Planar Minimum-Weight Triangulations

Jesper Jansson

November 1995

Abstract

The classic problem of finding a minimum-weight triangulation for a given planar straight-line graph is considered in this paper. A brief overview of known methods is given in addition to some new results. A parallel greedy triangulation algorithm is presented along with experimental data that suggest a logarithmic relationship between the number of input vertices and its running time in the average case. Next, some new graph-based algorithms for parallel dynamic programming for simple polygons are described and analyzed. They are combined into a general algorithm that can be employed when the number of available processors is anywhere between $O(n^2/\log(n))$ and $O(n^6/\log(n))$ to reach running times that vary from $O(n\log(n))$ to $O(\log(n)^2)$, where n denotes the number of vertices that the polygons contain. As a side effect, a method for constructing the greedy triangulation of simple polygons with $O(n^{4.75}/\log(n)^2)$ processors in $O(n^{0.75}\log(n))$ time is obtained. Finally, an $\Omega(n)$ lower bound for the nonoptimality of the MST-triangulation heuristic and an $\Omega(\sqrt{n})$ lower bound for the nonoptimality of the GST-triangulation heuristic are proved.

Sammanfattning

Det klassiska problemet att hitta en triangulering av minimal längd för en given planär graf representerad med räta linjer behandlas i denna uppsats. En kort översikt över kända metoder samt några nya resultat presenteras. En parallell, girig trianguleringsalgoritm presenteras tillsammans med experimentell data som antyder ett logaritmiskt förhållande mellan antalet givna punkter och dess körtid i medelfallet. Därefter beskrivs och analyseras några nya grafbaserade algoritmer för parallell dynamisk programmering för enkla polygoner. De kombineras i en generell algoritm som kan nyttjas då antalet tillgängliga processorer befinner sig mellan $O(n^2/\log(n))$ och $O(n^6/\log(n))$ för att uppnå körtider som varierar från $O(n\log(n))$ till $O(\log(n)^2)$, där n anger antalet hörn som polygonerna innehåller. Som en sidoeffekt erhålls en metod för att konstruera den giriga trianguleringen för enkla polygoner med $O(n^{4.75}/\log(n)^2)$ processorer i $O(n^{0.75}\log(n))$ tid. Avslutningsvis bevisas en $\Omega(n)$ undre gräns för icke-optimaliteten av MST-trianguleringsheuristiken och en $\Omega(\sqrt{n})$ undre gräns för icke-optimaliteten av GST-trianguleringsheuristiken.

Acknowledgments

This paper is my thesis for a Master of Science degree in Computer Science and Technology. I would like to thank my advisor Christos Levcopoulos for all his help and patience; many of the ideas in here originated from him. I am also very grateful to Jens Pommer for proofreading an earlier version of this paper. Finally, I thank Jan Åberg for lending me his LaTeX manual.

Contents

Chapter 1

Introduction

"Geometry is not about squares.
It's about triangles and stuff."
– Beavis and Butt-Head, *Washing the Dog*

Many problems in computational geometry involve decomposing a specified geometric object into simpler geometric objects. One way to do this is by covering it with primitive components that are allowed to overlap. Another possibility is to partition the given object into distinct, nonoverlapping components. The most basic partitioning of a set of planar points and line segments is the *triangulation*, which divides the region inside the boundary into triangles. This can be done in many different ways for most sets, and some triangulations have properties that are better than those of others.

Triangulations are used by a wide range of applications as a substep in solving a presented problem. Some examples, which all depend on efficient triangulation algorithms, are numerical surface interpolation, the finite-element method, Kirkpatrick's solution to the point location problem, Lee-Preparata's shortest path algorithm for simple polygons, and creating "triangleplasma" in computer graphics.

This paper is focused on planar triangulations, as seen from an algorithmic point of view.

Chapter 2 provides the basic definitions that are used in the rest of the paper and establishes some simple theorems that come in handy later on.

Chapter 3 attempts to convince the reader that it is not a good idea to generate all different possible triangulations for a given problem and then select the most suitable one because of the enormous combinatorial explosion that occurs when the number of vertices in the graph grows large. It is shown how dynamic programming can be applied to the minimum-weight triangulation problem for simple polygons to find an optimal solution in polynomial time (a well-known result), and famous heuristic methods that approximate the best solution for arbitrary sets of vertices are discussed.

Chapter 4 looks into how the greedy triangulation heuristic can be implemented using a parallel processor model of computation. The rest of the chapter presents the results obtained from a large number of simulations that were performed to study certain aspects of the algorithm's behavior (as suggested by Christos Levcopoulos). The results imply that there is a logarithmic relationship between the number of input vertices and its running time in the average case.

Chapter 5 introduces some new dynamic programming algorithms for solving the MWT problem with parallel processors (originally invented by Christos Levcopoulos, but never published). These methods are then combined into a general algorithm whose running time varies between $O(n\log(n))$ and $O(\log(n)^2)$, depending on the number of available processors, where n is the number of vertices in the given polygons. The algorithms can be modified to handle other types of triangulations (like the greedy triangulation, for instance).

Finally, the appendix contains a detailed description of how point sets can be constructed so that if they are triangulated by two of the subgraph-based heuristic methods introduced in Chapter 3 (the MST- and GST-triangulations, to be specific), the results are not optimal in the MWT sense.

Chapter 2

Terminology

A number of concepts and definitions will be introduced here. The definitions were collected from the sources listed in the bibliography.

A *point* in $\mathbf{E}^2$ (the two-dimensional Euclidean space) can be represented by a pair of real numbers (x_1, x_2). Given two different points $\mathbf{a} = (a_1, a_2)$ and $\mathbf{b} = (b_1, b_2)$, the linear combination $\alpha\mathbf{a} + (1-\alpha)\mathbf{b}$ ($\alpha \in \mathbf{R}$ and $0 \leq \alpha \leq 1$) describes the straight line segment between $\mathbf{a}$ and $\mathbf{b}$. The *length* of the line segment $(\mathbf{a}, \mathbf{b})$ is defined as $d(\mathbf{a}, \mathbf{b}) = \sqrt{(a_1 - b_1)^2 + (a_2 - b_2)^2}$ (the Euclidean distance between $\mathbf{a}$ and $\mathbf{b}$).

Let R be a subset of $\mathbf{E}^2$ bounded by a closed curve. If R is open, i.e. doesn't contain any points of its boundary, R is called a *region.* R is *convex* if all points on the line segment $(\mathbf{a}, \mathbf{b})$ belong to R for any two points $\mathbf{a}$ and $\mathbf{b} \in R$. (Otherwise, R is *nonconvex.*) The boundary of the smallest convex subset which contains a set of points S in $\mathbf{E}^2$ is called the *convex hull* of S.

Graphs can be viewed as special combinatorial objects that can be represented geometrically. If the undirected graph $G = (V, E)$, where V is a set of *vertices* (sometimes referred to as *nodes*) and E a set of open *edges* between pairs of different elements in V and whose insides are disjoint from V, can be embedded in $\mathbf{E}^2$ so that no two images of edges intersect, then G is said to be *planar.* Every vertex is mapped to a point in the plane and every edge is mapped to a simple curve between its two corresponding endpoints. Any planar graph of the type described above can be represented in the plane by an embedding in which all images of edges are straight line segments (this was first shown by Fáry in 1948). Such a representation is known as a *planar straight-line graph* (abbreviated PSLG). See Figure 2.1 for an example.

Let $G = (V, E)$ be a PSLG. If $E = \{\}$ and $V \neq \{\}$ then G is a *planar point set.* If G consists of a single cycle, it is a *polygon.* In a polygon, every edge is connected to exactly one other edge at each endpoint. No subset of edges may have this property; in other words, polygons can't have holes. The number of edges and vertices in a polygon are always equal, so a polygon can't contain isolated vertices either. As an example, a triangle is a polygon with three vertices and three edges. Finally, a polygon is called *simple* if no pair of nonconsecutive edges share a common point.

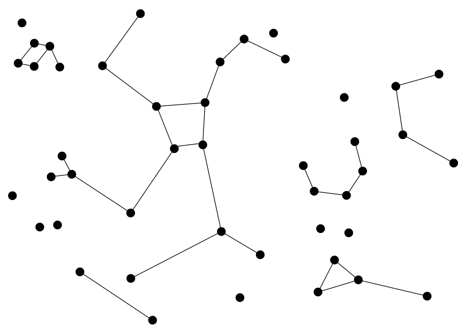


Figure 2.1: An example of a PSLG.

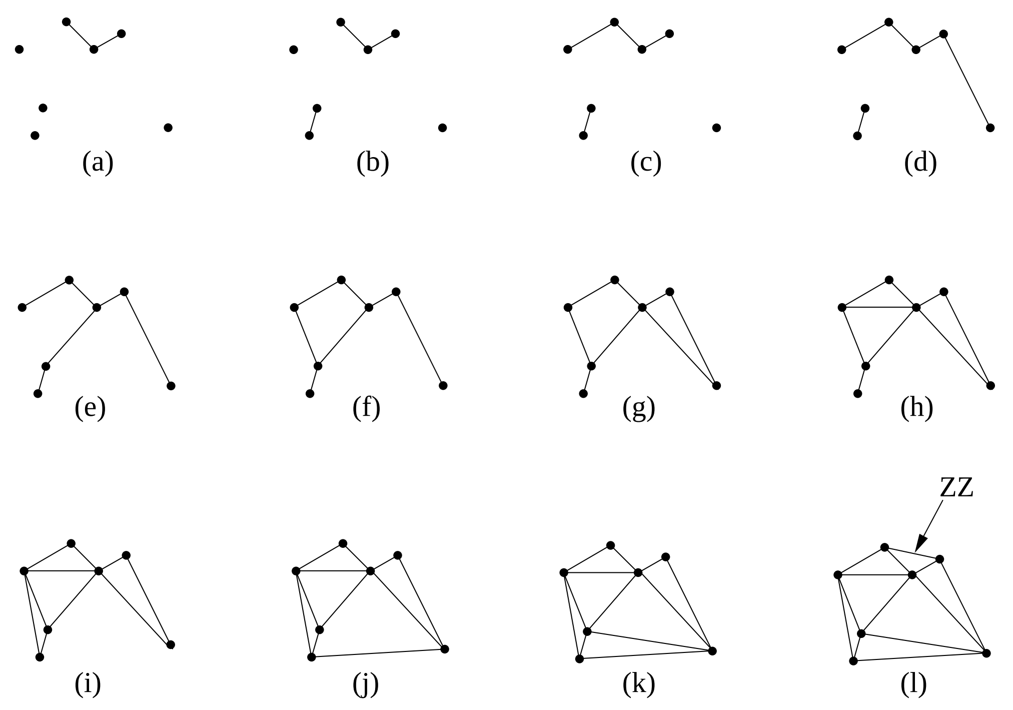


Figure 2.2: Adding nonintersecting diagonals to (a) results in a total, constrained triangulation (l). If the diagonal labelled ZZ was specified as illegal, then (k) is a triangulation since no more diagonals can be added.

The term "polygon" can be used to refer to the boundary (a curve) that separates the plane into two regions as well as to the boundary along with its interior region. We will use the term *perimeter* in this paper when we want to refer to just the boundary defining a polygon.

A *diagonal* of G is an open straight line segment with endpoints in V that doesn't intersect any edges from E nor contain any points belonging to V.

A *triangulation* of G is a PSLG $(V, E \cup E')$, where E' is defined as a maximal set of noncrossing diagonals of G. By joining the vertices of G with nonintersecting diagonals until every region inside the convex hull of G is bounded by an empty triangle (whose sides are edges or diagonals), a *total triangulation* is obtained. If G is a nonconvex polygon, diagonals on G's convex hull are considered illegal, which makes it impossible to construct a total triangulation of G. If $E \neq \{\}$ to start with, a triangulation is called a *constrained triangulation* and E its set of *constraining edges*. The process of constructing a triangulation of G is known as *triangulating G*. Figure 2.2 shows an example.

In *Steiner triangulations*, extra vertices (called *Steiner vertices* or *Steiner points*, as one might expect) can be added too. This is necessary if a triangulation with nonobtuse triangles is required for any given PSLG, for example. A constrained Steiner triangulation is a *conforming triangulation*. However, these special types of triangulations will not be discussed any further from here on.

By summing up the lengths of the edges in a triangulation T of a PSLG G, we get the *weight* of T (written as $|T|$). A *minimum-weight triangulation* (MWT in short) of G is any triangulation T with $|T|$ as small as possible. If T is a unique MWT of G, it can be denoted by $MWT(G)$. The expression $|MWT(G)|$ is well-defined even if G has more than one MWT since all MWTs of G have the same value.

Some results that will be used in subsequent chapters follow. We assume that $m \geq 3$ and $n \geq 3$.

Lemma 2.1 (Euler's Formula):
For any connected PSLG with v vertices, e edges, and r regions (the single outer infinite region included), the relation $v - e + r = 2$ always holds.

Proof:
See Grimaldi [9]. □

Theorem 2.2:
Let G be a PSLG with n vertices and let m be the number of vertices that lie on the convex hull of G. A total triangulation of G partitions G into $2n - m - 2$ triangles.

Proof:
The proof is by induction on m and n.

First of all, for $m = 3$ and $n = 3$, $2n - m - 2$ is equal to 1. This is correct since this case corresponds to a single triangle. It is marked **s** in Figure 2.3.

Next, assume that the relation is true for $m = 3$ and $n = k$. Then there are $2k - 3 - 2$ triangles in a total triangulation of G. If we add a vertex to the inside of G's convex hull, m is still equal to 3 but n is increased to $k + 1$. By drawing three new diagonals, a total triangulation of the resulting PSLG is obtained. See Figure 2.4 (a). One triangle is thus split into three smaller ones, so the total number of triangles increases by two. The total number of triangles is now equal to $2k - 3 - 2 + 2 = 2(k + 1) - 5 = 2(k + 1) - 3 - 2$. This proves all of the transitions represented by right arrows in Figure 2.3.

Finally, assume that theorem is true for $m = j$ and $n = k$, i.e. that a total triangulation of G contains $2k - j - 2$ triangles. Place a new vertex somewhere outside the convex hull of G so that it doesn't force any old vertices on G's convex hull off the resulting convex hull. The new PSLG has $m = j + 1$ and $n = k + 1$. A total triangulation is acquired by drawing two new diagonals as in Figure 2.4 (b). One new triangle is obtained, so the total number of triangles is $2k - j - 2 + 1 = 2(k + 1) - (j + 1) - 2$. This proves the rest of the transitions in Figure 2.3.

By the principle of mathematical induction, the proof for Theorem 2.2 is complete. □

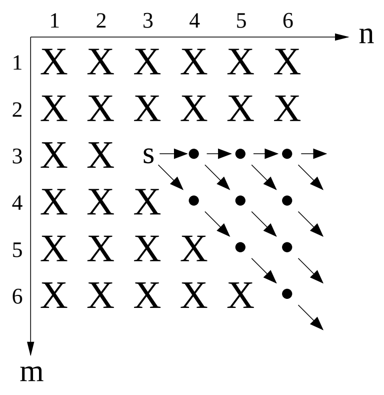


Figure 2.3: The induction proof for Theorem 2.2 starts out by showing that the theorem is valid for $m = 3$ and $n = 3$. Next, the transitions from $m = 3$ and $n = k$ to $m = 3$ and $n = k + 1$ (right arrows in the picture) are proved. Finally the remaining transitions (from $m = j$ and $n = k$ to $m = j + 1$ and $n = k + 1$) are taken care of. Thus, all cases ($n \geq m \geq 3$) are covered.

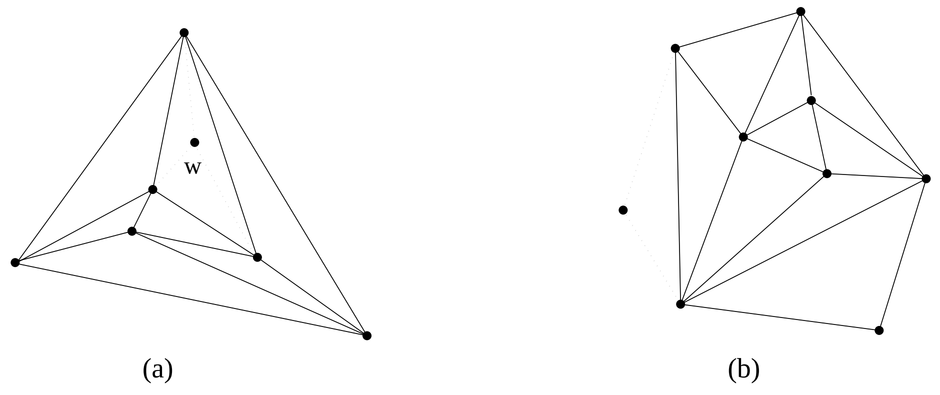


Figure 2.4: (a) A new vertex w is added to the inside of G's convex hull. Drawing diagonals from w to the three corners of the triangle in which w was placed gives us a total triangulation. (b) A new vertex is added so that the number of points on the convex hull is increased by one. A total triangulation is obtained by drawing two diagonals on the convex hull.

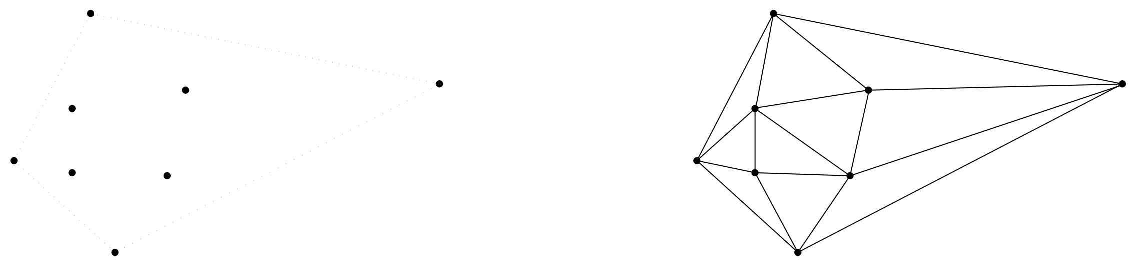


Figure 2.5: An example. The PSLG contains 8 vertices, 4 of which are located on the convex hull. Since $n = 8$ and $m = 4$, there have to be 10 triangles and 17 edges in a total triangulation.

Theorem 2.3:
Let G be a PSLG with n vertices and let m be the number of vertices that lie on the convex hull of G. Then the number of edges in a total triangulation of G is equal to $3n - m - 3$.

Proof:
Set $v = n$ and $r = 2n - m - 2 + 1$ in Lemma 2.1. Now, $e = 3n - m - 3$. □

The minimum possible value of m is 3, so we have:

Corollary 2.4:
A triangulation of a given PSLG with n vertices contains at most $3n - 6$ edges. □

Refer to Figure 2.5 for an example of how Theorem 2.2 and Theorem 2.3 can be applied.

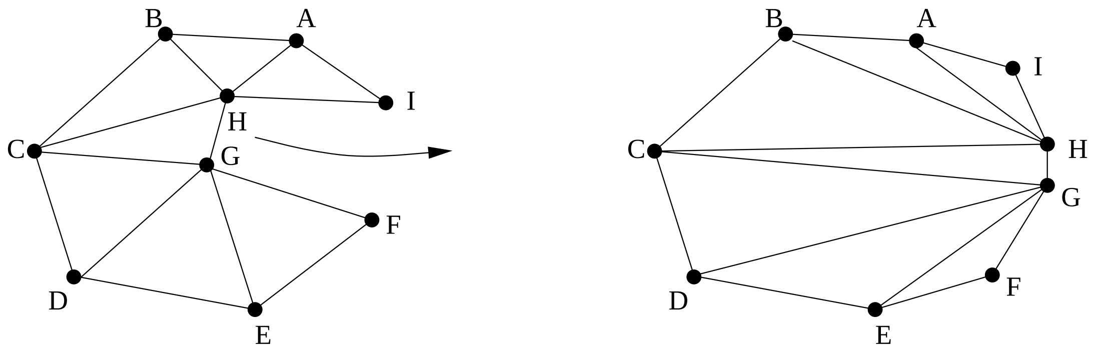


Figure 2.6: A triangulated nonconvex polygon is deformed into a triangulated convex polygon.

Theorem 2.5:
A triangulated polygon with n vertices always contains $n-2$ triangles.

Proof:
A triangulation of a convex polygon is necessarily total since every region inside the convex hull is bounded by a triangle. Furthermore, $m = n$ for all convex polygons, so Theorem 2.2 implies that any triangulated convex polygon always consists of $n-2$ triangles. A triangulated nonconvex polygon can be continuously deformed into a triangulated convex polygon without having to change any of the diagonals' endpoints. See Figure 2.6. The resulting triangulation contains the same number of triangles as the original triangulation. Thus, any triangulation of a polygon with n vertices consists of $n-2$ triangles. □

Chapter 3

Finding good triangulations

3.1 Combinatorial analysis

Depending on the criteria used to identify "good" and "bad" triangulations, searching for optimal triangulations can involve minimizing the lengths of the selected diagonals (locally or the total length of all diagonals), maximizing/minimizing the minimum/maximum induced angle, minimizing the number of obtuse angles, etc. In some cases, the data values associated with the vertices (along with their locations) can determine if one triangulation is better than another. Data dependent criteria arise in certain numerical interpolation methods.

The most straightforward way to find an optimal triangulation (there might be more than one!) for a given problem would be to generate all possible triangulations and then compare them to each other. This is feasible for PSLGs with few vertices, but when n grows large, the number of different triangulations grows even faster.

To illustrate, let P be a convex polygon with n vertices and t_n the number of different triangulations it admits. t_i for $i = 3, 4, 5, 6$ are displayed in Figure 3.1.

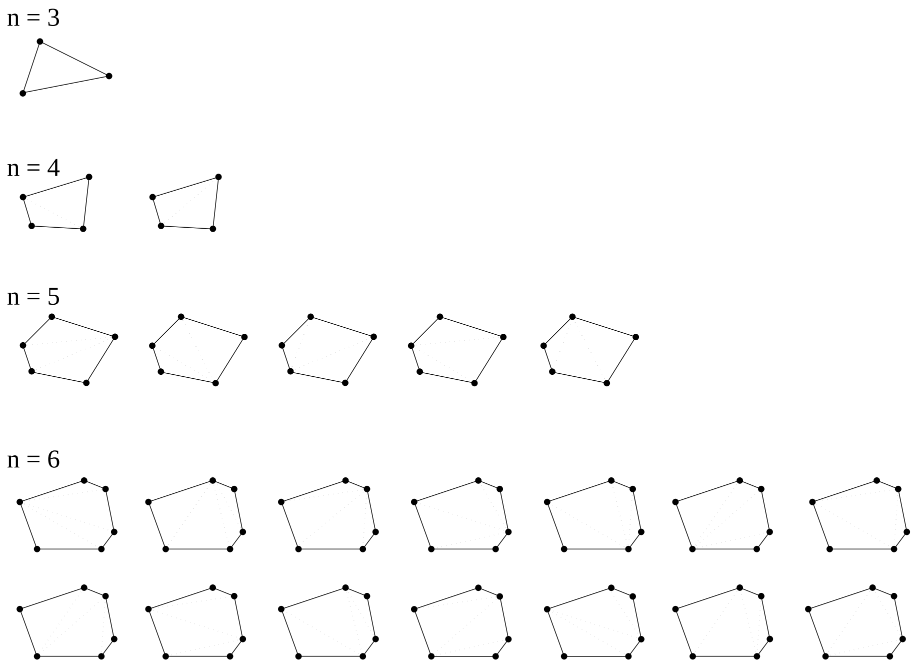


Figure 3.1: If $n = 3$, P is already a triangle, so $t_3 = 1$. For $n = 4$, there are two possibilities, which means $t_4 = 2$. When $n = 5$, there are five ways to triangulate P, and so $t_5 = 5$. Similarly, $t_6 = 14$.

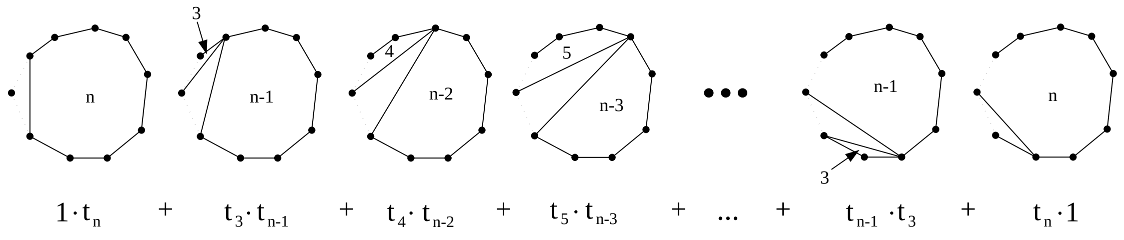

3
4
5
n
n-1
n-2
n-3
n-1
n
3
$1 \cdot t_n$ + $t_3 \cdot t_{n-1}$ + $t_4 \cdot t_{n-2}$ + $t_5 \cdot t_{n-3}$ + ... + $t_{n-1} \cdot t_3$ + $t_n \cdot 1$

Figure 3.2: Fix one point and let it belong to different triangles that cut the polygon into smaller polygons. By counting all the different triangulations for the resulting polygons and then adding them together, we get t_{n+1}.

Assume that we know t_i for $3 \le i \le n$. What is the value of t_{n+1}?

To count the number of ways in which a convex polygon with $n+1$ vertices can be triangulated, we can look at how many possible triangulations there are for different parts of the original polygon and combine the results according to Figure 3.2.

This gives us

$$t_{n+1} = 1 \cdot t_n + t_3 t_{n-1} + t_4 t_{n-2} + \ldots + t_{n-1} t_3 + t_n \cdot 1$$

Define $t_2 = 1$.
Then

$$t_{n+1} = t_2 t_n + t_3 t_{n-1} + t_4 t_{n-2} + \ldots + t_{n-1} t_3 + t_n t_2$$

Solving this relation by the method hinted at in Problem 13-4 in [4] yields

$$t_{n+1} = \frac{1}{2\,(2n-1)} \binom{2n}{n}$$

This can be rewritten as

$$t_n = \frac{1}{2\,(2n-3)} \binom{2n-2}{n-1} = \frac{1}{2\,(2n-3)} \cdot \frac{(2n-2)!}{(n-1)!\,(n-1)!} = \frac{(2n-4)!}{(n-2)!\,(n-1)!}$$

Expanding t_n, we get

$$t_n = \frac{(2n-4)!}{(n-2)!\,(n-1)!} = \frac{(2n-4)\,(2n-5)\,(2n-6)!\;2}{(n-1)\,(n-2)\,(n-3)!\,(n-2)!\;2}$$

which, when simplified into a recurrence relation, becomes

$$t_n = \frac{2\,(2n-5)}{(n-1)}\, t_{n-1} \tag{3.1}$$

Theorem 3.1:

$$t_n = \prod_{j=0}^{n-4}(1 + \frac{3\,(j+1)}{j+3})$$

for $n \geq 4$.

Proof:
Base step: $t_4 = 1 + 3 \cdot \frac{1}{3} = 2$.
Now assume that $t_{n-1} = \prod_{j=0}^{n-5}(1 + 3\frac{j+1}{j+3})$.
This, along with (3.1), yields

$$t_n = \frac{2\,(2n-5)}{n-1} t_{n-1} = \frac{4n-10}{n-1}\prod_{j=0}^{n-5}(1 + 3\frac{j+1}{j+3}) = (1 + 3\frac{n-3}{n-1})\prod_{j=0}^{n-5}(1 + 3\frac{j+1}{j+3}) = \prod_{j=0}^{n-4}(1 + 3\frac{j+1}{j+3})$$

By the principle of mathematical induction, the proof is complete. □

Examining t_n, we now see that

$$t_n = \prod_{j=0}^{n-4}(1 + 3\frac{j+1}{j+3}) \geq \prod_{j=0}^{n-4}(1 + 3 \cdot \frac{1}{3}) = \prod_{j=0}^{n-4} 2 = 2^{n-3}$$

The number of ways to triangulate a convex polygon grows exponentially with n.

In the more general case (a set of n vertices in $\mathbf{E}^2$), an upper bound of 10^{13n} on the number of triangulations has been found by Ajtai, Chvátal, Newborn, and Szemerédi [1]. What all this adds up to is that it is not practical to produce all the different triangulations and compare them to find the best one. Instead, people try to invent efficient algorithms that exploit geometric properties in clever ways so that optimal (or close to optimal) solutions can be discovered without having to check all possible alternatives.

3.2 Dynamic programming

Dynamic programming is a useful problem-solving method that combines solutions to subproblems systematically. When an encountered problem consists of subproblems that share certain sub-subproblems (i.e. the subproblems are not independent), the normal divide-and-conquer approach would repeatedly solve the same subsubproblems over and over, wasting time and resources. The dynamic programming method, however, would perform the necessary calculations once for each subsubproblem and store the answers in a table. Subsequently, the work needed to solve each such subsubproblem would already have been done.

Dynamic programming can be applied to many discrete optimization problems with optimal substructure. If an optimal solution to a given problem includes optimal solutions to its subproblems and the subproblems overlap, it might be a good idea to construct a dynamic programming algorithm.

As an example, consider the problem of finding a minimum-weight triangulation for a simple polygon (convex or nonconvex) with n vertices. It is easy to see that an optimal triangulation T of a simple polygon $P = (v_0, v_1, \ldots, v_{n-1})$ containing the diagonal $d = (v_0, v_k)$ for some k, $0 < k < n-1$, must consist of the minimal triangulations of $P_1 = (v_0, v_1, \ldots, v_k)$ and $P_2 = (v_0, v_k, v_{k+1}, \ldots, v_{n-1})$. (If P_j, $j = 1$ or 2, is not optimally triangulated, the weight of P_j's triangulation can be decreased. But this means that T is not optimal since the weight of T can be made less by using this triangulation of P_j instead.) A degenerate polygon (v_i, v_{i+1}) is considered to have an optimal triangulation with weight $|(v_i, v_{i+1})|$.

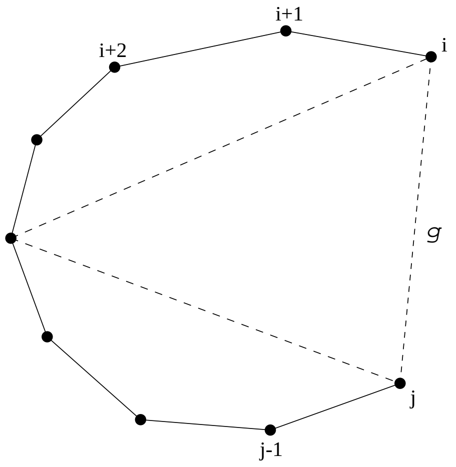


Figure 3.3: g is an edge of P' and one side of a separating triangle.

Let $t[i,j]$ be the total length of all edges belonging to an optimal triangulation for the subpolygon $P' = (v_i, v_{i+1}, \ldots, v_j)$. $(v_i, v_j) = g$ is the only edge of P' that might not be an edge of P. If g crosses any of the edges (v_k, v_{k+1}) where $i < k < j-1$, P' is not a simple polygon, and $t[i,j]$ is set to ∞. Otherwise, g must be one side of a triangle that further divides P' into two optimal (and possibly degenerate) subpolygons. See Figure 3.3. In this case,

$$t[i,j] := |(v_i, v_j)| + \min_{i<k<j} (t[i,k] + t[k,j]). \tag{3.2}$$

Note that $t[i,i+1] = |(v_i, v_{i+1})|$.

The following algorithm calculates the minimum-weight triangulation for a simple polygon by using recurrence relation (3.2). The values of $t[i,j]$ are computed and stored in a table called `t`.

```
FOR i:=0 TO (n-1) DO
   t[i,i+1 MOD n]:=|(v_i,v_(i+1 MOD n))|

FOR l:=3 TO n DO
   FOR i:=0 TO n-l DO
      j:=i+l-1
      k:=i+1
      cross:=False
      REPEAT
         cross:=Intersecting((v_i,v_j),(v_k,v_(k+1)))
         k:=k+1
      UNTIL cross OR k=j-1
      t[i,j]:=INFINITY
      S[i,j]:=-1
      IF NOT cross THEN
         FOR k:=i+1 TO j-1 DO
            q:=|(v_i,v_j)|+t[i,k]+t[k,j]
            IF q < t[i,j] THEN
               t[i,j]:=q
               S[i,j]:=k
```

The `REPEAT`-loop checks if (v_i, v_j) crosses the boundary of P, in which case $t[i,j] := \infty$. (See Chapter 4.2 for details on how the segment intersection test can be implemented.) The `k` that is selected to obtain each optimal $t[i,j]$ is stored in `S` (an auxiliary table). This way, an optimal triangulation can be traced afterwards.

The algorithm's nested loop structure gives it a running time of $O(n^3)$. The amount of memory needed is $O(n^2)$ because of the `t` and `S` tables.

Other optimal triangulation problems can be solved by modifying the algorithm. But dynamic programming is not always the most efficient method. For instance, the min-max angle problem (find a triangulation that minimizes the largest angle in all of its triangles) for a simple polygon can be solved by dynamic programming in $O(n^3)$ time and $O(n^2)$ space. Tan showed in [30] how this problem can be solved in $O(n^2 \log(n))$ time and $O(n)$ space by using edge insertion techniques.

There are other interesting problems that can be solved by dynamic programming and whose structures are identical to the one described above. Two famous examples are computing the optimal order of matrix multiplications (the number of required operations can be minimized since matrix multiplication is associative) and the construction of optimal binary search trees. Although these problems might seem totally unrelated, they can all be described by relation (3.2). Solving any one of these problems often means that the others can be solved with similar methods. The matrix multiplication problem, for example, can therefore be solved in $O(n^3)$ time and $O(n^2)$ space by dynamic programming.

3.3 Heuristics

For some difficult problems such as finding the MWT of an arbitrary PSLG, no polynomial-time methods have yet been found. However, if some sacrifices in performance are made, we can obtain powerful heuristic techniques that yield triangulations that are close to optimal and run in polynomial time.

Two popular and simple heuristics for the MWT problem are the greedy triangulation and the Delaunay triangulation. Sometimes more complicated, subgraph-based methods are employed to find slightly better solutions.

3.3.1 The greedy triangulation

Greedy algorithms can be employed in optimization problems that consist of a sequence of steps, where each step involves making a selection from a set of choices. A greedy algorithm makes a locally optimal choice in every step, never undoing what it has done before.

The greedy triangulation algorithm starts out with the set of edges and the set of vertices originally in a given PSLG G. In each step, it then inserts the shortest compatible diagonal between nodes in G until no more diagonals can be added. (A compatible diagonal is one that does not intersect any of the previously generated ones or any of the original edges.) The resulting triangulation is denoted by $GT(G)$. Figure 3.4 shows an example. If two or more distances between pairs of points in G are exactly the same, $GT(G)$ is not necessarily unique.

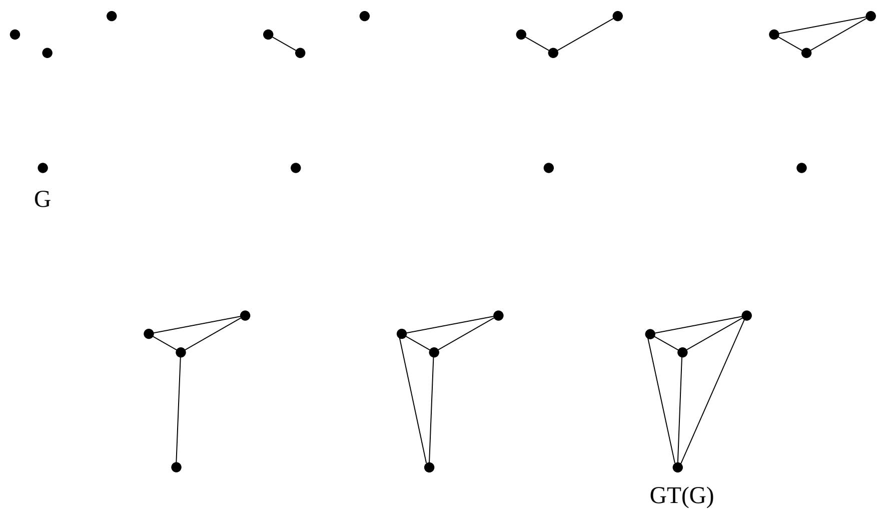


Figure 3.4: G, shortest compatible diagonal repeatedly added, and $GT(G)$.

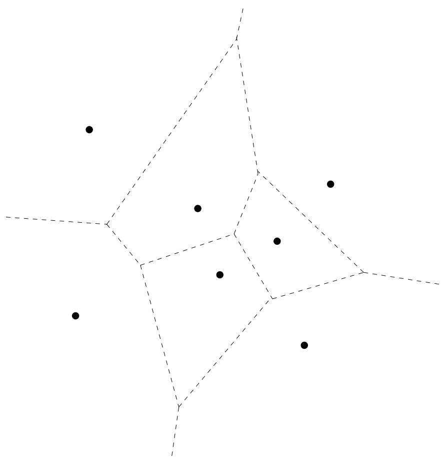


Figure 3.5: A set of points and its Voronoi diagram.

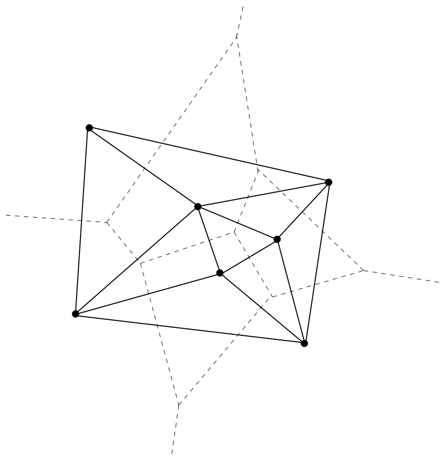


Figure 3.6: The straight-line dual of the Voronoi diagram results in $DT(S)$. Observe that a Voronoi edge and its dual line segment don't actually have to intersect.

3.3.2 The Delaunay triangulation

The Delaunay triangulation method is an extensively studied heuristic which has been shown to possess lots of good properties. For example, compared to all triangulations of a given PSLG, the Delaunay triangulation is the one that maximizes the minimum angle of all induced triangles.

Given a PSLG G, its Delaunay triangulation $DT(G)$ is obtained by taking the straight-line dual of its Voronoi diagram. If S is a set of n points in the plane, the Voronoi diagram of S is a partition of the plane into n distinct convex regions. Each region is associated with one point p_i in S, and consists of the locus of points closer to p_i than to any other point in S. Obviously, the Voronoi diagram contains information about point proximity. Refer to Figure 3.5 for an example.

By drawing line segments between each pair of points in S whose Voronoi regions share an edge and adding the points in S, we get the straight-line dual graph $DT(S)$. An example is given in Figure 3.6. If no four points of the original set S are cocircular (the nondegenerate case), this graph is in fact a triangulation of S. Examining adjacent Voronoi regions, one can see that every vertex of the Voronoi diagram is the common intersection of exactly three Voronoi edges and corresponds to one triangle in $DT(S)$. A detailed proof of why $DT(S)$ is a triangulation is presented in [27] and will not be given here.

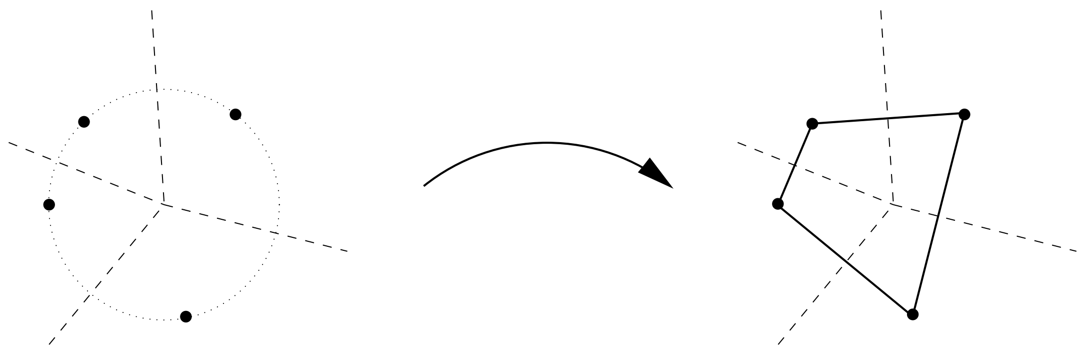


Figure 3.7: An example of a degenerate set of points (all four points lie on the same circle).

In the degenerate case, auxiliary segments have to be added to the straight-line dual to make a triangulation. See Figure 3.7.

If the given PSLG contains edges as well as vertices, the concept of Voronoi diagrams has to be generalized to Voronoi diagrams with barriers. In [18], $Vorb(G)$ for a given PSLG G $(= (V, E))$ is defined as the minimal set of straight-line segments and half-lines that complements G to the partition of the plane into regions $P(v)$, $v \in V$. (For any $v \in V$, the region $P(v)$ consists of all points p in the plane for which the shortest, open straight-line segment between p and a vertex of G that does not intersect any edge of G is the distance between p and v.)

If a metric other than the conventional Euclidean ($\mathbf{L}_2$) one is used, the Voronoi diagram is not always unique. Historically, this lead to the development of a more general definition of the Delaunay triangulation based on the fact that the circumcircle of each Delaunay triangle never contains any vertices in its interior. But since the Euclidean metric is employed in most applications, the simple definition based on the relationship between Voronoi diagrams and their straight-line duals is often used.

3.3.3 Implementations of the greedy and Delaunay triangulations

An obvious way to implement the greedy triangulation heuristic is by generating all diagonals between points in G, sorting them by increasing lengths, and then testing one diagonal at a time. In a PSLG with n vertices, there are $\binom{n}{2}$ pairs, so this method requires $O(n^2)$ space. The running time is

$$T(n) = O(n^2 \log(n)) + O(n^2) \cdot \alpha(n) + O(n) \cdot \beta(n)$$

where $O(n^2 \log(n))$ is the time it takes to sort the diagonal lengths, $\alpha(n)$ the time required to test a new diagonal for compatibility, and $\beta(n)$ the time required to update the data structure whenever a diagonal is added. A naive compatibility test (check every new potential diagonal against the $O(n)$ many edges and diagonals currently in the triangulation) would give us $\alpha(n) = O(n)$ and therefore $T(n) = O(n^3)$. Gilbert [7] came up with a more efficient edge test that uses segment trees, obtaining $\alpha(n) = O(\log(n))$ and $\beta(n) = O(n \log(n))$, and thus $T(n) = O(n^2 \log(n))$.

To get bounds lower than these, there are ways to generate compatible diagonals only. Lingas [22] (and independently Goldman [8]) invented $O(n)$ space implementations that utilize Voronoi diagrams with barriers. Levcopoulos and Lingas later found an $O(n^2)$ time and $O(n)$ space method by performing the update step in $O(n)$ time [17]. Also in [17], a linear time algorithm for producing the greedy triangulation of convex polygons was presented. Since then, Levcopoulos and Lingas have improved the total expected running time to $O(n)$ for a set of points distributed uniformly in a square [18]. Recently, Dickerson, Drysdale, McElfresh and Welzl published some practical algorithms in [5] for greedy triangulations that aren't as good asymptotically in the worst case, but that work for general convex regions (not just squares and rectangles) and are easier to implement.

The Delaunay triangulation of G can be obtained from the corresponding Voronoi diagram with barriers in $O(n)$ time. According to Wang and Schubert [31], $Vorb(G)$ can be constructed in $O(n \log(n))$ time and linear space, so $DT(G)$ can be found in $O(n \log(n)) + O(n) = O(n \log(n))$ time and $O(n)$ space.

Levcopoulos and Krznaric [14] have shown how $GT(S)$ can be calculated from $DT(S)$ in linear time, where S is a given set of planar points. Hence, they get an $O(n \log(n)) + O(n) = O(n \log(n))$ time algorithm for computing the greedy triangulation of planar point sets. For applications where the Voronoi diagram has to be constructed anyway, their approach is especially useful.

3.3.4 Performance of the greedy and Delaunay triangulations

Neither the greedy triangulation heuristic nor the Delaunay triangulation heuristic produce triangulations that are guaranteed to be optimal in the MWT sense. See Figure 3.8 for two classic examples. In fact, for a given planar point set S, $|GT(S)| = \Omega(\sqrt{n})\ |MWT(S)|$ (see [13]) and $|DT(S)| = \Omega(n)\ |MWT(S)|$ (see [24]).

However, for convex polygons, the weight of a greedy triangulation is always within a constant factor of the weight of a corresponding MWT [16].

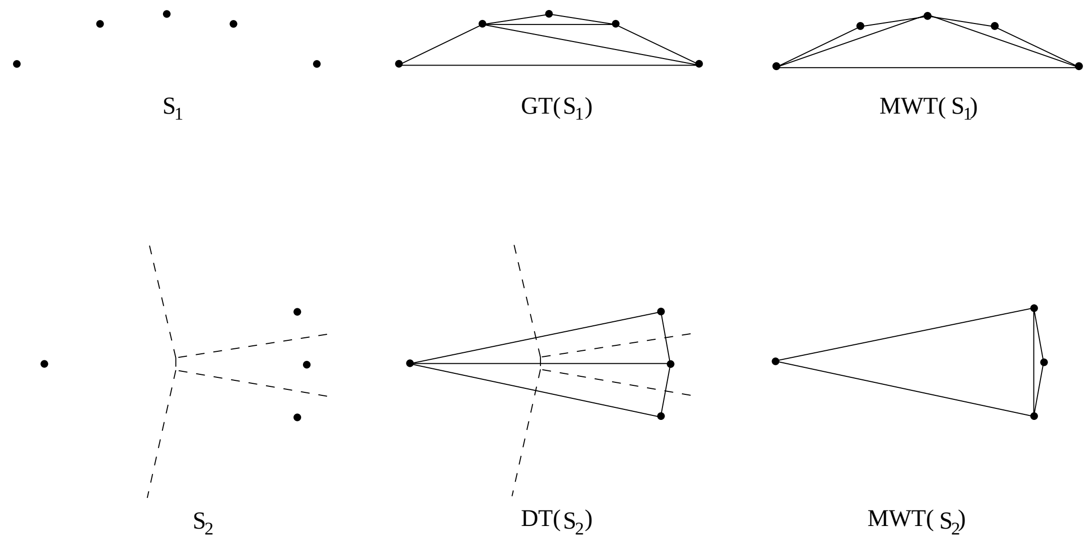


Figure 3.8: $GT(S)$ and $DT(S)$ are not always optimal.

3.3.5 Subgraph-based heuristics

As mentioned above, there exist other, more devious heuristics for the MWT problem. Subgraph-based heuristics divide an input PSLG into simple polygons which are then triangulated in polynomial time, e.g. by dynamic programming. The difficult part is finding diagonals in polynomial time that belong to the optimal solution and that separate the PSLG into simple polygons. However, it is not so hard to find and use interior diagonals that have a good chance of being part of the optimal solution together with the missing pieces of the convex hull. Some subgraph-based methods use the greedy triangulation or the Delaunay triangulation as a substep for this purpose.

The heuristics below assume that the given PSLG is a planar point set.

MST-triangulation:

Let S be a planar point set with n vertices. $T_{MST}(S)$ is the triangulation obtained by

- Computing the Delaunay triangulation of S
- Using the edges in $DT(S)$ to construct a minimum-spanning tree $MST(S)$ for S (edges belonging to the convex hull of S are considered to have cost 0)
- Augmenting it with the rest of the convex hull of S
- Triangulating the resulting polygons optimally

The method relies on the fact that $MST(S) \subseteq DT(S)$ (see [27]).

Sometimes "polygons" that look like the one in Figure 3.9 have to be triangulated in the last step. The dynamic programming algorithm from Section 3.2 can be used after a small modification: represent vertex X as two different vertices with the same coordinates. Even after adding new vertices, there will never be more than twice the number of original vertices. Thus, the time complexity is still $O(n^3)$. It can't be lower than this; in the worst case, only one polygon is created. See Figure 3.10.

Lingas proved in [21] that $T_{MST}(S)$ is at least as good as $DT(S)$, and that $|T_{MST}(S)|$ is within a factor of $O(\log(n))$ from $|MWT(S)|$ with high probability for a uniformly distributed set of points. But for every $n \geq 4$, there exists a planar point set S with n vertices such that $|T_{MST}(S)| \; / \; |MWT(S)| = \Omega(n)$. This is illustrated in Appendix A.1.

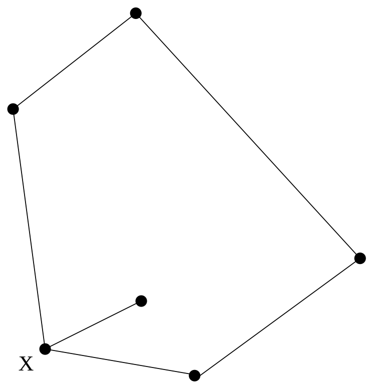


Figure 3.9: Figures such as this one might arise when the MST-triangulation method is applied.

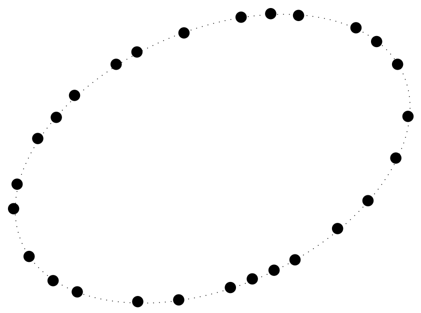


Figure 3.10: If the points lie distributed on a convex shape like in this example, a single polygon containing all of the n vertices is created by the MST-triangulation heuristic and then optimally triangulated.

GST-triangulation:

Let S be a planar point set with n vertices. $T_{GST}(S)$ is the triangulation obtained by

- Computing the greedy triangulation of S
- Using only edges in $GT(S)$ to construct a spanning tree for S of shortest possible length (edges belonging to the convex hull of S are considered to have cost 0)
- Augmenting it with the rest of the convex hull of S
- Triangulating the resulting polygons optimally

By using dynamic programming in the last step, an $O(n^3)$ implementation is obtainable.

Heath and Pemmaraju [10] showed that $T_{GST}(S)$ is at least as good as $GT(S)$. However, for every $n \geq 17$, there exists a planar point set S with n vertices such that $|T_{GST}(S)| \; / \; |MWT(S)| = \Omega(\sqrt{n})$. This is shown in Appendix A.2.

Plaisted & Hong-triangulation:

Plaisted and Hong developed a complex method in [26] that initially computes a graph (a set of "stars" from each point in a given set S) which is then modified by replacing crossed edges according to special rules so that a number of convex polygons are produced. A ring heuristic is employed to triangulate each of the induced polygons. The result turns out to be a solution whose total edge length is within a factor of $O(\log(n))$ of $|MWT(S)|$. Smith implemented the heuristic in [29] to run in $O(n^2 \log(n))$ time.

Chapter 4

Parallel greedy triangulation

4.1 Concepts

4.1.1 Model of computation

A *parallel random-access machine* (PRAM) is a model of computation in which p ordinary processors $P_0, P_1, \ldots, P_{p-1}$ (with local memories and registers) share a global memory. All processors can perform logical and arithmetic operations or read from or write to different parts of the global memory simultaneously. We will assume that a memory access takes one time unit.

PRAM algorithms can be divided into four categories:

- EREW (Exclusive Read and Exclusive Write)
- CREW (Concurrent Read and Exclusive Write)
- ERCW (Exclusive Read and Concurrent Write)
- CRCW (Concurrent Read and Concurrent Write)

The common-CRCW model (in which several processors wanting to write to the same memory address have to write the same value) will be used in this chapter.

The parallel instruction `FOR x:=1 TO s, in parallel, DO instruction(x)` assigns one processor to each `x` and executes `instruction(x)` for every `x` at the same time.

4.1.2 Parallel complexity

An *NC-algorithm* is an algorithm that runs in polylogarithmic ($\log^{O(1)}$) time and uses a polynomial number of processors. The set of problems that can be solved by NC-algorithms is called *NC*.

Some problems are *P-complete*. What this means is that they are solvable in polynomial time, and have an interesting property: all other problems solvable in polynomial time can be reduced to one of them in polylogarithmic time by using a polynomial number of processors. Thus, showing

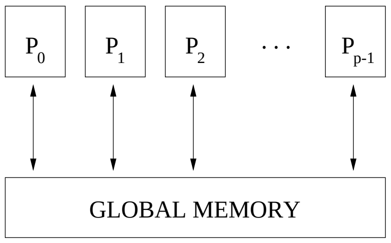


Figure 4.1: The PRAM model.

that a P-complete problem can be solved by an NC-algorithm would prove that NC = the complexity class P (the set of problems solvable in polynomial time). This is probably not true, however, and P-complete problems are generally considered to be more complex than those belonging to NC.

Levcopoulos, Lingas, and Wang proved in [19] that constructing the greedy triangulation of a finite set of planar points with integer coordinates is a P-complete problem.

4.2 A parallel greedy triangulation algorithm

Given a PSLG G consisting of n vertices, the greedy triangulation can be produced by $O(n^4)$ parallel processors in $O(n)$ time. This is accomplished by assigning one processor to each pair of possible line segments between two vertices in G (in reality there are only $\frac{\frac{n(n-1)}{2}(\frac{n(n-1)}{2}-1)}{2} = O(n^4)$ such pairs, but we might as well use $\frac{n(n-1)}{2}\frac{n(n-1)}{2} = O(n^4)$ processors to make things simpler), and employing the following common-CRCW algorithm:

```
s:=n(n-1)/2

FOR i:=1 TO s, in parallel, DO
   IF S[i] belongs to G THEN
      S[i].status:=Selected
   ELSE
      S[i].status:=Neutral

REPEAT
   finished:=True

   FOR i:=1 TO s, in parallel, DO
      IF S[i].status = Neutral THEN
         V[i]:=True
         W[i]:=True

   FOR i:=1 TO s  &  j:=1 TO s, in parallel, DO
      IF i<>j THEN
         IF S[j].status = Selected THEN
            IF Intersecting(S[i],S[j]) THEN
               W[i]:=False

   FOR i:=1 TO s, in parallel, DO
      IF S[i].status = Neutral THEN
         IF NOT W[i] THEN
            S[i].status:=Rejected

   FOR i:=1 TO s  &  j:=1 TO s, in parallel, DO
      IF i<>j THEN
         IF S[j].status = Neutral THEN
            IF Intersecting(S[i],S[j]) THEN
               IF (S[i].length > S[j].length) OR
                  ((S[i].length = S[j].length) AND (i>j)) THEN
                     V[i]:=False

   FOR i:=1 TO s, in parallel, DO
      IF S[i].status = Neutral THEN
        finished:=False
        IF V[i] THEN
            S[i].status:=Selected

UNTIL finished
```

Correctness

The line segments are represented by the S-array. Each element in S has a status field that tells us if the corresponding line segment is one of G's original edges or a diagonal that belongs to the greedy triangulation (Selected), if it is a discarded diagonal (Rejected), or if it is a diagonal whose usefulness hasn't been determined yet (Neutral). The way the segments are indexed depends on the input order of the vertices, S[1] being the line segment between vertex 1 and vertex 2, S[2] the line segment between vertex 1 and vertex 3, and so on. If two intersecting diagonals happen to be of the same length, the one with the lowest index is assigned higher priority.

The V- and W-arrays are modified when a diagonal is blocked by a shorter Neutral diagonal or by a Selected diagonal/edge, respectively. During each iteration, the remaining Neutral diagonals are either Selected, Rejected, or preserved as Neutral depending on the contents of V and W. (A Neutral diagonal blocked by a Selected diagonal or an edge has to be discarded.) The algorithm keeps on running until no Neutral diagonals remain.

When the algorithm ends, S[i].status will have been set to either Selected or Rejected (for all valid values of i). The line segments corresponding to Selected array elements are the ones that constitute the greedy triangulation of G.

Lemma 4.1:
The algorithm produces a greedy triangulation of G.

Proof:
First all of G's edges are Selected. Then every possible diagonal is checked against all the others. A diagonal is Selected if and only if it is shorter (or of the same length and has a lower index) than all other diagonals that it intersects and it doesn't intersect any of the original edges in G. This is because its status is changed from Neutral to Selected only if its V-element and W-element still are equal to True at the end of one set of tests. When no Neutral diagonals are left, finished is never changed to False, and the algorithm terminates. $\square$

Intersection test

To implement the intersection test, we can first test if the two segments' bounding boxes intersect. (A bounding box is the smallest rectangle with sides parallel to the x-axis and y-axis that contains the segment.) If the bounding boxes don't intersect, the segments can't intersect and we are finished. If the bounding boxes do intersect, we then proceed to check if segment (p_1, p_2) straddles segment (p_3, p_4) and vice versa. This is easily done by using cross-products; if

$$[(p_3 - p_1) \times (p_2 - p_1)]_z \cdot [(p_4 - p_1) \times (p_2 - p_1)]_z < 0$$

then (p_3, p_4) straddles (p_1, p_2), and if it is > 0 then (p_3, p_4) does not straddle (p_1, p_2). Two line segments that each straddle the line containing the other must intersect. In some boundary cases the product of cross-products above turns out to be equal to zero, which means p_3 or p_4 lie on the line containing segment (p_1, p_2). Since the bounding boxes intersect, the segments intersect. However, two segments sharing exactly one common endpoint are not considered to intersect (they are just connected). In these special cases a few simple checks involving the endpoints have to be carried out.

Analysis

Diagonals that belong to the greedy triangulation are called *greedy edges*. Depending on how the input vertices are related geometrically, the parallel greedy triangulation algorithm will require one or more iterations to complete. During iteration j in the `REPEAT`-loop, a number of greedy edges are produced (their S[i].status fields are set to Selected). The set of all greedy edges located in iteration j is called *generation* j.

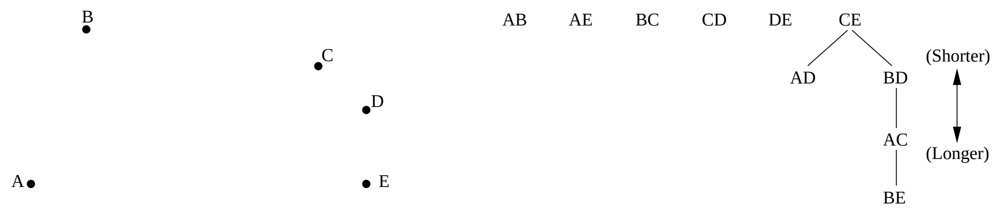


Figure 4.2: A set of points and the length relationship between blocking diagonals.

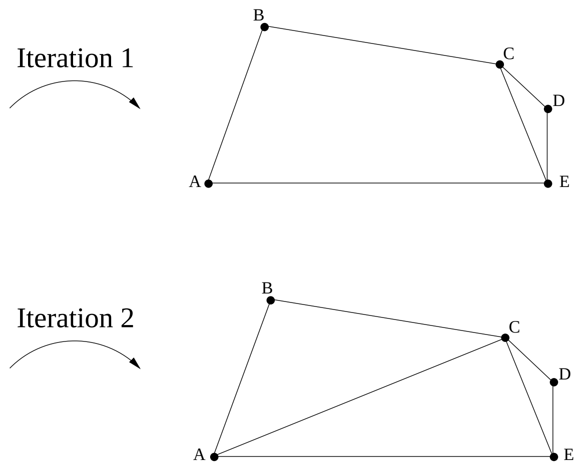


Figure 4.3: During iteration 1, AC is blocked by BD. In iteration 2, BD is no longer around to cause AC any trouble.

The convex hull will always belong to generation 1 since no line segments intersect it, but some of the other greedy edges can be more intricately embedded in the graph. See Figures 4.2 and 4.3 for an example.

The algorithm's running time is determined by the number of times it has to execute the instructions in the `REPEAT`-loop. One iteration takes $O(1)$ time, so the total running time has to be $O(k)$ if k is the number of iterations required to complete the greedy triangulation. Actually, after all greedy edges have been found, the algorithm has to go through one final iteration (in which no more greedy edges are found, of course) to make sure it is finished, but this can be ignored since we are only interested in what happens asymptotically.

Lemma 4.2:
Generation j consists of at least one greedy edge, $1 \leq j \leq k$.

Proof:
Assume there are p Neutral diagonals at the beginning of iteration j. Some of them are blocked by diagonals that were Selected in iteration $j-1$, and subsequently ignored. This leaves q Neutral diagonals. q has to be > 0 or finished would never be set to False (which only occurs in iteration $k+1$). At least one of the q Neutral diagonals has a length shorter or equal to the lengths of the $q-1$ other diagonals. (If two or more diagonals are of the same length, one will have an index lower than the others.) Its V-element is never changed to False, and so it gets Selected at the end. Lemma 4.1 guarantees that only greedy edges are produced, which completes the proof for Lemma 4.2. □

There are at most $3n - 6$ $(= O(n))$ greedy edges in $GT(G)$ (see Corollary 2.4), so Lemma 4.2 implies that the algorithm produces $O(n)$ generations in the worst case and thus requires $O(n)$ iterations to complete. This gives a total running time of $O(n)$. Since the greedy triangulation problem is P-complete, this is not surprising (a total running time that was polylogarithmic would have been, though!).

4.2.1 Average number of generations needed to complete

As one might suspect, the $O(n)$ bound on the number of generations is overkill most of the time. One question that naturally arises is: How many generations are needed to complete the parallel greedy triangulation in the average case?

To get an idea of what the average case looks like, the following experiment was repeatedly carried out on the computer:

- For a certain n, randomly place that many points in the plane. (To get statistically independent and uniformly distributed numbers between 0 and 1, a prime modulus multiplicative linear congruential generator like the one described in Park & Miller [25] was used.)
- Triangulate the set of points using the parallel greedy triangulation algorithm described above. (Parallel processors weren't actually employed, so they had to be simulated.)
- Count the number of generations needed to complete.

As an example, Figure 4.4 illustrates the results for $n = 600$.

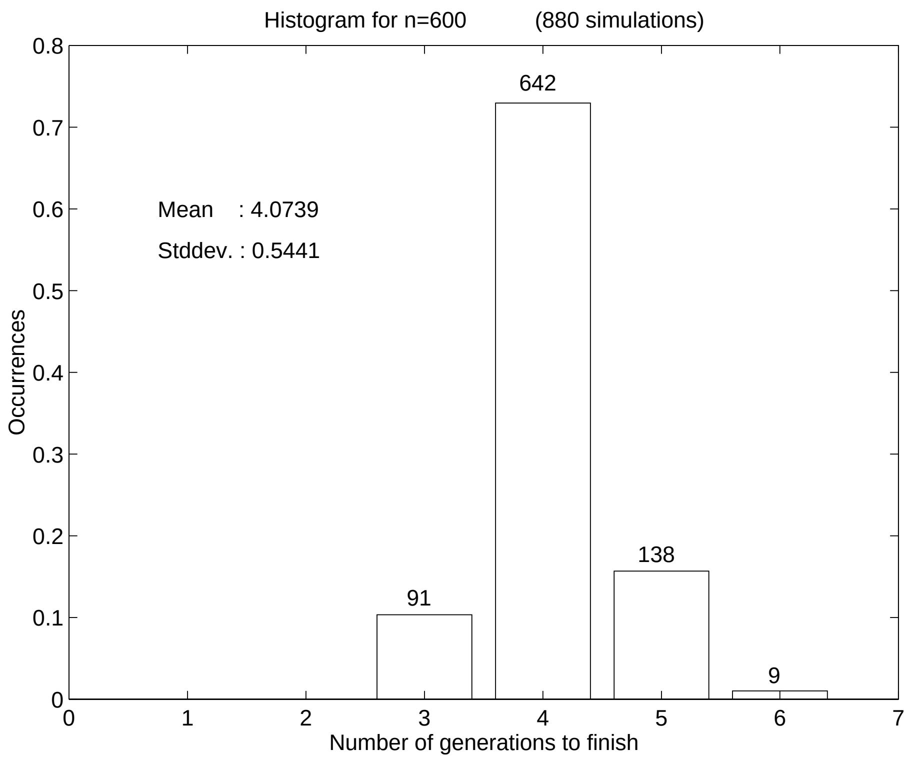


Figure 4.4: The parallel greedy triangulation algorithm applied to 880 sets of 600 randomly selected points in the plane produced a total of 3 generations in 91 of the cases, 4 generations in 642 of the cases, and so on.

The experiment was repeated for different values of n (up to 1000) 3.43 million times, as listed in Table 4.1. By plotting the average number of generations versus n, the graph in Figure 4.5 was obtained. The curve is close to $y = 0.3192 + 0.6100 \ln(n)$, as can be seen in Figure 4.6. (This is a least-squares approximation.) The results seem to indicate that only $O(\log(n))$ generations are required to finish the greedy triangulation in the average case.

For a fixed n, the number of generations needed to finish is a bounded, positive integer. Let $p_1, p_2, \ldots, p_r$ represent the different probabilities of it being 1, 2, ..., or r. If m independent tries are carried out and X_i denotes the number of times the result is i, the stochastic variable $(X_1, \ldots, X_r)$ is multinomially distributed. (See Blom [2] or any other elementary book on statistics for a detailed explanation of why.) Thus,

$$p_{X_1,\ldots,X_r}(k_1, \ldots, k_r) = \frac{m!}{k_1! \ldots k_r!} p_1^{k_1} \ldots p_r^{k_r}$$

where $\sum p_i = 1$, and all k_i assume non-negative integer values with $\sum k_i = m$.

n	Nr. of exp.	Average nr. of gen.	n	Nr. of exp.	Average nr. of gen.	n	Nr. of exp.	Average nr. of gen.
			51	5000	2.7946	110	1000	3.2300
			52	5000	2.7824	120	1000	3.2910
			53	5000	2.7972	130	1000	3.3410
			54	5000	2.8050	140	1000	3.3750
5	300000	1.0415	55	5000	2.8178	150	1000	3.3880
6	250000	1.1136	56	5000	2.8386	160	1000	3.4250
7	200000	1.2025	57	5000	2.8768	170	1000	3.4360
8	200000	1.2992	58	5000	2.8798	180	1000	3.4790
9	200000	1.3979	59	5000	2.8696	190	1000	3.5150
10	200000	1.4942	60	5000	2.8926	200	4000	3.5333
11	130000	1.5877	61	3000	2.9087	225	750	3.6066
12	100000	1.6696	62	3000	2.8961	250	1000	3.6400
13	100000	1.7540	63	3000	2.9184	275	500	3.6680
14	100000	1.8260	64	3000	2.9336	300	2500	3.7272
15	100000	1.8870	65	3000	2.9463	350	100	3.8400
16	100000	1.9501	66	3000	2.9514	400	1250	3.8528
17	100000	2.0019	67	3000	2.9555	450	50	3.9000
18	100000	2.0548	68	3000	2.9768	500	845	3.9929
19	100000	2.0991	69	3000	2.9727	550	50	4.0400
20	160000	2.1419	70	3000	2.9891	600	880	4.0739
21	50000	2.1782	71	1000	3.0120	700	154	4.0909
22	50000	2.2182	72	1000	3.0000	750	17	4.1765
23	50000	2.2515	73	1000	3.0010	800	45	4.1778
24	50000	2.2759	74	1000	3.0480	850	132	4.1591
25	50000	2.3082	75	1000	3.0220	900	59	4.2034
26	50000	2.3312	76	1000	2.9940	1000	20	4.3000
27	50000	2.3581	77	1000	3.0160			
28	50000	2.3860	78	1000	3.0360			
29	50000	2.4095	79	1000	3.0580			
30	50000	2.4300	80	1000	3.0680			
31	20000	2.4441	81	1000	3.0430			
32	20000	2.4766	82	1000	3.0670			
33	20000	2.4937	83	1000	3.0910			
34	20000	2.5071	84	1000	3.0990			
35	20000	2.5229	85	1000	3.1380			
36	20000	2.5400	86	1000	3.0940			
37	20000	2.5698	87	1000	3.1330			
38	20000	2.5878	88	1000	3.1430			
39	20000	2.5962	89	1000	3.1590			
40	20000	2.6134	90	1000	3.1560			
41	10000	2.6389	91	1000	3.1410			
42	10000	2.6457	92	1000	3.1620			
43	10000	2.6620	93	1000	3.1830			
44	10000	2.6847	94	1000	3.1220			
45	10000	2.7031	95	1000	3.1990			
46	10000	2.7087	96	1000	3.1690			
47	10000	2.7307	97	1000	3.1700			
48	10000	2.7423	98	1000	3.1800			
49	10000	2.7587	99	1000	3.1660			
50	30000	2.7638	100	40000	3.1957			

Table 4.1: The exact number of times the experiment in Section 4.2.1 was carried out for various values of n. The running time increases dramatically as n grows, which is why there are fewer results for large n's.

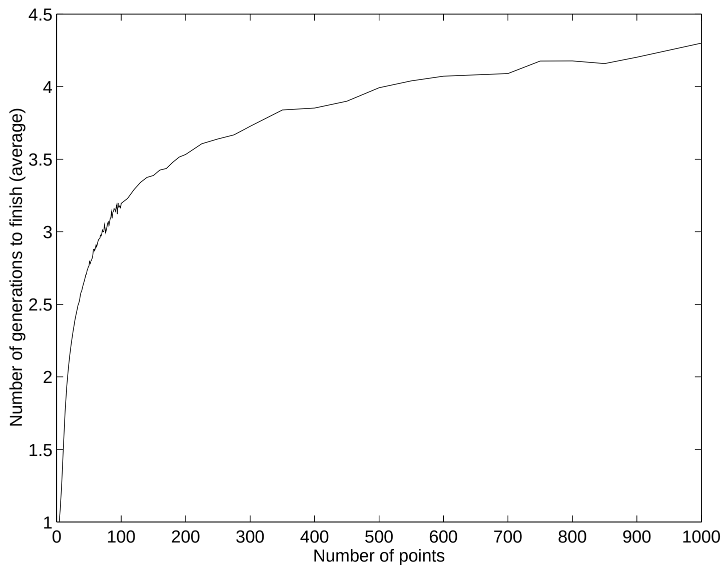


Figure 4.5: The average number of generations required for the parallel greedy triangulation algorithm to complete.

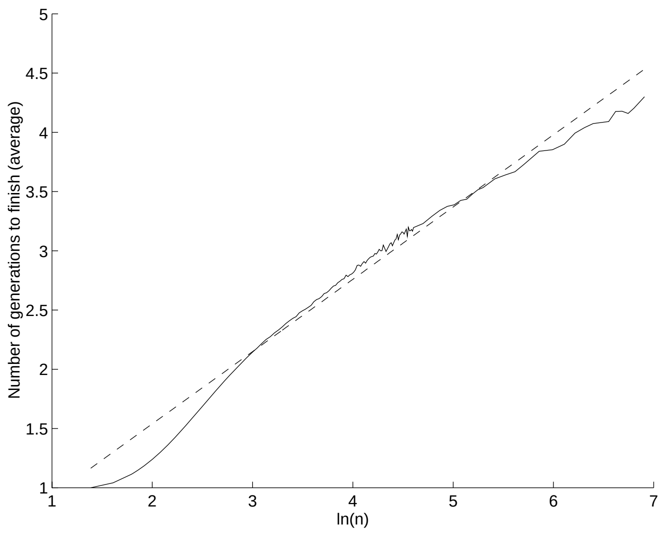


Figure 4.6: The average number of generations required to complete plotted as a function of $\ln(n)$. The dashed line is a least-squares approximation. Its equation is $y = 0.3192 + 0.6100\ln(n)$.

4.2.2 Average number of greedy edges in different generations

The number of greedy edges in a specified generation number depends not only on n, but also on the vertices' coordinates since the total number of edges equals $3n - m - 3$, where m tells how many points are on the convex hull (see Theorem 2.3). And just like before, complicated chains of geometric relationships between blocking diagonals add to the mayhem.

The next series of simulations were performed in order to study how many greedy edges belong to different generations in the average case. A slight modification to the program above resulted in histograms like the ones displayed in Figure 4.7. The experiment was conducted a total of 767877 times for various values of n between 4 and 1000 (see Table 4.2). Finally, the graphs in Figure 4.8 were obtained by combining all the collected data. The number of greedy edges in each generation appears to be linearly related to n.

For a given n and a fixed generation number, the number of greedy edges is a bounded, positive integer. If we let X_i represent the number of times the result is i, then the stochastic variable $(X_q, \ldots, X_r)$ is multinomially distributed, where q is the minimum and r the maximum possible outcome.

n	#exp.
4	100000
5	100000
6	100000
7	100000
8	100000
9	100000
10	100000
20	30000
50	20000
100	10000
200	3000
300	2000
400	1200
500	620
600	800
700	110
850	68
900	59
1000	20

Table 4.2: The exact number of times that the diagonal-counting experiment in Section 4.2.2 was carried out for different n's.

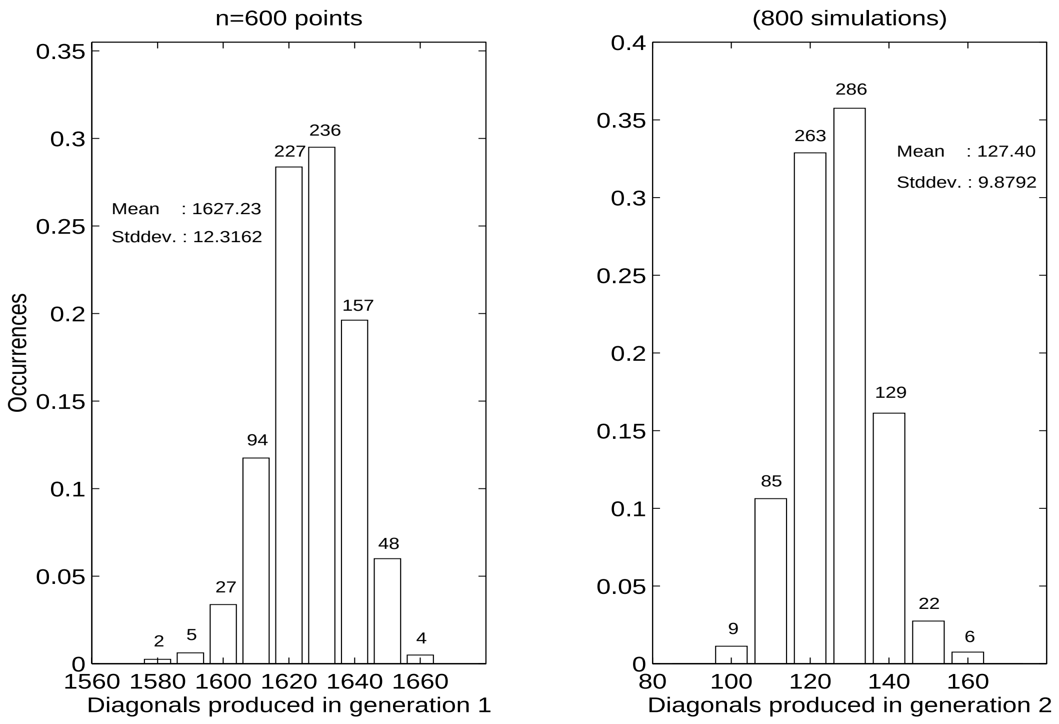


Figure 4.7: Two histograms showing the number of times that different quantities of greedy edges were found in generations 1 and 2 by the parallel greedy triangulation algorithm. 800 different sets of 600 random points were tested.

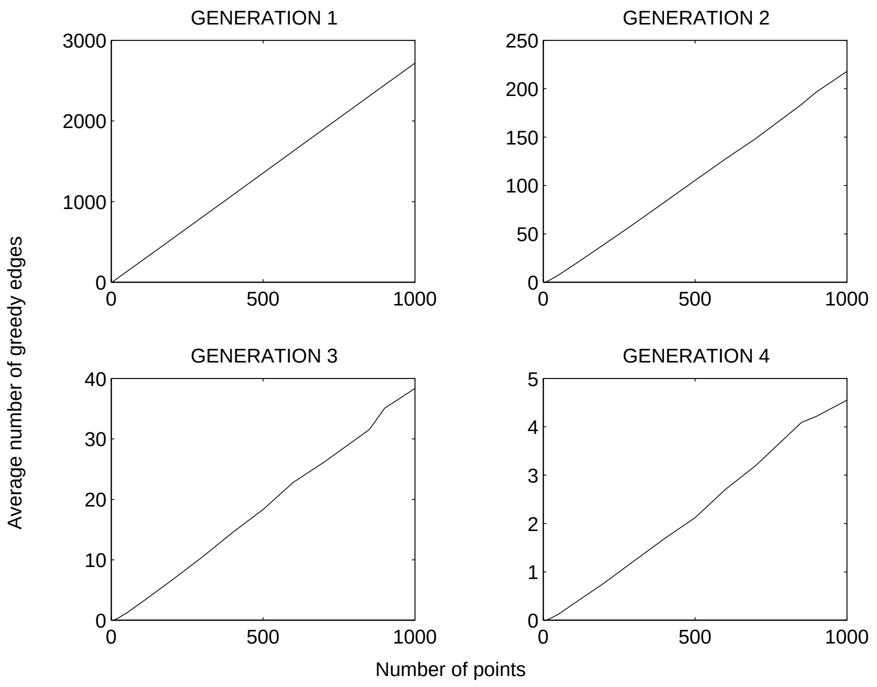


Figure 4.8: The average number of greedy edges in generations 1 through 4.

Chapter 5

Parallel dynamic programming for simple polygons

A cubic time algorithm for finding a minimum-weight triangulation of a simple polygon by using dynamic programming was presented in Chapter 3.2. If more than one processor is available, the work involved can be distributed so that the total running time is reduced. This chapter contains some CREW PRAM dynamic programming algorithms that accomplish this. They are focused on solving the MWT problem for simple polygons, but can be modified to solve other optimal triangulation problems.

A fundamental (and time-consuming) part of all dynamic programming algorithms is minimizing some function. With just one processor, finding the minimum of n arbitrary values takes $O(n)$ time since all values have to be checked. Finding the minimum value with a CREW algorithm takes $\Omega(\log(n))$ time, even with an unlimited number of processors (see [3]). But $O(n/\log(n))$ processors suffice. Divide the n different values into $\lfloor n/\log(n)\rfloor$ groups of $\lfloor \log(n)\rfloor$ values each and at most one group made up of the remaining values. Assign one processor to each group, and find the minimum of each one in parallel. Scanning the $O(\log(n))$ values in a group takes $O(\log(n))$ time. Next, let the $\lceil n/\log(n)\rceil$ locally minimal values be leaves of a complete binary tree. In parallel, compare all sibling leaves using $\lceil\lceil(n/\log(n))\rceil/2\rceil$ processors, and assign each minimum value to the corresponding parent node. Remove the leaves and repeat the process until only one value remains. In each stage, the number of values in the tree is halved, so $O(\log(n/\log(n))) = O(\log(n))$ stages are needed for the minimum value to reach the root. Thus, the total time required to find the minimum of n arbitrary values with an $O(n/\log(n))$ processor CREW algorithm is $O(\log(n)) + O(\log(n)) = O(\log(n))$.

The minimum value can be found in $O(\log(\log(n)) - \log(\log(2p/n)))$ time, where p is the number of processors, if the common-CRCW model is used instead (see Karp and Ramachandran [11]). If $p = n^{1+\epsilon}$ for any constant $\epsilon > 0$, the time needed is

$$\begin{aligned} O(\log(\log(n)) - \log(\log(2n^{1+\epsilon}/n))) &= O(\log(\log(n)) - \log(\log(2n^{\epsilon}))) \\ &= O(\log(\log(n)) - \log(1 + \epsilon\log(n))) \\ &= O(\log(\log(n)) - \log(\epsilon\log(n))) \\ &= O(\log(\log(n)) - \log(\epsilon) - \log(\log(n))) \\ &= O(-\log(\epsilon)) \\ &= O(1) \end{aligned}$$

In other words, the minimum can be found in $O(1)$ time by employing at least $n^{1+\epsilon}$ processors, where ϵ is some constant > 0. The running times for the CREW algorithms below can therefore be improved by a factor $\log(n)$ if concurrent writes are allowed and a polynomial number of extra processors are added. For simplicity, only the CREW versions are described.

5.1 $O(n^2/\log(n))$ processors, $O(n\log(n))$ time

The running time of the algorithm in Chapter 3.2 can easily be reduced by making a few modifications. As before, let n denote the number of vertices in the input polygon. If we have $O(n^2/\log(n))$ processors, the algorithm's i-loop can be carried out in $O(\log(n))$ time for a fixed value of l. The processors simultaneously perform the $O(n^2)$ (independent) crossing tests for all valid i's and k's in $O(\log(n))$ time by letting each processor handle $\log(n)$ crossing tests. Next, they combine the $O(n)$ resulting boolean values[1] for each i, once again needing a total of $O(n/\log(n))O(n) = O(n^2/\log(n))$ processors and taking $O(\log(n))$ time. The processors then calculate all the $O(n)$ different possible values for each $t[i,j]$, $0 \le i \le n-l$ and $j = i+l-1$, using the already determined values of $t[i,k]$ and $t[k,j]$, where $i < k < j$. Finally, the values that minimize each $t[i,j]$ are selected and stored, which also requires $O(\log(n))$ time and $O(n/\log(n))O(n) = O(n^2/\log(n))$ processors since there are $O(n)$ values to choose from for each i. Letting l run from 3 to n takes a linear amount of time. All in all, the algorithm's running time is improved from $O(n^3)$ to $O(n\log(n))$.

5.2 $O(n^9)$ processors, $O(\log(n)^2)$ time

The depth of the recursion in the previous method can be linear, which means that it has to be modified if a sublinear time algorithm is wanted. In general, logarithmic recursion depth can be obtained if, in every recursion step, all given sets of data are split into tinier ones that are smaller by a constant fraction. This is because if $R(m)$ is the maximum size of all sets in step m (i.e. the number of elements they are allowed to contain in step m), and
$R(m) \le R(m-1)\cdot c$, where c is a constant $(0 < c < 1)$, then

$$R(m) \le R(m-2)\cdot c^2 \le \ldots \le R(0)\cdot c^m$$

Let $R(m) = 1$ and $R(0) = n$, and we obtain

$$1 \le n \cdot c^m$$

$$\log(1) \le \log(n) + m\log(c)$$

$$m\log(1/c) \le \log(n) - 0$$

$$m \le \frac{\log(n)}{\log(1/c)}$$

So, the bottom of the recursion is reached in $O(\log(n))$ steps.

A dynamic programming method that (like usual) works backwards will be described in this section. It starts out with optimal solutions for small subpolygons and works its way up to the top by constructing optimal solutions for bigger and bigger subpolygons in each recursion step. (The size of a polygon/subpolygon is the number of vertices it contains.) It is essential that the new polygons acquired in each step are larger than the old ones by at least a constant factor $1/c$ $(0 < c < 1)$; if not, the logarithmic time bound will never be attained.

A more formal definition of what we mean by "subpolygon" is needed. A *q-subpolygon* of a simple polygon P, where $q \in \mathbf{Z}^+$, is a simple polygon whose perimeter consists of exactly q diagonals located inside P (called *d-edges*) as well as up to q maximal, disjoint pieces of P's perimeter called *chains*. We define a chain of a polygon P to be a section of its perimeter that includes at least two consecutive vertices. Chains in a q-subpolygon are always isolated from each other by d-edges, in other words no two chains can share a common point. A *corner vertex* of a subpolygon is a vertex that connects a chain and a d-edge or that connects two d-edges.

Now, examine pairs of nonintersecting diagonals instead of pairs of vertices (as in the original method) in a simple polygon. One such pair divides the polygon into three subpolygons. Each subpolygon contains at most two chains in addition to the one or two d-edges. Figure 5.1 shows an example.

From now on when referring to "subpolygons", we will always mean "subpolygons of the original polygon with n vertices".

[1]To compute the OR of n boolean values, a method analogous to the one for finding the minimum value can be employed, which results in an $O(n/\log(n))$ processor and $O(\log(n))$ time requirement for the CREW model.

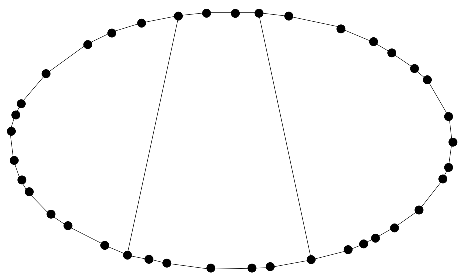


Figure 5.1: A pair of nonintersecting diagonals cuts the polygon into three subpolygons (two 1-subpolygons and one 2-subpolygon).

Before we can continue, we need to develop a special version of the renowned planar separator theorem by Lipton & Tarjan [23].

Lemma 5.1:
Let $T = (V, E)$ be a connected, undirected tree with degree$(v) \leq 3$ for all $v \in V$ and let $n = |V|$. There exists an edge $e \in E$ whose removal disconnects T into two disjoint subtrees, each with $\leq \frac{2n+1}{3}$ vertices.

Proof:
(From Kozen [12])
Create a directed tree T' by determining for each $e \in E$ the sizes of the two components obtained by deleting e (e.g. by a recursive depth-first search in linear time) and orienting e in the direction of the smaller component. If the two components are equal in size, let e point in either direction.

To see that T' really is a directed tree, we have to prove that no vertex has indegree > 1. Assume that the edges (x, z) and (y, z) both are oriented toward z (so that z has indegree > 1), and X, Y, and Z are subtrees of T' as shown in Figure 5.2. Then $|X| \geq |Y| + |Z|$ and $|Y| \geq |X| + |Z|$. But this results in $|Z| = 0$, which is a contradiction.

Call the unique root of T' r. Let A, B, and C be the maximal proper subtrees of r, where A is the one of maximum size. Because of the orientation of the edge between r and A,

$$|A| \leq \frac{n}{2} \leq \frac{2n+1}{3}. \tag{5.1}$$

Next, $|B| \leq |A|$ and $|C| \leq |A|$, so

$$|B| + |C| \leq 2|A|$$

and

$$n - 1 = |A| + |B| + |C| \geq \frac{|B| + |C|}{2} + |B| + |C| = 3\frac{|B| + |C|}{2}.$$

This means

$$|B| + |C| + 1 \leq \frac{2n+1}{3}. \tag{5.2}$$

(5.1) and (5.2) prove that the edge connecting r to A will do. □

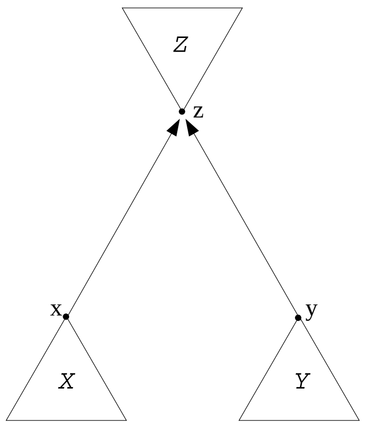


Figure 5.2: The edges (x, z) and (y, z) are directed toward z. X, Y, and Z are subtrees of T'.

Theorem 5.2:
Given a simple polygon P with n vertices and a triangulation T of P, there exists a diagonal in T that divides P into two parts with $\leq \frac{2n+7}{3}$ vertices each.

Proof:
Let TR be an undirected tree whose nodes correspond to triangles inside T and whose edges correspond to diagonals. See Figure 5.3.

Since P is a simple polygon, there are no cycles in TR, and TR is indeed a tree. (If there was a cycle in TR, the triangles in T would have to contain some vertex or vertices located in the interior of P, which contradicts the fact that P is a simple polygon. This is illustrated in Figure 5.4.)

Each triangle is adjacent to at most three other triangles, which means that each node has degree ≤ 3.

According to Lemma 5.1, TR can be split into two binary subtrees with $\leq \frac{2n+1}{3}$ nodes each by removing an edge e. Going back to the original representation, the diagonal that corresponds to e partitions P into two subpolygons. Each subpolygon must have less than $\frac{2n+1}{3}$ triangles, which means $\leq \frac{2n+1}{3} + 2 = \frac{2n+7}{3}$ vertices (a triangulated polygon with n vertices always contains $n-2$ triangles; see Theorem 2.5). $\square$

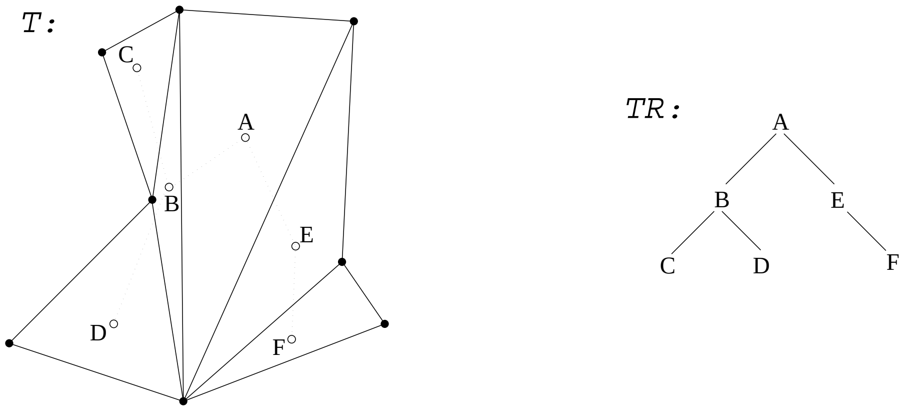


Figure 5.3: The relationship between T and TR.

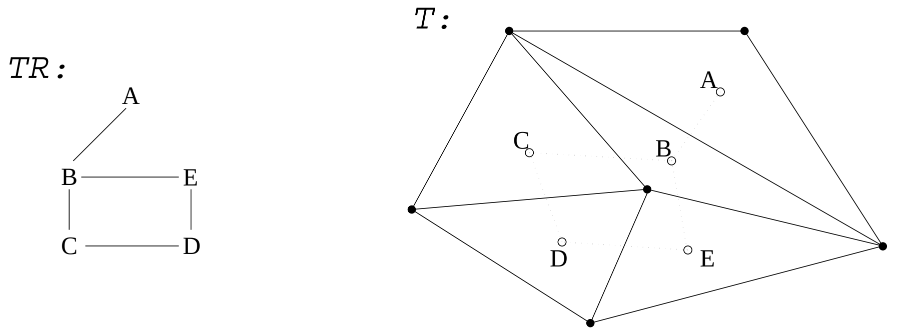


Figure 5.4: A cycle in TR would correspond to triangles with vertices inside P.

Corollary 5.3:
Given a simple polygon P with n vertices and a MWT of P, there exists a diagonal called a separator in the MWT that divides P in two parts with $\leq \frac{2n+7}{3}$ vertices each. □

Corollary 5.3 implies that optimal solutions for 1- and 2-subpolygons of maximum size m can be found in one recursion step if optimal solutions for all 1- and 2-subpolygons of size $\leq \frac{2m+7}{3}$ are available. The way to do this is by testing all diagonals that divide the specified subpolygon into two subpolygons of size $\leq \frac{2m+7}{3}$, computing the resulting triangulation weights using the stored MWTs of the smaller subpolygons, and selecting a triangulation with the smallest weight. Such a triangulation will be a MWT since Corollary 5.3 guarantees that at least one diagonal in the MWT is tested. (From Chapter 3.2 we recall that a diagonal belonging to the MWT of a simple polygon always joins two optimally triangulated subpolygons together.)

Before the recursion starts, the simple sequential method from Chapter 3.2 is used to optimally triangulate all 1- and 2-subpolygons of size ≤ 9 with $O(n^2)$ processors in $O(1)$ time. (There are $O(n^2 + d) = O(n^2)$ such subpolygons, where d is some constant.) Subsequently, if $R(k)$ is the maximum size of all 1- and 2-subpolygons taken care of in step k, $R(k+1) = \frac{3R(k)-7}{2}$.

There is one small problem with this approach. A discovered separator might partition a given 2-subpolygon into one 1-subpolygon and one 3-subpolygon (see Figure 5.5 (c)), and we don't have the MWTs for 3-subpolygons.

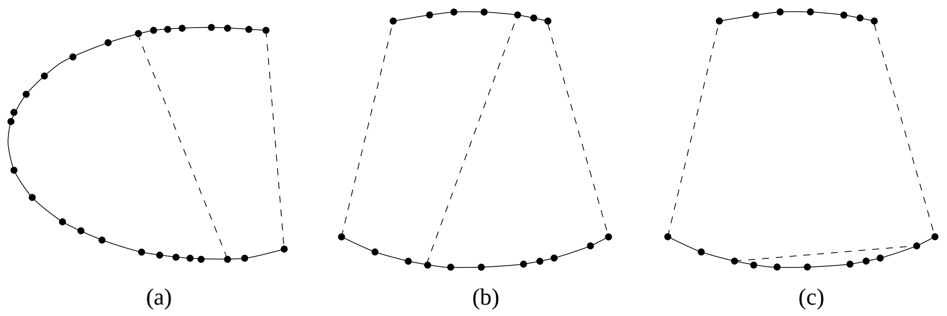


Figure 5.5: (a) A separator always divides a 1-subpolygon into one smaller 1-subpolygon and one 2-subpolygon. (b) A 2-subpolygon can be cut into two smaller 2-subpolygons by a separator. (c) A separator that splits a 2-subpolygon into one 1-subpolygon and one 3-subpolygon.

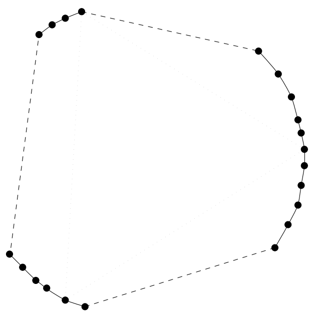


Figure 5.6: A 3-subpolygon divided into three 2-subpolygons and a triangle.

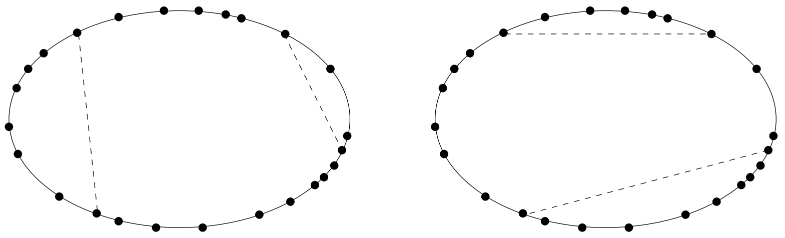


Figure 5.7: Two different subpolygons are defined by the same quadruple of vertices.

However, a 3-subpolygon can always be divided into at most three 2-subpolygons by some triangle like in Figure 5.6. If the 3-subpolygon is of size $\leq \frac{2m+7}{3}$, the 2-subpolygons have less than $\frac{2m+7}{3}$ vertices, which means that their optimal solutions are available.

By evaluating and comparing all valid configurations of 2-subpolygons and matching triangles, we cover the 3-subpolygon case as well since at least one of the triangles we test has to belong to a MWT of the 3-subpolygon.

In each recursion step, there are $O(n^4)$ 1- and 2-subpolygons to be taken care of. (There are $\binom{n}{4}$ alternative ways to select a quadruple of vertices. For every quadruple, there are at most two ways to draw two d-edges and get a valid subpolygon. See Figure 5.7.)

Every subpolygon requires $O(n^5)$ sets of calculations to be performed because first the separator can be chosen in $O(n^2)$ ways, and then $O(n^3)$ different triangles have to be tested. The optimal solutions for the induced 1- and 2-subpolygons are looked up and combined with the dividing triangle, whereupon the resulting $O(n^5)$ triangulations are evaluated and compared. The best solution is selected and recorded for future use.

By assigning one processor to each set of calculations and then using the same processors to find the best triangulation for each subpolygon, a total of $O(n^4) \cdot O(n^5) = O(n^9)$ processors are needed in each recursion step. The processors are recycled between successive steps in the recursion, so $O(n^9)$ parallel processors are enough for the whole algorithm. Finding the minimum value in each step takes $O(\log(n^5)) = O(5\log(n)) = O(\log(n))$ time, and there are $O(\log(n))$ steps in the recursion, so the algorithm's running time is $O(\log(n)^2)$.

5.3 $O(n^6/\log(n))$ processors, $O(\log(n)^2)$ time

Rytter invented a parallel pebble game on binary trees to find a simplified structure for solving some related dynamic programming problems in [28]. He proved that the MWT problem for a simple polygon with n vertices can be solved in $O(\log(n)^2)$ time using a CREW PRAM model with $O(n^6/\log(n))$ processors.

The method can be described geometrically, making it easier to understand. It is very similar to the method in Section 5.2, but reduces the number of processors required by a factor of $n^3\log(n)$ without slowing down the asymptotic running time!

As before, the main idea is to use the stored optimal triangulations for 1- and 2-subpolygons of size $\leq \frac{2m+7}{3}$ to get optimal triangulations for 1- and 2-subpolygons of maximum size m in one recursion step. (All 1- and 2-subpolygons of size ≤ 9 are triangulated optimally in $O(1)$ time with $O(n^2)$ processors with the sequential method from Chapter 3.2 before the first recursion step.) Each recursion step is split into four phases.

Phase 1:

Find optimal triangulations for the $O(n^2)$ different 1-subpolygons with m or fewer vertices by testing all $O(n^2)$ ways of drawing one diagonal that cuts a given 1-subpolygon into 1- and 2-subpolygons of size $\leq \frac{2m+7}{3}$. Optimal solutions for these smaller subpolygons are available from previous recursion steps. At least one separator will always be examined because of Corollary 5.3.

Phase 2:

Triangulate every 2-subpolygon with a maximum of m vertices and exactly one chain optimally by testing triangles that divide it into smaller parts that have been taken care of earlier. By testing all triangles that partition the 2-subpolygon as in Figure 5.8, every possible solution is checked. The dividing triangle may or may not be degenerate; for cases in which area C in Figure 5.8 is nonexistent, the triangle is in fact just a diagonal.

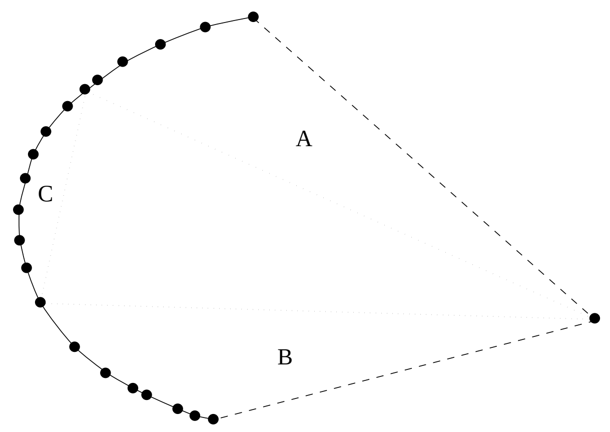


Figure 5.8: A 2-subpolygon of size m with one chain is cut into two 2-subpolygons (A and B) of size $\leq \frac{2m+7}{3}$, and one 1-subpolygon (C) of size $\leq m$ by a dividing triangle. Optimal solutions for A and B are available from earlier recursion steps; the optimal solution for C is fetched from phase 1.

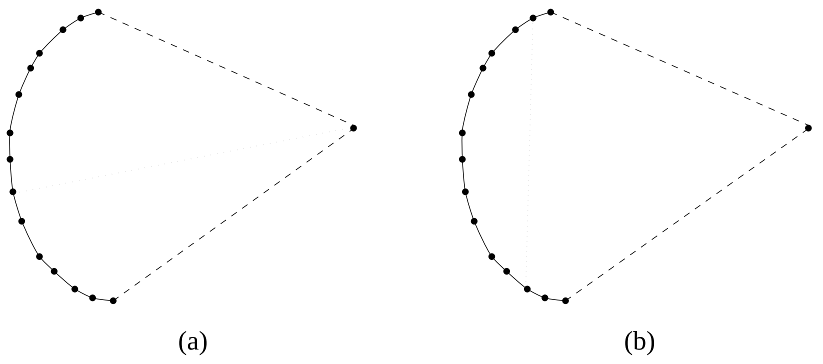


Figure 5.9: (a) Case 1 (b) Case 2

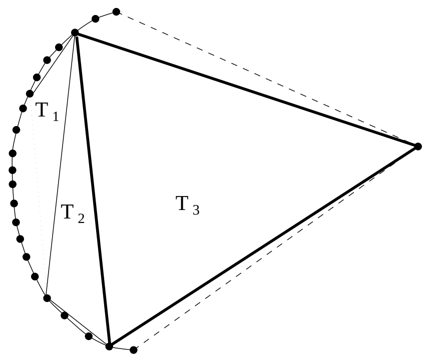


Figure 5.10: P is cut into two 2-subpolygons of size $\leq \frac{2m+7}{3}$ with one chain and one 1-subpolygon by T_3. The dotted line is the original separator.

To prove that all possibly optimal solutions are covered, we will once more use Corollary 5.3. It tells us that any MWT of a polygon P contains at least one separator. If a given separator of the 2-subpolygon connects the chain and the opposite corner vertex, we get two 2-subpolygons, both of size $\leq \frac{2m+7}{3}$ (case 1 in Figure 5.9). Otherwise, it connects two vertices on the chain, giving us one 1-subpolygon and one 3-subpolygon (case 2 in Figure 5.9).

In case 1, optimal solutions are available right away since the optimal subsolutions are known. The optimal solutions are available in case 2, too, but to see this requires a little more thought. The separator belongs to two triangles in the MWT of P. Let T_1 be the one whose third vertex lies outside the subchain defined by the separator. If the third vertex happens to be the corner vertex opposite of the chain, the other two sides of T_1 split P into one or two 2-subpolygons with one chain and one 1-subpolygon. If it isn't, it lies on the same chain as the separator, and we consider the side that cuts off T_1 together with two 1-subpolygons from P instead. This diagonal has to be a side in some other triangle T_2 in the MWT. By repeating this process, we eventually get to a triangle T_k whose third vertex is the opposite corner vertex. See Figure 5.10 for an example. By observing that T_k splits P into one or two 2-subpolygons of size $\leq \frac{2m+7}{3}$ with one chain and one 1-subpolygon of unrestricted size, we realize that any optimal triangulation must contain a dividing triangle of the type illustrated in Figure 5.8, and that an optimal solution can be found if we test all valid dividing triangles and compare the resulting triangulations.

There are $O(n^3)$ 2-subpolygons to be handled in this phase. For each one, there are $O(n^2)$ possible ways to define the dividing triangle. The induced subpolygons are triangulated optimally and compared as described above. Finally, the best triangulation (for each of the checked 2-subpolygons) is stored.

Phase 3:

Compute optimal solutions for all $2^\star$-subpolygons of size $\leq m$. A $2^\star$*-subpolygon of size* m is a special kind of 2-subpolygon that has one or two chains and a diagonal from a corner vertex to a vertex on the opposite chain in its MWT such that the two smaller 2-subpolygons satisfy the following conditions:

- A 2-subpolygon with only one chain can be of any size (up to m, that is).
- A 2-subpolygon that comprises two chains must have $\leq \frac{2m+7}{3}$ vertices.

Optimal solutions for all 2-subpolygons of size $\leq m$ with one chain have been found in phase 2 and optimal solutions for the other (smaller) 2-subpolygons in previous recursion steps.

For each one of the $O(n^4)$ 2-subpolygons, $O(n) + O(n) + O(n) + O(n) = O(n)$ diagonals are tested. The triangulations are put together, evaluated, and the best solution for each 2-subpolygon so far is recorded. 2-subpolygons that are $2^\star$-subpolygons will be optimally triangulated at the end of this phase, but 2-subpolygons like the one in Figure 5.12 will not.

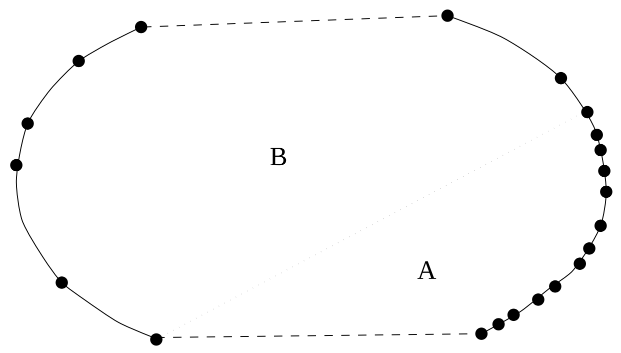


Figure 5.11: A $2^\star$-subpolygon is always divided into two 2-subpolygons A and B by some diagonal in its MWT. A has size $\leq m$ and B has size $\leq \frac{2m+7}{3}$.

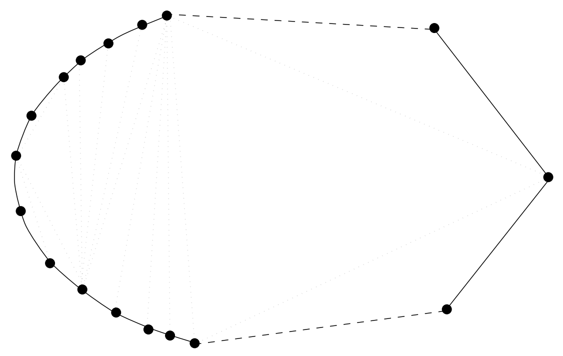


Figure 5.12: If the optimal triangulation for the given 2-subpolygon looks like this, it is not a $2^\star$-subpolygon. Consequently, its optimal triangulation will not have been discovered by the end of phase 3.

Phase 4:

In the last phase, optimal triangulations for the remaining 2-subpolygons of size less than or equal to m are found.

There are two ways that separators in a 2-subpolygon can be oriented. A given separator either splits the 2-subpolygon into two smaller 2-subpolygons (case 1 in Figure 5.13), or into one 1-subpolygon and one 3-subpolygon (case 2 in Figure 5.13). The situation is almost identical to phase 2. By using the same technique, we can prove that there always exists a dividing triangle T_k that partitions a 2-subpolygon of size $\leq m$ into one or two 2-subpolygons of size $\leq \frac{2m+7}{3}$ with one or two chains and one 2-subpolygon of size $\leq m$ with exactly one chain. See Figure 5.14.

One of the 2-subpolygons with one or two chains can be combined with the 2-subpolygon of size $\leq m$ to obtain a $2^{\star}$-subpolygon of size $\leq m$.

So, phase 4 of the algorithm is: For each of the $O(n^4)$ 2-subpolygons with two chains, independently test the $O(n^2)$ possible diagonals that connect the two chains and might partition the 2-subpolygon into one 2-subpolygon of size $\leq \frac{2m+7}{3}$ and one $2^{\star}$-subpolygon of size $\leq m$. (These have already been optimally triangulated and can be combined.) A MWT for each 2-subpolygon of size $\leq m$ is obtained by evaluating the different resulting triangulations and selecting one with the lowest weight.

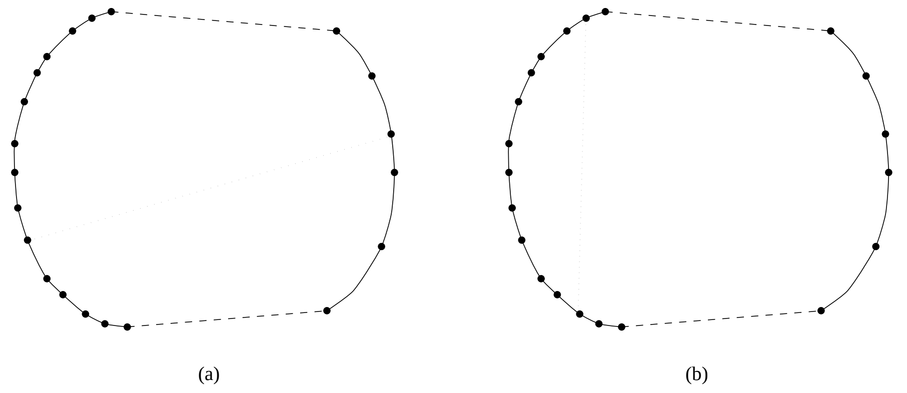


Figure 5.13: (a) Case 1 (b) Case 2

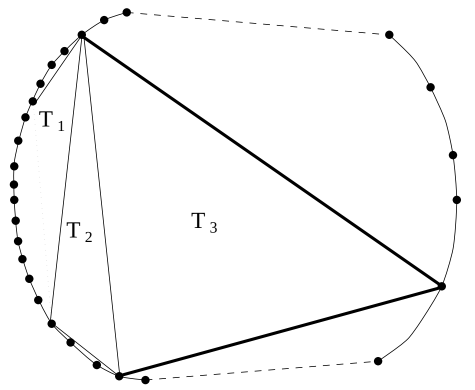


Figure 5.14: P is partitioned into two 2-subpolygons of size $\leq \frac{2m+7}{3}$ with two chains and one 2-subpolygon with one chain. The dotted line is the original separator.

Analysis

The method can be implemented on a CREW PRAM machine to run in $O(\log(n)^2)$ time. The recursion consists of $O(\log(n))$ steps, and each recursion step can be completed in $O(\log(n))$ time if sufficiently many parallel processors are employed. Phase 1 requires $O(n^2) \cdot O(n^2) = O(n^4)$ processors, phase 2 $O(n^3) \cdot O(n^2) = O(n^5)$, phase 3 $O(n^4) \cdot O(n) = O(n^5)$, and phase 4 $O(n^4) \cdot O(n^2) = O(n^6)$ to test all possibilities in $O(1)$ time and then find the best one in $\log(n)$ time. The phases occur one at a time, so the same processors can be used over and over. Thus, a total of $O(n^6)$ processors is needed. This number can be reduced by forcing each processor to simulate $\log(n)$ others. Only $O(n^6/\log(n))$ processors would then be needed for phase 4 (and hence, for the whole algorithm). Each recursion step could still complete in $O(\log(n))$ time ($O(\log(n))$ time to test all possibilities plus $O(\log(n))$ time to find the minimum value).

As a final note, phase 3 might seem redundant at a first glance since phase 4 can be modified to directly handle all 2-subpolygons with two chains. However, this would lead to an $O(n^7/\log(n))$ processor requirement (in the last phase, $O(n^3)$ different possibilities would have to be tested for every 2-subpolygon instead of just $O(n^2)$).

5.4 $O(n^{2.75}/\log(n))$ processors, $O(n^{0.75}\log(n))$ time

All the sublinear time dynamic programming methods described so far require a large number of processors. If we just want to break the $O(n)$ time bound, there are ways to do this without having to use so many processors. The method in this section finds a minimum-weight triangulation for a simple polygon with n vertices recursively in $O(n^{0.75}\log(n))$ time with $O(n^{2.75}/\log(n))$ parallel processors.

First, in parallel, find and store optimal triangulations for all 1- and 2-subpolygons containing up to n^α vertices. α is a parameter ($0 < \alpha < 1$) that will be specified later on. Using the ordinary sequential dynamic programming method from Chapter 3.2, this takes $O((n^\alpha)^3) = O(n^{3\alpha})$ time. According to Figure 5.15, there are $O(n^{2+2\alpha})$ different subpolygons to be considered, so $O(n^{2+2\alpha})$ processors are needed.

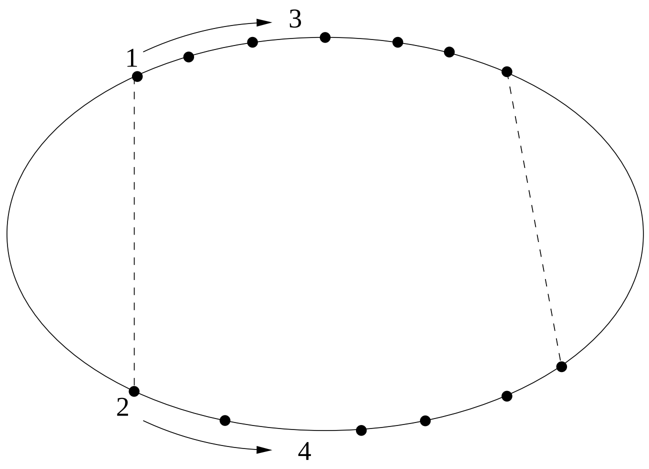
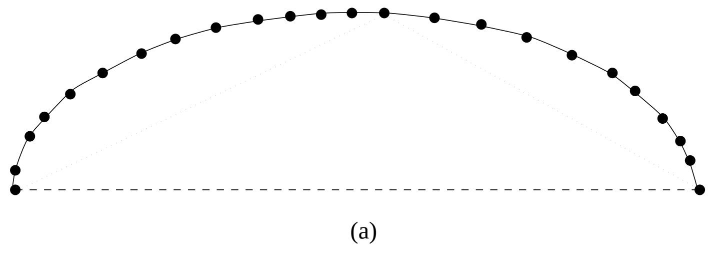


Figure 5.15: One vertex can be selected in $O(n)$ ways. For the second vertex, there are also $O(n)$ possible choices. For every given pair of vertices, the third and fourth vertices are chosen independently from a set of $O(n^\alpha)$. All in all, there are $O(n) \cdot O(n) \cdot O(n^\alpha) \cdot O(n^\alpha) = O(n^{2+2\alpha})$ ways to select four vertices in this manner.



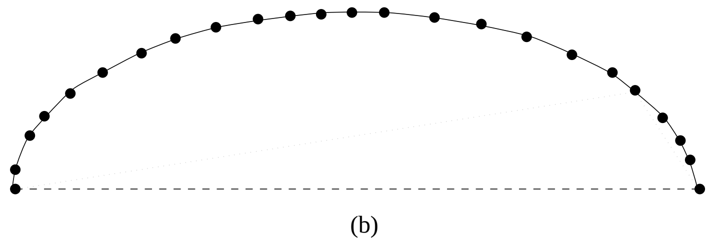


Figure 5.16: A base triangle of a 1-subpolygon of size $\leq m + n^{\alpha} - 2$ that induces:
(a) two 1A-subpolygons of size $\leq m$ ("the nice case")
(b) one 1A-subpolygon of size $\leq m$ and one 1A-subpolygon of size $> m$ ("the not-so-nice case")

After completing the initial work, the main recursive process is initiated. At the beginning of recursion step k (for some positive integer k), we assume that all 1-subpolygons of size $\leq m$, where $m = k \cdot n^{\alpha}$, have been triangulated optimally. We want to make the optimal solutions for all 1-subpolygons of size less than or equal to $m + n^{\alpha} - 2$ available by the end of recursion step k. (The '-2' will be accounted for later.)

Any d-edge that defines a 1-subpolygon has to be a side in some triangle belonging to an optimal triangulation T of the 1-subpolygon. Such a triangle is called a *base triangle* (of T). The other two sides in a base triangle cut the 1-subpolygon into two smaller 1-subpolygons called *1A-subpolygons*. If they were both of size $\leq m$ when the given 1-subpolygon had size $m + n^{\alpha} - 2$, their optimal triangulations would be available right away. Unfortunately, in some cases one 1A-subpolygon has more than m vertices. (See Figure 5.16(b).)

However, a base triangle in a 1A-subpolygon like this will cut it into two smaller 1-subpolygons (this time called 1B-subpolygons). At least one 1B-subpolygon has to have size $\leq m$; the other one can still have size $> m$. Continuing this process until all the induced 1-subpolygons have size $\leq m$, we might get 1C-polygons, 1D-polygons, etc. The important thing to notice is that until the last stage is reached, the total number of different vertices in all the 1-subpolygons of size $\leq m$ is $\leq n^{\alpha}$. (If it was larger than n^{α} then the remaining 1-subpolygon couldn't be of size $> m$.) Therefore, the last base triangle, called a *slicer triangle*, partitions the original 1-subpolygon into:

- The base triangle itself
- One 2-subpolygon of size $\leq n^{\alpha}$ (degenerate, i.e. size 2, in "the nice case")
- Two 1-subpolygons of size $\leq m$

Optimal solutions for all of these are available.

To find an optimal solution for a 1-subpolygon of size $m + n^{\alpha} - 2$, we have to locate a slicer triangle. The other base triangles are contained within the MWT of the 2-subpolygon induced by the slicer triangle since by definition, base triangles always belong to the optimal triangulation. The 1-subpolygon might have two (or more) different slicer triangles. Since all triangulations that originate from slicer triangles are optimal, any one of them can be selected.

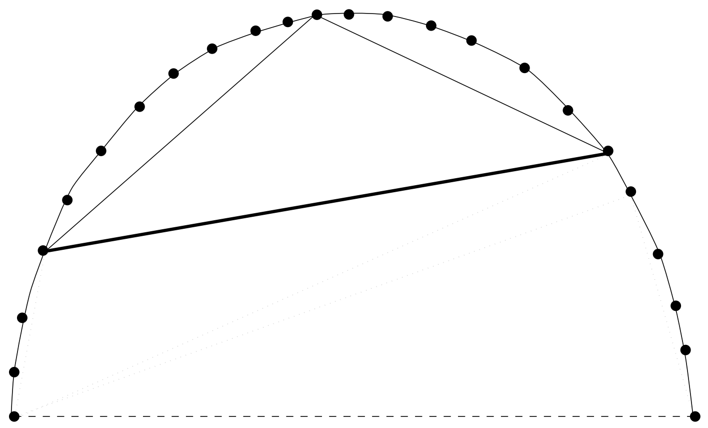


Figure 5.17: The 1-subpolygon of size $m + n^\alpha - 2$ is repeatedly divided into smaller pieces by base triangles. Note that the slicer triangle defines a 2-subpolygon of size $\leq n^\alpha$.

By combining a 2-subpolygon of size $\leq n^\alpha$, two 1-subpolygons of size $\leq m$, and a base triangle like we are doing here, we can obtain optimal solutions for any 1-subpolygon whose size is $\leq m + n^\alpha - 2$. We now see that the '-2' has to be included because two of the vertices are shared by the 2-subpolygon and the union of the 1-subpolygons.

In each recursion step, $O(n) \cdot O(n^\alpha) = O(n^{1+\alpha})$ 1-subpolygons are examined. (The same as the number of diagonals that cut off a 1-subpolygon of size z, where $m \leq z \leq m + n^\alpha - 2$.) Inside each 1-subpolygon, there are $O(n^\alpha) \cdot O(n^\alpha) \cdot O(n) = O(n^{1+2\alpha})$ triangles that might be slicer triangles. To simultaneously evaluate the resulting triangulations for all possible slicer triangles in $O(1)$ time, and select the best triangulation in $\log(n)$ time means $O(n^{1+\alpha}) \cdot O(n^{1+2\alpha}) = O(n^{2+3\alpha})$ parallel processors have to be used. As in the previous method, each processor can simulate $\log(n)$ others and still finish one recursion step in $O(\log(n))$ time, which lowers the required number of processors to $O((n^{2+3\alpha})/\log(n))$.

How many recursion steps are needed? Every step increases the maximum size of optimally triangulated 1-subpolygons by $n^\alpha - 2$, so by step k all 1-subpolygons of size $\leq k(n^\alpha - 2)$ will have been taken care of. Solving the equation $x(n^\alpha - 2) = n$ yields $x = n/(n^\alpha - 2)$. When n grows large, the '-2' can be ignored, and we get $x = n^{1-\alpha}$ asymptotically. Every step takes $O(\log(n))$ time, so the execution time for the main loop is $O(n^{1-\alpha}\log(n))$.

The preprocessing part used $O(n^{2+2\alpha})$ parallel processors and took $O(n^{3\alpha})$ time. The total number of processors is therefore $O(\max((n^{2+2\alpha}), (n^{2+3\alpha}/\log(n)))) = O(n^{2+3\alpha}/\log(n))$, and the total running time is $O(n^{3\alpha}) + O(n^{1-\alpha}\log(n)) = O(\max(n^{3\alpha}, n^{1-\alpha}\log(n)))$.

By setting $\alpha = 0.25$, we obtain a method that uses $O(n^{2.75}/\log(n))$ parallel processors and takes $O(n^{0.75}\log(n))$ time to complete.

5.5 General algorithm

The method from Section 5.4 can be adapted to situations where the allowed total running time is less than $O(n^{0.75}\log(n))$ by changing the parameter α. Naturally, the needed number of parallel processors changes too. This section shows how.

According to Section 5.4, the preprocessing requires $O(n^{2+2\alpha})$ processors and runs in $O(n^{3\alpha})$ time, where α is a parameter $(0 < \alpha < 1)$. The main loop uses $O(n^{2+3\alpha})$ processors and runs in $O(n^{1-\alpha}\log(n))$ time. The total number of processors is therefore $O(\max(n^{2+2\alpha}, n^{2+3\alpha}/\log(n))) = O(n^{2+3\alpha}/\log(n))$, and the total running time $O(n^{3\alpha})+O(n^{1-\alpha}\log(n)) = O(\max(n^{3\alpha}, n^{1-\alpha}\log(n)))$. We want the main loop to dominate, yielding a total running time which is $O(n^{1-\alpha}\log(n))$ for $0 < \alpha < 1$.

$0 < \alpha \leq 1/3$:

When $0 < \alpha \leq 0.25$, $n^{3\alpha} < n^{1-\alpha}\log(n)$, which means the running time is $O(n^{1-\alpha}\log(n))$. For $\alpha > 0.25$, the preprocessing stage takes more time than the main loop. To circumvent this, we have to change our tactics somewhat. Some processors are unused until the main loop is reached, so the first modification involves making the processor utilization more efficient.

The method from Section 5.1 uses $O(n^2/\log(n))$ processors to optimally triangulate a simple polygon with n vertices in $O(n\log(n))$ time. If every processor is assigned the work of n others, only $O(n/\log(n))$ processors are needed while the running time is changed to $O(n^2\log(n))$. Here, we have to triangulate 1- and 2-subpolygons of size $\leq n^\alpha$, so the preprocessing time would be $O((n^\alpha)^2\log(n^\alpha)) = O(n^{2\alpha}\log(n))$ with this method. The total number of processors is increased to $O(n^{2+2\alpha}) \cdot O(n^\alpha/\log(n)) = O(n^{2+3\alpha}/\log(n))$, but the main loop uses this many processors anyway. The algorithm's total running time is now changed to $O(n^{2\alpha}\log(n)) + O(n^{1-\alpha}\log(n)) = O(\max(n^{2\alpha}, n^{1-\alpha})\log(n))$, which is equal to $O(n^{1-\alpha}\log(n))$ since $0 < \alpha \leq 1/3$.

$1/3 < \alpha < 1$:

For $\alpha > 1/3$, we face the same problem as we did before. But this time we might have to use more processors in the preprocessing stage than in the main loop in order to keep the preprocessing time from growing faster than the main loop's executing time.

Use the method described in Section 5.3 to optimally triangulate all 1- and 2-subpolygons of size $\leq n^\alpha$ in $O(\log(n)^2)$ time with $O(n^{2+4\alpha}/\log(n))$ parallel processors. (Carry on like usual until the MWTs of all 1- and 2-subpolygons of size $\leq n^\alpha$ have been computed, then stop. At this point, $O(n^\alpha) \cdot O(n^\alpha) = O(n^{2\alpha})$ diagonals have been tested for each subpolygon. There are $O(n^{2+2\alpha})$ 1- and 2-subpolygons of the right size, and each processor handles $\log(n)$ of them. In other words, $O(n^{2\alpha}) \cdot O(n^{2+2\alpha}/\log(n)) = O(n^{2+4\alpha}/\log(n))$ processors are enough.)

Similarly to the technique used above, letting one processor do the work of $n^{1-\alpha}$ others reduces this number to $O((n^{2+4\alpha}/\log(n))/(n^{1-\alpha})) = O(n^{1+5\alpha}/\log(n))$ at the expense of increasing the time needed to $O(n^{1-\alpha}\log(n)^2)$. The main loop remains unchanged, so the whole algorithm requires $O(\max((n^{1+5\alpha}/\log(n)), (n^{2+3\alpha}/\log(n))))$ processors and takes $O(n^{1-\alpha}\log(n)^2)+O(n^{1-\alpha}\log(n)) = O(n^{1-\alpha}\log(n)^2)$ time. This is slightly worse than $O(n^{1-\alpha}\log(n))$, but still acceptable.

The total running time, $T(n,\alpha)$, and the number of processors needed, $P(n,\alpha)$, are thus given by:

$$T(n,\alpha) = \begin{cases} O(n^{1-\alpha}\log(n)), & 0 < \alpha \leq 1/3 \\ O(n^{1-\alpha}\log(n)^2), & 1/3 < \alpha < 1 \end{cases}$$

$$P(n,\alpha) = \begin{cases} O(n^{2+3\alpha}/\log(n)), & 0 < \alpha \leq 1/2 \\ O(n^{1+5\alpha}/\log(n)), & 1/2 \leq \alpha < 1 \end{cases}$$

$T(n,\alpha)$ and $P(n,\alpha)$ are plotted as functions of α in Figure 5.18.

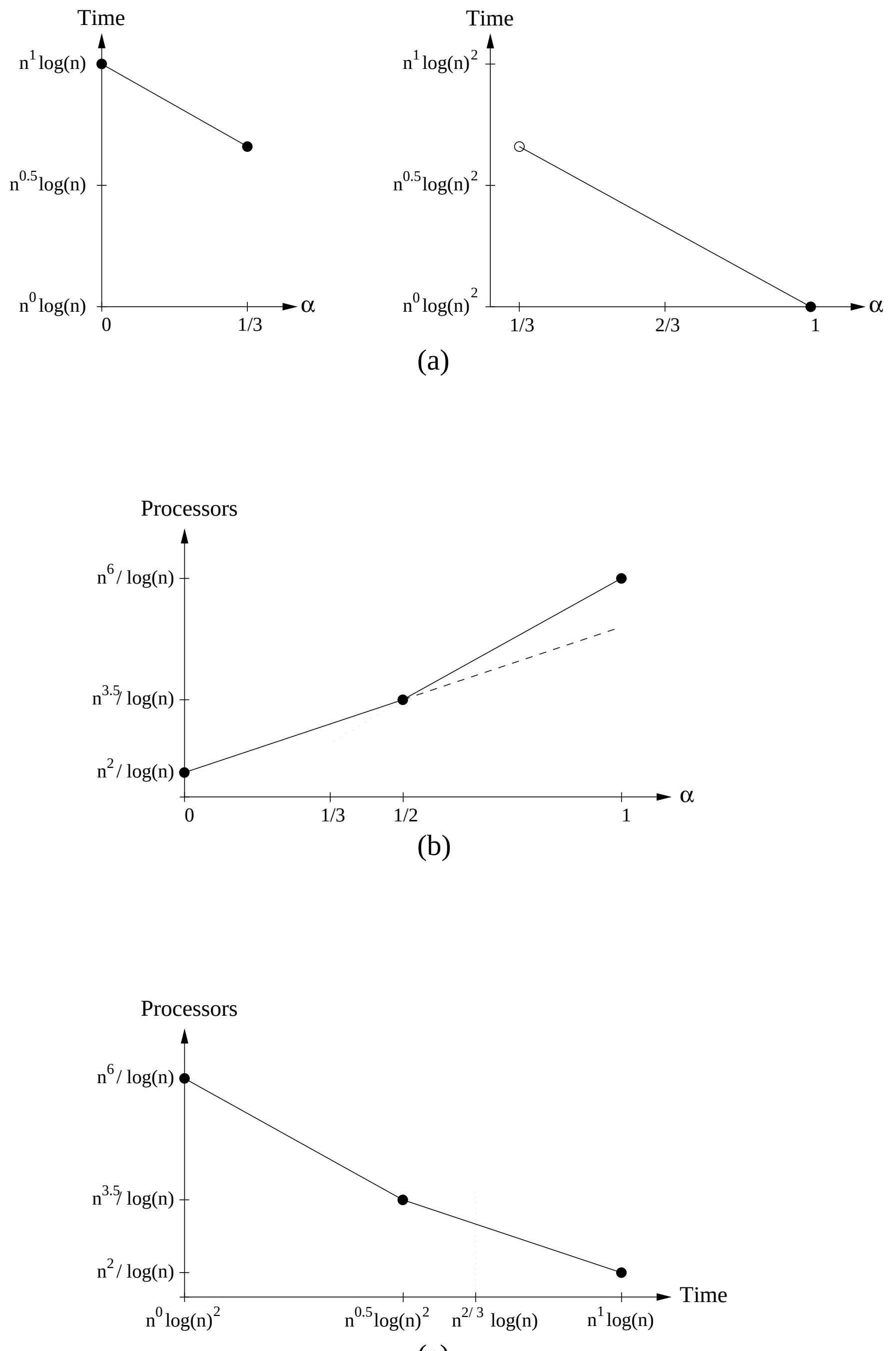


Figure 5.18:
(a) The total running time, $T(n, \alpha)$. A discontinuity occurs between $\alpha = 1/3$ and $\alpha > 1/3$.
(b) The required number of processors, $P(n, \alpha)$. The dotted line represents the preprocessing stage and the dashed line represents the main loop. (The preprocessing stage uses more processors than the main loop when $\alpha > 0.5$.)
(c) $T(n, \alpha)$ and $P(n, \alpha)$ combined. There is a discontinuity in the time scale at $n^{2/3}\log(n)$.

5.6 Conclusion

This chapter presented some CREW PRAM dynamic programming algorithms for solving the MWT problem for simple polygons. Their asymptotic running times depend on the number of parallel processors employed. A short overview of the algorithms is given below:

Number of processors	*Running time*	*Presented in*
1	$O(n^3)$	Section 3.2
$O(n^2/\log(n))$	$O(n\log(n))$	Section 5.1
$\downarrow$	$\downarrow$	Section 5.5
$O(n^{2.75}/\log(n))$	$O(n^{0.75}\log(n))$	Section 5.4
$\downarrow$	$\downarrow$	Section 5.5
$O(n^6/\log(n))$	$O(\log(n)^2)$	Section 5.3
$O(n^9)$	$O(\log(n)^2)$	Section 5.2

The general algorithm in Section 5.5 can be used when more than $O(n^2/\log(n))$ but less than $O(n^6/\log(n))$ processors are available. Its running time is somewhere between $O(n\log(n))$ and $O(\log(n)^2)$, depending on the number of processors.

In a recent article by Galil & Park [6], Rytter's approach is used to obtain a method that runs in $O(\log(\nu)^2 + (n\log(n))/\nu)$ time and requires $O(n^3 + n^2\nu^4 + n\nu^5\log(\nu))$ operations (ν corresponds to our n^α). For $\nu = n^{1/4}$, $O(n^{0.75}\log(n))$ time and only $O(n^3)$ operations are required. This is better than the algorithm presented here, which needs $O(n^{2.75}/\log(n))$ processors to complete in $O(n^{0.75}\log(n))$ time, i.e. $O(n^{3.5})$ operations. There might exist geometric methods like the ones in this chapter that attain the same (optimal) result as Galil & Park.

5.7 Generalization

The problem of constructing other types of triangulations such as the greedy triangulation can be solved after making a few changes. If

1. C is a class of triangulations of simple polygons with a special property: a C-triangulated polygon that is split along a C-diagonal yields two C-triangulated subpolygons
2. There is an efficient way of testing whether a given triangulation is a C-triangulation or not

then we can construct a C-triangulation of any simple polygon with methods based on the ones in this chapter. Proceed like usual, but whenever different ways of partitioning a given subpolygon into smaller subpolygons (which have been taken care of already) are evaluated, test if the union between the dividing diagonals and the smaller subpolygons is a C-triangulation instead of comparing the resulting weights. If the C-triangulation test uses $q(n)$ processors to complete in $O(\log(n))$ time, then a C-triangulation of a simple polygon can be obtained with $O(p(n)\cdot q(n))$ processors in $O(t(n))$ time, where $p(n)$ and $t(n)$ are the processor and time requirements for the corresponding unmodified dynamic programming algorithm.

In [19], Levcopoulos, Lingas, and Wang showed that triangulation classes which impose a partial order on their diagonals admit tests that can be carried out in $O(\log(n))$ time by $O(n^3/\log(n))$ processors (using a CREW PRAM). This result is valid for all PSLGs, but for partial orders induced by equivalence relations with linearly ordered equivalence classes, even fewer processors (i.e. $O(n^2/\log(n))$) are needed in the simple polygon case. Thus, it is possible to test if a given triangulation of a simple polygon is a greedy triangulation in $O(\log(n))$ time with $O(n^2/\log(n))$ processors. By combining this with what we said above, we get a method for constructing the greedy triangulation of simple polygons that uses $O(n^{2.75}/\log(n)\cdot n^2/\log(n)) = O(n^{4.75}/\log(n)^2)$ processors and takes $O(n^{0.75}\log(n))$ time.

Observe that the dynamic programming algorithms that run in $O(n)$ time or more are unsuitable for this kind of adaptation; there are methods that are much more processor efficient. For example, one processor is enough for computing the greedy triangulation in $O(n\log(n))$ time (see Chapter 3.3.3). The algorithm from Section 5.1 would also take $O(n\log(n))$ time, but with $O(n^2/\log(n)\cdot n^2/\log(n)) = O(n^4/\log(n)^2)$ processors.

Appendix A

Lower bounds for the nonoptimality of the MST- and GST-triangulations

In Chapter 3.3.5 we claimed that for every $n \geq 4$, there exists a planar point set S_1 with n vertices such that $|T_{MST}(S_1)| \ / \ |MWT(S_1)| = \Omega(n)$, and that for every $n \geq 17$, there exists a planar point set S_2 with n vertices such that $|T_{GST}(S_2)| \ / \ |MWT(S_2)| = \Omega(\sqrt{n})$. This appendix will show how such point sets can be constructed. The method that is used was inspired by Levcopoulos [13].

Both examples are divided into two cases:

- More than two vertices are allowed on the same line ("the simple case")
- No two vertices are allowed to be collinear ("the nondegenerate case")

Each example assumes that there are three exterior vertices located outside the convex hull of the given point set so that an outer convex hull is formed. The distance between an exterior vertex and the given point set is on the same scale as the diameter of the original convex hull.

Levcopoulos and Krznaric published similar proofs for these bounds in a recent report [15].

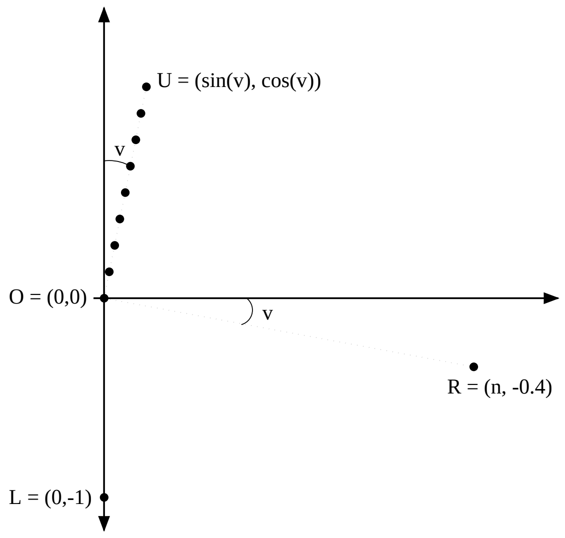


Figure A.1: The simple case.

A.1 An $\Omega(n)$ lower bound for the nonoptimality of the MST-triangulation

<u>**The simple case:**</u> (More than two vertices are allowed on the same line.)

Let S be the set of n (≥ 4) vertices with the following coordinates (see Figure A.1):

Right vertex:	$R = (n, -0.4)$
Lowest vertex:	$L = (0, -1)$
Crowd vertices:	A total of $n-2$ vertices evenly distributed along the line segment between $O = (0,0)$ and $U = (\sin(v), \cos(v))$, where $v = \arctan(\frac{0.4}{n})$.

The line segment between O and U is called *the crowd line*.
The line segment between O and R is called *the blocking line*.

$$|\overline{UR}| = \sqrt{(n-\sin(v))^2 + (-0.4-\cos(v))^2} = \sqrt{n^2 - 2n\sin(v) + 0.8\cos(v) + 1.16} = \Theta(n)$$

$$|\overline{OR}| = \sqrt{(n-0)^2 + (-0.4-0)^2} = \sqrt{n^2 + 0.16} = \Theta(n)$$

$$|\overline{LR}| = \sqrt{(n-0)^2 + (-0.4-(-1))^2} = \sqrt{n^2 + 0.36} = \Theta(n)$$

The length of the convex hull of S is
$|CH(S)| = |\overline{OL}| + |\overline{OU}| + |\overline{UR}| + |\overline{LR}| = \Theta(1) + \Theta(1) + \Theta(n) + \Theta(n) = \Theta(n)$

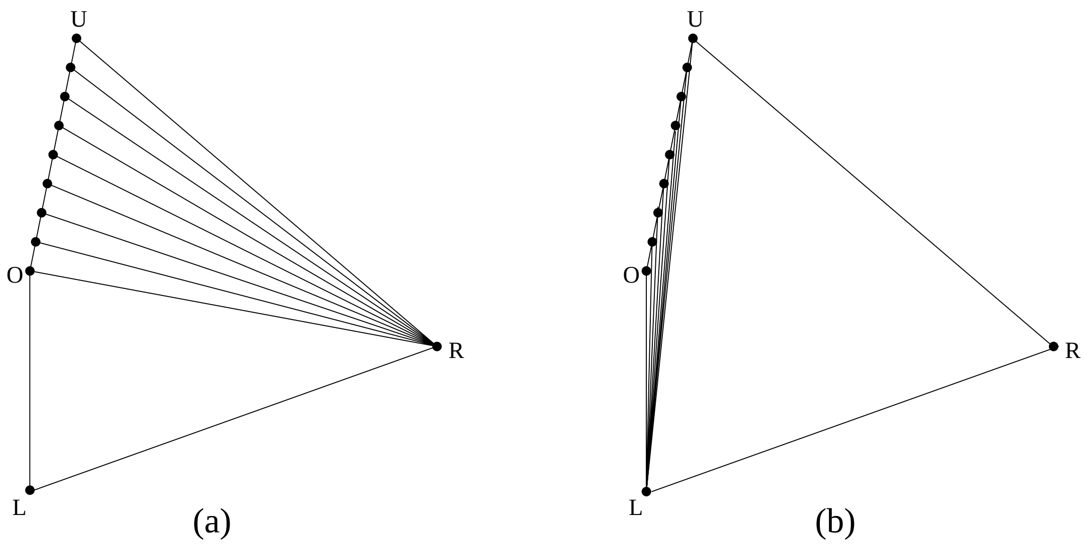


Figure A.2: (a) $T_{MST}(S)$ (b) $T(S)$

Since the blocking line is perpendicular to the crowd line, and -0.4 is closer to 0 than to -1, the blocking line has to belong to the minimum spanning tree of S.

$T_{MST}(S)$ is displayed in Figure A.2 (a).

$|T_{MST}(S)| = |CH(S)| + \Theta(n) \cdot ((n-2)-1) = \Theta(n) + \Theta(n^2) = \Theta(n^2)$

Let $T(S)$ be the triangulation of S obtained by drawing all possible diagonals from L to the vertices on the crowd line and adding the convex hull of S.
$T(S)$ is shown in Figure A.2 (b).

$|T(S)| = |CH(S)| + \Theta(1) \cdot ((n-2)-1) = \Theta(n) + \Theta(n) = \Theta(n)$

Thus,

$$\frac{|T_{MST}(S)|}{|T(S)|} = \frac{\Theta(n^2)}{\Theta(n)} = \Theta(n)$$

which means that

$$\frac{|T_{MST}(S)|}{|MWT(S)|} = \Omega(n)$$

□

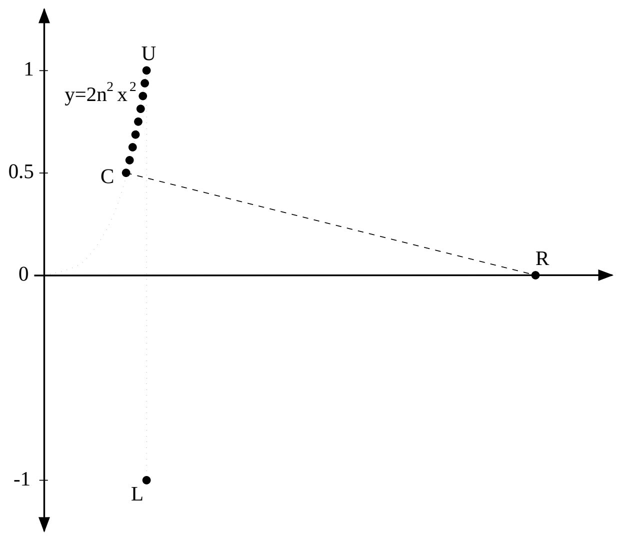


Figure A.3: The nondegenerate case.

The nondegenerate case: (No three vertices are allowed to be collinear.)

Let S' be the set of n (≥ 4) vertices with the following coordinates (see Figure A.3):

Right vertex:	$R = (n + \frac{1}{2n}, 0)$
Lowest vertex:	$L = (\frac{1}{\sqrt{2} \cdot n}, -1)$
Crowd vertices:	A total of $n-2$ vertices evenly distributed along the parabola $y(x) = 2n^2 \cdot x^2$ between $C = (\frac{1}{2n}, 0.5)$ and $U = (\frac{1}{\sqrt{2} \cdot n}, 1)$.

Fact 1: $\overline{CR}$ is perpendicular to the tangent of y in point C.
This is because $y'(x) = 4n^2 \cdot x$ (which yields $y'(0.5) = 2n^2$) and

$$y(0.5) + (-\frac{1}{y'(0.5)}) \cdot \Delta x \;=\; \frac{1}{2n} - \frac{1}{2n^2} \cdot ((n + \frac{1}{2n}) - \frac{1}{2n}) \;=\; \frac{1}{2n} - \frac{1}{2n} \;=\; 0$$

$\square$

Next,

$|\overline{CR}| = \sqrt{((n + \frac{1}{2n}) - \frac{1}{2n})^2 + (0 - 0.5)^2} = \sqrt{n^2 + 0.25}$
$|\overline{LR}| = \sqrt{((n + \frac{1}{2n}) - \frac{1}{\sqrt{2}n})^2 + (0 - (-1))^2} = \sqrt{n^2 + (2 - \sqrt{2}) + \frac{(\sqrt{2}-1)^2}{4n^2}}$

Since $0.25 < 2 - \sqrt{2}$ and $0 < \frac{(\sqrt{2}-1)^2}{4n^2}$, we have

Fact 2: $|\overline{CR}| < |\overline{LR}|$ $\square$

Fact 1 and Fact 2 imply that $\overline{CR}$ must belong to the minimum spanning tree of S'.

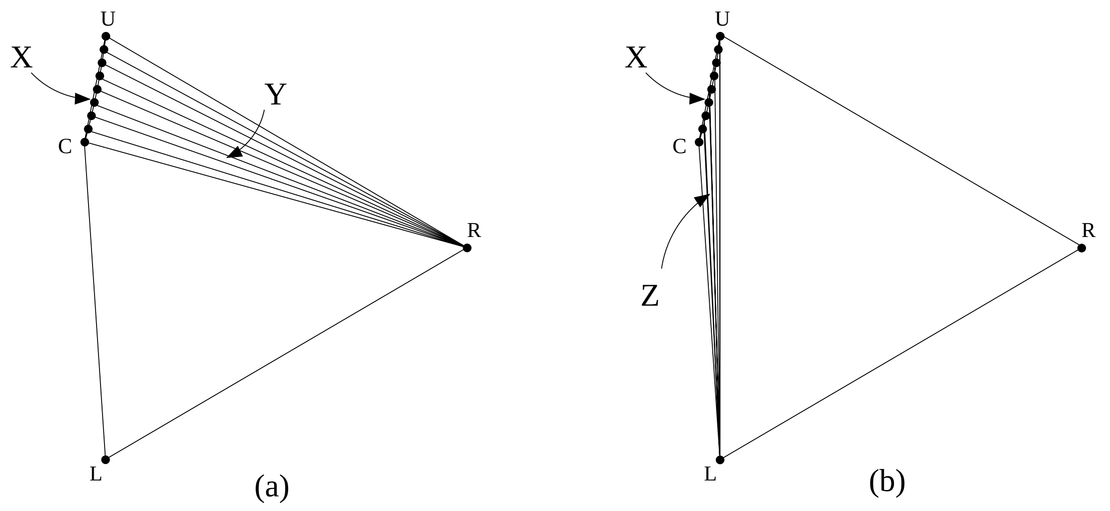


Figure A.4: (a) $T_{MST}(S')$ (b) $T(S')$

$T_{MST}(S')$ is displayed in Figure A.4 (a).

The length of the convex hull of S' is
$|CH(S')| = |\overline{LC}| + |\overline{CU}| + |\overline{UR}| + |\overline{LR}| = \Theta(1) + \Theta(1) + \Theta(n) + \Theta(n) = \Theta(n)$

which gives us

$$\begin{aligned}|T_{MST}(S')| &= |CH(S')| + \overbrace{\Theta(1)\cdot(n-3)}^{W} + |\overline{CR}| + \overbrace{\Theta(1)\cdot(n-5)}^{X} + \overbrace{\Theta(n)\cdot(n-4)}^{Y}\\ &= \Theta(n) + \Theta(n) + \Theta(n) + \Theta(n) + \Theta(n^2)\\ &= \Theta(n^2)\end{aligned}$$

where W corresponds to the set of diagonals on the parabola y between C and U, and X and Y correspond to the optimally triangulated areas (polygons) shown in Figure A.4 (a).

As a comparison, let $T(S')$ be the triangulation consisting of the convex hull of S', all diagonals between L and crowd vertices, and the same triangulation of area X as in T_{MST}.
See Figure A.4 (b).

$$\begin{aligned}|T(S')| &= |CH(S')| + \overbrace{\Theta(1)\cdot(n-3)}^{W} + |\overline{LU}| + \overbrace{\Theta(1)\cdot(n-5)}^{X} + \overbrace{\Theta(1)\cdot(n-4)}^{Z}\\ &= \Theta(n) + \Theta(1) + \Theta(n) + \Theta(n) + \Theta(n)\\ &= \Theta(n)\end{aligned}$$

Thus,

$$\frac{|T_{MST}(S')|}{|T(S')|} = \frac{\Theta(n^2)}{\Theta(n)} = \Theta(n)$$

which means that

$$\frac{|T_{MST}(S')|}{|MWT(S')|} = \Omega(n)$$

□

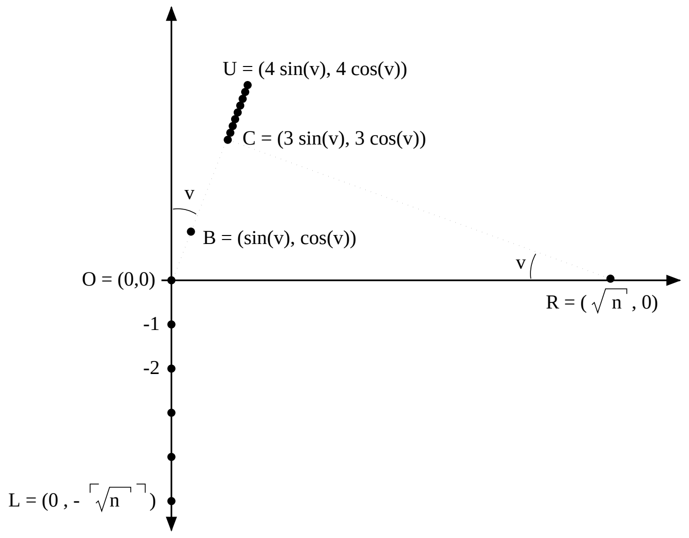


Figure A.5: The simple case.

A.2 An $\Omega(\sqrt{n})$ lower bound for the nonoptimality of the GST-triangulation

<u>**The simple case:**</u> (More than two vertices are allowed on the same line.)

Let S be the set of n (≥ 17) vertices with the following coordinates (see Figure A.5):

Blocking vertex:	$B = (\sin(v), \cos(v))$
Right vertex:	$R = (\sqrt{n}, 0)$
Crowd vertices:	A total of $n - \lceil\sqrt{n}\,\rceil - 3$ vertices evenly distributed along the line segment between $C = (3\sin(v), 3\cos(v))$ and $U = (4\sin(v), 4\cos(v))$, where $\sin(v) = \frac{3}{\sqrt{n}}$.
Pulling vertices:	A total of $\lceil\sqrt{n}\,\rceil + 1$ vertices with integer coordinates distributed along the negative y-axis between $O = (0,0)$ and $L = (0, -\lceil\sqrt{n}\,\rceil)$.

We start out by observing that $|\overline{BR}| < |\overline{CL}|$ since

$$|\overline{BR}| = \sqrt{(\sqrt{n} - \sin(v))^2 + (0 - \cos(v))^2} = \sqrt{n-5}$$
$$|\overline{CL}| = \sqrt{(0 - 3\sin(v))^2 + (-\lceil\sqrt{n}\,\rceil - 3\cos(v))^2} = \sqrt{\lceil\sqrt{n}\,\rceil^2 + 6\lceil\sqrt{n}\,\rceil\sqrt{1 - \tfrac{9}{n}} + 9}$$

and

$$n - 5 \;<\; n + 6\sqrt{n-9} + 9 \;\leq\; n + 6\frac{\lceil\sqrt{n}\,\rceil}{\sqrt{n}}\sqrt{n-9} + 9 \;\leq\; \lceil\sqrt{n}\,\rceil^2 + 6\lceil\sqrt{n}\,\rceil\sqrt{1 - \frac{9}{n}} + 9$$

In other words, $\overline{BR}$ will be selected before $\overline{CL}$ in a greedy triangulation.
Furthermore, any diagonal from C to a pulling vertex will be blocked because of B. (Assume that the diagonal from B to the pulling vertex at $(0, -j)$ has just been selected. When choosing between the diagonal from C to the pulling vertex at $(0, -j)$ and the diagonal from B to the pulling vertex at $(0, -j-1)$, the latter will be picked since it is shorter.)

$GT(S)$ is displayed in Figure A.6 (a).

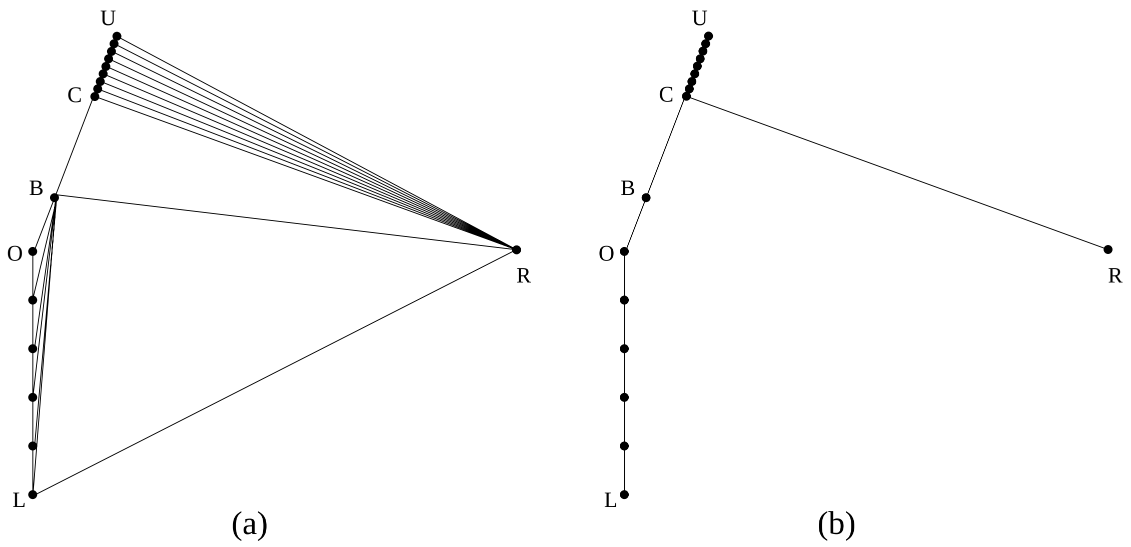


Figure A.6: (a) $GT(S)$ (b) $Sp(S)$

Let $Sp(S)$ be a spanning tree for S of shortest possible length made exclusively from edges that belong to $GT(S)$. It is easy to see that it will consist of the diagonals between adjacent pulling vertices, $\overline{OB}$, $\overline{BC}$, and the diagonals between adjacent crowd vertices. But how is R connected?

Consider the line segment $\overline{CR}$. It is perpendicular to $\overline{OU}$, so none of the crowd vertices, vertex B, or vertex O are closer to R than C is. Next, since $\overline{OR} \perp \overline{OL}$, the length of $\overline{LR}$ is greater than that of $\overline{OR}$ and therefore also greater than $|\overline{CR}|$. So, $\overline{CR}$ will belong to $Sp(S)$.

$Sp(S)$ is shown in Figure A.6 (b).

$T_{GST}(S)$ is obtained from $Sp(S)$ by adding the missing pieces of the convex hull of S and then optimally triangulating the induced polygons.
See Figure A.7 (a).

The length of the convex hull of S is
$|CH(S)| = |\overline{OL}| + |\overline{OU}| + |\overline{UR}| + |\overline{LR}| = \Theta(\sqrt{n}) + \Theta(1) + \Theta(\sqrt{n}) + \Theta(\sqrt{n}) = \Theta(\sqrt{n})$

which means that

$$\begin{aligned}|T_{GST}(S)| &= |CH(S)| + |\overline{BR}| + \overbrace{\Theta(\sqrt{n})\cdot\lceil\sqrt{n}\,\rceil}^{W} + \overbrace{\Theta(\sqrt{n})\cdot(n - \lceil\sqrt{n}\,\rceil - 4)}^{X} \\ &= \Theta(\sqrt{n}) + \Theta(\sqrt{n}) + \Theta(n) + \Theta(n\sqrt{n}) \\ &= \Theta(n\sqrt{n})\end{aligned}$$

where W and X correspond to the optimally triangulated areas shown in Figure A.7 (a).

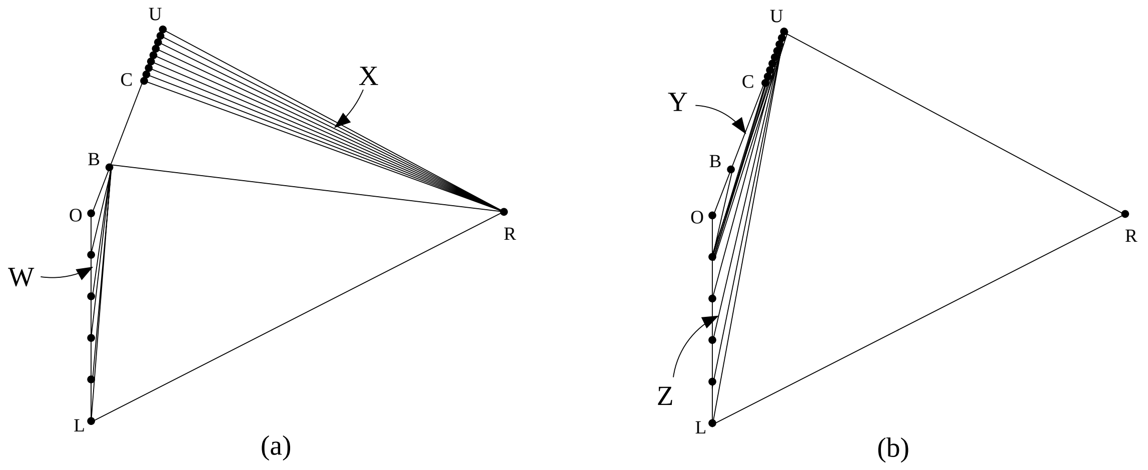


Figure A.7: (a) $T_{GST}(S)$ (b) $T(S)$

Let $T(S)$ be the triangulation of S obtained by starting with the convex hull of S and then drawing the diagonal between $(0,-1)$ and B (call it d), all the diagonals from crowd vertices to $(0,-1)$, and finally all the diagonals from pulling vertices to U.
$T(S)$ is shown in Figure A.7 (b).

$$\begin{aligned}|T(S)| &= |CH(S)| + |d| + \overbrace{\Theta(1)\cdot(n - \lceil\sqrt{n}\,\rceil - 3)}^{Y} + \overbrace{\Theta(\sqrt{n})\cdot(\lceil\sqrt{n}\,\rceil - 1)}^{Z} \\ &= \Theta(\sqrt{n}) + \Theta(1) + \Theta(n) + \Theta(n) \\ &= \Theta(n)\end{aligned}$$

Thus,

$$\frac{|T_{GST}(S)|}{|T(S)|} = \frac{\Theta(n\sqrt{n})}{\Theta(n)} = \Theta(\sqrt{n})$$

which means that

$$\frac{|T_{GST}(S)|}{|MWT(S)|} = \Omega(\sqrt{n})$$

□

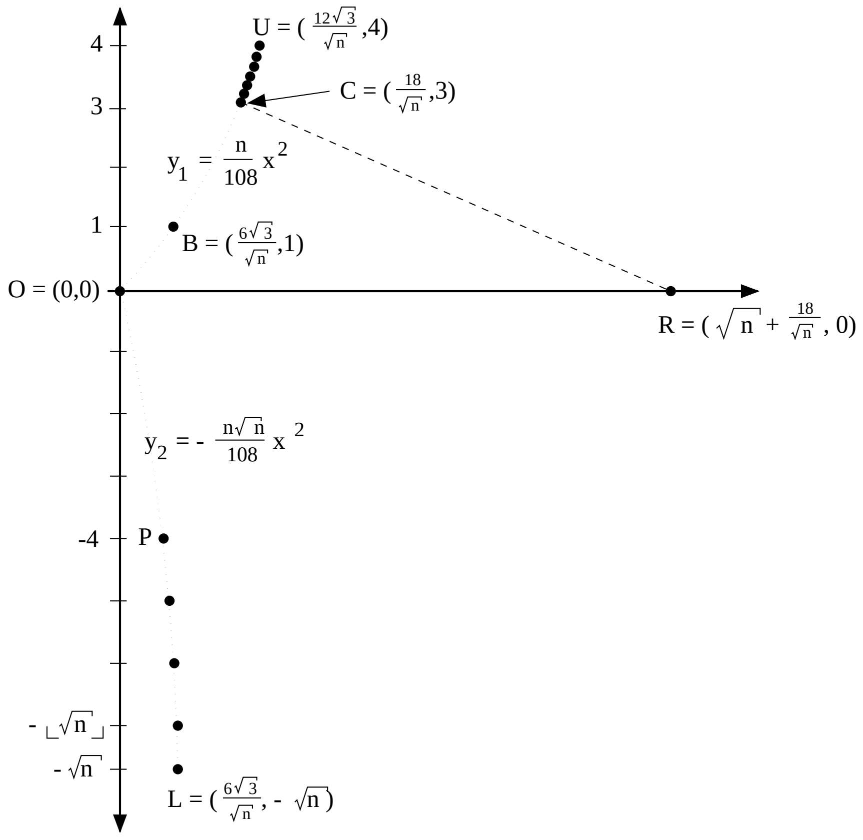


Figure A.8: The nondegenerate case.

The nondegenerate case: (No three vertices are allowed to be collinear.)

Let S' be the set of n (≥ 17) vertices with the following coordinates (see Figure A.8):

Origin vertex:	$O = (0,0)$
Blocking vertex:	$B = (\frac{6\sqrt{3}}{\sqrt{n}}, 1)$
Right vertex:	$R = (n + \frac{18}{\sqrt{n}}, 0)$
Crowd vertices:	A total of $n - \lceil\sqrt{n}\,\rceil$ vertices evenly distributed along the parabola $y_1(x) = \frac{n}{108}\cdot x^2$ between $C = (\frac{18}{\sqrt{n}}, 3)$ and $U = (\frac{12\sqrt{3}}{\sqrt{n}}, 4)$.
Pulling vertices:	A total of $\lceil\sqrt{n}\,\rceil - 3$ vertices located on the parabola $y_2(x) = -\frac{n\sqrt{n}}{108}\cdot x^2$. Let their y-coordinates be all the integers between -4 (at P) and $\lfloor\sqrt{n}\,\rfloor$. If $\sqrt{n} \neq \lfloor\sqrt{n}\,\rfloor$, place an extra vertex in $y = -\sqrt{n}$. The vertex in $(\frac{6\sqrt{3}}{\sqrt{n}}, -\sqrt{n})$ has the lowest y-coordinate, and is called L.

Note that B and L have the same x-coordinate.

Fact 1: $\overline{CR}$ is perpendicular to the tangent of y_1 in point C.
This is because $y_1'(x) = \frac{n}{54}\cdot x$ (which means $y_1'(\frac{18}{\sqrt{n}}) = \frac{\sqrt{n}}{3}$) and

$$y_1(\frac{18}{\sqrt{n}}) + (-\frac{1}{y_1'(\frac{18}{\sqrt{n}})})\cdot\Delta x \;=\; 3 - \frac{3}{\sqrt{n}}\cdot((\sqrt{n} + \frac{18}{\sqrt{n}}) - \frac{18}{\sqrt{n}}) \;=\; 3 - 3 \;=\; 0$$

□

Fact 2: $|\overline{BR}| < |\overline{CL}|$

Proof:

$|\overline{BR}| = \sqrt{((\sqrt{n} + \frac{18}{\sqrt{n}}) - \frac{6\sqrt{3}}{\sqrt{n}})^2 + (0-1)^2} = \sqrt{\frac{n^2+(37-12\sqrt{3})n+216(2-\sqrt{3})}{n}}$

$|\overline{CL}| = \sqrt{(\frac{6\sqrt{3}}{\sqrt{n}} - \frac{18}{\sqrt{n}})^2 + (-\sqrt{n}-3)^2} = \sqrt{\frac{n^2+6n\sqrt{n}+9n+216(2-\sqrt{3})}{n}}$

and $(37-12\sqrt{3})n < 6n\sqrt{n} + 9n$ since the inequality $\frac{28-12\sqrt{3}}{6} < \sqrt{n}$ is always satisfied (recall that we decided to set $n \geq 17$).

Thus, $|\overline{BR}| < |\overline{CL}|$. □

Fact 3: Let P_{-k} be the pulling vertex with y-coordinate $-k$.
For every integer k, where $4 \leq k < \sqrt{n} - 1$, the diagonal between B and the pulling vertex called P_{-k-1} is shorter than the diagonal between R and P_{-k}.

Proof:

$$\begin{aligned}|\overline{P_{-k-1}\ B}| &= \sqrt{(\frac{6\sqrt{3}}{\sqrt{n}} - \frac{\sqrt{108(k+1)}}{n^{3/4}})^2 + (1-(-k-1))^2} \\ &= \sqrt{\frac{(4+4k+k^2)n^{3/2}+108n^{1/2}-216\sqrt{k+1}\cdot n^{1/4}+108(k+1)}{n^{3/2}}}\end{aligned}$$

$$\begin{aligned}|\overline{P_{-k}\ R}| &= \sqrt{((\sqrt{n}+\frac{18}{\sqrt{n}}) - \frac{\sqrt{108k}}{n^{3/4}})^2 + (0-(-k))^2} \\ &= \sqrt{\frac{n^{5/2}+(36+k^2)n^{3/2}-12\sqrt{3}\sqrt{k}\cdot n^{5/4}+324n^{1/2}-216\sqrt{3}\sqrt{k}\cdot n^{1/4}+108k}{n^{3/2}}}\end{aligned}$$

First of all, $n \geq 17$ gives us $\begin{cases} n^{5/2} - 4n^2 \ > \ 0 \\ (32-12\sqrt{3})n^{3/2} + (1-\sqrt{3})216\cdot n^{1/2} \ > \ 0 \\ 216n^{1/4} - 108 \ > \ 0 \end{cases}$

This means that

$$n^{5/2} - 4n^2 + (32-12\sqrt{3})n^{3/2} + (1-\sqrt{3})216\cdot n^{1/2} + 216n^{1/4} - 108 \ > \ 0$$

which is the same as saying

$$4n^2 + 12\sqrt{3}n^{3/2} + 216\sqrt{3}\cdot n^{1/2} + 108 \ < \ n^{5/2} + 32n^{3/2} + 216n^{1/2} + 216n^{1/4}$$

Since $k < \sqrt{n}$ and $\sqrt{k+1} > 1$, we get

$$4k\cdot n^{3/2} + 12\sqrt{3}\sqrt{k}\cdot n^{5/4} + 216\sqrt{3}\sqrt{k}\cdot n^{1/4} + 108 \ < \ n^{5/2} + 32n^{3/2} + 216n^{1/2} + 216\sqrt{k+1}\cdot n^{1/4}$$

Further rearrangements lead to the relation

$$\begin{aligned}&(4+4k+k^2)n^{3/2} + 108n^{1/2} - 216\sqrt{k+1}\cdot n^{1/4} + 108(k+1) \\ &\quad < \ n^{5/2} + (36+k^2)n^{3/2} - 12\sqrt{3}\sqrt{k}\cdot n^{5/4} + 324n^{1/2} - 216\sqrt{3}\sqrt{k}\cdot n^{1/4} + 108k\end{aligned}$$

Putting everything together yields $|\overline{P_{-k-1}\ B}| < |\overline{P_{-k}\ R}|$. □

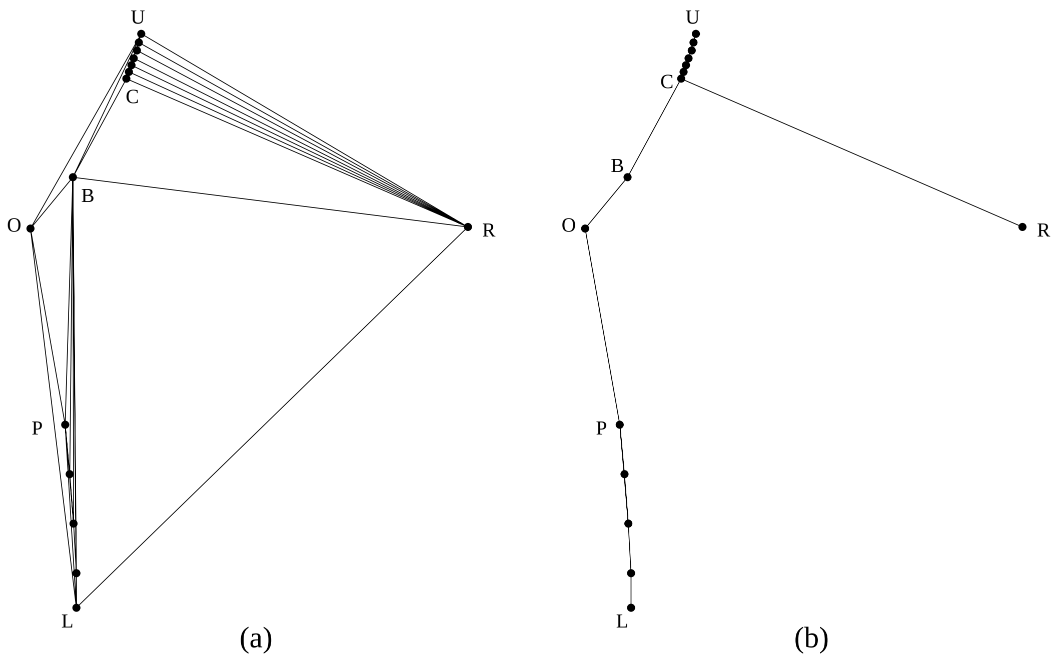


Figure A.9: (a) $GT(S')$ (b) $Sp(S')$

Fact 4: $|\overline{OP}| < |\overline{BP}|$

Proof:

$$|\overline{OP}| = \sqrt{\left(\sqrt{\tfrac{108\cdot 4}{n^{3/2}}} - 0\right)^2 + (-4-0)^2} = \sqrt{\tfrac{16n^{3/2}+432}{n^{3/2}}}$$

$$|\overline{BP}| = \sqrt{\left(\sqrt{\tfrac{108\cdot 4}{n^{3/2}}} - \tfrac{6\sqrt{3}}{\sqrt{n}}\right)^2 + (-4-1)^2} = \sqrt{\tfrac{25n^{3/2}+108n^{1/2}-432n^{1/4}+432}{n^{3/2}}}$$

We note that $432 < (9n+108)n^{1/4}$ because $n \geq 17$.
It then follows that $0 < 9n^{3/2}+108n^{1/2}-432n^{1/4}$, and finally $16n^{3/2} < 25n^{3/2}+108n^{1/2}-432n^{1/4}$.

Therefore, $|\overline{OP}| < |\overline{BP}|$. $\square$

Facts 1, 2, 3, and 4 cause the greedy triangulation of S' to look like Figure A.9 (a) and the spanning tree of minimal length consisting of greedy edges to look like Figure A.9 (b).

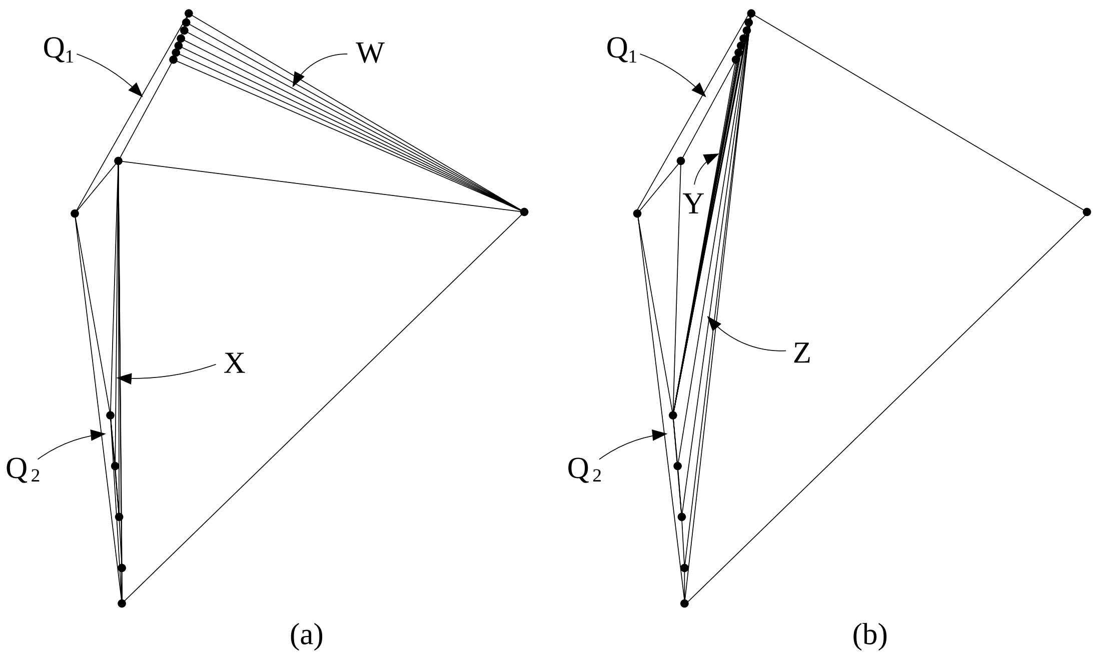


Figure A.10: (a) $T_{GST}(S')$ (b) $T(S')$

Just like before, $T_{GST}(S')$ is obtained from $Sp(S')$ by adding the missing pieces of the convex hull of S and then optimally triangulating the induced polygons.
See Figure A.10 (a).

Let Q_1 be the convex polygon whose edges are $\overline{OU}$ and the diagonals between consecutive vertices on y_1. Similarly, let Q_2 be the convex polygon whose edges are $\overline{OL}$ and the diagonals between consecutive vertices on y_2. According to [20], any convex polygon with n vertices (where no three are collinear) and perimeter of length p can be triangulated by inserting diagonals of total length $O(p\log(n))$. Using this fact, we can set $|Q_1| = O(1{\cdot}\log(n)) = O(\log(n))$ and $|Q_2| = O(\sqrt{n}\cdot\log(n))$, where $|Q_1|$ and $|Q_2|$ denote the total lengths of all inserted diagonals in two suitable triangulations of Q_1 and Q_2, respectively.

The length of the convex hull of S' is
$|CH(S')| = |\overline{OL}| + |\overline{OU}| + |\overline{UR}| + |\overline{LR}| = \Theta(\sqrt{n}) + \Theta(1) + \Theta(\sqrt{n}) + \Theta(\sqrt{n}) = \Theta(\sqrt{n})$

which gives us

$$\begin{aligned}
|T_{GST}(S')| = |CH(S')| &+ \overbrace{\Theta(1)\cdot(n-\lceil\sqrt{n}\,\rceil+1)}^{D_1} + \overbrace{\Theta(1)\cdot(\lceil\sqrt{n}\,\rceil-3)}^{D_2} + |Q_1| + |Q_2| + |\overline{BR}| \\
&+ \overbrace{\Theta(\sqrt{n})\cdot(n-\lceil\sqrt{n}\,\rceil-1)}^{W} + \overbrace{\Theta(\sqrt{n})\cdot(\lceil\sqrt{n}\,\rceil-3)}^{X} \\
&= \Theta(\sqrt{n}) + \Theta(n) + \Theta(n) + O(\log(n)) + O(\sqrt{n}\cdot\log(n)) + \Theta(\sqrt{n}) + \Theta(n\sqrt{n}) + \Theta(n) \\
&= \Theta(n\sqrt{n})
\end{aligned}$$

D_1 corresponds to the set of diagonals between consecutive vertices on y_1, and D_2 to the set of diagonals on y_2.
W represents the set of diagonals from the crowd vertices to R, and X the set of diagonals from the pulling vertices to B.
W and X are marked in Figure A.10 (a).

Let $T(S')$ be the triangulation of S' shown in Figure A.10 (b).

All crowd vertices are connected to P by the set of diagonals labelled Y in the figure.
Such a construction is possible since no diagonal from a crowd vertex to P crosses y_1.

Proof: The tangent of y_1 in U (whose equation is $y = \frac{2}{9}\sqrt{3n} \cdot x - 4$) crosses y_2 above $y = -4$ since they intersect in a point with the coordinates

$$(x_c, y_c) \;=\; (\frac{12\sqrt{3}}{n} \cdot (-1 + \sqrt{1 + n^{1/2}}) \;,\; \frac{8}{\sqrt{n}} \cdot (-1 + \sqrt{1 + n^{1/2}}) - 4)$$

and

$$y_c \;=\; -\frac{8}{\sqrt{n}} + \frac{8}{\sqrt{n}} \cdot \sqrt{1 + \sqrt{n}} - 4 \;>\; -\frac{8}{\sqrt{n}} + \frac{8}{\sqrt{n}} \cdot \sqrt{1} - 4 \;=\; -4.$$

Area Z in the figure consists of diagonals from the pulling vertices to U.

Finally, we have

$$\begin{aligned}|T(S')| = {} & |CH(S')| + \overbrace{\Theta(1) \cdot (n - \lceil\sqrt{n}\,\rceil + 1)}^{D_1} + \overbrace{\Theta(1) \cdot (\lceil\sqrt{n}\,\rceil - 3)}^{D_2} + |Q_1| + |Q_2| + |\overline{BP}| \\ & + \overbrace{\Theta(1) \cdot (n - \lceil\sqrt{n}\,\rceil)}^{Y} + \overbrace{\Theta(\sqrt{n}) \cdot (\lceil\sqrt{n}\,\rceil - 4)}^{Z} \\ = {} & \Theta(\sqrt{n}) + \Theta(n) + \Theta(n) + O(\log(n)) + O(\sqrt{n} \cdot \log(n)) + \Theta(1) + \Theta(n) + \Theta(n) \\ = {} & \Theta(n)\end{aligned}$$

Thus,

$$\frac{|T_{GST}(S')|}{|T(S')|} = \frac{\Theta(n\sqrt{n})}{\Theta(n)} = \Theta(\sqrt{n})$$

which means that

$$\frac{|T_{GST}(S')|}{|MWT(S')|} = \Omega(\sqrt{n})$$

□

Bibliography

[1] M. Ajtai, V. Chvátal, M. M. Newborn, E. Szemerédi. "Crossing-Free Subgraphs". *Annals of Discrete Mathematics 12* (1982), pages 9-12.

[2] G. Blom. *Sannolikhetsteori med tillämpningar*. Studentlitteratur, Lund (1984).

[3] S. Cook, C. Dwork, R. Reischuk. "Upper and Lower Time Bounds for Parallel Random Access Machines Without Simultaneous Writes". *SIAM Journal on Computing 15(1)* (1986), pages 87-97.

[4] T. Cormen, C. Leiserson, R. Rivest. *Introduction to Algorithms*. The MIT Press (1990).

[5] M. T. Dickerson, R. L. S. Drysdale, S. A. McElfresh, E. Welzl. "Fast greedy triangulation algorithms". *Proc. of the Tenth ACM Symp. on Computational Geometry* (1994), pages 211-220.

[6] Z. Galil, K. Park. "Parallel Algorithms for Dynamic Programming Recurrences with More Than O(1) Dependency". *Journal of Parallel and Distributed Computing 21* (1994), pages 213-222.

[7] P. D. Gilbert. "New Results in Planar Triangulations". M.S. Thesis, Coordinated Science Lab., University of Illinois at Urbana (1979).

[8] S. Goldman. "A Space Efficient Greedy Triangulation Algorithm". *Information Processing Letters 31* (1989), pages 191-196.

[9] R. Grimaldi. *Discrete and Combinatorial Mathematics: An Applied Introduction*. Addison-Wesley Publishing Company (1989).

[10] L. Heath, S. Pemmaraju. "New Results for the Minimum Weight Triangulation Problem". *Algorithmica 12* (1994), pages 533-552.

[11] R. Karp, V. Ramachandran. "Parallel algorithms for shared-memory machines". *Algorithms and Complexity*. Elsevier Science Publishers B.V. (1990), pages 871-941.

[12] D. C. Kozen. *The Design and Analysis of Algorithms*. Springer-Verlag (1992).

[13] C. Levcopoulos. "An $\Omega(\sqrt{n})$ Lower Bound for the Nonoptimality of the Greedy Triangulation". *Information Processing Letters 25* (1987), pages 247-251.

[14] C. Levcopoulos, D. Krznaric. "The Greedy Triangulation Can Be Computed from the Delaunay in Linear Time". Technical Report LU-CS-TR:94-136, Department of Computer Science, Lund University (1994).

[15] C. Levcopoulos, D. Krznaric. "Tight Lower Bounds for Minimum Weight Triangulation Heuristics". Technical Report LU-CS-TR:95-157, Department of Computer Science, Lund University (1995).

[16] C. Levcopoulos, A. Lingas. "On Approximation Behavior of the Greedy Triangulation for Convex Polygons". *Algorithmica 2* (1987), pages 175-193.

[17] C. Levcopoulos, A. Lingas. "Fast Algorithms for Greedy Triangulation". *BIT 32* (1992), pages 280-296.

[18] C. Levcopoulos, A. Lingas. "The Greedy Triangulation Approximates the Minimum Weight Triangulation and Can Be Computed in Linear Time in the Average Case". Technical Report LU-CS-TR:92-105, Department of Computer Science, Lund University (1992).

[19] C. Levcopoulos, A. Lingas, C. Wang. "On Parallel Complexity of Planar Triangulations". (1994).

[20] A. Lingas. "A Linear-Time Heuristic for Minimum Weight Triangulations of Convex Polygons". *Proc. 23rd Allerton Conf. on Computing, Communication, and Control, Urbana, Illinois* (1985).

[21] A. Lingas. "A New Heuristic for Minimum Weight Triangulation". *SIAM Journal of Algebraic and Discrete Methods 8* (1987), pages 646-658.

[22] A. Lingas. "Greedy Triangulation Can Be Efficiently Implemented in the Average Case". *Proceedings of Graph-Theoretic Concepts in Computer Science* (1988), pages 253-261.

[23] R. Lipton, R. E. Tarjan. "Applications of a planar separator theorem". *Proc. 18th Conf. Foundations of Computer Science* (1977), pages 162-170.

[24] G. K. Manacher, A. L. Zobrist. "Neither the Greedy nor the Delaunay Triangulation of a Planar Point Set Approximates the Optimal Triangulation". *Information Processing Letters 9* (1979), pages 31-34.

[25] S. K. Park, K. W. Miller. "Random Number Generators: Good Ones are Hard to Find". *Communications of the ACM* (1988), Volume 31, Number 10, pages 1192-1201.

[26] D. A. Plaisted, J. Hong. "A Heuristic Triangulation Algorithm". *Journal of Algorithms 8* (1987), pages 405-437.

[27] F. P. Preparata, M. I. Shamos. *Computational Geometry: An Introduction.* Springer-Verlag (1985).

[28] W. Rytter. "On Efficient Parallel Computations for Some Dynamic Programming Problems". *Theoretical Computer Science 59* (1988), pages 297-307.

[29] W. D. Smith. "Studies in Computational Geometry Motivated by Mesh Generation". Ph.D. Thesis, Princeton University (1989).

[30] T.-S. Tan. "Optimal Two-Dimensional Triangulations". Department of Computer Science, University of Illinois at Urbana-Champaign (1993).

[31] C. Wang, L. Schubert. "An Optimal Algorithm for Constructing the Delaunay Triangulation of a Set of Line Segments". *Proceedings of the Third Annual ACM Symposium on Computational Geometry* (1987), pages 223-232.