot 10^{-10}$
18	5	23.0442	1013.6	24135.9	209247.5	0.350 ± 0.079	$2.440 \cdot 10^{-10}$
21	6	24.9047	1541.3	76764.8	863987.5	1.345 ± 0.454	$1.030 \cdot 10^{-10}$
24	7	26.5899	2195.9	229059.4	3214433.1	5.119 ± 1.902	$4.833 \cdot 10^{-11}$
27	8	28.0598	2957.7	694656.8	12039561.9	20.534 ± 8.944	$2.610 \cdot 10^{-11}$
30	9	29.4855	3858.2	2001491.8	41397261.7	76.538 ± 32.125	$1.402 \cdot 10^{-11}$

Table 2. Experimental results for the purely path-based algorithm.

Another technical issue is that for Algorithm 4 (second combined algorithm), we chose the vertex at the clockwise end of the only bounding path as the anchor. In contrast to this, in Algorithms 2 and 3 the anchor of a type A subproblem is the leftmost vertex of the path, so that the anchor may be at the clockwise or at the counterclockwise end of the path, depending on the exact structure of the subproblem. The reason for this deviation are specific implementation problems of Algorithm 4, which can be resolved more easily if the anchor is always at the clockwise end of the path bounding a type A subproblem.

Example results, which we obtained with our implementation for different numbers of holes and perimeter vertices, are shown in Tables 1 to 4. The test system was an Intel Pentium 4C@2.6GHz with 1GB of main memory running S.u.S.E. Linux 10.0 and Sun Java 1.5.0_03. All execution times, numbers of subproblems etc. are averages of 100 runs, carried out on randomly generated convex pointgons (exactly the same set of pointgons was generated for all algorithms). 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).

$n-k$	k	weight	triangles	subprobs.	splits	time in seconds	time/$n^4 4^k k$
3	1	7.8745	3.0	3.0	1.4	0.011 ± 0.000	$1.080 \cdot 10^{-5}$
6	1	11.5363	29.0	20.3	32.3	0.012 ± 0.000	$1.231 \cdot 10^{-6}$
9	2	15.4201	124.2	112.5	300.6	0.018 ± 0.002	$3.904 \cdot 10^{-8}$
12	3	18.3976	309.9	407.8	1579.2	0.056 ± 0.004	$5.725 \cdot 10^{-9}$
15	4	20.8347	602.7	1263.5	6902.7	0.084 ± 0.005	$6.325 \cdot 10^{-10}$
18	5	23.0442	1013.6	3616.5	26661.2	0.117 ± 0.010	$8.196 \cdot 10^{-11}$
21	6	24.9047	1541.3	10041.7	96350.7	0.223 ± 0.054	$1.709 \cdot 10^{-11}$
24	7	26.5899	2195.9	25871.3	308325.8	0.581 ± 0.196	$5.490 \cdot 10^{-12}$
27	8	28.0598	2957.7	70460.8	1052407.3	1.865 ± 0.748	$2.371 \cdot 10^{-12}$
30	9	29.4855	3858.2	184448.5	3301765.8	5.962 ± 2.399	$1.092 \cdot 10^{-12}$

Table 3. Experimental results for the first combined algorithm.

$n-k$	k	weight	triangles	subprobs.	splits	time in seconds	time/$n^3 4^k k$
3	1	7.8745	3.0	3.0	1.4	0.011 ± 0.000	$8.375 \cdot 10^{-5}$
6	1	11.5363	29.0	17.7	32.1	0.011 ± 0.001	$1.660 \cdot 10^{-5}$
9	2	15.4201	124.2	74.0	247.9	0.016 ± 0.001	$1.505 \cdot 10^{-6}$
12	3	18.3976	309.9	231.5	1161.7	0.039 ± 0.005	$4.825 \cdot 10^{-7}$
15	4	20.8347	602.7	600.9	4047.1	0.058 ± 0.004	$1.329 \cdot 10^{-7}$
18	5	23.0442	1013.6	1471.9	12649.9	0.080 ± 0.004	$4.096 \cdot 10^{-8}$
21	6	24.9047	1541.3	3509.7	36999.8	0.109 ± 0.009	$1.446 \cdot 10^{-8}$
24	7	26.5899	2195.9	8052.1	101129.0	0.171 ± 0.030	$6.420 \cdot 10^{-9}$
27	8	28.0598	2957.7	18577.3	268779.7	0.352 ± 0.089	$4.005 \cdot 10^{-9}$
30	9	29.4855	3858.2	42601.0	708736.8	0.895 ± 0.271	$3.275 \cdot 10^{-9}$

Table 4. Experimental results for the second combined algorithm.

The tables show in their first two columns the number of perimeter vertices and the number of holes, which define the problem size. The next two columns contains the averages of the weights of the found MWTs and the averages of the number of empty triangles considered by the algorithms. The fact that these two columns are the same for all four algorithms demonstrates the sanity of the implementation, as all algorithms should yield the same results. The fifths column lists the average number of subproblems that were considered and stored with the solutions in a trie structure. The six column states the average number of (valid) splits considered by the algorithms. However, when considering these numbers one has to bear in mind that all algorithms generate a lot more splits, but discard most of them because they are invalid (lead to intersections, form non-empty triangles etc.). The seventh column contains the average running times in seconds, together with the standard deviation of these running times, to give an idea of their fairly large variability.

To check our theoretical result about the time complexities, we computed the ratios of the measured execution times to the theoretical values (see the last columns of all tables; note that the four algorithms have different theoretical

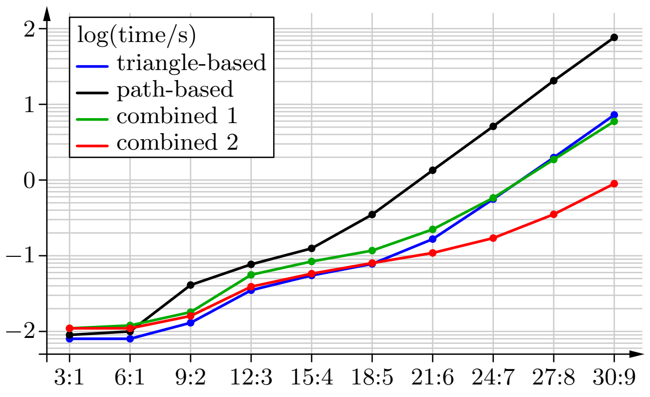


Fig. 16. Execution time over problem size (stated as $n - k : k$).

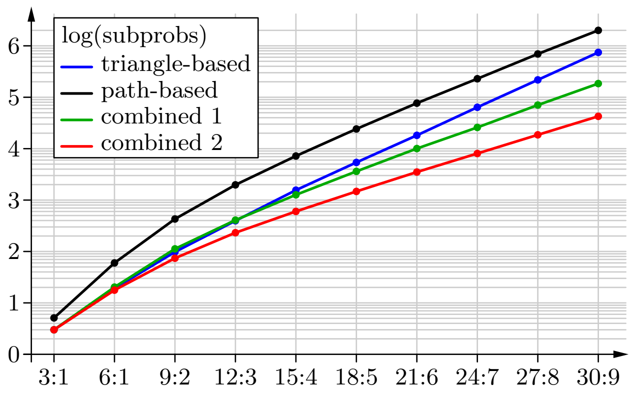


Fig. 17. Number of subproblems over problem size (stated as $n - k : k$).

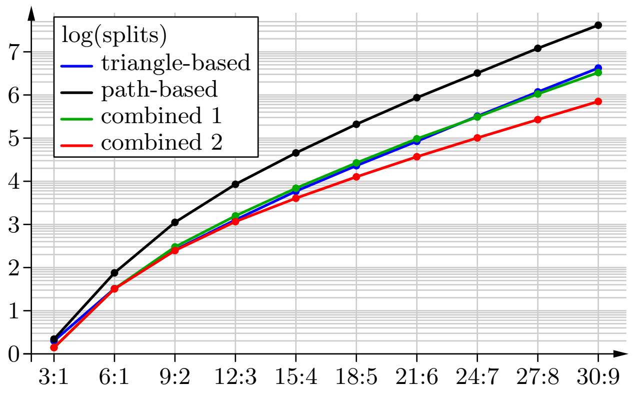
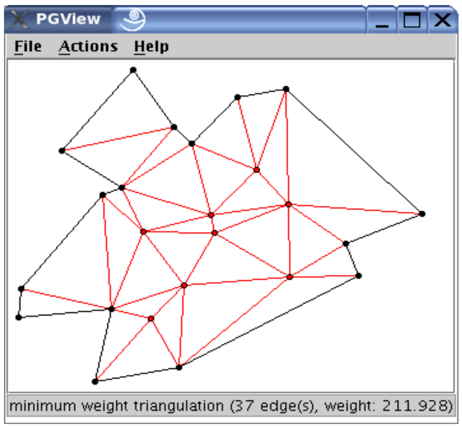


Fig. 18. Number of (valid) splits over problem size (stated as $n - k : k$).



minimum weight triangulation (37 edge(s), weight: 211.928)

Fig. 19. 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 splits 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 0.1 to 0.3 seconds (on an Intel Pentium 4C@2.6GHz system with 1GB of main memory running S.u.S.E. Linux 10.0 and Sun Java 1.5.0_03).

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.

To be able to study the relative performance of the different algorithms more easily, we plotted the decimal logarithms of the average execution time, average number of subproblems, and average number of (valid) splits over the problem sizes. Figures 16 to 18 show that the purely triangle-based algorithm, presumably due to its simpler processing scheme, is slightly better than the other algorithms up to about 4 to 5 hole vertices. Beyond that, due to its bad time complexity in the number of hole vertices, the combined algorithms are performing clearly better, winning by a considerable margin for 9 hole vertices. In addition, it becomes well visible that for larger numbers of holes the terms $k!$ (or rather b^k), 4^k, or 2^k govern the runtime as well as the number of subproblems, since all curves become almost straight lines (in this logarithmic plot) for $k \geq 5$.

To give an impression of the graphical user interface (GUI) of the program, Figure 19 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.

7 Conclusions

We discussed four fixed parameter algorithms for the minimum weight triangulation problem, which can be arranged into a nice sequence, in which ideas are added from one algorithm to the next. By studying the algorithms in this way, starting from special versions that use the two split types (triangle splits and path splits) individually, and proceeding to algorithms which combine then in order to achieve a higher efficiency, the ideas underlying these algorithms could be brought out very clearly. The final algorithm, which is a considerably simplified version of an algorithm by [31], exhibits excellent performance. From an implementation point of view, however, the purely triangle-based algorithm [13] also deserves attention, as it is definitely the easiest to implement. The theoretical time complexities of the algorithms are in overview:

purely triangle-based algorithm:	$O(n^3 k!\, k)$ or $O(n^3 b^k k)$,
purely path-based algorithm:	$O(n^4 4^k k)$,
first combined algorithm:	$O(n^4 4^k k)$,
second combined algorithm:	$O(n^3 2^k k)$,

where b is some constant. The experiments we carried out with our implementation show that these are worst case estimates, since average results in practice are considerably better (decreasing ratio of actual time and theoretical estimate).

8 Software

The Java source code of our implementation as well as an executable Java archive (jar) for the GUI version of the program can be downloaded free of charge at:

`http://fuzzy.cs.uni-magdeburg.de/~borgelt/pointgon.html`.

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 source package of the program contains two `bash` shell scripts we used for these experiments (files `bench` and `benchall` in the directory `pointgon`), so that all experiments reported here can easily be repeated in order to check our claims.

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. P. Belleville, M. Keil, M. McAllister and J. Snoeyink. On Computing Edges that are in all Minimum-Weight Triangulations. *Proc. 12th Ann. Symp. on Computational Geometry (SoCG'96, Philadelphia, PA, USA)*, 507–508. ACM Press, New York, NY, USA 1996
3. P. Bose, L. Devroye and W. Evans. Diamonds are not a Minimum Weight Triangulation's Best Friend. *Int. J. Comput. Geom. Appl.* 12:445–453. World Scientific, Hackensack, NJ, USA 2002

4. S. Cheng, M. Golin, J. Tsang. Expected-case Analysis of β-Skeletons with Applications to the Construction of Minimum Weight Triangulation. *Proc. 7th Canadian Conf. on Computational Geometry (CCCG'1995, Quebec, Canada)*, 279–283. CCCG Organization Committee, Canada 1995
5. S. Cheng, N. Katoh and M. Sugai. A Study of the LMT-Skeleton. *Proc. 7th Ann. Int. Symp. on Algorithms and Computation (ISAAC'96, Osaka, Japan)*, LNCS 1178:256–265. Springer-Verlag, Heidelberg, Germany 1996
6. S. Cheng and Y. Xu. On β-Skeleton as a Subgraph of the Minimum Weight Triangulation. *Theo. Comput. Sci.* 262:459–471. Elsevier, Amsterdam, Netherlands 2001
7. T.H. Cormen, C.E. Leiverson, and R.L. Rivest. *Introduction to Algorithms.* MIT Press/McGraw Hill, Cambridge, MA, and New York, NY, USA 1989
8. M. Dickerson and M. Montague. A (Usually) Connected Subgraph of the Minimum Weight Triangulation. *Proc. 12th Ann. Symp. on Computational Geometry (SoCG'96, Philadelphia, Pennsylvania, USA)*, 204–213. ACM Press, New York, NY, USA 1996
9. R. Downey and M. Fellows. *Parameterized Complexity.* Springer-Verlag, New York, NY, USA 1999
10. M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to Theory of NP-Completeness.* Freeman, New York, NY, USA 1979
11. P.D. Gilbert. *New Results in Planar Triangulations.* Report R-850, University of Illinois, Coordinated Science Lab, Urbana, IL, USA 1979
12. 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
13. M. Grantson, C. Borgelt and C. Levcopoulos. Minimum Weight Triangulation by Cutting Out Triangles. *Proc. 16th Ann. Int. Symp. on Algorithms and Computation (ISAAC'05, Sanya, China)*, LNCS 3827:984–994. Springer-Verlag, Heidelberg, Germany 2005
14. L. Heath and S Pemmaraju. New Results for the Minimum Weight Triangulation Problem. *Algorithmica* 12:533–552. Springer-Verlag, New York, NY, USA 1994
15. M. Hoffmann and Y. Okamoto. The Minimum Triangulation Problem with Few Inner Points. *Proc. 1st Int. Workshop on Parameterized and Exact Computation (IWPEC 2004)*, 200–212. Springer-Verlag, Berlin, Germany 2004
16. J. Keil. Computing a Subgraph of the Minimum Weight Triangulation. *Computational Geometry* 4:13–26. Elsevier, Amsterdam, Netherlands 1994
17. G.T. Klincsek. Minimal Triangulations of Polygonal Domains. *Annals of Discrete Mathematics* 9:121–123. ACM Press, New York, NY, USA 1980
18. G.T. Klincsek. A Note on Delaunay and Optimal Triangulations. *Inform. Process. Letters* 10:127–128. Elsevier, Amsterdam, Netherlands 1980
19. Y. Kyoda, K. Imai, F. Takeuchi, and A. Tajima. A Branch-and-Cut Approach for Minimum Weight Triangulation. *Proc. 8th Ann. Int. Symp. on Algorithms and Computation (ISAAC'97, Kent Ridge, Singapore)*, LNCS 1350:384–393. Springer-Verlag, Heidelberg, Germany 1997
20. C. Levcopoulos. An $\Omega(\sqrt{n})$ Lower Bound for Non-optimality of Greedy Triangulation. *Inform. Process. Letters* 25:247–251. Elsevier, Amsterdam, Netherlands 1987
21. C. Levcopoulos and D. Krznaric. Quasi-Greedy Triangulations Approximating the Minimum Weight Triangulation. *Journal of Algorithms* 27(2):303–338. Academic Press, San Diego, CA, USA 1998

22. A. Lingas. A New Heuristics for the Minimum Weight Triangulation. *SIAM J. Algebraic and Discrete Methods* 8:646–658. Society of Industrial and Applied Mathematics, Philadelphia, PA, USA 1987
23. E.L. Lloyd. On Triangulations of a Set of Points in the Plane. *Proc. 18th IEEE Symp. Found. Comp. Sci. (FOCS'77, Providence, RI)*, 228–240. IEEE Press, Piscataway, NJ, USA 1977
24. 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
25. 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
26. G. Manacher and A. Zobrist. Neither the Greedy nor the Delaunay Triangulation Approximates the Optimum. *Inform. Process. Letters* 9:31–34. Elsevier, Amsterdam, Netherlands 1979
27. D. Moitra. Finding a Minimum Cover for Binary Images: An Optimal Parallel Algorithm. *Algorithmica* 6:624–657. Springer-Verlag, Heidelberg, Germany 1991
28. W. Mulzer and G. Rote. Minimum-weight triangulation is NP-hard. *Proc. 22nd Ann. Symp. on Computational Geometry (SoCG'06, Sedona, AZ, USA)*, to appear. ACM Press, New York, NY, USA 2006
29. D. Plaisted and J. Hong. A Heuristic Triangulation Algorithm. *Journal of Algorithms* 8:405–437 Academic Press, San Diego, CA, USA 1987
30. 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'06, Florida, USA)*, 860–869. Society of Industrial and Applied Mathematics, Philadelphia, PA, USA 2006
31. A. Spillner. A Faster Algorithm for the Minimum Weight Triangulation Problem with Few Inner Points. *Proc. 1st Workshop Algorithms and Complexity in Durham (ACiD'05, Durham, UK)* 135–146. ACiD Organization Committee, Durham, UK 2005
32. C. Wang and B. Yang. A Lower Bound for β-Skeleton Belonging to Minimum Weight Triangulations. *Computational Geometry* 19:35–46. Elsevier, Amsterdam, Netherlands 2001
33. Y. Xu. *Minimum Weight Triangulation Problem of a Planar Point Set.* Ph.D. thesis, Institute of Applied Mathematics, Academia Sinica, Beijing, China 1992