Utrecht University

Master's Thesis

Impossible Geometry: Advancements in the Field of Counting Triangulations of Planar Point Sets

Author:
Jelle Schukken

Supervisors:
dr. Tillmann Miltzow
Ivor van der Hoog

August 19, 2020

Utrecht University

Abstract

Given a set of n points P in the plane, a triangulation of P is a maximal set of non-crossing edges between points in P. Counting the number of triangulations for a point set is a well known problem in the field of computational geometry. In this paper we present an overview of the previous 20 years of publications in the field of counting triangulations. We present four papers in the field that each represent significant advancements or interesting developments in the field. Additionally, we provide an implementation of the most recent and complex algorithm we present.

1 Introduction

A triangulation of a planar point set P is a maximal set of non-crossing edges between points of P [22]. See Figure 1 for an example.

Triangulations have a wide range of applications. For example, in the field of computer graphics, triangulations are used for the purpose of rendering because triangles allow for easy to render components, easier collision detection between components, exact definition of objects using only ordered points, and other advantages for graphical applications [25].

While counting the total number of possible triangulations of a point set is strongly related to the triangulations themselves, its applications are very different. The total number of triangulations can be used to compute upper and lower bounds on the total number of other geometric objects [27]. For example the number of plane graphs of a set with n points is at least $2^{3n/2}$ times the number of triangulations and at most 2^{3n} times the number of triangulations [28]. Similarly, the number of plane perfect matchings of a set with n points is $O(1.1067^n)$ times the number of triangulations [6]. Another possible application is that some approaches to counting triangulations can be used to generate a random triangulation from a uniform distribution of all triangulations. This, in turn, can be used to better test performance of algorithms requiring a triangulation as input or in randomized algorithms focusing on minimizing average running time [15]. Additionally, the total number of triangulations of a point set is of particular interest in the academic field of computational geometry. Many papers have been written describing upper and lower bounds of the number of triangulations of a planar point set and related problems [30, 7, 2, 29, 31, 4, 19]. Computing triangulations tends to have more practical applications, while counting triangulations has applications more geared towards research and theory.

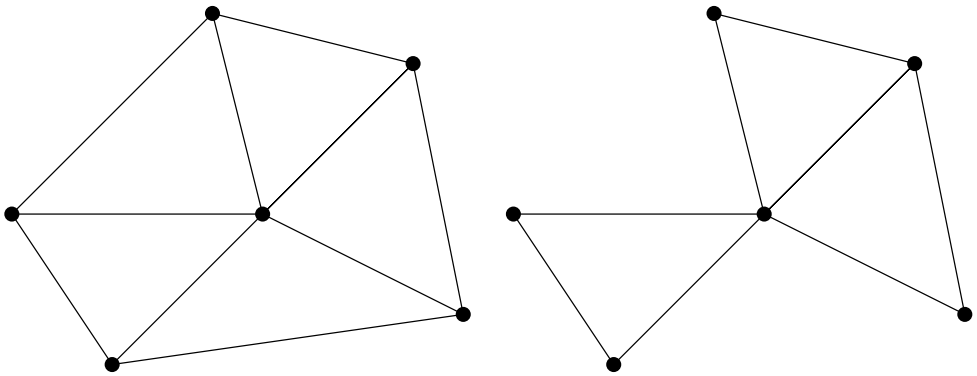


Figure 1: (a) A triangulation of the given point set. (b) While the given point set is subdivided into triangles, the set of edges is not maximal. This is not a triangulation of the point set.

Triangulations can be computed relatively quickly. Theory suggests that an arbitrary triangulation of a polygon can be computed in linear time [14]. Counting the total number of unique triangulations for any given point set is a more difficult problem. In fact, the similar problem of counting the triangulations of a polygon with holes, has been proven to be #P-complete [16]. While both constructing a triangulation and counting the total number of triangulations are related to triangulations, they are distinct problems. Counting triangulations does not explicitly require the construction of any triangulations. See Figure 2 for an example of all distinct triangulations of a regular pentagon.

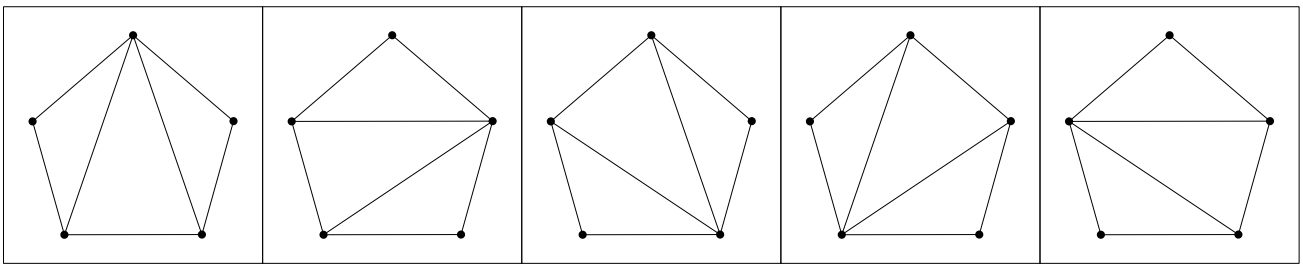


Figure 2: All five triangulations of a regular pentagon

In this paper we present a description and brief analysis of four different algorithms that count the number of distinct triangulations of a planar point set. While we focus on arbitrary planar point sets, algorithms for more specialized counting triangulation problems also exist [10, 13]. Each of the four algorithms we present takes a unique approach to counting triangulations and, for the most part, represents significant advances in the field. We present these papers for a number of reasons:

1. In presenting the papers we provide a practical overview of the field of counting triangulations.
2. Each presented algorithm is a novel approach to counting triangulations.
3. The presented papers provide a historical overview of the field. Each paper is presented in chronological order allowing for an interesting perspective on how the field has changed and advanced over time.
4. Finally, we give a different perspective and explanation for each presented paper in the hopes that we can deepen the readers understanding of the original papers.

The first paper we present is Avis and Fukuda's "Reverse search for enumeration"[8]. In their paper, Avis and Fukuda present a flexible algorithm that can solve many different geometric enumeration problems. We focus on their algorithm specifically applied to counting triangulations. This paper is an excellent introduction to the field; it is an early publication in the field and its basic concept is simple. While the concept is simple, the details of the algorithm are more complex.

We present an extended abstract titled "A Simple and Less Slow Method for Counting Triangulations and Related Problems" by Ray and Seidel as our second paper [26]. This paper is presented due to its distinct approach to counting triangulations.

The third paper, "A Simple Aggregative Algorithm for Counting Triangulations of Planar Point Sets and Related Problems" by Alvarez and Seidel, is a significant advancement over Avis and Fukuda's algorithm [6]. Alvarez and Seidel's algorithm was the first algorithm that could count the number of triangulations with a running time asymptotically faster than the order of the minimum number of triangulations for any planar point set. Their paper won the SOCG best paper in 2013 for this advancement. It has also proven influential with follow up papers generalizing their technique for other crossing-free geometric graphs [32].

Finally we present Marx and Miltzow's paper "Peeling and Nibbling the cactus: Subexponential-Time Algorithms for Counting Triangulations and Related Problems." [22]. Their algorithm is the first to have subexponential running time. Since Marx and Miltzow's algorithm is particularly complex, we present their algorithm implemented as a graphical application as a part of this thesis. The purpose of the application is to help the user understand Marx and Miltzow's algorithm.

While the papers we present in this paper were chosen to give an overview of the approaches taken to count triangulations, there are still other approaches we do not discuss [5, 3, 1, 23, 12].

In the remainder of this paper, the point set P consists of n points, $(p_1, p_2, \ldots, p_n)$, on the plane. Each point p_i has *order label* i.

2 General Preliminaries

In this section we introduce any concepts that are not specific to one algorithm.

The *convex hull* of P is the smallest convex area containing all points of P. Note that the convex hull is a simple polygon whose points are in P and every triangulation of P must contain every edge of the convex hull.

Let Δ(a,b,c) be a triangle formed by the points a, b, and c.

A *circumcircle* of a triangle $\Delta(a, b, c)$ is a circle with the points of the triangle, a, b, and c, lying on the boundary.

A *graph* G is a pair (V, E) where V is a set of points and E is a set of edges between points in V. A *directed acyclic graph* or DAG is a graph where edges are directed and no cycles exist.

A graph $G(V, E)$ with all points in V on a plane naturally partitions the plane into faces, segments, and a set of points. The *outer face* is the unique unbounded face of such a partition.

Some of the presented algorithms use a *dynamic programming* approach. Dynamic programming was introduced by mathematician Richard Bellman in 1954 as a technique to optimally solve certain problems [11]. The idea of the technique is to break complex problems into smaller ones, then recursively solve the smaller problems until the problems become so small as to be trivially computed. In recursively breaking the problem it is likely that the same exact sub-problem is repeatedly encountered. To avoid solving the same problems numerous times a database of sub-problems is kept. Whenever a sub-problem is encountered the database is checked, if that sub-problem is missing from the database it is solved and then added to the database.

Using this technique, difficult problems can be solved quickly. Additionally, if bounds can be placed on the size of the database, then the running time and space complexity of the algorithm can generally be computed easily.

3 Avis and Fukuda

3.1 Background

To give an introduction into algorithms for counting triangulations, we start with a paper by David Avis and Komei Fukuda titled "Reverse search for enumeration". In their paper they propose a diverse set of applications for an existing *reverse search* algorithm [8]. They published their paper in 1996 making it one of the earliest papers in the field of counting triangulations of an arbitrary point set. In this paper, we present and describe Avis and Fukuda's algorithm not only for its age, but also influence on computational geometry as a whole.

The paper describes the reverse search algorithm as a general framework, then goes on to detail how the algorithm can be applied to a number of potential applications. The applications listed in the paper include the explicit enumeration of:

1. All triangulations of a set of n planar points.
2. All cells in a hyperplane arrangement in R^d.
3. All spanning trees of a graph.
4. All Euclidean trees spanning a set of n points in the plane.
5. All connected induced sub-graphs of a graph.
6. All topological orderings of an acyclic graph.

The paper is influential because the algorithmic framework is flexible and easy to apply to a wide variety of problems. The algorithm has several attractive properties:

1. Its running time is proportional to the output size multiplied by a polynomial.
2. Its space complexity is polynomial in size of input.
3. The approach is easy to parallelize.

While describing this approach in the following subsection, we assume that P has no four co-circular points. While the algorithm can still work without this assumption, this assumption removes a number of degenerate cases thereby making the algorithm easier to understand.

3.2 Preliminaries

In this section we define any new concepts for understanding the general approach of Avis and Fukuda's algorithm.

Delaunay triangulations are triangulations of point sets such that the circumcircle of every triangle is empty.

Given a triangulation T, let $p1$, $p2$, $p3$, and $p4$ be four points in T. Let points $p1$, $p2$, and $p3$ form a triangle t_1 in T and let points $p2$, $p3$, and $p4$ form a triangle t_2 in T. The edge from $p2$ to $p3$ can be removed and an edge from $p1$ to $p4$ can be added. This process is called a *flip*. A flip is not always possible; see Figure 4 for an example of an impossible flip. If a flip f of an edge e results in e no longer violating the criteria for a delaunay triangulation then f is considered a *delaunay flip*. See Figure 3 for an example of a delaunay flip. With the assumption that no four points are co-linear, every point set has a single unique delaunay triangulation [9]. This is a non-trivial observation that we use to avoid complex degenerate cases resulting from multiple delaunay triangulations.

3.3 Concept

In order to enumerate all triangulations, Avis and Fukuda propose an approach that explicitly constructs every triangulation. With such an approach the challenge lies in only constructing each triangulation exactly once. Constructing the same triangulation multiple times results in unnecessary computation as triangulations constructed multiple times are not used and therefore wasted. Additionally, in a naive implementation, a list of triangulations would need to be maintained in order to check if a triangulation is new or if it had already been counted. Such a list would result in very large space complexity. Avis and Fukuda propose their implementation of the reverse search algorithm to deal with these problems.

The core idea of reverse search algorithm is straightforward. It can be explained by making an analogue to local search. Local search is an optimization technique where one starts with an arbitrary valid solution to a problem

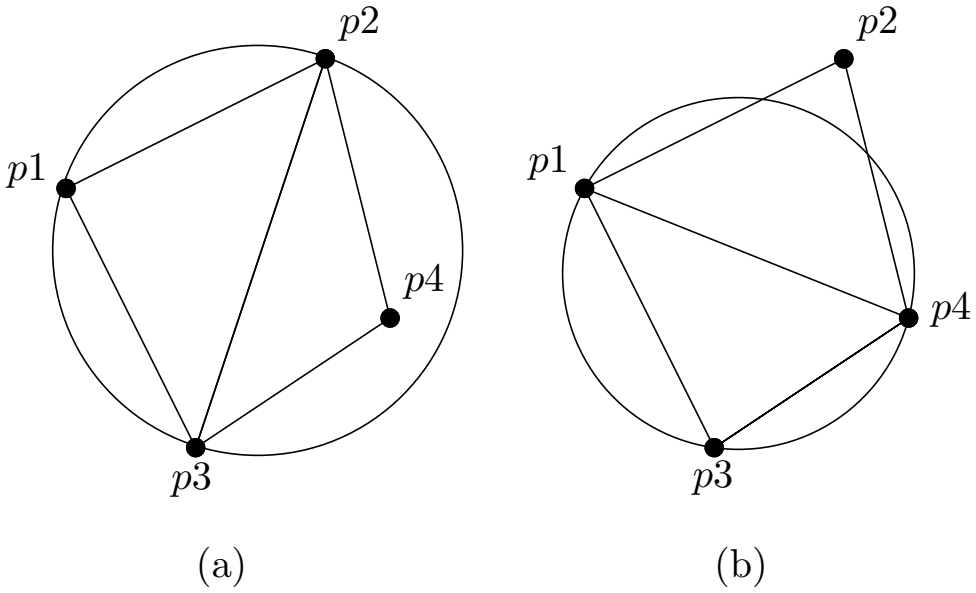


Figure 3: (a) A triangulation of 4 points where $p4$ is inside the circumcircle of the remaining points. (b) The same point set as (a) after a delaunay flip. It is now a delaunay triangulation.

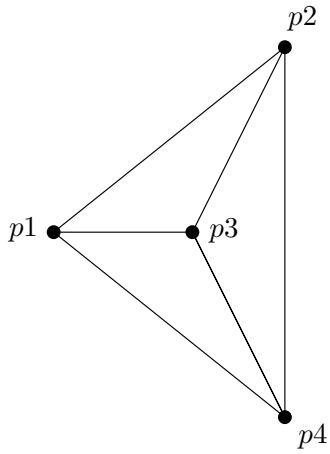


Figure 4: In this example we consider the quadrilateral formed by $p1$, $p2$, $p3$, and $p4$. Flipping edge $(p1, p3)$ would remove edge $(p1, p3)$ and introduce an edge $(p2, p4)$. The resulting graph would not be a triangulation hence edge $(p1, p3)$ is considered impossible to flip.

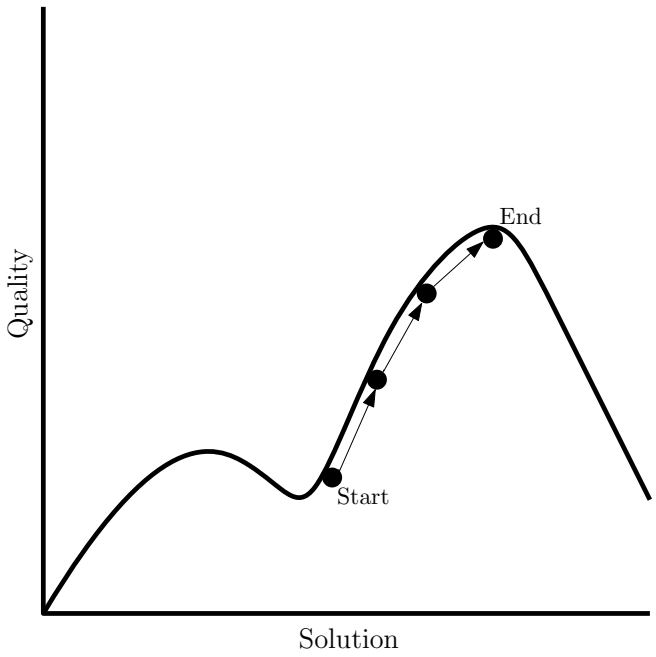


Figure 5: An example of local search. Here the x-axis represents a solution and the y-axis represents the quality of the solution. Each Solution points to a nearby superior solution until none exist and a local optimum is returned.

and iteratively improves the solution by selecting a similar but better solution until no more improvements are possible. Important components of local search are the *initial solution*, some *evaluation function*, and some *neighborhood function*. The initial solution is any arbitrary valid solution. The evaluation function is a function that given a solution determines its *quality*. This can be a value or it can be a *comparison function*; a function that takes two solution and returns the one that is better. Lastly, the neighborhood function, given a solution, returns a list of all similar solutions. For example, say one wanted to compute the optimal ratio of flour and milk for perfect pancakes. The initial solution could be 50% milk and 50% flour, the evaluation function takes two pancakes and returns the one that tastes better, and finally the neighborhood of a ratio is that ratio with 10% difference i.e. 55% milk and 45% flour or 45% milk and 55% flour. In this example, the algorithm adjusts the ratio 10% at a time until no ratio in the neighborhood provides better pancakes. See Figure 5 for a visualization of local search. Reverse search does the same procedure in reverse. It starts at the optimal solution and iteratively searches neighbors until all other solutions are found.

The important components of the reverse search mirror those of local search, namely: an initial solution, a neighborhood function, and an evaluation function. In reverse search the initial solution must be the best solution. The algorithm implicitly creates a tree of solutions. Each solution represents a node in the tree. Edges exist between neighboring solutions. Starting from the initial solution, reverse search performs a depth first search on the tree and counts all solutions it encounters.

Reverse search is applied to problems that have a unique configuration whose evaluation produces a maximum and no configurations that produce local maxima. i.e. every solution has a neighbor that is better except for the single solution that produces a maximum. As a result every other solution has a sequence of neighbors that ends in the single maximum solution henceforth known as the best solution.

Any triangulation can be transformed into a delaunay triangulation simply by flipping every edge that violates a delaunay properties until no such edges exist. This forms the base of reverse search. Since every possible triangulation has at least one sequence of flips that transforms it into a delaunay triangulation [18]. The delaunay triangulation can be transformed back into any triangulation simply by reversing that sequence of flips.

In applying reverse search to counting triangulations of P a neighborhood function, a comparison function, and an initial solution must be proposed. Avis and Fukuda define the initial solution as the delaunay triangulation, $D(P)$, the evaluation function for a triangulation T as the number of flips necessary to transform T into $D(P)$, and the neighborhood of a triangulation T as all triangulations that can result from a single flip of an edge of T.

In this way, the algorithm starts at the delaunay triangulation then spreads across the set of all triangulations until every triangulation has been counted.

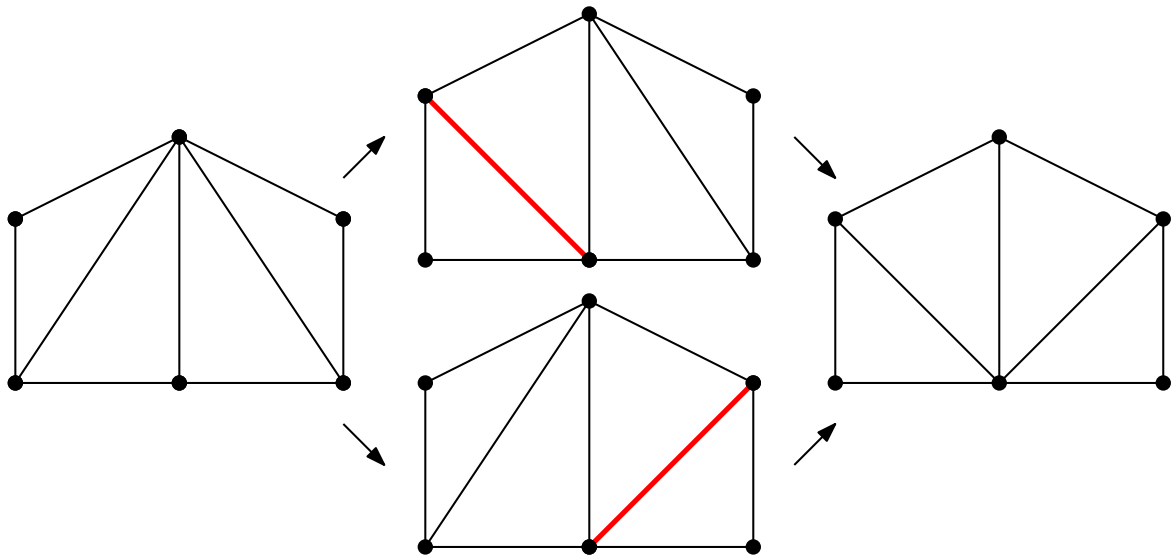


Figure 6: An example where the triangulation on the left has two distinct sequences of flips to reach the delaunay triangulation (on the right). The red edge is the edge flipped in that step.

This, however, introduces a problem. Since one triangulation may have multiple sequences of flips, the same triangulation could be counted multiple times. See Figure 6 for an example. To avoid this issue, Avis and Fukuda apply addition constraints to flip sequences to ensure that each triangulation has exactly one valid sequence of flips to reach the delaunay triangulation.

3.4 Algorithm

The reverse search algorithm explores every valid sequence of flips. The trick is in defining what exactly is a valid sequence of flips. If a triangulation T of P can be transformed into the delaunay triangulation, $D(P)$, via two distinct sequences of flips, then T would be considered twice. To ensure that each triangulation considered is unique, the algorithm defines the set of valid sequences of flips such that each triangulation has exactly one valid sequence of flips to reach the $D(P)$.

There are a number of ways in which reverse search can consider all possible triangulations in a unique order. One approach creates a tree where every node is a triangulation and each node has a child for every flippable edge (following a strict ordering). Another approach creates a binary tree where every leaf is a unique triangulation. The first approach is detailed in the original paper while the second approach is explained in more detail below.

Like the first approach, the second approach to reverse search for triangulations starts from a delaunay triangulation of P, $D(P)$. Then it orders all edges of $D(P)$, that are not a part of the convex hull, lexicographically. The algorithm then creates a tree graph $G(V, E)$. In G, each node $r \in V$ contains a set of edges E_v such that those edges form a triangulation of P. We denote the edges of a node r forming a triangulation T as "r represents T". Each edge can be marked as *fixed*. Edges marked as fixed cannot be flipped. We call the first edge in lexicographical order that can be flipped in a node r edge $e_f(r)$. The root node r of G contains the set of edges E_r that form $D(P)$. Each node, except the leaves, has at most two children, one child where the e_f is flipped, $r_{flip}(e_f(r))$, and one child where $e_f(r)$ is not flipped, $r_{fix}(e_f(r))$. The edge $e_f(r)$ is marked "fixed" in both children. Aside from $e_f(r)$ and its marking, children are identical to their parents. If no edges of a node can be flipped, either because they are fixed or because such a flip is impossible, that node is a leaf node. The total number of leaves of this tree is equal to the total number of triangulations.

In G, the path from any leaf triangulation T to the root represents the exact sequence of flips to transform T into the delaunay triangulation. Let L(r) be the number of leaf nodes of the tree rooted at node r and let $r_{D(p)}$ be the root node of G.

Formally, the number of triangulations of P is:

$$T(P) = L(r_{D(P)})$$

$$L(r) = 1 \iff e_f(r) \text{ does not exist}$$

$$L(r) = L(r_{flip}(e_f(r))) + L(r_{fix}(e_f(r))) \iff e_f(r) \text{ exists}$$

If an edge is impossible to flip it may become possible to flip after other edges have been flipped. Since this is the case, the lexicographic ordering of edges in a sequence may not be monotonically increasing.

Formally, a sequence of flips $q = (f_1, f_2, \ldots, f_k)$ for a triangulation T is *valid* if f_i is the the first edge in lexicographical order that can be flipped after flips f_1 - f_{i-1} have been performed regardless of whether f_i comes before or after f_{i-1} in lexicographic ordering.

If edges can be flipped in arbitrary order then there can exist multiple sequences of flips from one triangulation T to $D(P)$. In this case, the algorithm would count T more then once.

Lemma 1 *A triangulation T of P has exactly one valid sequence of flips to transform T into $D(P)$*

Proof: Take an arbitrary triangulation T with at least two distinct sequences of flips, S_1 and S_2, resulting in $D(P)$. Take edge e_1 and e_2 as the first edges in S_1 and S_2 (respectively) that are not the same. Both e_1 an e_2 must be flippable and the be the lowest in lexicographic order among all flippable edges. Since $e_1 \leq e_2$ and $e_2 \leq e_1$ where $\leq$ is with respect to lexicographic ordering, and no two edges can have the same position in the ordering, $e_1 = e_2$. This is a contradiction, therefore there must exist at most one sequence of flips from a triangulation to the delaunay triangulation.

The unique sequence of flips for a triangulation T is represented in G by the path from the leaf node representing T to the root node representing $D(P)$. In this way, once all paths have been explored, all triangulations have been counted exactly once.

3.5 Analysis

From Euler's formula the number of edges in a connected simple planar graph with n points is at most $3n - 6$ [17].

Hence, in each node at most $O(n)$ edges need to checked if they can be flipped. Flipping or checking an edge is a simple geometric operation on a constant number of points. It can be done $O(1)$ time. As a result, the running time per node is then $O(n)$.

Since the tree G is a binary tree with unique triangulations as leaves, there can be at most twice the number of nodes in the tree as there are triangulations. Since it takes $O(n)$ time for each node, the total running time of the algorithm is then $O(n \cdot Tr)$ where Tr is the total number of triangulations. The current best upper bound on Tr is 30^n [30]. Hence the order of the running time is bounded by $O(n \cdot 30^n)$

The space complexity is $O(n)$. The intuition here is that G is never explicitly constructed. Instead, it is traversed in a depth first manner one node at a time. As a result, only one node of the tree needs to be in memory at any time. A node only consists of all points in the point set, all edges in the triangulations, and a binary flag for each edge e marking whether or not e is fixed. Hence the space complexity of a single node is $O(n)$. Since only a single node is stored at a time, the total space complexity of the algorithm is also $O(n)$.

The algorithm is easy to parallelize. Each branch can be computed entirely independently since they are branches in a tree and are, therefore, completely independent of other branches. Parallelizing the algorithm can be done by assigning the sub-trees of each node to be computed to an idle thread. If no idle threads are available compute the sub-trees on the current thread. In this way the work is easily distributed among multiple parallel threads.

4 Ray and Seidel

4.1 Background

Saurabh Ray and Raimund Seidel proposed an algorithm in an extended abstract published in 2005 [26]. Titled "A Simple and Less Slow Method for Counting Triangulations and Related Problems", it is not a full paper and as a result is missing some of the in-depth analysis standard among published papers. On the other hand, the presented algorithm takes a distinct approach and is therefore interesting to consider alongside the other algorithms presented in this paper.

The algorithm they proposed is an extension of a simpler algorithm that counts the total number of triangulations in a polygon [15]. Their extension allows it to count the total number of triangulations in a polygon with internal points. This can be easily used count the number of triangulations for a planar point set simply by taking the convex hull of the point set as the polygon and all points not on the hull as the interior points. For an instance of a polygon with internal points S, we denote the boundary polygon as $bp(S)$ and the internal points as $ip(S)$.

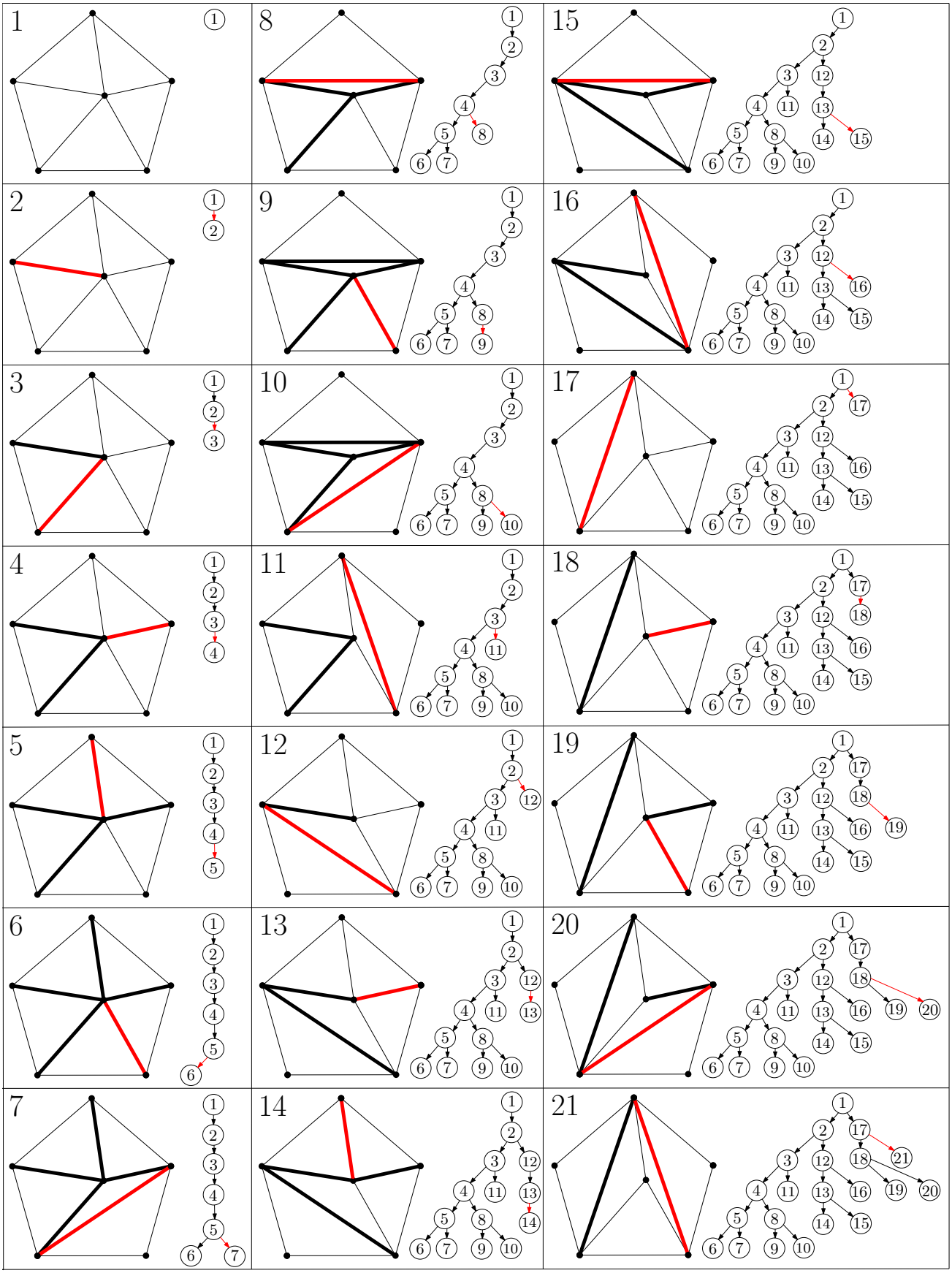


Figure 7: An example of the reverse search algorithm. The tree on the right of each step is G while the triangulation on the left is the triangulation the algorithm is working on at that step. The red edge is the most recently flipped or fixed edge and bold edges are fixed. Each node in G is labeled with its step number. Here the total number of triangulations is 11.

4.2 Preliminaries

A triangulation of a polygon with interior points is slightly different then a triangulation of a point set. Given a polygon with interior points S, a triangulation T of S is any maximal set of edges such that each edge in T crosses no other edge in T, is contained entirely within $bp(S)$, and has endpoints that are either in $ip(S)$ or in $bp(S)$. The number of triangulations of a polygon with interior points S is denoted $T(S)$.

4.3 Concept

The idea of the algorithm proposed by Ray and Seidel follows a dynamic programming approach. The approach is to repeatedly divide the problem into smaller sub-problems until each sub-problem is trivially easy to solve. Then, using the solutions to the sub-problems, solve the more complex problems. Continuing in this way until the original problem is solved. The important components of such an algorithm are: how is a problem split into sub-problems and how can the solutions to the sub-problems be combined to find the solution to the original problem.

Consider a triangulation T of a polygon with interior points S. It is easy to observe that any edge e on $bp(S)$ must be a part of exactly one triangle Δ in T. The total number of triangulations of S can be split into sets based on this observation. The total number of triangulations of S is equal to the sum of the total number of triangulations that contains Δ for each valid triangle Δ. Where a *valid* triangle is a triangle contained entirely within $bp(S)$ that contains no points in $ip(S)$ and is incident to an edge e. We denote the set of valid triangles for an edge e as $valid\text{-}\Delta(e)$.

Lemma 2 *The number of triangulation of S can be expressed as:*

$$T(S) = \sum_{\Delta \in valid\text{-}\Delta(e)} T_\Delta(S)$$

Where $T_\Delta(S)$ is the number of triangulations of S that contain the triangle Δ and e is an arbitrary edge on $bp(S)$.

Proof: Consider an arbitrary triangulation of S, T. Within T, edge e must be a part of a triangle Δ as that is a basic property of a triangulation. Since Δ is a part of a triangulation it must be a empty. Since Δ is empty and incident to e on $bp(S)$, Δ is a valid triangle. Hence T must be counted at least once since the total sum contains the sum of all triangulations that contain Δ. Since e is a part of exactly one triangle within T, T is counted at most once. From this we can conclude each triangulation is counted exactly once.

4.4 Algorithm

Formally, the algorithm described by Ray and Seidel works as follows for a point set P. The initial call to the algorithm has the convex hull of P as the polygon S and all points in P not on the convex hull as interior points of S. The algorithm starts with an arbitrary edge, $e = (a, b)$, of $bp(S)$. It then computes the set of *candidate points*, $C(S, e)$, from all points in S. A candidate point is a point in either $bp(S)$ or $ip(S)$ such that, for any candidate point $c \in C(S, e)$, triangle $\Delta(a, b, c)$ forms an valid triangle.

The number of triangulations of this problem can then be defined, for an arbitrary edge $e = (a, b)$, as the sum across all candidate points, $c \in C(S, e)$, of the number of triangulations of S that contain the triangle $\Delta(a, b, c)$, denoted S_c.

$$T(S) = \sum_{c \in C(S,e)} T(S_c)$$

If $c \in ip(S)$ then the triangulations of S_c can be defined as $T(S_c)$ where $bp(S_c)$ is $bp(S)$ with the edge (a, b) replaced by edges (a, c) and (c, b) and where $ip(S_c)$ is $ip(S)$ with c removed. See Figure 8 for an example.

In the case that c is a point on $bp(S)$ adjacent to e, the triangulations of S_c can be defined as $T(S_c)$ where $bp(S_c)$ is $bp(S)$ with the two edges of $\Delta(a, b, c)$ on $bp(S)$ replaced by the remaining edge of $\Delta(a, b, c)$.

In the case that c is a point on $bp(S)$ not adjacent to the selected edge e, the polygon is split in two polygons, $S_{c\text{-}left}$ and $S_{c\text{-}right}$. In this case the number of triangulations of S can be computed as the product of $T(S_{c\text{-}left})$ and $T(S_{c\text{-}right})$. In this case:

$$T(S) = T(S_{c\text{-}left}) \cdot T(S_{c\text{-}right})$$

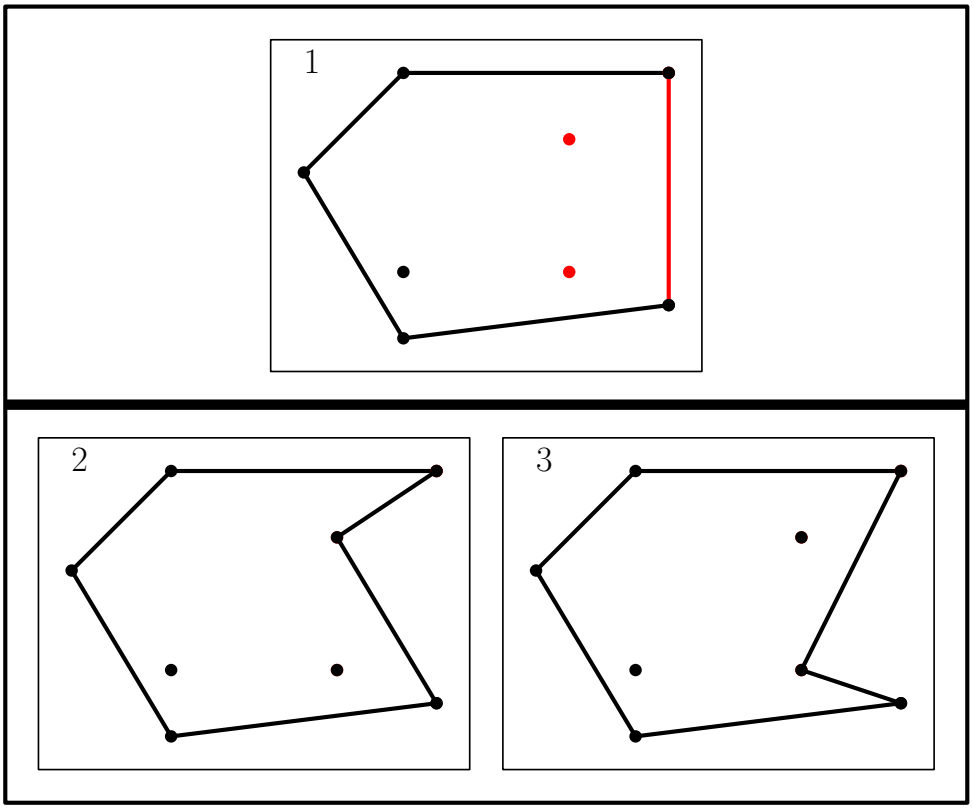


Figure 8: The red edge in sub-problem 1 is the selected edge with the red points as the candidate set. In this case the number of triangulations for sub-problem 1 is the sum of the number of triangulations of sub-problems 2 and 3.

See Figure 9 for an example.

Finally, if S consists of only 3 points with no interior points, $T(S)$ is simply one. In this case, the problem is not divided into sub-problems. This is the base case.

Formally, the number of triangulations of P is defined as follows:

$$T(P) = T(S)$$

$$T(S) = 1 \iff |bp(S)| = 3 \cap \ |ip(S)| = 0$$

$$T(S) = \sum_{c \in C(S,e)} T(S_c) \iff c \in bp(S)$$

$$T(S) = T(S_{c\text{-}left}) \cdot T(S_{c\text{-}right}) \iff c \in ip(S)$$

Since this is a dynamic programming algorithm, once a sub-problem S is solved a pair $(S, T(S))$ is inserted into a database d. Before solving a sub-problem S database d is checked to see if it already contains S. If d does contain S then $T(S)$ in d is used. If it does not, S is solved as described above and inserted into d. See Figure 10 for a full example of the algorithm.

4.5 Analysis

Ray and Seidel do not provide a running time in their abstract. We present our own simple upper bound on the running time below.

In analyzing the runtime of a dynamic programming algorithm one must consider two things: the number of possible sub-problems and the time taken per sub-problem.

Each sub-problem S on P is defined by its bounding polygon, $bp(S)$. The interior points $ip(S)$ is simply all points in P that are contained within $bp(S)$. A polygon is a combination of edges, hence the number of possible

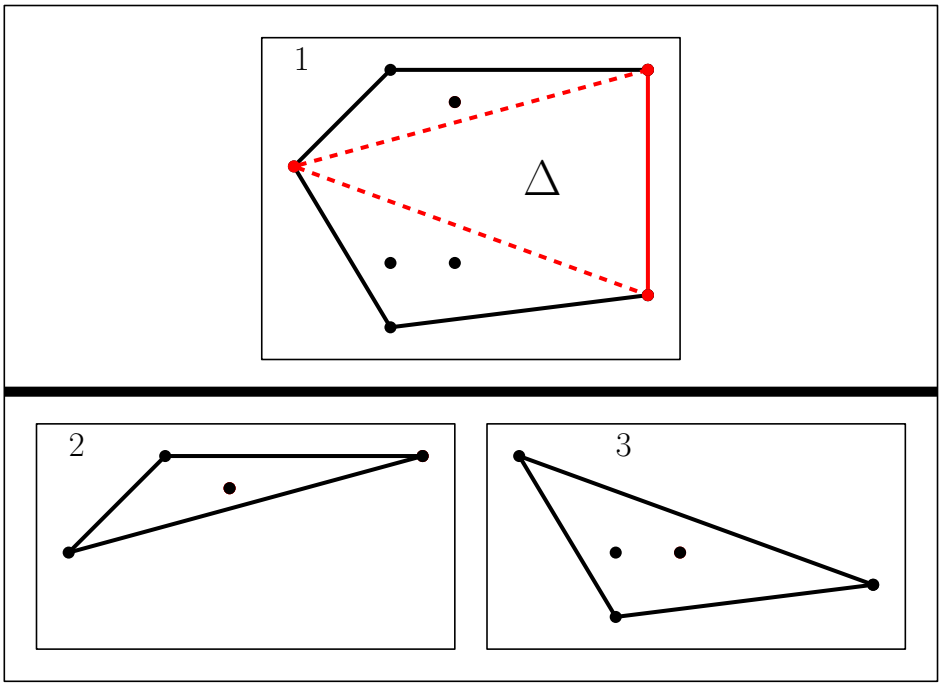


Figure 9: The number of triangulations of sub-problem 1 that contain the red triangle Δ is the product of the number of triangulations of sub-problems 2 and 3.

polygons is bounded by the number of possible combinations of edges. In a set of n points there are n^2 possible edges and therefor 2^{n^2} possible combinations of edges. Hence, the number of possible sub-problems is bounded by 2^{n^2}.

Given that the size of the data base is bounded by 2^{n^2}, the binary search algorithm can be used to perform each search in $O(log(2^{n^2})) = O(n^2) \cdot log(2)) = O(n^2)$ time.

The time taken per sub-problem is based on two components: time taken for database searches and time taken to compute each possible triangle Δ.

There are at most $n-2$ possible valid triangles Δ because each point not on the selected edge can potentially form a valid triangle. The number of database look ups is at most two per triangle (one for each resulting sub-problem) so the order of the time taken for lookups is $O(n^2) \cdot 2 \cdot (n-2) = O(n^3)$.

Each triangle Δ for a sub-problem Smust be checked for validity. A naive method to do this is to check each point in $ip(S)$ if it lies inside Δ and to check each edge of Δ to see if it lies within $bp(S)$. Checking if a point lies within a triangle is a basic geometric operation that can be done in $O(1)$ time. hence checking n points takes $O(n)$ time. Checking if an edge lies within a polygon can be done in time linear to the number of point in the polygon using the Ray Crossings algorithm [24]. In this case, $O(n)$ time. The total running time for computing all Δ is then $(n-2) \cdot (O(n) + O(n)) = O(n^2)$.

Finally, the total bound on the running time of this algorithm is $O(n^2) \cdot O(n^3) \cdot 2^{n^2} = O(2^{n^2})$. In layman's terms : The total number of lookups per sub-problem times the amount of time taken per lookup times the total number of sub-problems.

Ray and Seidel's experimental results, despite this theoretical bound, show that this algorithm computes triangulation counts very quickly. This may indicate that a more strict bound can be calculated or that the average case running time is much less then the worst case bound.

The approach cannot be easily parallelized because there is a shared data structure. The database of sub-problems is shared between all sub-problems.

5 Alvarez and Seidel

5.1 Background

By 2013 the field of counting triangulations had been expanded. Research was published proving tighter bounds, introducing new algorithms, and finding new applications. Then a paper titled "A Simple Aggregative Algorithm

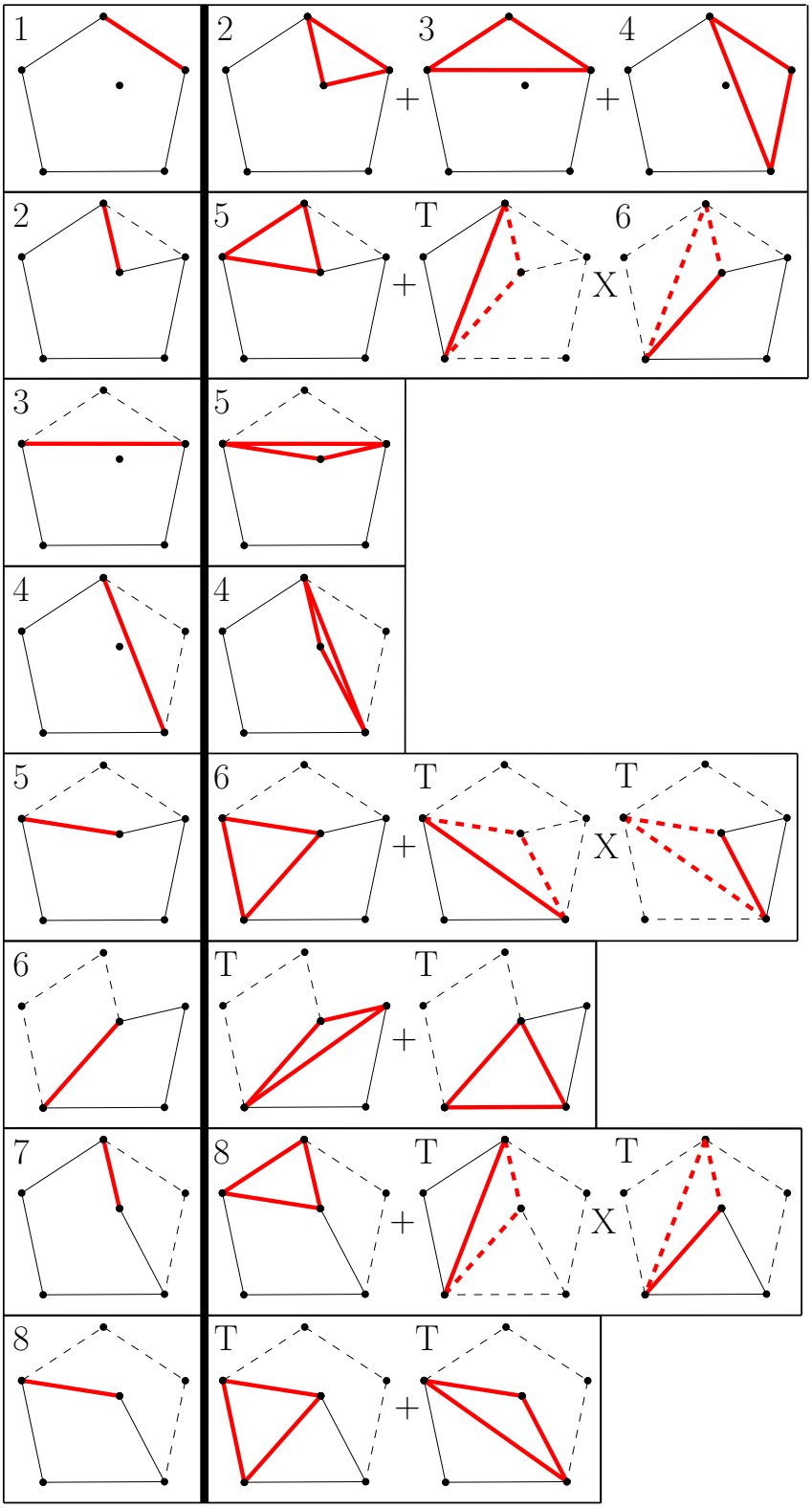


Figure 10: Each row represents a single step. The problem shown on the left of the black bar has a number of triangulations represented by an equation on the right side. Red edges on the left are the selected edge for that sub-problem. On the right, the Δ is highlighted in red. Sub-problems are labeled by the step number that solves that sub-problem. Sub-problems labeled with T are triangles and have only 1 possible triangulation. This example has 11 total triangulations.

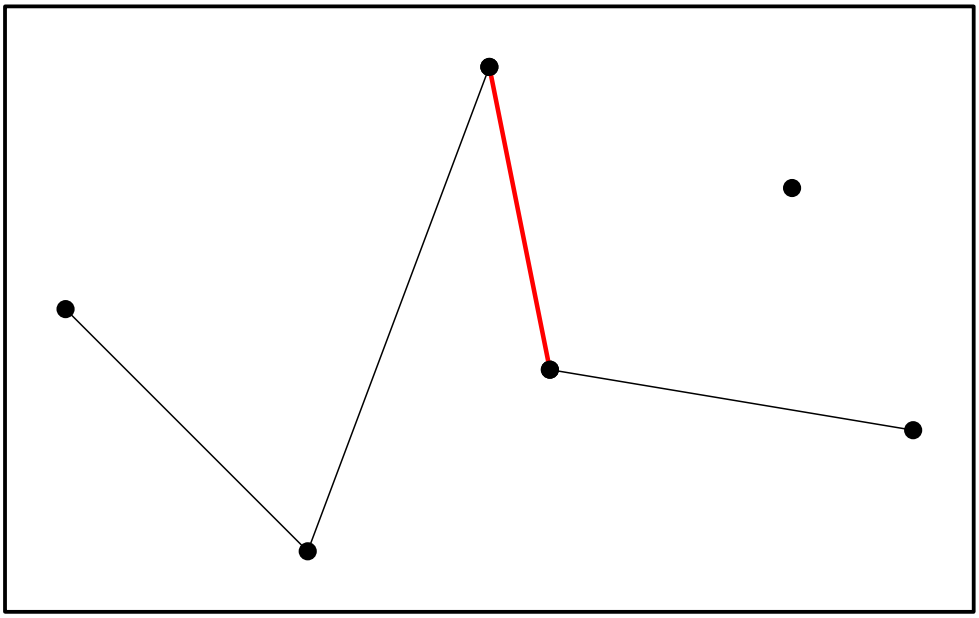


Figure 11: An example of a marked y-monotone chain. The black line is the chain and the bold red edge is the marked edge

for Counting Triangulations of Planar Point Sets and Related Problems" by Victor Alvarez and Raimund Seidel was published [6]. Their paper won SOCG best paper in 2013.

Their paper proposes a more specialized aggregative algorithm for counting triangulations. While the concept is more complex then the reverse search algorithm, Alvarez and Seidel's algorithm is more simple to implement. What makes it relevant is that it is the first paper to introduce an algorithm that counts the number of triangulations such that the order of the running time is asymptotically faster than the order of the minimum number of triangulations for any point set. Specifically, in 2013 the best known lower bound on the number of triangulations was $\Omega(2.43^n)$ and the running time of this algorithm is $O(n^2 \cdot 2^n)$.

For this algorithm P is assumed to be in general position and no two point in P lie on the same vertical line.

5.2 Preliminaries

The algorithm works using a concept of *marked y-monotone chains* through a point set. A *y-monotone chain* $C(P) = (e_0,\ e_1, \ldots,\ e_k)$ is a path consisting only of edges between points in P. All chains in this paper are of edges between points in P hence $C(P)$ is shortened to C. A y-monotone chain starts from the leftmost point and ends at the rightmost point in P and intersects every vertical line at most once. A marked y-monotone chain (C, m) is a y-monotone chain with an integer marker, m, pointing to edge e_m on the chain. For example (C, m) is the chain C with a marker pointing to the m-th edge (counting from left to right). See Figure 11 for an example.

A union of two chains C and $C`$ is a set of edges containing every unique edge in $C$ and $C`$. Every triangulation can be thought of as a union of y-monotone chains. Not all unions of y-monotone chains form a triangulation: e.g. one y-monotone chain might have an edge that crosses that of a different y-monotone chain or the set of non-crossing edges formed by a union of y-monotone chains may not be maximal. We define two y-monotone chains whose union has no crossing edges as compatible chains.

Since we assume no two points in P lie on the same vertical line, the convex hull H of P must have a rightmost and leftmost point, h_0 and h_c respectively. The *upper hull* is the chain of all edges encountered when traversing the convex hull clockwise from h_0 to h_c. The *lower hull* is the chain of all edges encountered when traversing the convex hull counter clockwise from h_0 to h_c. Both the lower and upper hulls start at h_0 and end at h_c. The union of the upper and lower hull forms the original convex hull.

The paper also introduces a concept of *advancing triangles*. An *advancing triangle* can exist in two ways: If, for a chain $C = (e_0,\ e_1, \ldots,\ e_k)$, where e_i is an edge e with index i in C, there exists a point p_j above edge $e_i = (l, h)$ such that p_j lies in the vertical slab bound by the endpoints l and h and the triangle $\Delta(l, h, p_j)$ contains no points then that triangle is an advancing triangle of C. See Figure 12 for an example.

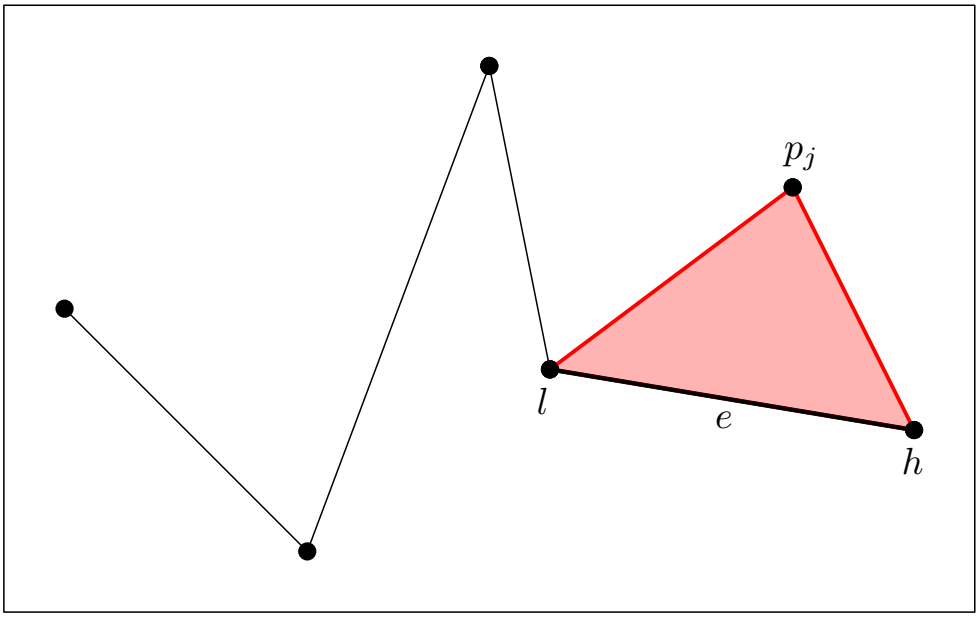


Figure 12: The first type of advancing triangle shown in red. The advancing chain replaces the edge e of the triangle with the two red edges (l, p_j) and (p_j, h).

The other advancing triangle, for a chain $C = (e_0,\ e_1, \ldots,\ e_k)$, exists for a point p_j if two edges $e_i = (l, p_j)$ and $e_{i+1} = (p_j, h)$ share p_j as an end point and p_j lies below the edge (l, h) and the triangle $\Delta(l, h, p_j)$ is empty. In this case the triangle $\Delta(l, h, p_j)$ is the advancing triangle. See Figure 13 for an example.

In the context of Alvarez and Seidel's algorithm we use *constrained advancing triangles*. A constrained advancing triangle can exist in two ways:

Recall the first type of advancing triangle for a chain C in a marked y-monotone chain (C, m) in P. The triangle $\Delta(l, h, p_j)$ is a constrained advancing triangle of (C, m) if $i \geq m$ (recall $e_i = (l, h)$).

Recall the second type of advancing triangle for a chain C in a marked y-monotone chain (C, m) in P. The triangle $\Delta(l, h, p_j)$ is a constrained advancing triangle of (C, m) if $(i + 1) \geq m$ (recall $e_{i+1} = (p_j, h)$).

In the remainder of this paper all advancing triangles, chains, or sweeps(defined later) are constrained advancing triangles, chains, or sweeps unless explicitly stated otherwise.

The *leftmost advancing triangle* for a y-monotone chain C is the advancing triangle of C with the leftmost point. Advancing triangles define *advancing chains*. A chain C together with a point p_j can define an advancing chain in one of two possible ways depending on p_j; one for each type of advancing triangle. In the first case the advancing chain $C`(p_j)$ includes the upper hull of the advancing triangle by replacing the edge $e = (l, h)$ in the chain with two edges, $(l,\ p_j)$ and $(p_j,\ h)$. The edge $(l,\ p_j)$ becomes the new marked edge. In the second case, for a chain $C$ the advancing chain $C`(p_j)$ includes the upper hull of the advancing triangle by replacing the edges e_i and e_{i+1} with the edge $(l,\ h)$. In this case the new edge $(l,\ h)$ becomes the marked edge for the new chain. If the advancing chain is formed from a leftmost advancing triangle, it is a *leftmost advancing chain*.

5.3 Concept

The purpose of Alvarez and Seidel's algorithm is to count all triangulations without explicitly enumerating them. Their strategy is to enumerate features of a triangulation and count the number of ways they could be combined to form valid triangulations.

Consider you have 10 items and you want to know how many ways you could order those 10 items with no restrictions on the ordering. One way you could compute this is to simply take the items and lay them out; keeping track of each ordering you create. Eventually you would count 3628800 possible permutations. Alternately you can note that there are 10 possibilities for the first item, 9 for the second (since one must have been used in the first spot), 8 in the third, and so on. In this way you can quickly compute $10 * 9 * 8 * 7 * \ldots * 1 = 10! = 3628800$ total permutations.

The algorithm proposed by Alvarez and Seidel works in a similar way. It defines a set of possible features and then computes the total number of ways these features can be combined to form a unique triangulation.

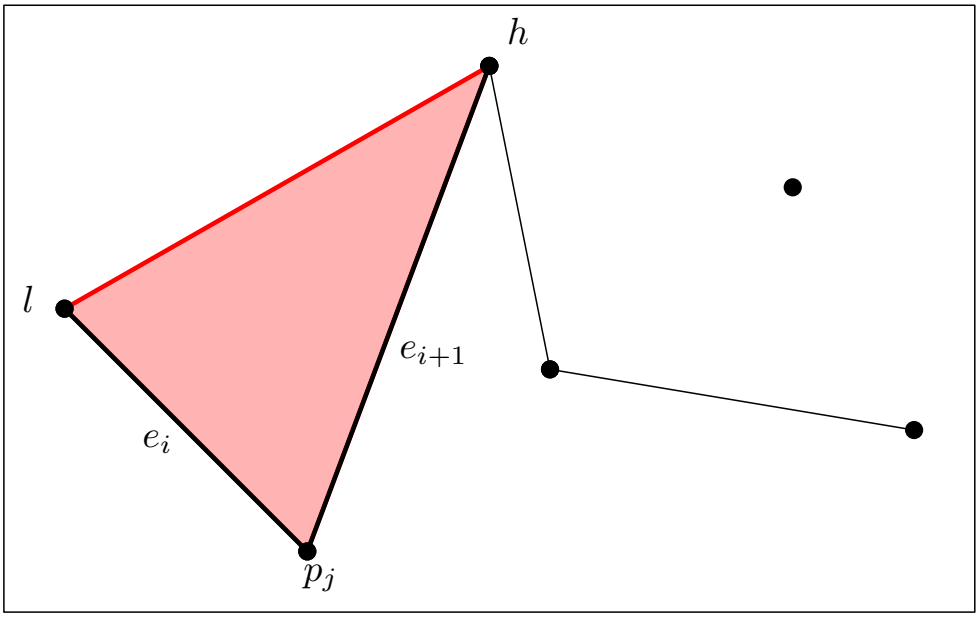


Figure 13: The second type of advancing triangle shown in red. The advancing chain replaces the two edges e_i and e_{i+1} with the red edge (l, h).

To assist in understanding, we define a term: a *feature* of a triangulation T. The purpose of the term is not to provide mathematical clarity nor is it used by the algorithm directly. Instead it is used to give a intuitive understanding of the approach taken by Alvarez and Seidel.

A set of features F of a triangulation T of P is anything such that, for each feature $f \in F$, there exists some function $func(f, T)$ that outputs either true of false. We consider f a feature of T if $func(f, T) = true$. Additionally, every triangulation must have at least one set of features $F_T \subset F$ such that T can be uniquely derived from F_T. For example, the set of all possible edges $E(P)$ between points P is a feature set for T. For any given edge $e \in E(P)$, T either contains e or it does not. The set of all edges in T is a subset of $E(P)$ and it uniquely describes T. No other triangulation of P can be created using only the set of edges of T.

Consider two edges $e, f \in E(P)$ that intersect and do not share endpoints. No set of edges that contain both e and f could possibly represent a triangulation of P. We consider two features *incompatible* if they no triangulation can contain both features. Additionally, consider a triangulation T. The set of edges containing one of each edge in T and the set of edges representing two of each edge in T are different feature sets but both describe the same triangulation T. As a result, naively counting all combinations of features that define a triangulation does not suffice to count all triangulations.

Marx and Miltzow consider each marked y-monotone chain of P as a potential feature of a triangulation of P. Y-monotone chains are a sequence of edges; if a triangulation T contains all edges of a chain C then C is a feature of T.

The challenge Alvarez and Seidel face is to identify, for P, a set of features for all triangulations of P and restrict the way that these features can be combined such that each combination is a unique triangulation.

5.4 Algorithm

It's obvious that all triangulations of a point set P can have the marked y-monotone chain representing the lower hull of P and the chain representing the upper hull of P as features since all triangulations contain the edges of the convex hull.

The authors show that every y-monotone chain within a triangulation other then the chain forming the upper hull must have at least one unconstrained leftmost advancing triangle. They then make the observation that every distinct triangulation has a unique sequence of leftmost advancing y-monotone chains that together define the triangulation. In all such sequences, $(C_0, C_1, \ldots, C_u)$, C_0 is the lower hull, C_u is the upper hull, and each C_i is formed by a leftmost advance from C_{i-1}. Analogue to flipping edges in lexicographic order in Avis and Fukuda's algorithm, markings on edges force an ordering on advancing triangles. The marked edge of an advancing chain C marks the leftmost edge of the advancing triangle Δ used to form C. It is used to ensure

that no subsequent advancing chain has a triangle that intersects C and is more left then the marked edge. This ensures that, in all subsequent chains, Δ was the leftmost advancing triangle for its chain.

Lemma 3 *Each triangulation T is defined by exactly one sequence of advancing chains.*

Proof: Let $J = (j_0,\ j_1, \ldots,\ j_u)$ and $K = (k_1,\ k_2, \ldots,\ k_l)$ be two distinct sequences of advancing chains such that the union of the chains in J and the union of the chains in K are both a triangulation T. Let j_i be the first advancing chain in J different then in K. Let j_i be the advancing chain formed by taking advancing triangle Δ_j from j_{i-1} and let k_i be the advancing chain formed by taking advancing triangle Δ_k from k_{i-1}. Finally, let the left most point of Δ_j be more left then the leftmost point of Δ_k. Since the union of both K and J is T, both K and J must each contain an advancing chain formed using Δ_j and an advancing chain formed using Δ_k. Since all chains before chain i in both J and K are identical, K must have an advancing chain formed using Δ_j, k_l at some point after k_i. This is a contradiction. Since a point of Δ_j is more leftmost and k_l comes after k_i in K, k_l cannot exist as it breaks the ordering enforced by the marked edge. As a result, the sequence J must be unique for T.

From these observations it follows that, to count all triangulations, it suffices to count all unique sequences of advancing chains. Such a unique sequence is known as a *leftmost advancing sweep*. See Figure 14 for an example. Such a sweep represents a set of compatible features that combine to form a unique triangulations. Two features are considered *compatible* if there exists a triangulation that contains both features.

To count the number of unique sweeps Alvarez and Seidel propose the use of a Directed Acyclic Graph (DAG), $G(V, E)(P)$. Referred to as G. Each node in G is a Marked y-monotone chain (C, m). An edge from from a node (C, m) to (C', m') exists in E if and only if (C', m') represents an advancing chain of (C, m). Let $successors(C, m)$ be the set of nodes to which (C, m) has an edge. In this way, a path from a node representing lower hull chain to a node representing the upper hull chain in the graph corresponds to a leftmost advancing sweep. Counting the unique paths defined as such suffices to count the total number of sweeps. It is well known that all paths between two nodes in a DAG can be counted in $O(|V| + |E|)$.

To formally introduce the DAG $G(V, E)(P)$ we start with a source node $S \in V$. The node S stores a chain $(C, 0)$ such that C is the lower hull boundary. Successors of each node are then added. Finally, a sink node T representing the upper hull boundary is added as a successor of every node (C, m), where C is the upper hull boundary. The number of unique paths from node S to node T is then the total number of triangulations for P

The use of a DAG ensures that no marked y-monotone chains are added to G twice. If a marked y-monotone chain (C, m) is encountered multiple times, no new node is created. Instead the node (C, m) will simply be the successor of multiple nodes.

The edges in G for P represent compatibility of features. Every feature is compatible with all its parents. Since successors of a node (C, m) in G are defined such that they are leftmost advancing chains of (C, m), every unique path through G from the lower hull, S, to the upper hull, T, is a leftmost advancing sweep of P.

The total number of paths through G can be computed in linear time with respect to the number of edges and nodes by traversing the nodes in reverse topological order. Let $paths((C, m))$ be the number of paths from a node (C, m) to the sink T. The number of paths from the sink to itself, $paths(T)$, is one, namely the empty path. Each other node (C, m) has a number of paths to T equal to the sum for each $(C', m') \in successor((C, m))$, of the number of paths to T from (C', m').

Formally, the triangulations of P can be defined as follows:

$$T(S) = paths(S)$$

$$paths(T) = 1$$

$$paths((C, m)) = \sum_{(C',m') \in successors((C,m))} paths((C', m')) \iff (C, m) \neq T$$

5.5 Analysis

Each y-monotone chain $C = (e_0,\ e_1, \ldots,\ e_k)$ can be uniquely defined by a sequence of points in P, $(p0, p1, \ldots, p(k + 1))$, such that $e_i = (pi, p(i + 1))$. Since C is a y-monotone chain pi is always more left then $p(i + 1)$. Consider all points ordered left to right, a y-monotone chain can be uniquely defined by some subset of these points in that order. Therefore each point can be thought of as either in the chain or not in the chain, leading to 2^n possible chains. For each chain there can be at most $n - 1$ markings since a chain cannot have more then $n - 1$ edges. In this way there are $O(n \cdot 2^n)$ total nodes.

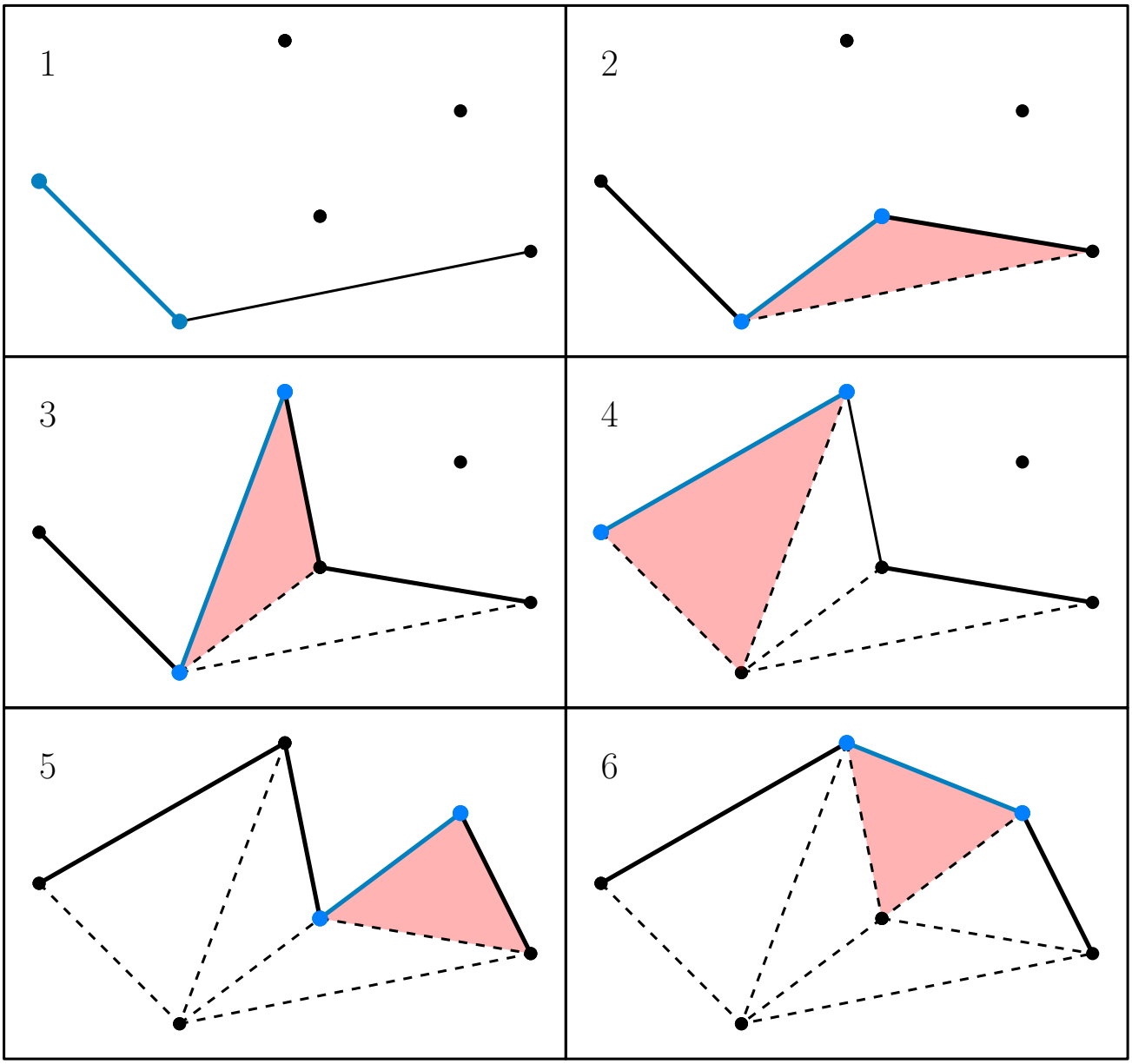


Figure 14: An example of a sweep from the lower convex hull boundary to upper convex hull boundary forming a triangulation. In each step the newest chain is shown in bold and previous chains are shown with dashed edges. The marked edge is the edge in blue and the advancing triangles are shown in red.

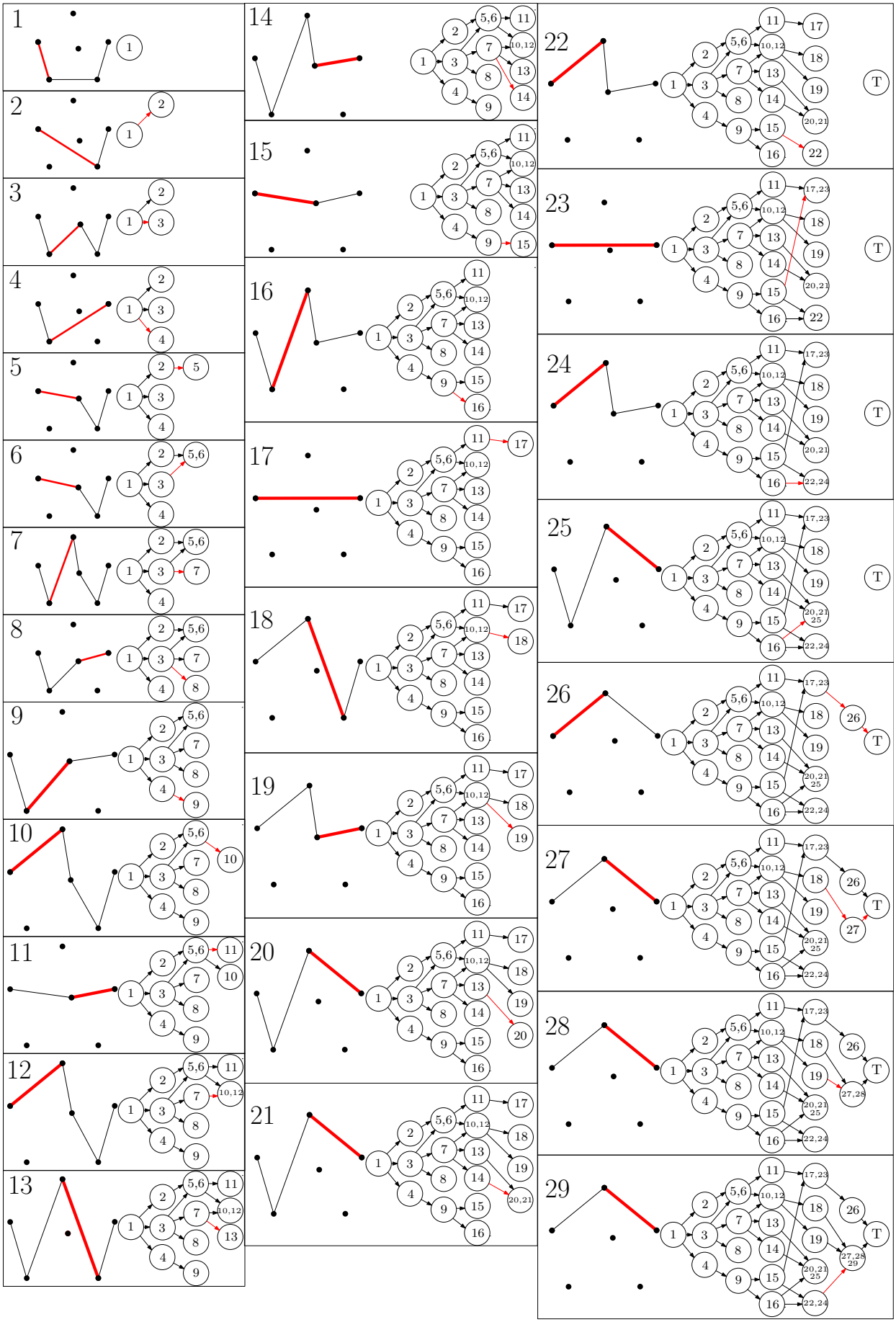


Figure 15: A complete example of Alvarez and Seidel's algorithm. In each step current DAG and the newly added chain are shown. The red edge on the chain is the marked edge. Nodes are labeled with the step(s) that generated that node. This example has 11 triangulations.

A successor (C', l) for a node (C, m) is a leftmost advancing chain formed from C with a point p_j. The leftmost advancing chain C' either contain p_j where as C did not or does not contain p_j while C did. Since each successor either contains one additional point or excludes a point from this chain, there are at most n successors for any node. To clarify, in the worst case there is one successor for each point in the point set P.

Alvarez and Seidel claim that the successors of a node can be computed in $O(n)$ time after polynomial preprocessing.

In this way the running time of the algorithm is $O(n)$ time for each node or $O(n^2 \cdot 2^n)$ total running time.

Since edges can be computed in $O(n)$ time after polynomial preprocessing, they don't need to be stored. A node contains only its marked y-monotone chain C, m hence it uses only linear space. The space complexity of this algorithm is then simply equal to the number of nodes to be stored multiplied by space required for each node: $O(n \cdot 2^n) \cdot O(n) = O(n^2 \cdot 2^n)$. Alvarez and Seidel make the assumption that their counters which can take on exponentially large values, can be stored in a single word, and can be added in constant time. Therefor the resulting space complexity is $O(n \cdot 2^n)$

Since the space complexity is quite large, the paper recommends storing at most two *layers* of the DAG at a time. Layers refer to topological layers of the DAG. i.e. the first layer L_1 is just S, the each subsequent layer l_i is composed of the successors of prior layer, $l_i = \{successors(r) | r \in l_{i-1}\}$. In this way the total number of triangulations can be computed without storing the entire graph at one time and without changing the order of the running time. While storing only two layers at a time does reduce the amount of space used in practice, it does not impact order of the space complexity of the algorithm.

6 Marx and Miltzow

6.1 Background

In 2016 Marx and Miltzow published a paper titled "Peeling and Nibbling the Cactus: Subexponential-Time Algorithms for Counting Triangulations and Related Problems.". In their paper they propose an algorithm capable of counting the number of triangulations of a point set in sub-exponential time. Specifically, their algorithm can count the number of triangulations in $n^{(11+O(1))\cdot\sqrt{n}}$ time. While similar results had been obtained for counting triangulations approximately, this algorithm is the first and, currently, only algorithm capable of counting the exact number of triangulations in sub-exponential time [4]. The paper represents significant advancement over the previous best running time of $O(n^2 \cdot 2^n)$ from Alvarez and Seidel's algorithm.

We present this algorithm last because it is the most recent, the most complex, and the order of its running time is the smallest.

For the purpose of simplicity we assume all points are in general position: no three points are collinear and no four points are cocircular.

6.2 Preliminaries

Marx and Miltzow introduce their own definition of *cactus graphs* that is different then the standard [20]. Their definition for a cactus graph is any graph $G(V, E)$ such that every point in V and edge in E is incident to the outer face induced by G. Unlike in a standard cactus graph, their cactus graphs may have multiple connected components. See Figure 16 for an example.

An *interior component* of a cactus graph G is polygon formed by the set of edges bounding an enclosed face of G. A cactus graph can have 0 or more interior components.

A triangulation T on P naturally defines a unique set of nested cactus graphs on P. The outermost cactus graph of T is a graph of all edges and points in T incident to the outer face. The i'th cactus graph is the graph of all edges and points incident to the outer face after cactus graphs 0 through $(i-1)$ and all edges incident to those graphs have been removed from T.

A *cactus layer* l is a cactus graph with a *layer index*. The layer index of l, *layer-index*(l), is an integer defining the number of cactus graphs in which l is nested in any triangulation. For example, the i'th cactus graph of T is a cactus layer l with layer index i. The cactus layer l could also be a layer of a different triangulation T'; however, it would still need to be the i'th cactus graph of T'.

The size of a cactus layer is the number of points of which it is composed. We refer to the number of cactus layers of a triangulation T as the width of T. Every point in the triangulation T lies on exactly one of it's layers. We say that T *contains* l if layer l is a layer of triangulation T.

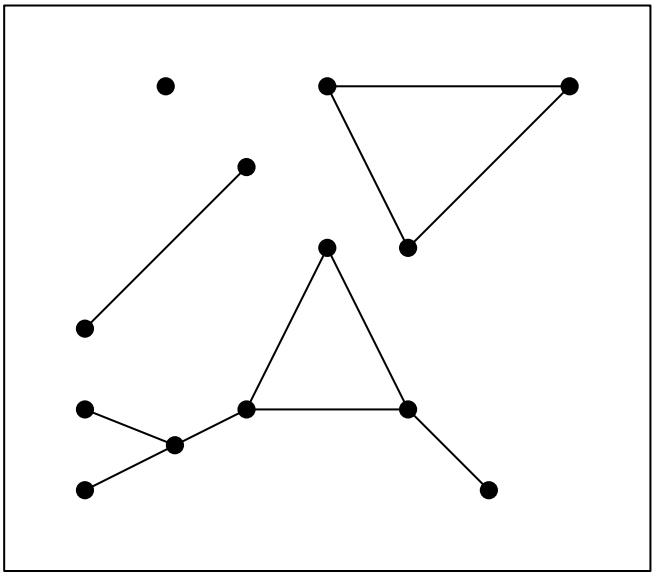


Figure 16: An example of a cactus graph.

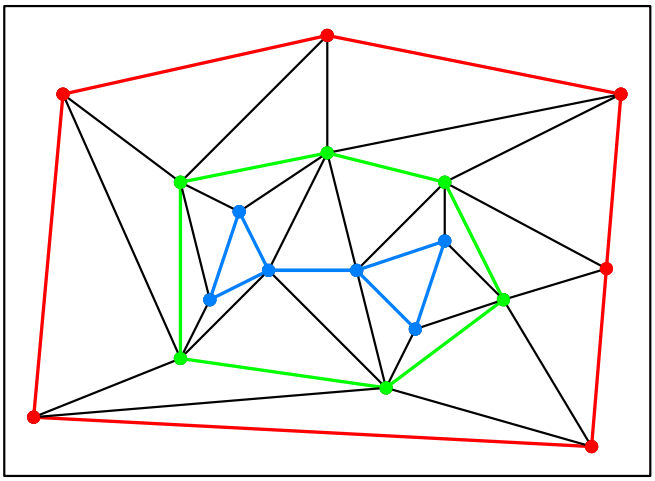


Figure 17: An example of the cactus layers of triangulation. The red edges are the cactus layer with layer index 0, green edges are the cactus layer with layer index 1, and the blue cactus layer has layer index 2.

While similar, the cactus layers of a triangulation T on P and the "onion layers" of P are distinctly different [21]. Onion layer i of a point set P is the convex hull of P after onion layers 0 through $i-1$ have been removed. The onion layers of P are dependent only on P while the cactus layers are dependent on T as well. For example in Figure 17 the blue cactus layer with layer index 2 is not a convex hull of its points and therefor not an onion layer. Onion layers and onion decompositions are not used in Marx and Miltzow's approach.

Marx and Miltzow also define *ring problems*. A ring problem R consists of:

1. *outer layer*: A sequence of points defining a simple polygon. Often the set of edges bounding an interior face of a cactus layer. Denoted *outer-layer*(R).
2. *inner layer*: A cactus layer contained entirely within the outer layer. The inner layer can be empty. Denoted *inner-layer*(R).
3. *interior components*: The set of interior components of R is the set of interior components of *inner-layer*(R). The set of interior components of R is denoted *interior-components*(R).
4. *free region*: The area within the outer layer excluding bounded faces of the inner layer.
5. *restricted regions*: The area outside the outer layer and the area within the interior components are restricted regions.

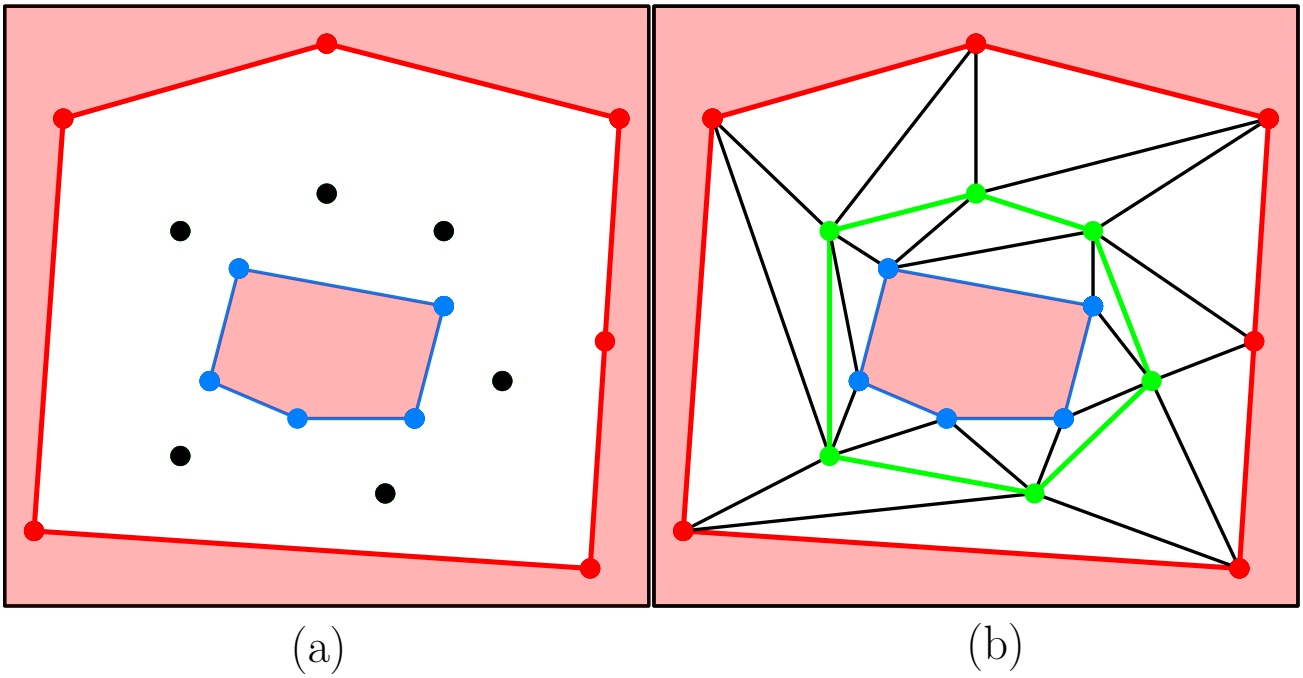


Figure 18: (a) An example of a ring problem. The outer layer is shown in red, the inner layer in blue, and free points in black. (b) A sample triangulation for the ring problem in (a). The red edges, green edges, and blue edges are cactus layers 0, 1, and 2 respectively. In this example, the width is 3. In both images the regions in red are restricted regions.

6. *free points*: A set of points contained within the free region. Free points do not lie on the inner or outer layer. Denoted $free\text{-}points(R)$.
7. *layer indexes*: Both the outer layer and the inner layer have layer indexes.
8. *width*: The difference between the inner layer index and the outer layer index plus one. Denoted $width(R)$

See Figure 18 for an example of a ring problem.

A ring problem R is dependent only on $outer\text{-}layer(R)$, $inner\text{-}layer(R)$, $free\text{-}points(R)$, and the layer indexes of the outer and inner layer.

Two cactus layers l_i and l_j of a point set P, such that l_j is contained entirely within a component of l_i, induce a ring problem R. In this case:

1. The outer layer of R is the polygon bounding the inner face of l_i that contains l_j.
2. The inner layer is l_j.
3. The free points are all point in P that lie in the free region.
4. The layer index of the outer layer is $layer\text{-}index(l_i)$ and the layer index of the inner layer is $layer\text{-}index(l_j)$

The points of R is the set of points contained in the outer layer, the inner layer, and the free points.

For a ring problem R let the triangulations of R be the set of all maximal subsets of non-crossing edges between points of R that are contained within the free region that include all edges of the outer and inner layer of R. For a ring problem R, $T(R)$ is the number of triangulations of R with width equal to $width(R)$.

There is a special case where the inner layer is a layer consisting of no points or edges. We refer to such a layer as an empty layer. In the case where the inner layer of a ring problem R is empty, $T(R)$ is the number of triangulations of R with width less then or equal to $width(R)$ instead of only triangulations of R with width equal to $width(R)$. In this way counting triangulations of any point set can be represented as a ring problem R. To convert P into a ring problem R such that every triangulation of R has a corresponding triangulation of P and vice versa, take the convex hull as the outer layer and an empty cactus layer with layer index $n+1$ as the inner layer. The triangulations of R are the triangulations of P that contain all edges of the convex hull and have at most n cactus layers. Since no triangulation can have more layers then there are points and every triangulation of P includes all edges of the convex hull, the number of triangulations of R is equal to the number of triangulations of P.

6.3 Concept

The fundamental approach of Marx an Miltzow's algorithm can be divided into two primary components described later: the *nibbled ring sub-problem* and the *general layer-unconstrained ring sub-problem*. The number of triangulations of P is denoted $T(P)$. The general layer-unconstrained ring sub-problem is a sub-problem of of the primary problem of solving for $T(P)$. The number of triangulations of a general layer-unconstrained ring sub-problem S is denoted $T(S)$. The nibbled ring sub-problem is a sub-problem of the general layer-unconstrained ring sub-problem. The approach detailed in their paper first takes P and defines a general layer-unconstrained ring problem, S, Such that every triangulation of S has a corresponding triangulation in P and vice versa. Then two dynamic programming algorithms, one for each type of sub-problem, are used to solve for $T(S)$.

Their approach is based on an observation that ring problems with width at most $\sqrt{n}$ can be solved in $O(n^{\sqrt{n}})$ time. They propose one algorithm to divide ring problems with width greater then $\sqrt{n}$ into a set of ring problems with width at most $\sqrt{n}$ and another algorithm to efficiently solve ring problems with width at most $\sqrt{n}$.

Marx and Miltzow propose the *nibbled ring algorithm* to solve ring problems with width at most $\sqrt{n}$ and they propose the *general layer-unconstrained ring algorithm* to solve ring problems with width greater then $\sqrt{n}$.

The nibbled ring algorithm works in a similar way to Ray and Seidel's algorithm. Both algorithms consider all possible triangles incident to a selected edge and split the problem into sub-problems for each such triangle. Additionally, they both use a dynamic programming approach. The nibbled ring algorithm, however, has a significant amount of added complexity. The added complexity is a result of techniques that efficiently impose a specific set of constraints on the triangulations that the algorithm counts. The set of constraints dictates the size of each cactus layer in the counted triangulations.

While the nibbled ring algorithm is fast for ring problems with width at most $\sqrt{n}$, it becomes slow for thicker rings. To account for this Marx and Miltzow use another algorithm to divide ring problems with width greater then $\sqrt{n}$ into ring problems with width at most $\sqrt{n}$ that can then be solved with the nibbled ring algorithm.

Similar to the nibbled ring algorithm, the general layer-unconstrained ring algorithm is also a dynamic programming algorithm.

The base case of this algorithm is simple, if the problem cannot have more then $\sqrt{n}$ layers then it can be solved using the nibbled ring algorithm. The recursive case, however, is more complex.

From the pigeon hole principle it is easy to see that, for any triangulation T of ring problem R with more then $\sqrt{n}$ cactus layers, at least one of the first $\sqrt{n}$ layers must have size at most $\lfloor \frac{free\text{-}points(R)}{\sqrt{n}} \rfloor$. The *outermost small layer* of T, $OST(T)$, is the layer in T with the minimum layer index of layers with size at most $\lfloor \frac{free\text{-}points(R)}{\sqrt{n}} \rfloor$.

Let $layers(P)$ be the set of all possible cactus layers of P. The set of all triangulations of P can be denoted as the sum for all cactus layers l of all triangulations T such that $OST(T) = l$. Formally:

$$T(P) = \sum_{l \in layers(P)} T_l(P)$$

Where $T_l(P)$ is the number of triangulations T such that $OST(T) = l$. Since each triangulation with width greater then or equal to $\sqrt{n}$ has exactly one such l, such a sum contains triangulations with width greater then or equal to $\sqrt{n}$ exactly once.

Intuitively, such a layer l is a perfect separator for splitting a general layer-unconstrained ring sub-problem S into smaller components. The outer component is a ring problem with width at most $\sqrt{n}$ and the inner components, if any, are smaller instances of the general layer-unconstrained ring sub-problem. See Figure 19 for an example. The outer component is constrained.

Since l is defined to be the outermost small layer, all layers j with $layer\text{-}index(j) < layer\text{-}index(l)$ must have size greater then $\lfloor \frac{free-points(S)}{\sqrt{n}} \rfloor$. If l has any interior components then each of those components define a smaller general layer-unconstrained ring sub-problem. It is possible for l to have zero interior components.

In this way the number of triangulations for a general layer-unconstrained ring sub-problem S on P can, informally, be defined as the sum, for each $l \in small\text{-}layer(P)$ of the product of the number of triangulations of constrained outer ring problem defined by l and each ring problem defined by an interior component of l.

The outer ring problem fits the definition of a nibbled ring problem. Its triangulations can therefore be counted by solving the nibbled ring problem. The inner ring problem fits the definition for the general layer-unconstrained ring problem. Its triangulations can then be counted by solving the general layer-unconstrained ring problem. By dividing problems in this way, Marx and Miltzow's algorithm can count the triangulations of any point set in sub-exponential time.

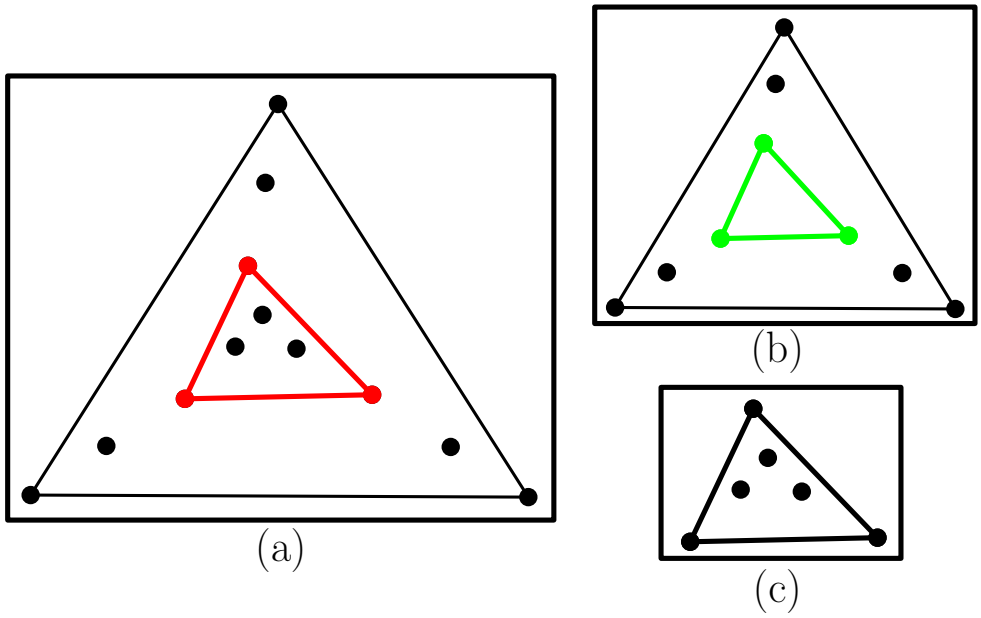


Figure 19: A general layer-unconstrained ring sub-problem where the red cactus layer is the separator layer. The separator layer splits the problem into a nibbled ring sub-problem (b) and a smaller general layer-unconstrained ring sub-problem (c). In (b) the inner layer is shown in green.

6.4 Algorithms

6.4.1 Nibbled Ring Sub-Problem

The nibbled ring algorithm uses a dynamic programming approach to divide and conquer the ring problem. Similar to Ray and Seidel's algorithm, at each step there is a single "selected" edge e on the boundary polygon. The problem is then divided into sub-problems by computing the number of triangulations that contain a certain element dependent on e for every element of that type. While Ray and Seidel's algorithm uses triangles incident to e as the element, the nibbled ring algorithm uses specially constrained paths. The paths are constrained such that every triangulation of P has exactly one such path for an edge e. The paths are designed such that the nibbled ring sub-problem can be split into two nibbled ring sub-problems along the path. This path is called a *separator path*. Separator paths allow the nibbled ring algorithm to enforce constraints on layer sizes. Defining a valid separator path is the primary challenge with this approach.

See Figure 20 for a labeled example of a nibbled ring sub-problem. A nibbled ring sub-problem N has the following components:

1. *boundary polygon*: Denoted *b-polygon*(N), the boundary polygon is a polygon defined by the set of edges incident to the outer face.
2. *outer layer*: Denoted *outer-layer*(N). Unlike in a standard ring problem, the outer layer is a non-crossing polygonal chain of edges. The outer layer is a subset of the boundary polygon.
3. *boundary paths*: Two directed paths, denoted $b1(N)$ and $b2(N)$, ending at different end points of the outer layer. Each path is edge disjoint of the outer layer, a subset of the boundary polygon, and the two paths differ in length by at most 1.
4. *inner layer*: Denoted *inner-layer*(N), the inner layer is a cactus layer. It may be empty. The intersection between *inner-layer*(N) and *b-polygon*(N) must be connected or empty. It is always disjoint from the outer layer. It is edge disjoint from the boundary paths.
5. *interior components*: The interior components of N are the interior components of *inner-layer*(N). The set of interior components of N is denoted *interior-components*(N).
6. *layer indexes*: Both the outer layer and the inner layer have layer indexes. Denoted *outer-index*(N) and *inner-index*(N) respectively.
7. *width*: The difference between the inner layer index and the outer layer index plus one. Denoted *width*(N).
8. *free region*: The area within the boundary polygon that is not within the inner layer.

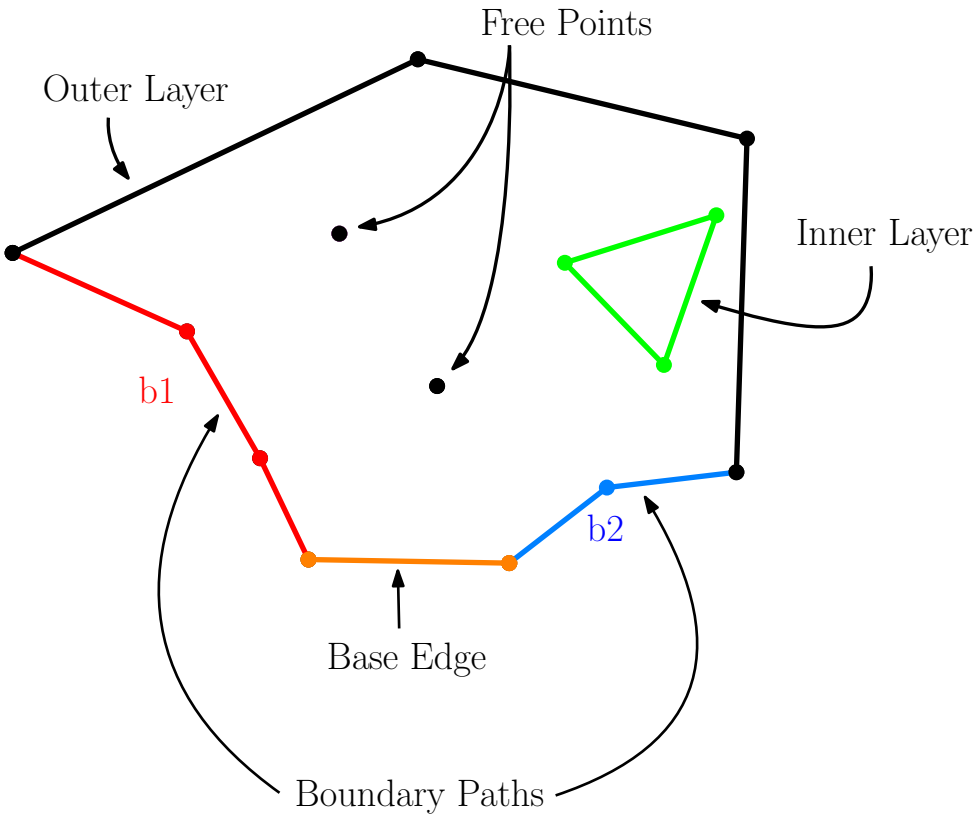


Figure 20: A labeled diagram of a nibbled ring sub-problem.

9. *base edge*: The base edge is an edge of the boundary polygon. It is edge disjoint from the boundary paths and the outer layer except in one special case when the free region is empty. The base edge of N is denoted $base\text{-}edge(N)$.
10. *restricted region*: The are outside the boundary polygon and the area inside the interior components of the inner layer are restricted regions.
11. *free points*: Denoted $free\text{-}points(N)$, the free points are a set of points contained within the free region. A free point cannot be incident to any other component of the nibbled ring problem.

The points of a nibbled ring sub-problem refer to the points of all its components.

A nibbled ring problem N is dependent only on $outer\text{-}layer(N)$, $inner\text{-}layer(N)$, $free\text{-}points(N)$, boundary paths $b1(N)$ and $b2(N)$, $base\text{-}edge(N)$, and the layer indexes. Additionally, for Marx and Miltzow's approach, $free\text{-}points(N)$ is always all point in P that satisfy the constraints on free points.

For a nibbled ring problem N let the triangulations of N be the set of all maximal subsets, S, such that S consists of non-crossing edges that are contained within the free region and span between points of N and S contains all edges in $outer\text{-}layer(N)$, $inner\text{-}layer(N)$, $b1(N)$, $b2(N)$, and $base\text{-}edge(N)$.

Let $T(N)$ be the number of triangulations of N with width equal to $width(N)$.

A ring problem R can easily be converted into a nibbled ring problem N by taking an arbitrary edge of $outer\text{-}layer(R)$, $e = (l, h)$, and defining each component in the following way:

1. $outer\text{-}layer(N)$ is the chain resulting from removing e from $outer\text{-}layer(R)$
2. $base\text{-}edge(N) = e$
3. $b1(N) = (l)$
4. $b2(N) = (h)$
5. $free\text{-}points(N) = free\text{-}points(R)$
6. $inner\text{-}layer(N) = inner\text{-}layer(R)$
7. $outer\text{-}layer(N) = outer\text{-}layer(R)$
8. $inner\text{-}layer(N) = inner\text{-}layer(R)$

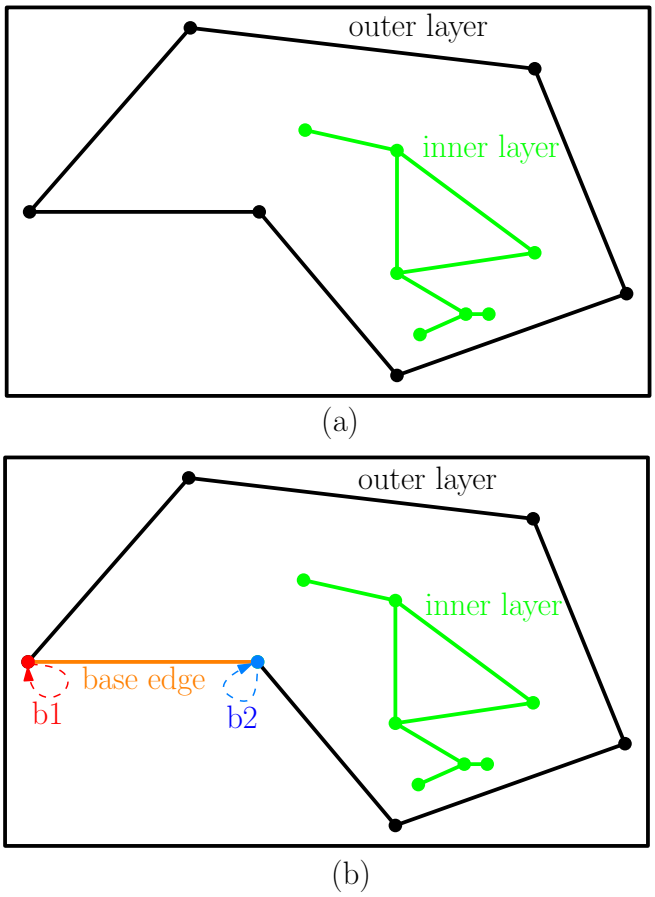


Figure 21: (a) An example ring problem. (b) The nibbled ring problem corresponding to (a).

See Figure 21 for an example.

For the purpose of Marx and Miltzow's algorithm, all nibbled ring sub-problems are *constrained nibbled ring sub-problems* unless explicitly stated otherwise. In addition to the normal components a constrained nibbled ring sub-problem N has a constraint vector $C = (C_1, \ldots, C_k)$ of length k where k is the number of points of N. Each entry C_j in C is an integer defining the exact number of points in layer j of any triangulation of N or 0 to ensure layer j does not exist in any counted triangulation N. Using the constraint vector allows us to compute the number of triangulations that meet certain layer constraints.

As a dynamic programming algorithm, the nibbled ring algorithm has two components, the recursive case and the base case.

The recursive case relies heavily on separator paths. To formally define what separator paths are, consider a triangulation T of a nibbled ring sub-problem N. A separator path $p(T, N) = w_1, w_2, \ldots, w_\alpha$ is a path in T such that:

1. The first point of the separator path, w_1, and *base-edge*(N) form an empty triangle Δ such that the edges of Δ lie within the free region of N.
2. The length of p is at most the $width(N)$
3. The final point of the separator path, w_α, lies on *outer-layer*(N)
4. Any edge in p with both endpoints on *inner-layer*(N) must also be an edge of *inner-layer*(N)
5. Each point w_i on p must lie on a layer with layer index one less then then the layer index of the layer in T which contains w_{i-1}.
6. For each point w_i on p, w_{i+1} is the neighbor in T with the smallest order label among all neighbors of w_{i+1} in T with layer index one less then *layer-index*(w_i).

Lemma 4 *A triangulation T of a nibbled ring problem N has at most one layer separator.*

Proof: Let $p = (w_1, w_2, \ldots, w_d)$ and $k = (k_1, k_2, \ldots, k_d)$ be two distinct separator paths in T. Since *base-edge*(N) can only be a part of one triangle in T, $w_1 = k_1$. Take i as the smallest index such that $w_i \neq k_i$. Since $w_1 = k_1$, $i > 1$. From property 5: Since both w_i and k_i are neighbors of w_{i-1} in T ($w_{i-1} = k_{i-1}$), w_i and k_i lie on the same layer of T. From property 6: Since they lie on the same layer and are neighbors of w_{i-1}, both w_i and k_i must have the smallest order label. All points have a unique order label so $w_i = k_i$. This is a contradiction hence there cannot be two distinct separator paths for a triangle Δ.

This definition of separator paths has an inherent problem. It is dependent on an underlying triangulation T for the layer indexes necessary in points 5 and 6. We want to generate all possible separator paths with no knowledge of their underlying triangulations. Since layer indexes are not available, we introduce a new label, the *index* of a point. The index of a point p is denoted $index(p, N)$. The index can be undefined for some points. For others, it is defined inductively: Given a nibbled ring sub-problem N, $index(w_\alpha, N) = $ *outer-index*(N) and $index(w_i, N) = index(w_{i+1}, N) + 1$. Index is defined in the same way for the two boundary paths $b1$ and $b2$. All points on the outer layer have index equal to the *outer-index*(N) and all points on the inner layer has index equal to *inner-layer*(N). In this way the index for the points of N only depends on N itself rather then any underlying triangulation.

Using the index we can redefine $p(T, N)$ as $p(N)$ by replacing conditions 5 and 6 with new conditions:

1. No two adjacent points with an index differ in index by more then 1.
2. For each pair of adjacent points w_i and w_{i+1} on $pN()$ or on a boundary path, w_{i+1} has the smallest order label among neighbors of w_i with $index = index(w_{i+1})$.

We refer to the set of all Δ induced by the separator paths of a nibbled ring sub-problem N as $\Delta(N)$. A separator path is unique for a triangle $\Delta \in \Delta(N)$.

Using these conditions we can generate all separator paths with no knowledge of the underlying triangulations. We denote the set of possible separator paths of N with $Paths(N)$.

The recursive case of the nibbled ring algorithm uses these separator paths to define the nibbled ring sub-problem N in terms of smaller sub-problems. Each triangulation T of N has exactly one separator path from *base-edge*(N). The algorithm uses this observation to split a nibbled ring sub-problem into smaller sub-problems; the number of triangulations of a nibbled ring sub-problem is equal to the sum, for each $p \in Paths(N)$, of the number of triangulations of the nibbled ring sub-problem that contain p as the separator path. Formally:

$$T(N) = \sum_{p \in Paths(N)} T_p(N)$$

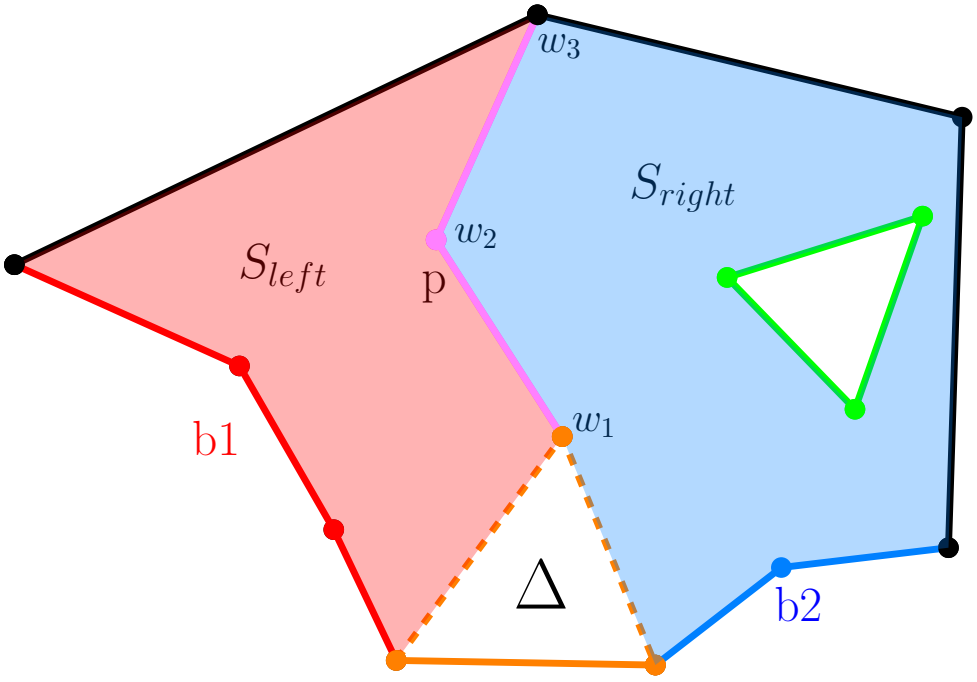


Figure 22: An example of a nibbled ring sub-problem split into a left and a right half by a separator path. The base edge is in orange, the separator path is in pink and labeled p, the inner layer is in green, and the two boundary paths are labeled and in red and blue.

where $T_p(N)$ is the number of triangulations of N that contain p.

What remains is to solve for $T_p(N)$. To do so, a separator path p is used to split a nibbled ring sub-problem N into two smaller sub-problems, $N_{left}(p)$ and $N_{right}(p)$, such that $T_p(N) = T(N_{left}(p)) \cdot T(N_{right}(p))$.

The base edge, $e = (i, j)$, and the first point w_1 in a path $p \in Paths(N)$ form the triangle Δ. The left sub-problem, $N_{left}(p)$, has a new edge (i, w_1) as the base edge, $b1$ as the new left boundary path, p as the new right boundary edge, and the segment of the original outer layer between the endpoints of $b1$ and p as the new outer layer. These components form a boundary polygon. Any edges of the original inner layer inside the boundary polygon are included as the new inner layer. Finally any free points within the boundary polygon are included as the new free points.

The right sub-problem, $N_{right}(p)$, is defined similarly, but with p as the new left boundary path, $b2$ as the right boundary path, and (w_1, j) as the new base edge. See Figure 22 for an example of a separator path splitting a nibbled ring sub-problem.

This division has a number of special cases:

1. If the base edge lies on a component of the inner layer the two boundary paths b1 and b2 may not be incident to the base edge. See Figure 23 (a) for an example.
2. Similarly, if the base edge is incident to an edge of the inner layer and that edge is on the boundary polygon, then the boundary paths may not be incident to the base edge. See Figure 23 (b) for an example.
3. If the new base edge of $N_{left}(p)$ or $N_{right}(p)$ would lie on a component c of the inner layer such that the new base edge would not incident to the free region of that sub-problem, a different edge of c is chosen as the new base edge instead. See Figure 23 (c) for an example.
4. If the separator path merges with a boundary path then the point where the two paths merge becomes the new outer layer for the sub-problem that no longer contains any of its parents outer layer. For example, in Figure 24 w_3 becomes the new outer layer of $N_{left}(p)$ and the two boundary paths of $N_{left}(p)$ end at w_3.
5. If the nibbled ring sub-problem consists of only the base edge, then the free region is empty, and one of the boundary paths coincides with the base edge while the other is a single point. This case is one of the base cases of the nibbled ring algorithm. See Figure 25 for an example of the base cases.

Refer to Marx and Miltzow's original publication for more detail on the special cases.

The index of points and the division into sub-problems are defined in such a way that the labeling of points with indexes is consistent in subsequent sub-problems. For example if point v_i is labeled with index k all sub-

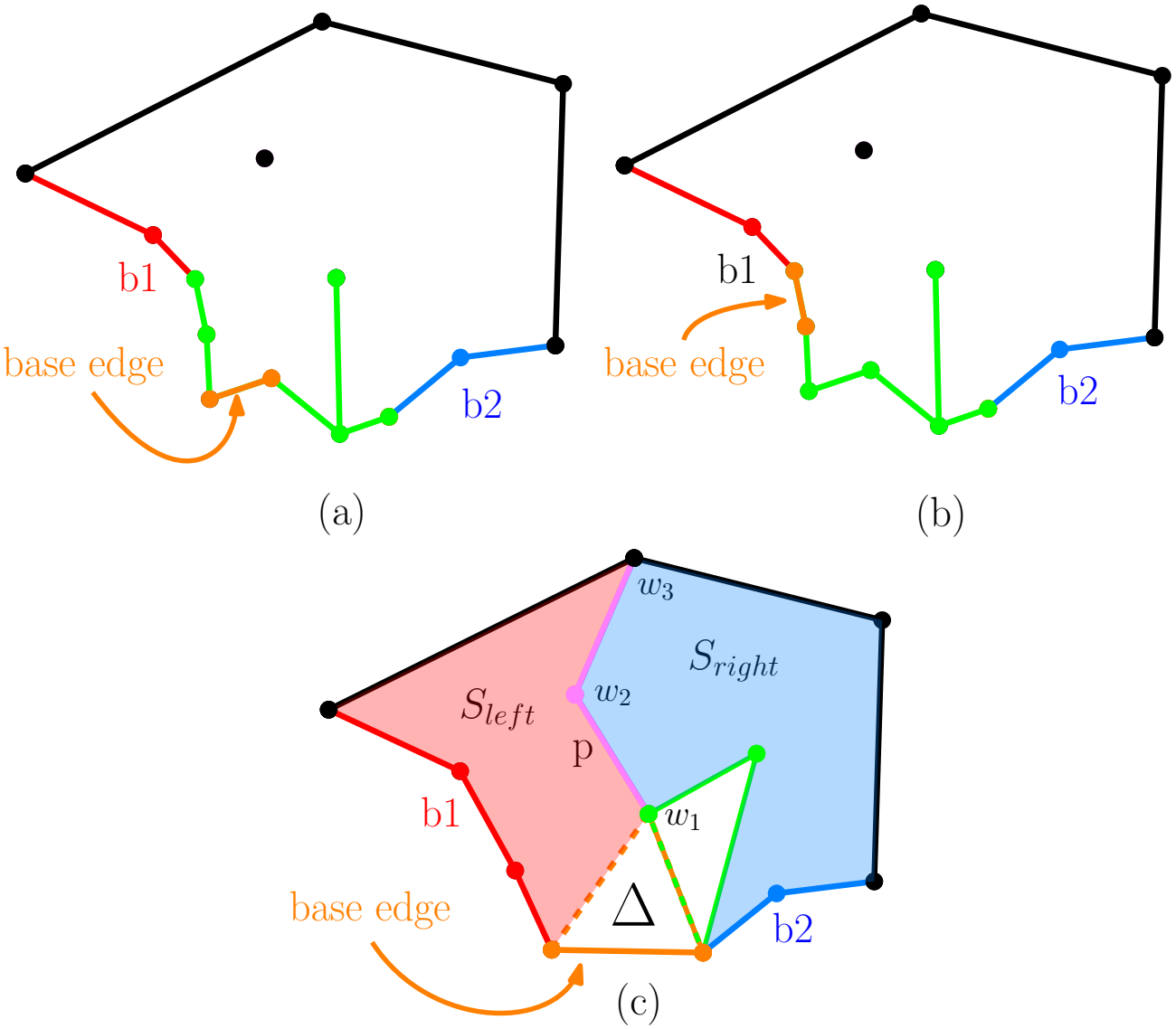


Figure 23: In all examples above the orange edge is the base edge, green edges are in inner layer, red and blue edges are b1 and b2 respectively. (a) An example of a nibbled ring sub-problem where the base edge is an edge of the inner layer. (b) An example of a nibbled ring sub-problem where the base edge is incident to the inner layer and both are on the boundary polygon. (c) An example of a nibbled ring sub-problem where the base edge of $N_{right}(p)$ would not be incident to the free region.

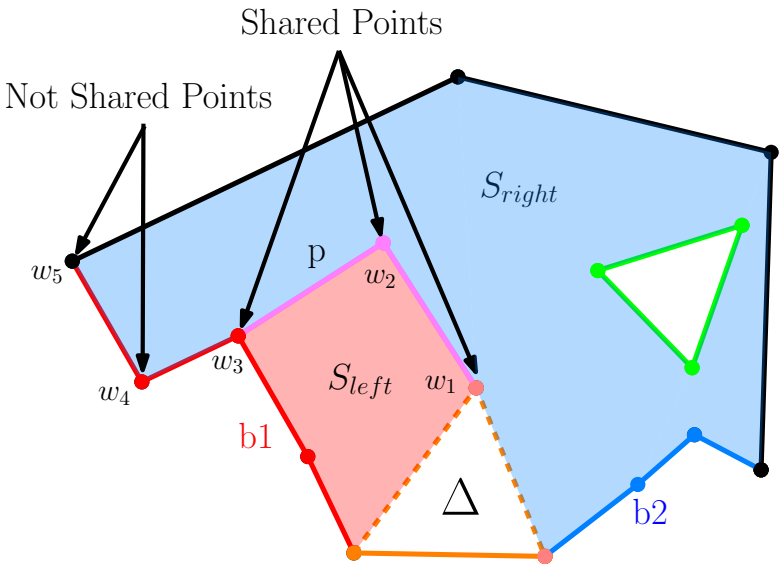


Figure 24: An example of a nibbled ring sub-problem where not all points on the separator path p are shared. In this example, w_4 and w_5 are not shared.

problems containing v_i will also label v_i with index k. This is an important property. If a point is stated to be part of a specific cactus layer, it cannot then be a part of a different cactus layer in subsequent steps.

The product of the left sub-problem and the right sup-problem for a separator path p is the total number of triangulations that contain path p. In this way, the total triangulations of a nibbled ring sub-problem N is the sum, for each separator path $p \in Paths(N)$, of the products of the left and right sub-problems.

$$T(N) = \sum_{p \in Paths(N)} T(N_{left}(p)) \cdot T(N_{right}(p))$$

However, this is not sufficient. The algorithm must only count triangulations that satisfy its constraint vector. In order to do so, the constraint vector must still be maintained in the sub-problems. For a nibbled ring sub-problem N with constraint vector C, all points in layer l_i across both the left and right sub-problem must sum to $j_i \in C$. To enforce this Marx and Miltzow define constraints $C_{left}(p)$ and $C_{right}(p)$ for the left sub-problem $N_{left}(p)$ and the right sub-problem $N_{right}(p)$ respectively. The general idea being that $C_{left}(p)$ and $C_{right}(p)$ sum to C thereby enforcing that any triangulation counted satisfies C. This introduces a problem. If $C_{left}(p)$ and $C_{right}(p)$ sum to C then points shared between $N_{left}(p)$ and $N_{right}(p)$ would be counted twice for layer size.

To address this, Marx and Miltzow first considered when points might be shared. Points shared between $N_{left}(p)$ and $N_{right}(p)$ lie on the dividing line between the two, the separator path. It is not always the case that all points on a separator path are shared. See Figure 24 for an example. To account for the shared points, Marx and Miltzow propose the use of a indicator vector $CI(k_1, \ldots, k_n) : k_i \in [0, 1]$. Each $k_i \in CI$ is 1 if layer i contains a shared point and 0 otherwise. Using CI we can define, for a nibbled ring problem N and a path p, a set of paired constraint vectors $C^p(N, p)$. The pair $(C1, C2) \in C^p \iff C1 + C2 = C + CI$. This solution essentially acknowledges that shared points are counted twice and increases the size of the constraint for each layer with a shared point in order to account for it. An indicator vector is used because, since shared points lie on the separator path, there can be at most one shared point per layer.

The nibbled ring algorithm counts the number of triangulations for each pair of constraint vectors in C^p and sums them up. The number of triangulations for a recursive case nibbled ring sub-problem N can then be defined as:

$$T(N) = \sum_{p \in Paths(N)} \left[\sum_{(C1,C2) \in C^p(N,p)} [T(N(C1, C2, p))] \right]$$
$$T(N(C1, C2, p)) = T(N_{left}(C1, p)) \cdot T(N_{right}(C2, p))$$

Where:

1. $T(N_{left}(C1, p))$ is the triangulation count of the left sub-problem of N formed using path p and constrained with constraint vector $C1$.

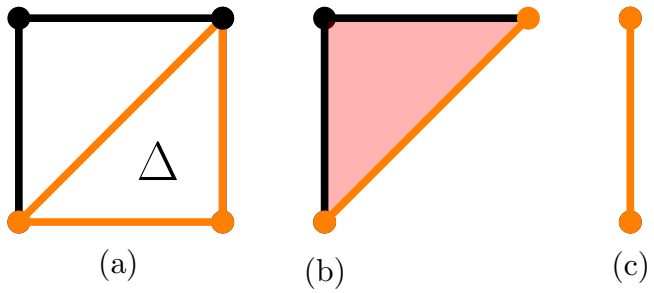


Figure 25: An example of both base cases of the nibbled ring sub-problem. The point set in (a) is split by Δ into sub-problems (b) and (c). Both (b) and (c) are base cases with one triangulation.

2. $T(N_{right}(C2, p))$ is the triangulation count of the right sub-problem of N formed using path p and constrained with constraint vector $C2$.

The base cases of the nibbled ring algorithm is interesting. Obviously, if the input is simply a triangle that satisfies the constraint vector, the number of triangulations is 1. If any nibbled ring problem does not satisfy the constraint vector, the number of triangulations is always 0. Less intuitively however, if the sub-problem consists of only an edge and it satisfies the constraint vector then it is also considered to have one triangulation. See Figure 25 for an example of both base cases.

6.4.2 General Layer-Unconstrained Ring Sub-Problem

The nibbled ring algorithm can solve ring problems with width at most $\sqrt{n}$. What remains is the technique for dealing with ring problems with width greater then $\sqrt{n}$. Marx and Miltzow propose a *general layer-unconstrained ring algorithm* to count the triangulations of these ring problems.

This algorithm is relatively simple. The crux is to generate all cactus layers satisfying certain constraints that are disjoint from the convex hull then, for each such generated layer l, split the ring problem into two ring problems. A nibbled ring sub-problem between the outer layer and l, and a smaller general layer-unconstrained ring sub-problem inside l. Marx and Miltzow refer to this process as "peeling". A nibbled ring sub-problem is peeled off the outside leaving a smaller instance of the original problem on which to recurse.

Formally, a general layer-unconstrained ring sub-problem S has:

1. *outer layer*: Denoted with $\textit{outer-layer}(S)$, the outer layer is a polygon.
2. *outer index*: The outer index is the layer index of the outer layer. Denoted by $\textit{outer-index}(S)$
3. *free points*: A set of points contained entirely within the outer layer. A free point cannot be incident to any other component of the nibbled ring problem. Denoted by $\textit{free-points}(S)$

A general layer-unconstrained ring sub-problem S is dependent on all its components. However, for Marx and Miltzow's approach, $\textit{free-points}(S)$ is always all point in P that satisfy the constraints on free points.

The points of S is the set of points contained in either $\textit{outer-layer}(S)$ or $\textit{free-points}(N)$.

For a general layer-unconstrained ring sub-problem S let the triangulations of S be the set of all maximal subsets of non-crossing edges spanning between points of S that are contained within $\textit{outer-layer}(S)$ and that include all edges of the $\textit{outer-layer}(S)$. For a general layer-unconstrained ring sub-problem S, $T(S)$ is the number of triangulations of S.

In order to peel a nibbled ring sub-problem from a general layer-unconstrained ring sub-problem, Marx and Miltzow propose the use of *layer separators*.

To define a layer separator we must first define an *m-peripheral layer*. An m-peripheral layer for a triangulation T is a cactus layer with layer index i such that: $\textit{outer-index}(T) < i \leq \textit{outer-index}(T) + \lfloor\sqrt{n}\rfloor + 1$.

A layer separator l of a general layer-unconstrained ring sub-problem R has the following properties:

1. l is an m-peripheral layer.
2. l has size less then $\lfloor\frac{\textit{free-points}(S)}{\lfloor\sqrt{n}\rfloor}\rfloor$.
3. l is entirely contained within the outer layer: each edge lies within the outer layer and is completely disjoint from the outer layer.
4. All m-peripheral layers with a smaller layer index then l have size at least $\lfloor\frac{\textit{free-points}(S)}{\lfloor\sqrt{n}\rfloor}\rfloor$.

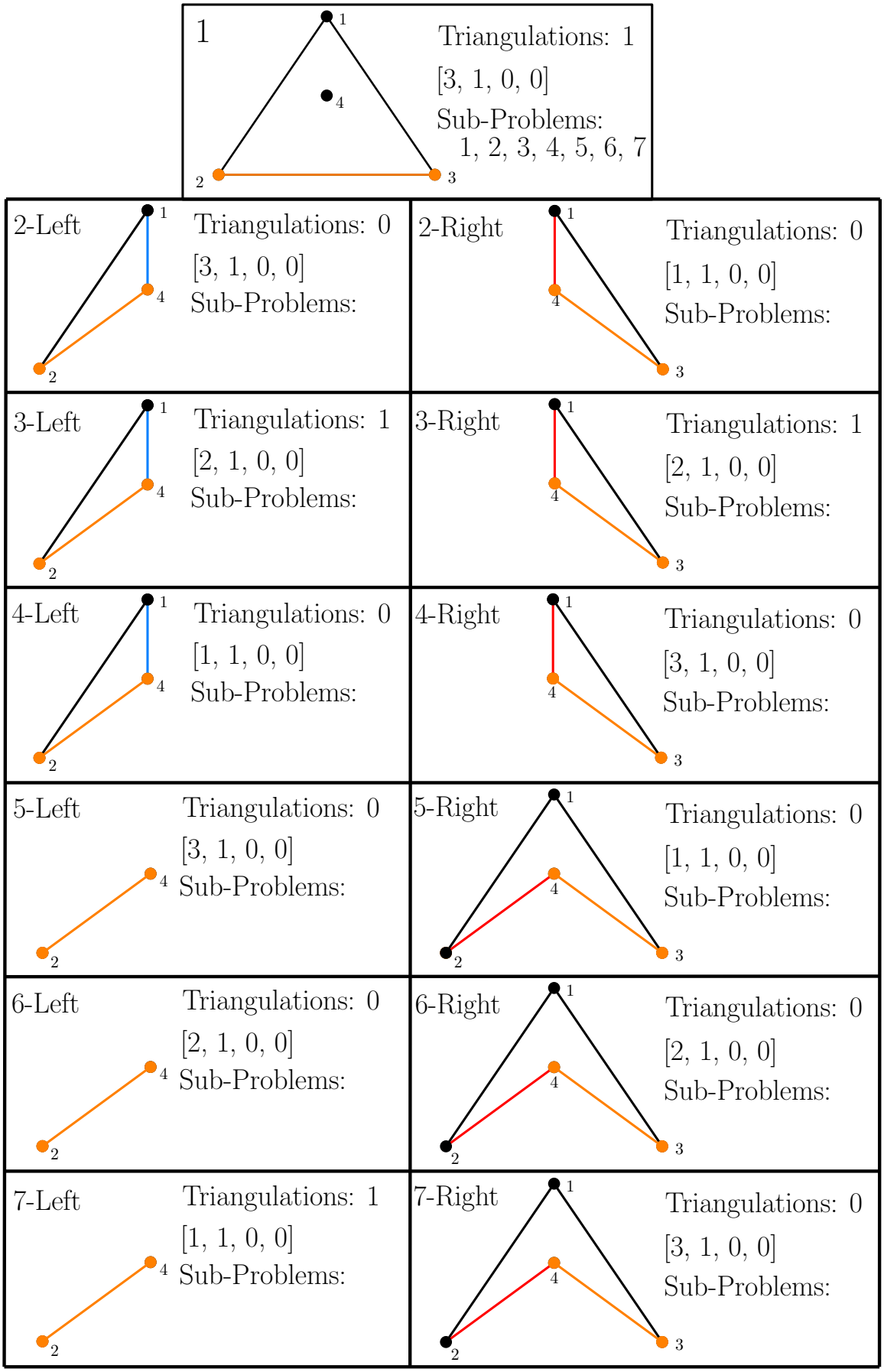


Figure 26: An example of the nibbled ring algorithm. The black edges are the outer layer, the orange edge is the base edge, red and blue edges are b1 and b2 respectively, and the small number by each point is its order label. Interestingly, 7-Right does not have any triangulations, this is because point 1 cannot be a separator path due to condition 6.

We define $separators(S)$ as the set of all layer separators S. Every triangulation has exactly one layer separator. If the triangulation has no layer of size less then $\lfloor \frac{free\text{-}points(S)}{\lfloor \sqrt{n} \rfloor} \rfloor$ then it can have at most $\lfloor \sqrt{n} \rfloor$ layers and an empty layer with index $\lfloor \sqrt{n} \rfloor + 1$ is the separator layer.

Since every triangulation has exactly one layer separator, the number of triangulations of a general layer-unconstrained sub-problem S can be denoted:

$$T(S) = \sum_{l \in separators(S)} T_l(S)$$

Where $T_l(S)$ is the number of triangulations that contain layer l.

Given a separator layer $l \in separators(S)$, the procedure is straight forward. The separator layer l is completely disjoint from the outer layer by the definition of m-peripheral layer. As such the ring formed by the outer layer and l forms an unconstrained nibbled ring problem $N_u(l)$.

Notably, each layer of $N_u(l)$ must be of size at least $\lfloor \frac{free-points(S)}{\lfloor \sqrt{n} \rfloor} \rfloor$ from the definition of a layer separator. To enforce this we use a set of constraint vectors.

Given a general layer-unconstrained ring sub-problem S, a layer separator l, we define a set of constraint vectors $C_v(S, l)$. A vector $C(i_1, i_2, \ldots, i_n) \in C_v(S, l)$ if:

1. i_j is equal to the size of the outer layer of S if
   $j = outer\text{-}index(S)$
2. i_j is equal to the size of l if
   $j = layer\text{-}index(l)$
3. i_j is equal to 0 if
   $j \notin [outer\text{-}index(S), layer\text{-}index(l)]$
4. $\lfloor \frac{free\text{-}points(S)}{m} \rfloor \leq i_j \leq n$ if
   $j \in [outer\text{-}index(S) + 1, layer\text{-}index(l) - 1]$

The sum of the number of triangulations of $N_u(l)$ constrained with a vector $C \in C_v(S, l)$ for each vector in $C_v(S, l)$ counts every possible triangulation of the outer ring problem where each layer has size at least $\lfloor \frac{free\text{-}points(S)}{\lfloor \sqrt{n} \rfloor} \rfloor$. What remains is any interior components of l.

Each interior component of l, if any, forms the outer layer of a new, smaller, general layer-unconstrained ring sub-problem. In the case that there are multiple interior components of l the number of triangulations of the inner components is the product of the number triangulations of each component. See Figure 27 for an example of a separator layer with multiple interior components.

Formally, the number of triangulations of the interior components of l in a general layer-unconstrained ring problem S is:

$$\prod_{S_{in} \in comp_{in}(l)} T(S_{in})$$

Where the set of interior components of a layer l is denoted $comp_{in}(l)$.

The product of number of triangulations of the outer ring problems and the number of triangulations of the inner ring problems is the total number of triangulations of the original sub-problem that contain the separator layer l.

Formally, The number of triangulation of a general layer-unconstrained ring sub-problem S that contain the layer l is:

$$\left(\sum_{c \in C_v(S,l)} N(l, c) \right) \cdot \left(\prod_{S_{in} \in comp_{in}(l)} T(S_{in}) \right)$$

The number of triangulations of a general layer-unconstrained ring sub-problem S is the sum across all $l \in separators(S)$ of the product of the sum of all nibbled ring sub-problems with each vector $C \in C_v$ times the product of each interior general layer-unconstrained ring sub-problem.

Formally the number of triangulations of a general layer-unconstrained sub-problem S is:

$$\sum_{l \in separators(S)} \left[\sum_{C \in C_v(S,l)} N(l, C) \cdot \prod_{S_{in} \in comp_{in}(l)} T(S_{in}) \right]$$

Where $N(l, C)$ is the total number of triangulations of a nibbled ring sub-problem formed by $outer\text{-}layer(S)$ and l with the constraint vector C.

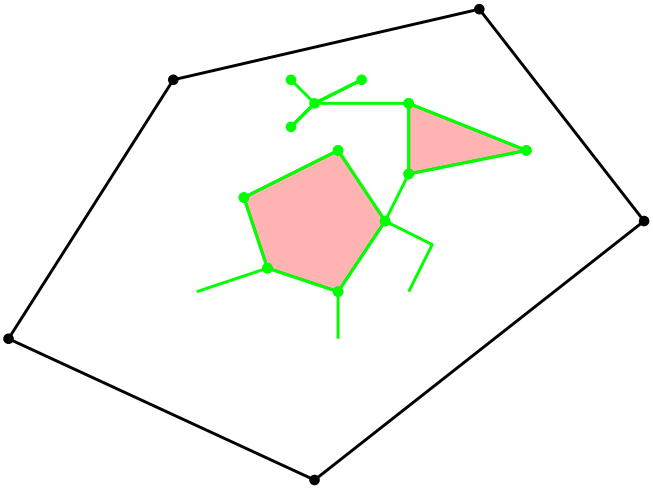


Figure 27: An example of the separator layer, shown in green, having two interior components, shown in red.

Algorithm 1 General Layer-Unconstrained Ring Algorithm

Precondition: planar point set P
function INITIALIZE(P)
    Convert P into its corresponding general layer-unconstrained sub-problem S
    Initialize a database d of all computed values
    return COUNT(S)

Precondition: general layer-unconstrained sub-problem S
function COUNT(S)
    if S in d **then**
        return $d(S)$
    else if S has maximum width $\leq \lfloor\sqrt{n}\rfloor$ **then**
        $result$ = NIBBLING(S)
    else if S has maximum width $> \lfloor\sqrt{n}\rfloor$ **then**
        $result$ = SPLITBYSEPERATOR(S)
    add $(S, result)$ to d
    return $result$

Precondition: general layer-unconstrained sub-problem S
function SPLITBYSEPERATOR(S)
    $s_1 \leftarrow 0$
    for each layer separator $l \in separators(S)$ **do**
        $s_2 \leftarrow 0$
        $s_3 \leftarrow 0$
        Define the sub-problem $N_{out}(l)$ and the set of sub-problems $S_{in}(l)$ by splitting S using l
        for each constraint vector $C \in C_v(S)$ **do**
            $s_2 = s_2 +$ NIBBLING($N_{out}(l), C$)
        for each sub-problem $S_k \in S_{in}(l)$ **do**
            $s_3 = s_3 \cdot$ COUNT(S_k)
        $s_1 = s_1 + s_2 \cdot s_3$
    return s_1

Algorithm 2 Nibbled Ring Algorithm

Precondition: nibbled ring problem N, constraint vector C

```
1  function NIBBLING(N, C)
2      if N in d then
3          return d(N)
4      else if N does not satisfy C then
5          insert (N, 0) into d
6          return 0
7      else if N is a base case then
8          insert (N, 1) into d
9          return 1
10     else
11         result ← 0
12         for each valid triangle Δ do
13             for each separator path p from Δ do
14                 for each constraint pair (C_left, C_right) ∈ C^p(N, p) do
15                     result = result + NIBBLING(N_left(p), C_left) · NIBBLING(N_right(p), C_right)
16         insert (N, result) into d
17         return result
```

6.5 Analysis

We recall the proof of running time and correctness given in Marx and Miltzow's paper [22]. Marx and Miltzow use a series of formal theorems and proofs to prove the running time bounds of their algorithm. Their approach involves proving the running time of the nibbled ring algorithm and using that to prove the running time of the general layer-unconstrained algorithm. In this paper we reiterate key components contributing to the running time of both algorithms in order to cultivate an understanding. For a formal proof please refer to the original paper.

Proving an upper bound on the running time of any dynamic programming algorithm can be done with a bound on the total number of unique sub-problems, and a bound on the time taken in each sub-problem.

Finding a bound on the total number of unique nibbled ring sub-problems relies on an observation that any nibbled ring sub-problem stemming from one original ring problem N can be uniquely defined by its two boundary paths and its constraint vector. Take n as the number of points in N. Since boundary paths can have length at most one more then the width of the nibbled ring sub-problem and nibbled ring sub-problems have width at most $\sqrt{n}$, there are at most $n^{\sqrt{n}+O(1)}$ possible boundary paths. Similarly, since all layers with index greater then the width must have size 0, there are only $n^{\sqrt{n}}$ possible constraint vectors. In this way, there are at most $n^{\sqrt{n}+O(1)} \cdot n^{\sqrt{n}+O(1)} \cdot n^{\sqrt{n}} = n^{O(\sqrt{n})}$ possible nibbled ring sub-problems for N.

The majority of the time spent in a single nibbled ring sub-problem N is the time spent searching the database for each of N's sub-problems. There is one search of the database for each of N's sub-problems. The number of sub-problems for a single instance of a nibbled ring sub-problem is bounded by the number of constraint pairs, (c_{left}, c_{right}), multiplied by the number of separator paths. Since separator paths have the same size restriction as boundary paths, there are also at most $n^{\sqrt{n}+O(1)}$ of them. For every C_{left} there is exactly one C_{right}, therefor there are $n^{\sqrt{n}}$ total constraint pairs. In this way, there are $n^{\sqrt{n}+O(1)} \cdot n^{\sqrt{n}} = n^{O(\sqrt{n})}$ total searches of the database per nibbled ring sub-problem.

Given that there are at most $n^{O(\sqrt{n})}$ sub-problems, searching the database using, for example, a binary search would take $log(n^{O(\sqrt{n})}) = O(\sqrt{n}) \cdot logn < O(n^2)$ time.

Finally, since there are at most $n^{O(\sqrt{n})}$ searches per sub-problem, at most $n^{O(\sqrt{n})}$ sub-problems, and each search takes at most $O(n^2)$ time, the running time of the nibbled ring algorithm is bounded by $n^{O(\sqrt{n})} \cdot n^{O(\sqrt{n})} \cdot O(n^2) = n^{O(\sqrt{n})}$.

Similar to the nibbled ring algorithm, computing a bound on the running time of the general layer-unconstrained algorithm involves putting a bound on the number of possible sub-problems stemming from a single super-problem. It also involves putting a bound on the running time of each sub-problem. The majority of time spent in a sub-problem is either the time spent in the nibbled ring algorithm for this sub-problem or the time spent searching the database for sub-problems of this sub-problem.

For a point set P of size n, a general layer-unconstrained ring sub-problem can be completely defined by

its outer layer and outer layer index. Marx and Miltzow show that there are at most $n^{(3+O(1))\sqrt{n}} = n^{O(\sqrt{n})}$ total possible outer layers. Obviously there cannot be more then n non-empty layers so the total number of sub-problems is at most $n \cdot n^{O(\sqrt{n})} = n^{O(\sqrt{n})}$.

The time spent in the nibbled ring algorithm has already been shown to be $n^{O(\sqrt{n})}$ per constraint vector and there are at most $n^{\sqrt{n}}$ constraint vectors. As a result the time spent in the nibbled ring algorithm per layer separator is at most $n^{O(\sqrt{n})} \cdot n^{\sqrt{n}} = n^{O(\sqrt{n})}$. The number of recursive calls per layer separator l is dependent on the number of interior components of l which Marx and Miltzow claim to be polynomial in n. We denote the number of recursive calls per layer separator as J.

What remains is the bound on the number of layer separators. It is well known that any outer planar graph of a set of $\sqrt{n}$ points has at most $2\sqrt{n} - 3$ edges. Since cactus graphs are a type of outer planar graph this bound holds for them as well. Using basic statistics, it is easy to see that, on a set of $\sqrt{n}$ points, there are at most $\binom{\sqrt{n}^2}{2\sqrt{n}-3} \leq \sqrt{n}^{4\sqrt{n}}$ possible cactus graphs. Similarly, from a set of n points, there are at most $\binom{n}{\sqrt{n}} \leq n^{\sqrt{n}+1}$ point sets of size at most $\sqrt{n}$. Since a layer separator is a cactus graph of size at most $\sqrt{n}$ in a set of n points, the number of layer separators is bounded by $n^{\sqrt{n}+1} \cdot \sqrt{n}^{4\sqrt{n}} = n^{O(\sqrt{n})}$

Finally, these components can be combined to compute the running time of the general layer-unconstrained ring algorithm.

$$n^{O(\sqrt{n})} \cdot n^{O(\sqrt{n})} \cdot (n^{O(\sqrt{n})} + J) = n^{O(\sqrt{n})}$$

In layman's terms: The number of sub-problems times the amount of time taken per sub-problem. Each sub-problem takes time equal to the number of layer separators times the sum of the time spent in the nibbled ring algorithm and the time spent making recursive calls.

Its easy to see that both nibbled ring sub-problems and general layer-unconstrained ring sub-problems require $O(n)$ space. The largest component contained in each sub-problem is a list of points bounded in size by n. Since there are at most $n^{O(\sqrt{n})}$ sub-problems of each type and each sub-problem is stored at most once the total space complexity of this algorithm is $n^{O(\sqrt{n})} \cdot O(n) + n^{O(\sqrt{n})} \cdot O(n) = n^{O(\sqrt{n})}$.

While Marx and Miltzow's algorithm has far worse space complexity then Avis and Fukuda's algorithm, its running time is significantly better. Additionally, its space complexity is better then that of Alvarez and Seidel's algorithm.

7 Experimentation

As a compliment to this paper, an application implementing Marx and Miltzow's algorithm with step-by-step visualizations is provided. This algorithm was chosen for two primary reasons: To supplement this paper's explanation of their algorithm with a visual aid for the user's own point set and to convince the user of the correctness of Marx and Miltzow's algorithm.

To provide an indication of correctness Avis and Fukuda's algorithm and Alvarez and Seidel's algorithm were also implemented. The two other algorithm's were used to independently verify, for each test point set, that Marx and Miltzow's algorithm counts triangulations correctly.

The application was tested on the point sets listed in Table 1 and shown in Figure 28. Each algorithm returned the same number of triangulations for every point set tested with the exception of the cases where implementation of Avis and Fukuda's algorithm timed out. In those cases, Marx and Miltzow's algorithm and Alvarez and Seidel's algorithm still returned the same result.

While all three algorithms were implemented with step by step visualizations, special attention was given to Marx and Miltzow's algorithm. See Appendix A for a user manual of the implementation of Marx and Miltzow's algorithm.

Marx and Miltzow's algorithm, while fast in theory, is significantly slower then Alvarez and Seidel's algorithm in practice for any reasonable point set. This is due to significant constants hidden in the order notation of the running time.

The implementation described in Appendix A has a number of such improvements. Most of these improvements check to see if a sub-problem can possibly have any triangulations then discard the sub-problem if no triangulations are possible. Optimizations in the implementation include the following:

1. Once a layer has size less then 3, no subsequent layer can have size larger then 0. As a result any constraint vector that violates this can be discarded. This is simply because at least 3 points are necessary to create a polygon with volume.
2. If one of the inner ring problems or the outer ring problem defined by a layer separator l has zero triangulations, then no triangulations that contain l can exist and l and its remaining unsolved sub-problems can be discarded.

Set 1	Set 2	Set 3	Set 4
(0.5, 1)	(0, 0)	(0.15, 0.1)	(62, 140)
(0.2 0.85)	(500, 0)	(0.85, 0)	(72, 365)
(0.1,0.6)	(250, 501)	(0, 0.75)	(297, 443)
(0.4, 0.7)	(151, 100)	(1, 0.75)	(474, 371)
(0.3, 0.35)	(351, 100)	(0.5, 1)	(481, 210)
(0.7, 0.5)	(251, 352)	(0.5, 1)	(396, 95)
(0.6, 0.3)	(226, 156)	(0.51, 0.5)	(271, 56)
	(273, 152)	(0.52, 0.25)	(168, 82)
		(1.1, 1)	(120, 250)
		(0.9, 0.78)	(212, 356)
		(0.6, 0.9)	(380, 326)
			(383, 212)
			(291, 157)
			(179, 165)
36	135	824	249434

Table 1: Each Column above is a point set with a row for each point. The last row lists the number of triangulations for that point set. Avis and Fukuda's algorithm timed out on Set 4 so it was only verified on Alvarez and Seidel's algorithm.

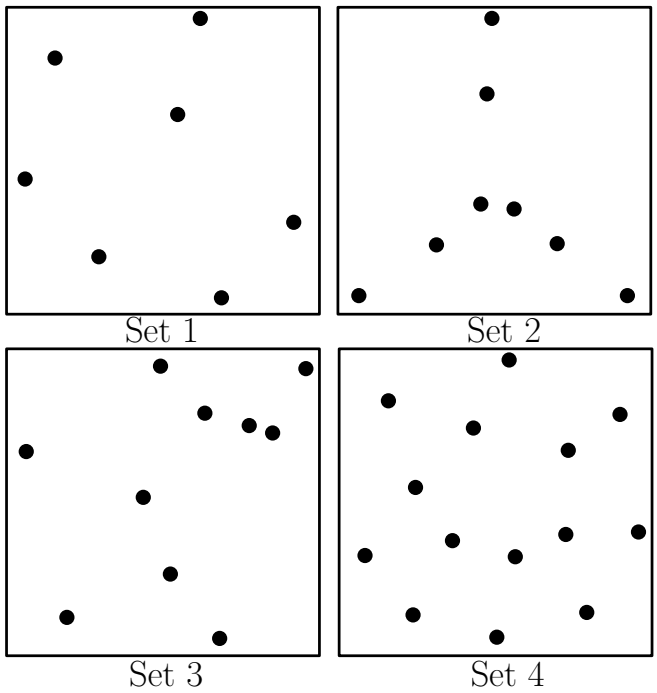


Figure 28: Each of the 4 point sets.

3. If a layer separator l defines an outer ring problem with less then $(width - 2) \cdot \lfloor \frac{free\text{-}points(S)}{\lfloor \sqrt{n} \rfloor} \rfloor$ free points, then there exist zero triangulations with layer separator l. In this case layer l and all its sub-problems can be discarded. This is simply because the outer layer is constrained such that each layer must have size at least $\lfloor \frac{free\text{-}points(S)}{\lfloor \sqrt{n} \rfloor} \rfloor$. If there do not exist enough points to satisfy the constraints then the constrained outer ring cannot have any triangulations and the triangulations of the inner ring do not matter.
4. If at any point the sum of the sizes of each layer in a constraint vector c does not equal the total number of points in c's nibbled ring problem, then that problem can be discarded. This is because there is no way to satisfy c.

Similar techniques are used to reduce space complexity. For example, while the theoretical number different of constraint vectors is enormous, in practice there are relatively few unique constraint vectors and significant number of duplicates. Since constraint vectors are immutable, all sub-problems that require a specific constraint vector c can refer to one instance of c. In this way there are no duplicate constraint vectors. In the test cases, this example reduced memory usage by a factor of 10.

A fair comparison of the algorithm's experimental run times is not possible with the implementation due to the vast amount of optimizations performed and the different structure of Marx and Miltzow's algorithm's implementation. Even with the optimizations of Marx and Miltzow's algorithm, Alvarez and Seidel's algorithm is orders of magnitude faster in practice for any reasonable point set.

8 Conclusion

In the twenty years between 1996 and 2016 algorithms for counting triangulations have significantly improved. Avis and Fukuda's algorithm, published in 1996, was a general framework. Only one of its applications was counting triangulations. Nine years later Ray and Seidel proposed a specialized algorithm specifically to count triangulations. They claim their algorithm is more intuitive then Avis and Fukuda's and Ray and Seidel's testing seems to indicate that it is faster as well. Eight years later, Alvarez and Seidel publish a significant advancement. The first algorithm that can count triangulations of a point set such that the order of its running time is asymptotically less then the order of the number of triangulations. Aside from being fast, their algorithm is also intuitive and flexible. Alvarez and Seidel's paper won the SOCG best paper award in 2013 showing that counting triangulations is very relevant in the wider field of computational geometry. In 2016, three years after Alvarez and Seidel's paper, Marx and Miltzow publish the first algorithm capable of counting the triangulations of a point set in sub-exponential time.

In only twenty years algorithms progressed from $O(n \cdot 30^n)$ running time to $n^{(11+O(1))\sqrt{n}}$ running time.

In this paper we discussed four papers dispersed over the twenty year period. Each paper proposed an algorithm more efficient then the one before it and each algorithm took a distinct approach to counting triangulations. To help convince the reader of the correctness of Marx and Miltzow's algorithm and to help explain Marx and Miltzow's algorithm, we presented an application that implements it. The application was tested along side an implementation of Avis and Fukuda's algorithm and an implementation of Alvarez and Seidel's algorithm. In all tested cases all three returned the same result.

References

[1] Oswin Aichholzer, "The Path of a Triangulation", June 1999

[2] Oswin Aichholzer, Ferran Hurtado, and Marc Noy, "A lower bound on the number of triangulations of planar point sets", October 2004.

[3] Victor Alvarez, Karl Bringmann, Radu Curticapean, and Saurabh Ray "Counting Triangulations and other Crossing-free Structures via Onion Layers", March 2015

[4] Victor Alvarez, Karl Bringmann, Saurabh Ray and Raimund Seidel "Counting Triangulations and other Crossing-Free Structures Approximately", July 2015

[5] Victor Alvarez, Karl Bringmann, and Saurabh Ray, "A Simple Sweep Line Algorithm for Counting Triangulations and Pseudo-triangulations", April 2018

[6] Victor Alvarez and Raimund Seidel, "A simple aggregative algorithm for counting triangulations of planar point sets and related problems", June 17 2013

[7] Emile E. Anclin, "An upper bound for the number of planar lattice triangulations", August 2003

[8] David Avis and Komei Fukuda, "Reverse search for enumeration", March 1996

[9] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars, “Computational Geometry: Algorithms and Applications”, March 2008

[10] Roland Bacher and Frederic Mouton, “Triangulations of nearly convex polygons”, December 2010

[11] Richard Bellman, “The Theory of Dynamic Programming”, July 1954

[12] Sergei Bespamyatnikh, “An efficient algorithm for enumeration of triangulations”, November 2002

[13] S. Chattopadhyay and P.P. Das, “Counting thin and bushy triangulations of convex polygons”, March 1991

[14] Bernard Chazelle, “Triangulating a Simple Polygon in Linear Time”, September 1991

[15] Peter Epstein and Jörg-Rüdiger Sack, “Generating triangulation at random”, July 1994

[16] David Eppstein, “Counting Polygon Triangulations is Hard”, March 2019 revised March 2020

[17] Leonhard Euler, “Elementa doctrinae solidorum”, 1758

[18] Stefan Felsner, “Geometric Graphs and Arrangements: Some Chapters from Combinatorial Geometry”, February 2004

[19] Tamal Krishna Dey, “On counting triangulations in d dimensions”, December 1993

[20] Yu-Feng Lan, Yue-Li Wang, and Hitoshi Suzuki. “A linear-time algorithm for solving the center problem on weighted cactus graphs”, February 1999 revised September 1999

[21] Maarten Loffler and Wolfgang Mulzer, “Unions of Onions: Preprocessing Imprecise Points for Fast Onion Decomposition”, February 2013 revised January 2014

[22] Daniel Marx and Tillmann Miltzow, “Peeling and Nibbling the Cactus: Subexponential-Time Algorithms for Counting Triangulations and Related Problems.”, March 23 2016

[23] Marc Noy and Juanjo Rue, “Counting Polygon Dissections in the Projective Plane”, October 2008

[24] Joseph O’Rourke, “Computational Geometry in C Second Edition”, 2011

[25] Renato Pajarola, “Advanced 3D Computer Graphics”, October 20 2000

[26] Saurabh Ray and Raimund Seidel, “A Simple and Less Slow Method for Counting Triangulation and for Related Problems”, 2004

[27] Andreas Razen and Emo Welzl, “Counting Plane Graphs with Exponential Speed-Up”, November 2007

[28] A. Razen and E. Welzl, “Counting plane graphs with exponential speed-up”, 2011

[29] Francisco Santos and Raimund Seidel, “A better upper bound on the number of triangulations of a planar point set”, April 2003

[30] Micha Sharir and Adam Sheffer, “Counting Triangulations of Planar Point Sets”, March 2011

[31] Emo Welzl, “The Number of Triangulations on Planar Point Sets”, 2006

[32] Manuel Wettstein, “Counting and Enumerating Crossing-free Geometric Graphs”, June 2014

A Appendix: Application User Manual

The application and the source code are both available at https://github.com/Jschukken/MarxMiltzowApplication.

Table of Contents

A.1 Basic Start-Up

In this section we describe how to launch the application on a Windows computer after downloading the .zip file.

Step 1:
Create a new empty folder in a location of your choice.

Step 3:
Download "CactusApplication(.jar file).zip" from https://github.com/Jschukken/MarxMiltzowApplication.

Step 3:
Open the .zip folder, select all files, and drag them into the new folder. This unzips them automatically. There will be 3 files: a folder called "resources", a .jar called "CactusApplication", and a .bat called "Run". See Figure 29 for the required files and see Figure 30 for a labeled image of each file in the resources folder.

Step 4:
Double click on "CactusApplication.jar". The application will launch.

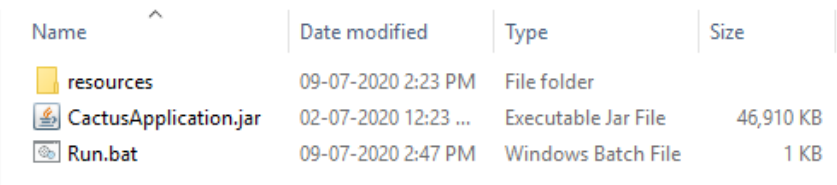
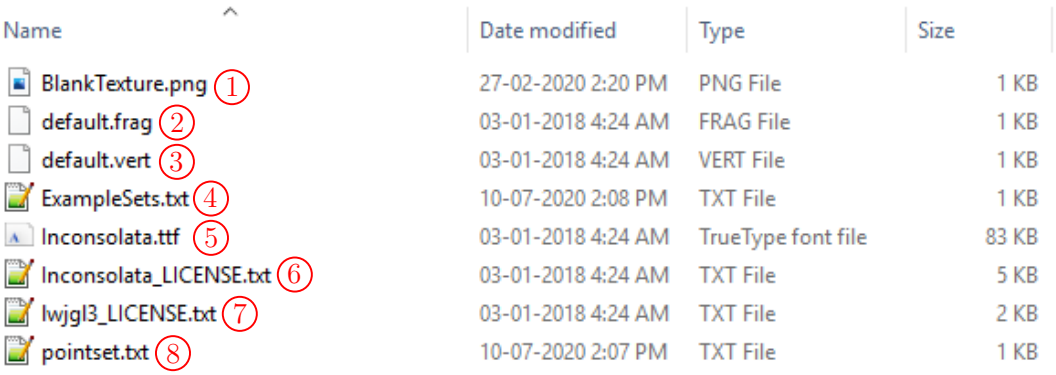


Figure 29: An image of each expected file in the .zip.



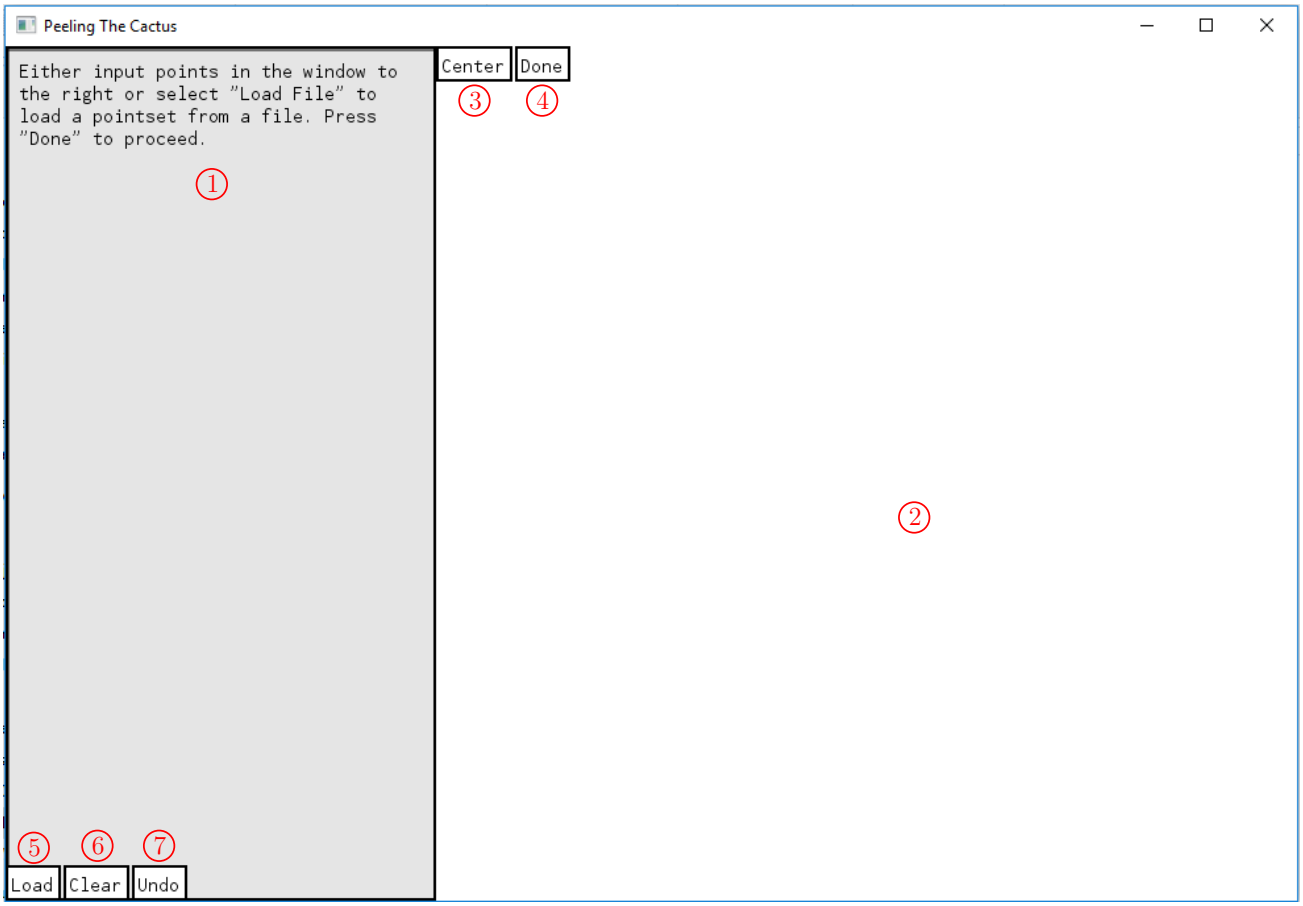
(1)(2)(3)(5): Files essential to the rendering of the application.
(6)(7): Licenses for copyrighed material used in application.
(4): File containing a number of example point sets and their triangulation counts.
(8): File from which the application loads a point set.

Figure 30: A labeled image of each expected file in the resources folder.

A.2 Algorithm Navigation

In this section we provide a labeled diagram and a description for each state in the application.

A.2.1 Point Set State



1: Text Window. In this window a short set of instructions is given.

2: Display Window. In this window the user can click to place points, click and drag to move viewing window around, and use the scroll wheel to zoom in or out. The window will display the current point set and the outer layer of the resulting ring problem, if any.

3: Center Button. Resets the Display Window to default position and zoom.

4: Done Button. Transitions from Point Input State to Verification State with the current point set.

5: Load Button. Clears the current point set and loads a new one from pointset.txt

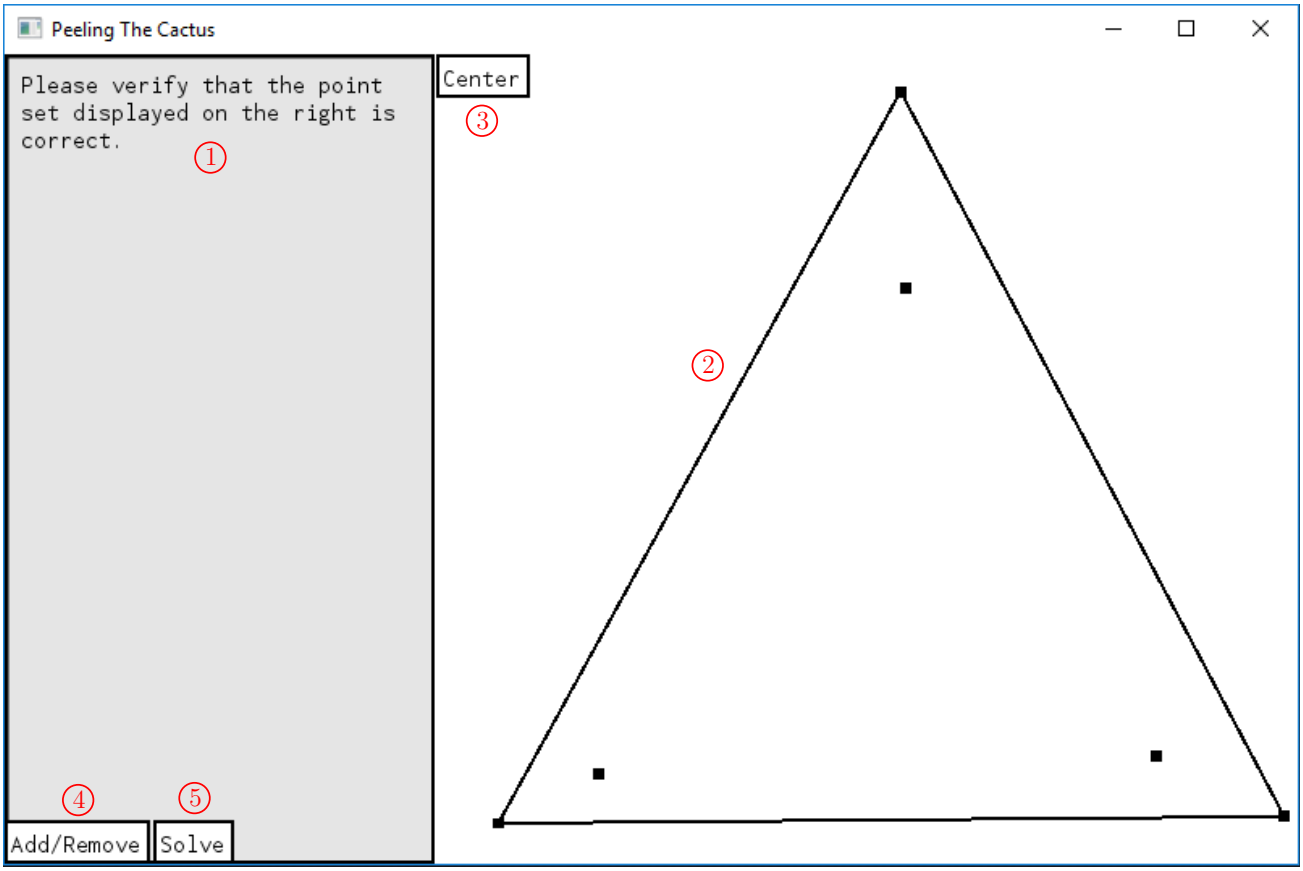
6: Clear Button. Removes all points from the point set.

7: Undo Button. Removes the most recently added point from the point set.

Figure 31: A labeled image of the Point Set State.

See Figure 31 for a labeled diagram. When the application is launched the initial state is the Point Set State. In this state the user can input their own point set by clicking in the display window or by loading from a file. When the user is happy with their point set, they can press “Done” button to proceed to the Verification State.

A.2.2 Verification State



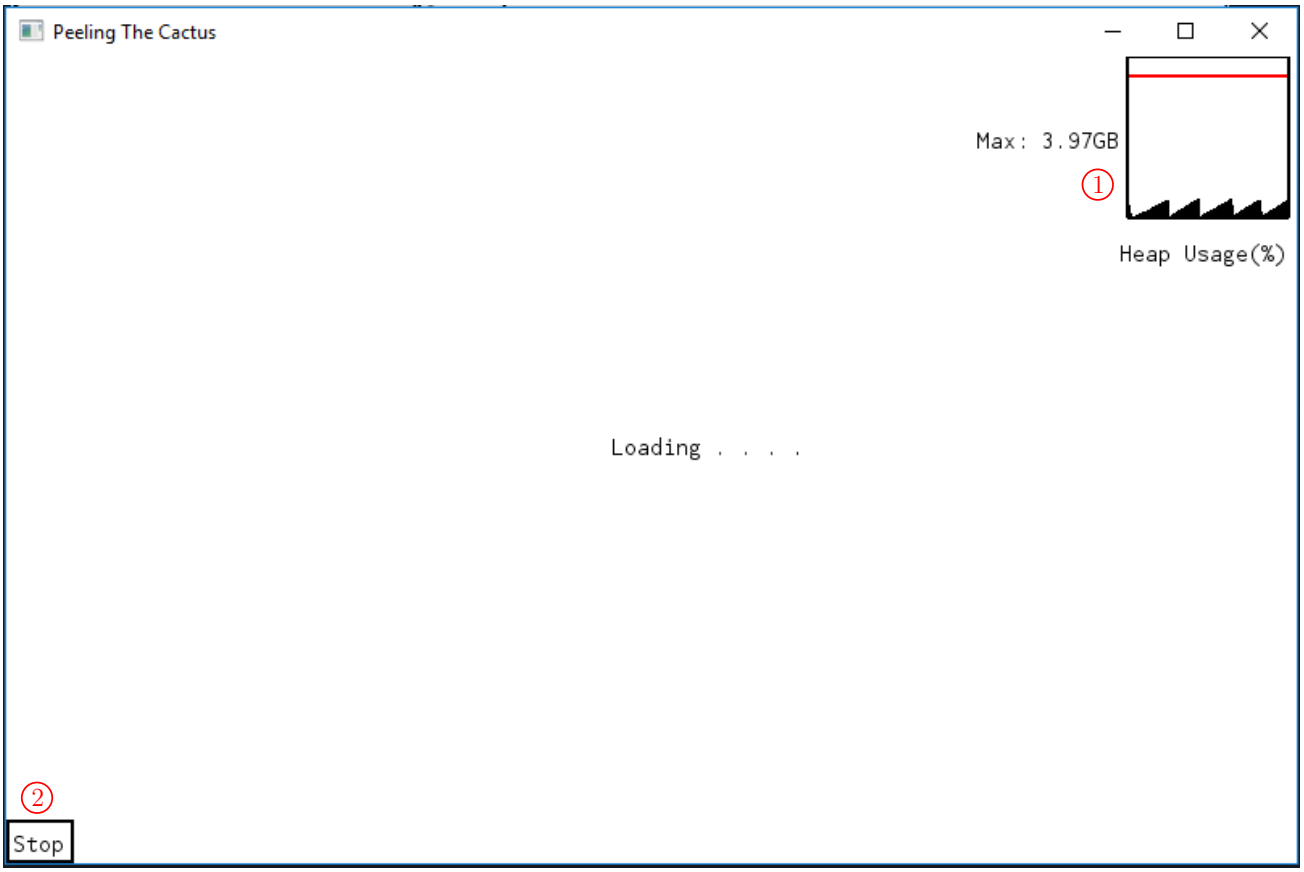
①: Text Window. This window will inform the user if the provided point set is not in general position.
②: Display Window. In this window the user can click and drag to move viewing window around, and use the scroll wheel to zoom in or out. The window will display the current point set.
③: Center Button. Resets the Display Window to default position and zoom.
④: Add/Remove Button. Transitions from the Verification State back to the Point Input State with the current point set.
⑤: Solve Button. Run's Marx and Miltzow's algorithm on the displayed point set.

Figure 32: A labeled image of the Verification State.

See Figure 32 for a labeled diagram. In this state the user can confirm that their point set is displayed correctly. Additionally, in this step the application checks the point set to see if it is in general position. If the point set is not, the application prompts the user with a relevant error message. In this case the user can use the "Add/Remove" button to return to the Point Set State and adjust their point set.

If the point set is in general position, the user can press "Solve" to run Marx and Miltzow's algorithm on the point set. Once "Solve" is pressed the application proceeds to the Loading State.

A.2.3 Loading State



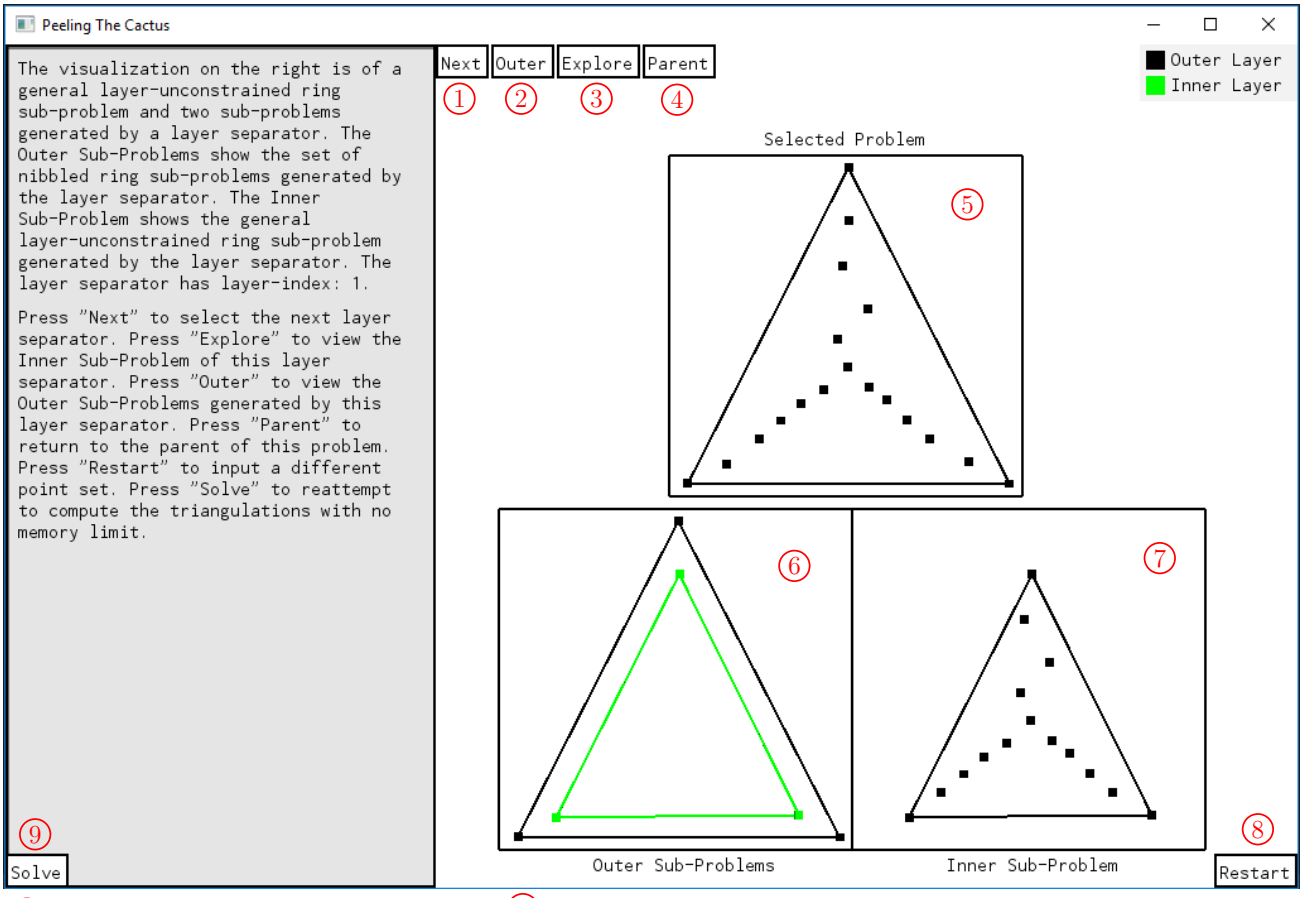
1: Heap Space Display. This can be used to monitor the current heap space usage. The window represents the heap space usage of this application over the last 300 frames. The red bar is the safety cut off limit. If the memory usage reaches the red bar the Stop Button is automatically pressed.

2: Stop Button. This button requests the algorithm to stop so the user can explore what it has managed to compute up to this point.

Figure 33: A labeled image of the Loading State.

See Figure 33 for a labeled diagram. In this state the user can monitor the application's memory usage and manually stop the algorithm before it is complete. Once in the loading state the application will remain in this state until the algorithm has terminated unless "Stop" is pressed. If the algorithm terminates, or "Stop" is pressed, the application proceeds to the General Layer-Unconstrained State. If stop is pressed then the algorithm is stopped prior to completion and only the generated sub-problems will be displayed in subsequent states.

A.2.4 General Layer-Unconstrained State



1. Next Button. Select the next layer separator.
2. Outer Button. Explore the outer ring problem.
3. Explore Button. Explore the inner ring problem.
4. Parent Button. Return to the parent problem.
5. Selected Problem.
   The currently selected parent problem.
6. Outer Ring Sub-Problems. The outer ring problem.
7. Inner Ring Sub-Problem / Layer Separator.
   The inner ring problem or the layer separator if no inner ring problem exists
8. Restart Button. Return to the Point Set State.
9. Solve Button. Reattempt to solve with no memory limit.
   Only present if algorithm was stopped prior to completion.

Figure 34: A labeled image of the General Layer-Unconstrained State. In this example, the algorithm was unable to terminate so no triangulation counts are provided.

See Figure 34 for a labeled diagram.

In this state the user can explore the sub-problems of the currently selected problem. When the application first reaches this state the selected sub-problem is the general layer-unconstrained ring problem with the same triangulations as the original point set.

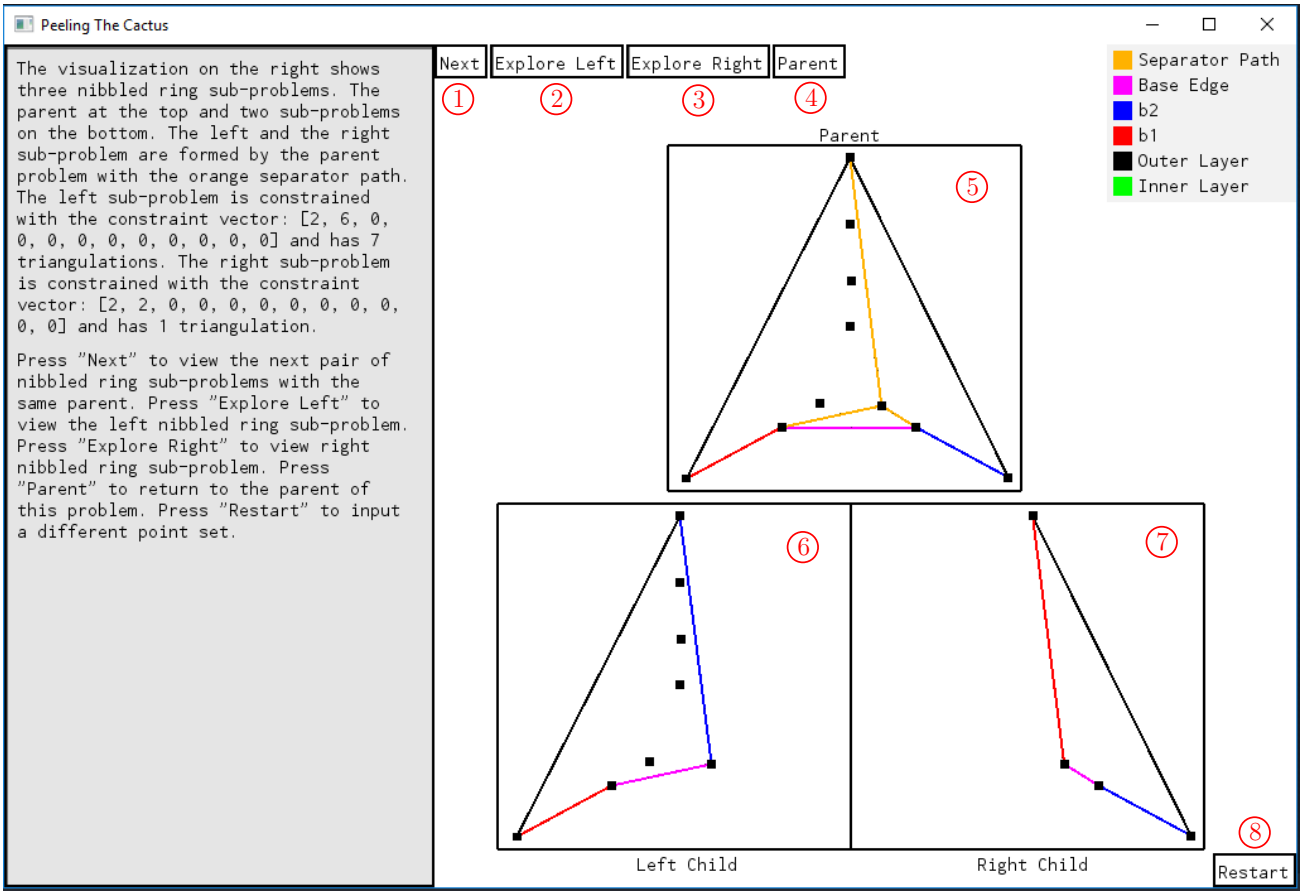
The user can press the “Explore” button to explore the general layer-unconstrained ring sub-problem of the currently selected layer separator (shown in green), if any. If there is a general layer-unconstrained ring sub-problem with this layer separator it is shown in the bottom left, otherwise the layer separator is shown instead.

The user can press the “outer” button to explore the outer layer ring sub-problems for the currently selected layer separator, if any.

The user can press the “Next” button to select the next layer separator. If there are no remaining layer separators, the first separator is selected instead. Finally, the user can press “Parent” to return to the parent problem of this sub-problem.

If a general layer-unconstrained sub-problem with no separator layers of size at least 3 is selected, the algorithm proceeds to the Nibbled Ring State instead. In this case, all sub-problem of the selected problem are nibbled ring sub-problems and can be viewed and explored in the Nibbled Ring State.

A.2.5 Nibbled Ring State



1. Next Button. Select the next pair of sub-problems.
2. Explore Left Button. Explore the left sub-problem.
3. Explore Right Button. Explore the right sub- problem.
4. Parent Button. Return to the parent problem.
5. Parent. The parent problem with the the separator path in orange.
6. Left Child. The left sub-problem.
7. Right Child. The right sub-problem.
8. Restart Button. Return to the Point Set State.

Figure 35: A labeled image of the Nibbled Ring State.

See Figure 35 for a labeled diagram.

In this state the user can explore the sub-problems of the currently selected nibbled ring sub-problem. The constraint vector of the selected sub-problem is displayed in the Display Window. If the algorithm was able to terminate, the number of triangulations of the selected sub-problem is also displayed.

The user can press the "Explore Left" or "Explore Right" button to explore the left and right sub-problems (respectively) of the currently selected separator path (shown in orange). The user can press the "Next" button to select the next separator path or, in the case that the parent of this sub-problem is a general layer-unconstrained ring problem, the next nibbled ring sub-problem of its parent. If there are no remaining sub-problems or separator paths, the first sub-problem or separator path (respectively) is selected instead. Finally, the user can press "Parent" to return to the parent problem of this sub-problem.

A.3 Advanced Options

Large Point Sets:

Large point sets (15+ points depending on your system) may be unreasonable to compute. However, the user can still view sub-problems of these point sets. If the user presses the "Stop" button while the application is loading then the application will complete its current sub-problem then stop and allow the user to view the sub-problems that were computed. If the algorithm is stopped in this way, no triangulation count can be computed

and many sub-problems may be unsolved.

Loading Points From a File:

The application supports loading from a file. When inputting points, the "load" button will replace all current points with the points in the file "pointset.txt". The file "pointset.txt" is located in the resources folder. The file must be formatted as follows: Each line contains a single point. A line consists only of the x-coordinate followed by a space followed by the y-coordinate.

In all cases the first point has order label 1 the second point has order label 2 and so on. Points do not need to be integers.

Solve Anyway:

When the application has used more than 85% of its allocated memory, it automatically stops the algorithm before it completes. It does this to prevent a majority of heap space related crashes. When this occurs or the algorithm is stopped manually via the "Stop" button it is still possible to attempt to continue solving the problem. Pressing the "Solve" button will resume the algorithm without the memory limit. In this case it is very likely that the application will encounter a heap space error and crash.

Allocating Additional Memory:

By default, Java allocates a maximum of 4GB of RAM to any Java application. If the application is started by running "CactusApplication.jar" 4GB of RAM will be allocated. To allocate additional memory, the file "run.bat" is provided. It runs the application with a desired maximum memory allocation.

To change the amount of memory allocated, right click on "run.bat" and select edit. This will open it as a text file. You will see one line: "java -Xmx8000M -jar CactusApplication.jar". The command -Xmx8000M allocates 8000MB, or 8GB, to the application. To change the amount of memory allocated change the number 8000 to the number of Megabytes you desire and save the file. Do not change anything else in the file. Double click on "run.bat" to launch the application with the desired allocated memory.

A.4 Frequently Asked Questions

Application launches and immediately crashes:

This can occur if the resources file is missing or if the files have not been properly unzipped. Ensure that the resources file is present and contains the files shown in Figure 30.

CactusApplication.jar is an unrecognized file type and prompts the user to select an application with which to open it:

This is the case if you do not have Java installed on your system. Install Java and try again.

I pressed solve and the application stopped responding:

If the point set is very large (20+ points) the first step may take a long time. The application may appear to be not responding, but it is still working. If the point set is not large, there may be something wrong with the installation or your system. Try reinstalling a fresh download of the application.

The memory indicator on the loading screen has flattened out, has the program stopped working?

The memory usage will flatten out when the vast majority of newly computed sub-problems are already in the dynamic programming database. This is expected behavior, it is still working as intended.

I solved a point set, but it immediately shows me nibbled ring sub-problems, where are the general layer-unconstrained ring sub-problems?

If the point set is too small to possibly generate layer separators with inner ring problems (less then 12 free points) then the application will skip showing the layer separator state since there are no layers with general layer-unconstrained ring sub-problems of their own. In this case each layer separator is an internal layer of a nibbled ring problem and has no internal components. Pressing "next" will allow the user to see the nibbled ring sub-problems for each layer separator.

If the user's system is not powerful enough to solve a point set of adequate size, then they can input a larger point set and stop it before it completes. In this way the user can explore the general layer-unconstrained ring sub-problems.

I pressed "load" and the application crashes:

If the file "pointset.txt" has unexpected characters the application will crash when it attempts to read it. Please ensure that the file has the correct format.

I pressed "load" but nothing happened:
If the file "pointset.txt" is missing, renamed, or empty nothing will happen when the "load" button is pressed. Ensure that the file is present and contains a point set in the correct format.

The application crashes when I press "solve", even for small point sets:
This is a known issue on some Windows machines. Using the .bat file to run the application generally solves this problem.