Thinnest V-shapes, Biggest Angles, and Lowest Potentials

DISSERTATION

Submitted in Partial Fulfillment of

the Requirements for

the Degree of

DOCTOR OF PHILOSOPHY (Computer Science)

at the

New York University
Tandon School of Engineering

by

Mark V. Yagnatinsky

May 2016

Approved:

Department Chair Signature

Date

University ID:	**N18183721**
Net ID:	**mvy204**

Approved by the Guidance Committee:

Major: Computer Science

Boris Aronov
Professor of Computer Science

Date

John Iacono
Professor, Computer Science and Engineering

Date

Lisa Hellerstein
Professor, Computer Science and Engineering

Date

Pat Morin
Professor, Carleton University

Date

Microfilm or copies of this dissertation may be obtained from:

UMI Dissertation Publishing
ProQuest CSA
789 E. Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106-1346

Vita

Mark V. Yagnatinsky was born in Brooklyn on March 25, 1987. He started his undergraduate education at Polytechnic University in September 2005 and in January 2010 he graduated with a BS in CS from the Polytechnic Institute of New York University, because in 2008 the school changed names on him. In September of 2010 he came back to Polytechnic to pursue a PhD in computer science, and before too long the school changed names on him again, first to New York University Polytechnic School of Engineering in 2014, and then to New York University Tandon School of Engineering in 2015. His goal as he writes this paragraph (in December) is to finish the thesis by February 2016 or thereabouts, so he can graduate before the name changes yet again.[1]

His research has been supported by NSF Grants CCF-11-17336 and CCF-12-18791, NSA MSP Grant H98230-10-1-0210, and GAANN Grant P200A090157.

[1]It's now May 2016. He is almost done. He's been saying that for a few days.

Acknowledgments

I would like to thank my mom, without whose help I'd never have finished, and my dad, for reminding me to not neglect my health (though he may think I don't listen). I'd also like to thank those who made my PhD years at Poly a joy, when they might otherwise have been a drag: Muriel, Pooya, Linda, Devorah, Trishank, Hossein, Oz, Patrick, Khadijeh, Dimitrios, Anirudh, and Sarah. Thank you to John Sterling, without whose encouragement I'd never applied in the first place. Thank you to Florin, for that all-nighter of productivity surpassing that of a typical "good week". Thank you to John Iacono, for trying to instill in me some measure of discipline.

And thank you to Boris Aronov, for more than words can say. Thank you for being who you are.

Mark Yagnatinsky
May 10, 2016

To my sister, Doreen.

ABSTRACT

Thinnest V-shapes, Biggest Angles, and Lowest Potentials

by

Mark V. Yagnatinsky

Advisor: Boris Aronov

Co-Advisor: John Iacono

Submitted in Partial Fulfillment of the Requirements for

the Degree of Doctor of Philosophy (Computer Science)

May 2016

This thesis addresses two algorithmic questions in computational geometry, and presents a partial asymptotic analysis of a widely used data structure whose asymptotic behavior is not fully understood.

The first algorithmic question is motivated by curve reconstruction: given a set of points sampled from a curve in the plane, find a shape approximating the original curve. Aronov and Dulieu have suggested that near a sharp turn, it makes sense to model the curve by a V-shape. The authors remark that it would be natural to investigate a variant that can handle a small number of outliers, to accommodate a few bad data points. We show how to find the thinnest V-shape that encloses all but k points in a given point set, for any specified integer k.

The second algorithmic question is motivated by mesh refinement: in applications one often needs a triangulation of a point set in the plane and the triangulation should be as "nice" as possible. One common niceness criterion is avoiding long skinny triangles, and a common way of formalizing that into an optimization target is to say that the smallest angle in the triangulation should be as big as possible (and ties are broken by making the second smallest angle as big as possible, etc). The Delaunay triangulation has precisely this property and is unique for a given point set as long as it is in general position. Unfortunately, the smallest angle in a Delaunay triangulation can be arbitrarily small. Sometimes, it is acceptable to introduce extra points, so as to get a better triangulation. However, it is desirable to avoid introducing too many, because they increase the memory and time requirements of all algorithms that operate on the triangulation. Here, we investigate how to find the best place to add one point to a point set so as to maximize the smallest angle in the Delaunay triangulation of the combined point set. We will present a small zoo of algorithms for this problem.

In the second part of the thesis, we analyze is the pairing heap, a data structure for implementing priority queues introduced in the mid-1980s. The inventors conjectured that pairing heaps and Fibonacci heaps have the same amortized complexity for all operations. While this was eventually disproved, pairing heaps are, unlike Fibonacci heaps, both easy to implement and also fast and are thus widely used in practice, while their asymptotic performance is still not fully understood. Here, we show that a short sequence of operations on a pairing heap takes less time than was previously known.

Contents

List of Figures

Chapter 1

Introduction

1.1 Organization of the thesis

This thesis is divided into three essentially independent pieces. In Part I, we present two algorithmic results and in Part II we present an amortized analysis of pairing heaps. Both algorithmic questions are instances of a large class of problems known as *geometric optimization* questions; see [AS98] for a survey. While a full survey is beyond the scope of this thesis, below we will look at some tools from the geometric optimization literature which were used to obtain the results here. (Not all these techniques are unique to computational geometry; for instance "characterizing the solution" is used practically everywhere.)

Later in this chapter, we will review amortized analysis, the implementation of pairing heaps, and some related previous work.

1.2 A mini-survey of geometric optimization

A geometric optimization problem is, loosely speaking, a type of optimization problem where the input is a collection of geometric objects. As a simple example, one might have a set of points in the plane and wish to find the closest pair. Alternatively, one might have a set S of points to preprocess and wish to find the closest point in S to a given query point.

One common ingredient in many geometric optimization algorithms is characterizing the solution. This may involve showing that the space of locally optimal solutions has lower dimension than the space of all solutions (for instance, maybe some point must lie on a circle rather than at an arbitrary location in the plane), or perhaps that the optimum solution must be one of a finite set of candidates. Very often, a characterization leads directly to a trivial brute-force algorithm, and further algorithmic improvements amount to tweaking the brute-force algorithm to make it less wasteful. (Often, these improvements are enabled by better characterizations.) As a silly example, consider finding the convex hull of a set of points in the plane. This can be cast (admittedly somewhat unnaturally) as an optimization question: find the smallest convex shape enclosing all the points. As phrased, there does not seem to be any obvious algorithm for this problem, even a brute force one. Once one realizes that the hull must be a polygon with vertices selected from the input points, a brute force algorithm immediately suggests itself, as there are only a finite number of such polygons (albeit far from polynomial). After a bit more thought, one has the well-known gift wrapping algorithm. Characterizing the solution is at the heart of the V-shape algorithm in Chapter 2 and the fastest algorithm for placing a point in Chapter 3. (Indeed, for the case of V-shapes, the characterization alone is enough to yield a polynomial-time algorithm, and most of the work goes into reducing that polynomial from $O(n^7)$ to something closer to quadratic.)

Randomized algorithms are not particularly rare elsewhere, but in computational geometry they are seemingly universal: any sufficiently famous problem seems to have at least one. Both the V-shape algorithm and the fastest point-placement algorithm rely directly on random sampling, and some of the other point-placement algorithms rely on it indirectly by using black boxes that themselves use randomization. The V-shape algorithm uses randomization in a different way than the point placement one. In the case of V-shapes, there is a large number of locally optimal V-shapes, and we wish to only look at those with k outliers. Using random sampling, we can look at all V-shapes with k outliers (with high probability), even though we don't know how to enumerate them all deterministically without losing a factor of n in the running time. (It may well be possible to do so; using the Clarkson-Shor technique, it can be shown that the number of V-shapes we sample is proportional to the number of V-shapes with at most k outliers, rather than exactly k.) On the other hand, the point-placement algorithm uses random sampling in essentially the same way that quicksort does: to evenly partition a space where most partitions are good but many are not.

Finally, we mention generalized linear programming, which will be useful when we discuss one of our point-placement algorithms in Section 3.4. A linear program is a type of optimization problem. It is specified by a set of n linear inequalities in d variables. The goal is to find values of the variables that satisfy all the inequalities while maximizing a specified linear function. Linear programming has the following nice property: there exist d inequalities, such that the optimum for just those is the optimum for all n. An LP-type problem is a generalization of linear programming where instead of linear inequalities, there are black-box constraints on the solution, and a black box objective function. As long as the constraints satisfy some basic properties, such as the analog of the property just mentioned for linear programs (there exists a set of d constraints such that the optimum for them is the optimum for all), then there are algorithms that can solve such problems, and furthermore, the algorithms run in time $O(n)$ for any constant d. If d is a small constant, such as three, they are fast in practice as well as theory.

To give the reader a small taste of LP-type problems, we present two problems; the first is an LP-type problem and the second is not. This is: given a set of points in the plane, find the smallest circle that covers all of them. The smallest circle for a point set is defined by two or three points from that set: the points that lie on its boundary. (Note that for a point set in general position, no four points lie on the same circle.) Even if the rest of the point set is removed, the circle can shrink no further. Therefore the optimum for the complete point set is the optimum for the set of just three points. Thus, $d = 3$. The V-shape problem which we will discuss in the next section is not an LP-type problem, although superficially it looks like it might be; see the next section for details.

1.3 V-shapes

Motivation. The motivation for this problem comes from curve reconstruction: given a set of points sampled from a curve in the plane, find a shape approximating the original curve. It has been suggested in [AD11] that in an area where the curve makes a sharp turn, it makes sense to model the curve by a *V-shape*; see Figure 2.1. The authors remark that it would be natural to investigate a variant that can handle a small number of outliers, to accommodate a few bad data points. We investigate that variant here.

Previous work. In [AD11], the authors develop an algorithm for covering a point set in general position with a V-shape of minimum width (allowing no outliers) that runs in $O(n^2 \log n)$ time and uses quadratic space. (We use the same general position assumptions as [AD11]: no vertical line goes through two points, no three points are collinear, and no lines defined by pairs of points are parallel.) They also find a constant-factor approximation algorithm with running time $O(n \log n)$, and a $(1 + \varepsilon)$-approximation algorithm with a running time of $O((n/\varepsilon) \log n + n/(\varepsilon^{3/2}) \log^2(1/\varepsilon))$, which is $O(n \log n)$ for a constant $\varepsilon > 0$.

Our Result. We describe a randomized algorithm that, given n points and an integer $k > 0$, finds the minimum-width V-shape enclosing all but k of the points with probability $1 - 1/n^c$ for any $c > 0$, with expected running time $O(cn^2k^4 \log n(\log n \log\log n + k))$, or $\tilde{O}(cn^2k^5)$, where $\tilde{O}(\cdot)$ is meant to suppress polylogarithmic terms.

This result is joint work with B. Aronov, Ö. Özkan, and J. Iacono [AIÖY13].

Remark. Even though a locally optimal V-shape (without outliers) is defined by at most six points on its boundary, this is not enough for it to be an LP-type problem (at least not one with $d = 6$). Indeed, if we take the globally optimal V-shape and remove some points not on its boundary from the point set, the optimal V-shape may change.

1.4 Placing a point to maximize angles

Often in applications one needs a triangulation (or more generally a mesh) of a point set in the plane, and the triangulation should be as "nice" as possible. (See for instance [BCKO08, introduction of Chapters 9 and 14].) What counts as "nice" varies with the application, but one common desire is for the triangles produced to be fat, rather than long and skinny. (See again [BCKO08, Sections 9.6, 14.1, and 14.4].) The ideal in this case is a triangulation composed of equilateral triangles, but this is usually impossible. However, it is also overkill, since it often suffices that all angles are in a Goldilocks range of "not too big and not too small"; say, between 30 and 120 degrees.

One common way to formalize the wish for fat triangles is to ask for the smallest angle in the triangulation to be as big as possible [O'R98, Section 5.5.2]. It is well known that the Delaunay triangulation of a planar point set has precisely this property [BCKO08, Theorem 9.9]. Unfortunately, the smallest angle in a Delaunay triangulation can be arbitrarily small. Sometimes, it is acceptable to introduce extra points, known as *Steiner points*, so as to get a better triangulation [BCKO08, Section 14.1]. However, it is desirable to avoid introducing too many, because they increase the memory and time requirements of all algorithms that operate on the triangulation. There are two natural versions this problem. One is: if we want all angles to measure at least z degrees, how many additional points do we need? The other is: given a budget of k points, how large can we force the smallest angle to be? The fixed-budget question was addressed in [AAF07], which presented an algorithm that, given a point set P, finds the best placement of k additional points $q_1, \ldots, q_k$, so that the minimum angle in the Delaunay triangulation of $P \cup \{q_1, \ldots, q_k\}$ is maximized. In fact, the problem they solved was slightly more general, in that the input also included a set C of non-crossing line segments with endpoints in P, which must appear as edges in the final triangulation. These are sometimes called *constrained edges*, or simply *constraints*, and the resulting triangulation is called a *constrained Delaunay triangulation*. This generalization allows one, for example, to triangulate a simple polygon, by specifying the polygon boundary as the set of mandatory edges, and more generally handle real-world applications with boundary conditions.

Unfortunately, the running time of the algorithm in [AAF07] is $n^{\Theta(k)}$, because it relies on explicit construction of high-dimensional arrangements. They also present an algorithm for the case $k = 1$, which runs in time $O(n^{4+\varepsilon})$. In Chapter 3 of this thesis, we focus on improving the case $k = 1$, and develop several asymptotically faster algorithms, at least one of which is simpler than that of [AAF07]. (We say at least one is simpler. It may be that two of them are simpler, but simplicity is subjective and the second is a borderline case.) We also ask whether these running times are likely to happen in practice, or whether realistic inputs (as opposed to adversarial ones) are unlikely to trigger them.

The algorithms described here are joint work with Boris Aronov. An early version of the LP Algorithm was published in the 2013 proceedings of the Canadian Conference on Computational Geometry [AY13a]. That version could not handle constraints; a variation that could was later uploaded to arXiv [AY13a]; we describe this variant in Section 3.4. The Envelope Algorithm (Section 3.5) was presented at the 2013 Fall Workshop on Computational Geometry [AY13b]. Finally, the Feasible Region Algorithm was published in the proceedings of the 2014 Canadian Conference on Computational Geometry [AY14], but without the input-sensitive analysis that we present in Section 3.6.

1.5 A potential function for pairing heaps

The last chapter of this thesis will be devoted to an amortized analysis of pairing heaps. We go through some background in this section.

We will first try to formulate a definition for the term *data structure* that matches the everyday understanding of the term. After that, we do the same for abstract data types. We then review amortized analysis in general and potential functions in particular. Afterwards, we go over the abstract data type known a *priority queue* and a family of data structures known as *heaps* for implementing them. Finally, we will discuss a specific type of heap called a *pairing heap*, which will be the focus of the last chapter of this thesis.

1.5.1 Basics first: what on Earth is a data structure?

Niklaus Wirth famously titled one of his books *Algorithms + Data Structures = Programs* [Wir76]. Even people who are not computer scientists know what a program is. Those who have taken even a single programming class have an intuitive idea of what an algorithm is, and even an idea of the difference between an algorithm and a program. If an intuitive idea does not suffice, there are various formalisms (such as Turing machines, lambda calculus, real RAM, or assembly language) that say essentially “anything that you can express in this notation is an algorithm.” For example, *factorial* is not an algorithm, but merely a function, and there exist several algorithms for computing this function: multiply the numbers in increasing order, in decreasing order, or multiply the odd numbers and the even numbers separately and then multiply the results together. Likewise, *sort* is a function, for which there are many algorithms, such as selection sort, merge sort, and heap sort.

There does not seem to be any such formal definition of a data structure. Instead, people typically "know one when they see it." In practice, this seems to cause few if any problems, and I've only gotten into an argument on this topic once in my life. However, that one argument has given me a reason to be cautious now, as I attempt a very vague and informal definition. Although this is one reason to be cautious, it is not the primary one. The primary reason is that even a definition at first sounds so vague as to be nearly vacuous, turns out to nevertheless have counterexamples, in the form of things that are widely considered to be data structures, but do not meet the definition. For instance consider: "A data structure is a way of arranging data in RAM, that allows information to be put in by performing operations called updates, and allows that data to be retrieved by performing operations called accesses." But some data structures (such as B-trees [BM72]) are designed to work on disk instead of main memory, so we should get rid of the "in RAM" clause. Others (such as Bloom filters [Blo70]) do not support retrieving the data that was used to create it. And some data structures do not support efficient updates, so that if the data changes, the simplest course of action is to run an algorithm to rebuild the structure from scratch. Our definition has been reduced to "a way of arranging data," which will not enlighten anyone who does not already know what a data structure is, which limits its usefulness as a definition.

There is a further problem we've been glossing over: the term "data structure" actually has two distinct meanings, which are not always carefully distinguished in everyday speech, but which will be vital, if we are to avoid circular definitions while staying sane. One meaning simply describes the layout of the data. For instance, a red-black tree [GS78] is a binary search tree that satisfies certain balance properties, and some more properties about node color. Indeed, if you look in a data structures textbook, the first thing you will see under the heading for red-black trees is probably an enumeration of these properties. But this takes up a relatively small part of the section, because the interesting thing about red-black trees (and indeed the reason why the colors are useful) is that there exist efficient algorithms for *maintaining* these balance (and color) properties while doing something useful. These are also part of what defines a red-black tree. Note that the two definitions are not even in the same grammatical equivalence class. For instance, in the first case, it makes sense to talk about plurals: it is perfectly reasonable to use two red-black trees in the same program. But in the second case, it does not: there is only one standard algorithm for deleting a node from a red-black tree. In common usage it is not rare to conflate the two meanings, sometimes even within a single sentence, such as "red-black trees support merging in linear time," which really means: "given two red-black tree instances, there is an algorithm that can combine them into one tree in time proportional to their combined size."

Let's try again. As for the layout aspect, we simply say that it is a set of invariants that the contents of a region of storage must satisfy. In the case of a red-black tree, this is simply the invariant that it must look like a red-black tree, as opposed to an unbalanced search tree, or binary heap, or a linked list. And for the algorithmic aspect, we say that there is a set of algorithms that take an instance of such a storage region, and either modify it while maintaining the invariant, or compute something useful from its current state.

This definition still has problems, and we won't attempt to fix most of them. Some are easy, such as the fact that this doesn't allow for initializing the data structure, or that some algorithms need two or more instances (such as the merging algorithm mentioned above). Others are trickier, such as deciding which operations really define a data structure, and which it merely happens to support. If it takes a few decades to discover an algorithm to efficiently merge two Sparkling binary search trees, is that one of the defining algorithms or not? Does the answer depend on whether it is implemented in terms of things that are clearly definitional primitives, such as insert and delete? Does it matter if at first it seems not to be, but it turns out that it can be rephrased so that it is, at no loss of efficiency?

Instead of addressing these problems, let's look at a few examples. Consider first the array. It is among the oldest of data structures and has built-in support in every popular programming language. Its layout is simply a sequence of data items arranged contiguously, and the two supported operations are retrieving the ith item, and setting the ith item. We've already described red-black trees layout; the operations supported include insertion and deletion of an item. Bloom filters gave our last definition trouble, but now the layout is just an array of bits, and the operations are insertion of an item, and querying for an item's presence.

1.5.2 Abstract data types

The primary use for data structures is in the design of algorithms, and in particular, for implementing abstract data types. A data structure is to an *abstract data type* (or ADT) what an algorithm is to a mathematical function. That is, the details of the layout are suppressed, and the algorithms to maintain the invariants are not mentioned, leaving only the functions. For example, an array and a doubly-linked list can both be used to implement the *sequence* ADT, which is just an ordered list of values. As another example, dynamic arrays are very commonly used to implement the *multi-set* ADT. Adding an item to the set is an append to the array, a search for an item involves a linear scan, and so on.

The advantage of phrasing algorithms in terms of abstract data types is twofold. First, it enables a higher-level description of an algorithm, which helps to keep the whole algorithm in your head at once, or at least larger pieces of it. Second, it makes it more modular, so that it is often possible to substitute a data structure with better performance for the problem at hand, without changing the high-level description at all. This works even at the level of source code instead of pseudo-code: in a well-designed library, a switch from one implementation of a sequence to another can often be accomplished by changing a single line of code. If a faster implementation of an ADT is discovered, then this automatically improves the speed of all algorithms that make use of it. (Of course, it only speeds up algorithms and not programs; to speed up a program one needs to actually change that one line of code, as well as code up the faster implementation.) As a real-life example, Dijkstra did not originally phrase his shortest path algorithm in terms of abstract data types, and in that formulation it took time proportional to the square of the number of nodes in the graph; for a dense graph this is optimal, but for a sparse graph it is not. If we recast his algorithm in terms of a *priority queue*, then his original implementation used a priority queue backed by an arbitrarily ordered list. By using a binary search tree or some kind of a heap, it is possible to achieve good running times even for sparse graphs.

Some abstract data types are esoteric, in that they are used by only a few algorithms and have very few data structures that implement them, but there are a few (such as sequences, sets, stacks, queues, and priority queues) that arise again and again, in many diverse algorithms. Since they are used so often, there has been much interest in developing data structures that improve on the best known implementations of widely used ADTs, and analyzing those data structures that are already in wide use to better understand their performance, and perhaps find variants that overcome known weak spots.

1.5.3 A review of amortization and potential functions

Some applications of data structures require that the worst-case time of operations be bounded, even at the expense of the average case. For instance, a text editor that takes an average of a millisecond to respond to a user's keystrokes will not be popular if a keystroke occasionally takes a minute to process. Far better that the average be increased to two milliseconds, if by doing so we can guarantee that no keystroke will take more than four. (Note: any connection to reality that these numbers have is utterly accidental.)

But many applications of data structures are in batch algorithms, and in that context, the worst-case runtime of any particular operation on the data structure doesn't matter. An algorithm performs not one operation, but a sequence of them, and it is the runtime of the entire sequence that matters.

Consider a sequence of operations on some data structure, and let t_i denote the time to execute the ith operation in this sequence. The goal of amortized analysis is to assign to each operation an *amortized time* a_i, such that the sum of the amortized times is at most the sum of the actual times. There is of course an easy way to achieve this, by setting $a_i = t_i$. The real goal is to come up with times that are somehow more convenient, usually by showing an amortized time that is a function of the current size of the structure, but not its detailed internal state.

A key tool in amortized analysis is the *potential function.* Given a sequence of m operations $o_1, o_2, \ldots, o_m$ executed on a particular data structure, and an integer $i \geq 0$, a potential function is a function Φ that maps i to a real number Φ_i. The real number Φ_i is called the *potential* after the ith operation. Given Φ, we define the amortized time a_i of an operation o_i as the actual time t_i of the operation, plus the change in potential $\Phi_i - \Phi_{i-1}$. Often, instead of explicitly providing the mapping from integers to reals, a potential function is specified implicitly: given the current state of the data structure, there is an algorithm to calculate the current value of the potential function, without needing to know how many operations have taken place. Another common approach (which we use in combination with the previous one) is to specify Φ_0, and then specify how to update Φ after an operation.

Observe that since $a_k = t_k + \Phi_k - \Phi_{k-1}$, we have $t_k = a_k - \Phi_k + \Phi_{k-1}$. Thus, the total time taken for the subsequence of consecutive operations from

i to j is

$$\begin{aligned}
\textstyle\sum_{k=i}^{j} t_k &= \textstyle\sum_{k=i}^{j}(a_k - \Phi_k + \Phi_{k-1}) \\
&= \textstyle\sum_{k=i}^{j} a_k - \sum_{k=i}^{j} \Phi_k + \sum_{k=i}^{j} \Phi_{k-1} \\
&= \textstyle\sum_{k=i}^{j} a_k - \sum_{k=i}^{j} \Phi_k + (\Phi_{i-1} + \sum_{k=i+1}^{j} \Phi_{k-1}) \\
&= \textstyle\sum_{k=i}^{j} a_k - \sum_{k=i}^{j} \Phi_k + \Phi_{i-1} + \sum_{k=i}^{j-1} \Phi_k \\
&= \Phi_{i-1} - \Phi_j + \textstyle\sum_{k=i}^{j} a_k.
\end{aligned}$$

From this formula, we can derive the motivation for our result: namely, that a potential function with a small range is useful. (When we speak of the range of a potential function, we mean its maximum value as function of n minus its minimum value. Often the minimum is zero, and thus the range is simply the maximum value.) To see this, suppose you have a data structure with n elements, and you perform a sequence of k operations on it that don't change the size. If each operation takes $O(\log n)$ amortized time, then the total actual time is bounded by $O(k \log n)$ plus the loss of potential. Thus, if the range of the potential function is $O(n \log n)$, then the total time is $O(k \log n + n \log n)$, but if the range of the potential function is linear, this is improved to $O(k \log n + n)$, which is asymptotically better whenever k is $o(n)$. Thus, a reduced range of a potential function improves the time bounds for short sequences that don't start from an empty data structure.

1.5.4 Heaps and priority queues

A priority queue is an abstract data type that maintains a totally ordered set under the following operations:

$Q =$ **make-pq**(): Create an empty priority queue Q.

$p =$ **insert**(Q, x)**:** Insert key x into Q, and return a handle p which can be passed to decrease-key().

$x =$ **get-min**(Q)**:** Return the minimum key currently stored in Q.

delete-min(Q)**:** Delete the minimum key currently stored in Q.

decrease-key(p, y)**:** Decrease the key at p to y. Precondition: y must be strictly less than the original key.

Priority queues come up in such a wide variety of applications that they are covered in (nearly?) all data structure textbooks. As already mentioned, Dijkstra's shortest path algorithm (and its close relative, Prim's algorithm for minimum spanning trees) relies on a good priority queue for efficiency on sparse graphs. As another example, the heapsort sorting algorithm (insert all items into a heap, then remove them one by one) is an efficient in-place sorting algorithm given the right priority queue. (Specifically, when using a binary heap.)

The simplest implementation of a priority queue is probably a singly linked list. Initializing a new heap is trivial, and inserting a new item is simply an append, which takes constant time. If we return a pointer to the node an item is stored in, we have a suitable handle to perform decrease-key in constant time. Unfortunately, finding the minimum item takes linear time. Likewise, deleting the minimum item also takes linear time, since we must first find it. If we modify the insert operation to maintain the current minimum, then the current minimum can be found in constant time, but deletion still takes linear time, because we then have to find the *new* minimum. For this reason, this implementation is rarely used if the priority queue has any chance of being the algorithmic bottleneck.

Another approach is to use a balanced binary search tree. Its advantage over using a linked list is that deletion of the minimum now takes only logarithmic time, instead of linear. The disadvantage is that insertion slows down to logarithmic, as does decrease-key. Furthermore, the work required to implement a binary search tree noticeably exceeds that needed for a linked list or array.

The problem is that a binary search tree is trying too hard: it must be ready to support finding an arbitrary element at all times, despite the fact that we will only ever ask it for the smallest. Meanwhile, the unordered list has the opposite problem of trying too little. It turns out that instead of a search tree, it helps to store the set of items in a *heap*. There are many varieties of heaps, but they all have the following property in common. Like binary search trees, they store the elements of the set in nodes of a rooted tree (or sometimes a forest of such trees). Unlike search trees, heaps have the invariant that all children of a node have larger key values than their parent.

1.5.5 Pairing heaps

A *pairing heap* [FSST86] is a heap-ordered general rooted ordered tree. That is, each node has zero or more children, which are listed from left to right, and a child's key value is always larger than its parent's. The basic operation on a pairing heap is the *pairing* operation, which combines two pairing heaps into one by attaching the root with the larger key value to the other root as its leftmost child. For the purposes of implementation, pairing heaps are stored as a binary tree using the leftmost-child, right-sibling correspondence. That is, a node's left child in the binary tree corresponds to its leftmost child in the general tree, and its right child in the binary tree corresponds to its right sibling in the general tree. In order to support decrease-key, there is also a parent pointer which points to the node's parent in the binary representation. Priority queue operations are implemented in a pairing heap as follows:

make-heap(): return null

get-min(H): return H.val

insert(H, x): create new node n containing x; If the root H is null, then n becomes the new root; if H is not null then pair n with H and update root; return pointer p to the newly created node.

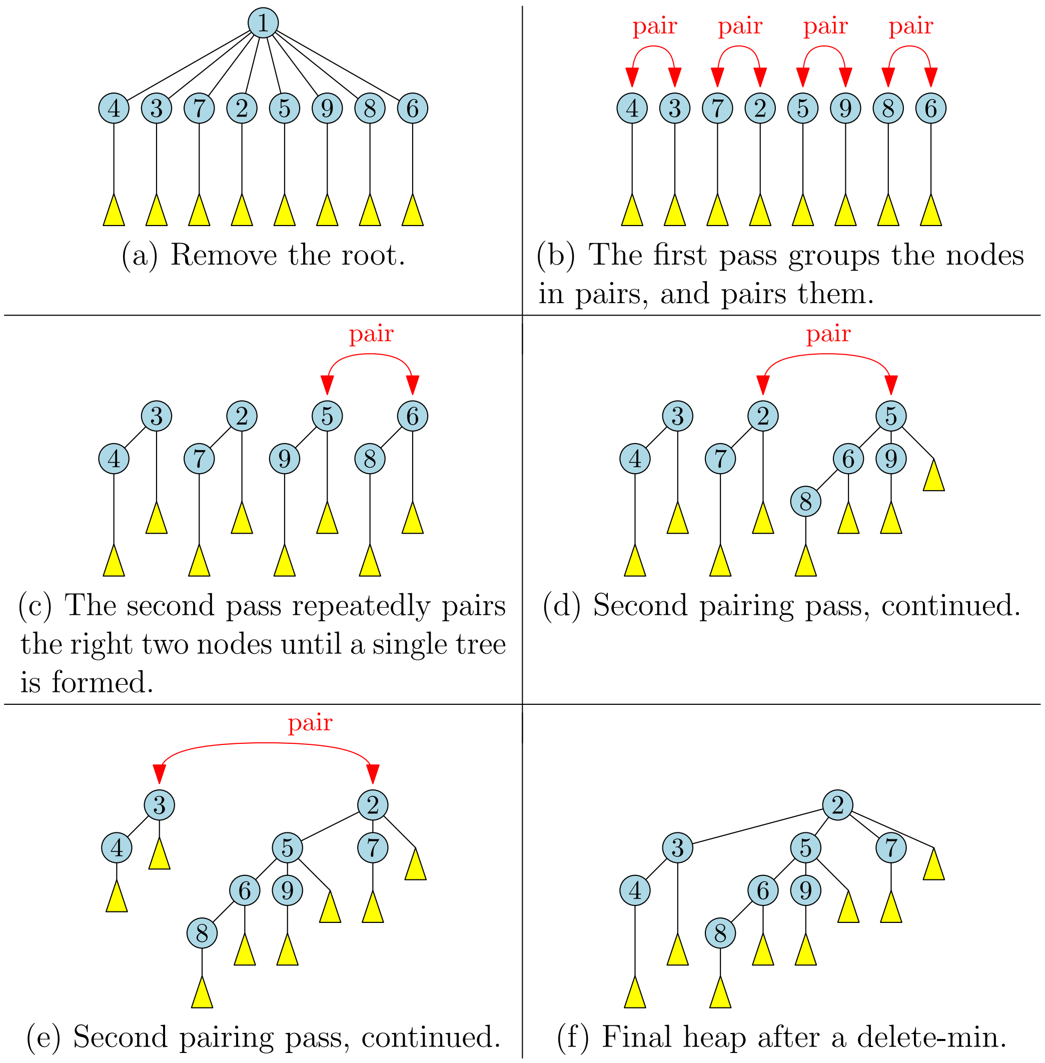
decrease-key(p, y): Let n be the node p points to. Set the value of n's key to y, and if n is not the root, detach n from its parent and pair it with the root

delete-min(H): remove the root, and then pair the remaining trees in the resultant forest in groups of two. Then incrementally pair the remaining trees from right to left. Finally, return the new root. See Figure 1.1 for an example of a delete-min executing on a pairing heap. (Readers familiar with splay trees may notice that in the binary view, a delete-min resembles a splay operation.)

All pairing heap operations take constant actual time, except delete-min, which takes time linear in the number of children of the root.

History

Pairing heaps were originally inspired by splay trees [ST85]. Like splay trees, they are a self-adjusting data structure: the nodes of the heap don't store any information aside from the key value and whatever pointers are needed to traverse the structure. This is in contrast to, say, Fibonacci heaps [FT84], which store at each node an approximation of that node's subtree size. Fibonacci heaps support delete-min in logarithmic amortized time, and all the other heap operations in constant amortized time. However, they are complicated to implement, somewhat bulky, and therefore slow in practice [SV86]. Pairing heaps were introduced as a simpler alternative to Fibonacci heaps, and it was conjectured that they have the same amortized complexity for all operations, although [FSST86] showed only an amortized logarithmic bound for insert, decrease-key, and delete-min. The conjecture was eventually disproved when it was shown that if insert and delete-min both take $O(\log n)$ amortized time, an adversary can force decrease-key to take $\Omega(\log \log n)$ amortized time [Fre99].



(a) Remove the root.

(b) The first pass groups the nodes in pairs, and pairs them.

(c) The second pass repeatedly pairs the right two nodes until a single tree is formed.

(d) Second pairing pass, continued.

(e) Second pairing pass, continued.

(f) Final heap after a delete-min.

Figure 1.1: Delete-min on a heap where the root has eight children.

Present

Nevertheless, pairing heaps are fast in practice. For instance, the authors of [LST14] benchmarked a variety of priority queue data structures. They also tried to estimate difficulty of implementation, by counting lines of code, and pairing heaps were essentially tied for first place by that metric, losing to binary heaps by only two lines. Despite (or rather because of) their simplicity, pairing heaps had the best performance among over a dozen heap variants in two out of the six benchmarks. In one of the benchmarks in which pairing heaps did not come in first, they were within ten percent of the performance of the heap which did, and in two others, they were within a factor of two of the best. The one benchmark in which they did poorly was pure sorting (add everything to the heap and then remove it), where they were over four times slower than the fastest heap. (Although it might be worth pointing out that the heap that won that benchmark does not support decrease-key in sub-linear time at all, so the comparison is not quite apples-to-apples, especially since it is possible to save one pointer per node in a pairing heap if you know you won't perform decrease-key. Pairing heaps were only thirty percent slower in the sorting benchmark than the fastest heaps that did support decrease-key.)

1.5.6 Previous work and our result

In [Col00, Theorem 3], Cole develops a linear potential function for splay trees (that is, the potential function ranges from zero to $O(n)$), improving on the potential function used in the original analysis of splay trees, which had a range of $O(n \log n)$ [ST85]. As explained above, this allows applying amortized analysis over shorter operation sequences.

There are several variants of pairing heaps such as [IÖ14] and [Elm09a], and one of them also has a potential function that is $o(n \log n)$ [IÖ14]. The main theme in all the variants is to create a heap with provably fast decrease-key, while maintaining as much of the simplicity of pairing heaps as possible.

Result	Range	Insert	Decrease-key	Delete-min
Pairing heap [FSST86]	$\Theta(n \lg n)$	$O(\lg n)$	$O(\lg n)$	$O(\lg n)$
Pairing heap [Pet05]	$O(n \cdot 4^{\sqrt{\lg \lg n}})$	$O(4^{\sqrt{\lg \lg n}})$	$O(4^{\sqrt{\lg \lg n}})$	$O(\lg n)$
Pairing heap [Iac00]	$O(n \lg n)$	$O(1)$	$O(\lg n)$	$O(\lg n)$
Pairing heap [This thesis]	$\Theta(n)$	$O(1)$	$O(\lg n)$	$O(\lg n)$
Stasko/Vitter [SV86]	$O(n \lg n)$	$O(1)$	$O(\lg n)$	$O(\lg n)$
Elmasry [Elm09a, Elm09b]	$O(n \lg n)$	$O(1)$	$O(\lg \lg n)$	$O(\lg n)$
Sort heap [IÖ14]	$\Theta(n \lg \lg n)$	$O(\lg \lg n)$	$O(\lg \lg n)$	$O(\lg n \lg \lg n)$
Binomial heap [Vui78]	$\Theta(\lg n)$	$O(1)$	$O(\lg n)$	$O(\lg n)$
Fibonacci heap [FT84]	$\Theta(n)$	$O(1)$	$O(1)$	$O(\lg n)$
Rank-pairing heap [HST09]	$\Omega(n)$	$O(1)$	$O(1)$	$O(\lg n)$

Table 1.1: Various heaps and amortized bounds on their running times. Top: analyses of pairing heaps. Middle: close relatives of pairing heaps. Bottom: more distant relatives of pairing heaps. Note: $\lg = \log_2$.

Our result. We present a potential function for pairing heaps that is much simpler than the one found for splay trees in [Col00] and also simpler than the only previously known potential function for pairing heaps that is $o(n \log n)$ [Pet05]. Further, it is simpler than the only other potential function known to give constant amortized time for insertion [Iac00], and perhaps more importantly, it is the first potential function for pairing heaps whose range is $O(n)$, which allows the use of amortized analysis to bound the run times of shorter operation sequences than before. In the case of pairing heaps, this bound on the potential function range is asymptotically the best possible, since the worst-case time for delete-min is linear, and thus we need to store at least a linear potential to pay for it.

This result is joint work with John Iacono.

Previous work. In Table 1.1, we list the amortized operation costs and ranges of several potential functions. Each row of the table corresponds to a single analysis of a specific heap variant. The table is divided into three parts. The top part is devoted to analyses of pairing heaps. The middle is for variants of pairing heaps, and the bottom is for heaps that are sufficiently different from pairing heaps that calling them a variant seems inaccurate. It is of course somewhat subjective whether a result belongs in the second or third group. For instance, a case could be made that rank-pairing heaps could go either way. Also, note that the use of $O()$ notation in the Range column is deliberate. First, because the hidden constants in the various results differ, sometimes dramatically. Second, because most papers do not state the range of the potential function (with [IÖ14] being a notable exception), using $\Theta()$ would force us to expand this subsection into its own chapter devoted to proving matching upper and lower bounds for each heap in the table. There are several cases in which we did use Θ; this is not meant to imply that the potential function is always this large, only that there exists at least one family of heaps for which it is that large. For instance, the classic potential function for pairing heaps is $\Theta(n \log n)$ because inserting n items in sorted order (increasing or decreasing both work) creates a heap with a potential that large, although some other operation sequences do not. Below we justify the correctness of the Range column of the above table; for the other columns, see the respective papers.

We use lg for $\log_2$ hereafter.

Classic pairing heap potential [FSST86]: The upper bound here is trivial, since the heap potential is the sum of the node potentials, and the potential of a node is the binary logarithm of the size of its subtree in the binary view. To get the lower bound, consider insertion of a sorted sequence (increasing or decreasing). Then in the binary view, all nodes (except the one leaf) have exactly one child. Thus, at least half of all nodes have a subtree size of $n/2$, and thus a potential of $\lg \frac{n}{2} = \lg n - 1$, which makes the heap potential at least $\frac{n}{2}(\lg n - 1) = \frac{1}{2} n \lg n - n/2$, which is $\Omega(n \log n)$.

Seth Pettie's analysis [Pet05]: Here the potential function is, to say the least, not very simple. As a warm-up, the paper starts with a simpler potential function which is far simpler to analyze. (Even the simple one has some interesting features, such as depending on the current state of the heap, and on its past, and on the future.) Deriving bounds for the general one is left as an exercise for the reader. (A grueling exercise, at that.) As for the simpler one, a node's potential is clearly upper-bounded by $2\sqrt{\lg n}$, so it follows that the range is at most n times that. (Actually this does not follow immediately, since the potential function has another term which we're neglecting, but this term is too small to matter in this case.) As for the lower bound, consider inserting n items into the heap in increasing order. (Decreasing order doesn't work here, because the potential function is based on the general view of the heap instead of the binary view, so it is less symmetric.) Then at least half of all nodes have $\hbar = \lg \frac{n}{2} = \lg n - 1$ (see the paper for the definition of $\hbar$), and thus a potential of $\Omega(\sqrt{\lg n})$. Therefore, the total heap potential would be $\Theta(n\sqrt{\lg n})$. Conjecture: the same construction works for the fancy potential function, which would give a bound of $\Theta(n \cdot 4^{\sqrt{\lg \lg n}})$.

John Iacono's analysis [Iac00]: The upper bound of $O(n \log n)$ follows immediately from the definition of the potential function (which we omit). To show that this bound is tight, some background will be needed. This potential function depends on the future even more strongly than that of Seth Pettie. For this potential function, it makes little sense to speak of the amortized run time of a single operation, even in a sloppy, informal sense, unless one has in mind, at least implicitly, the operation sequence that this operation is part of. Thus, if we wish to speak of the range of this potential function, it makes sense to look at its value at the beginning, and at the end, and then subtract the two. All intermediate values are essentially internal to the analysis. And at the end of an operation sequence, the potential is always at least zero and at most linear. However, the potential function depends only on the future and not the past, so it is legitimate to start the operation sequence from a nonempty heap. If we start it from the heap that results from inserting n items in decreasing order, and let the operation sequence be "delete-min, n times," then the initial potential equals $-\Theta(n \log n)$ (yes, negative!), and thus the bound is tight. Note, however, that this is not as interesting as it seems. Recall that the point of having a small range is to gain the ability to amortize over short sequences of operations. But we have created a sequence of n delete-min operations, which is not short, since such a sequence naturally takes $\Theta(n \log n)$ time anyway, so the large range does no harm. Thus the tightness of the upper bound remains mysterious.

Stasko and Vitter's variant(s) [SV86]: This paper actually defines two variants of pairing heaps. They both use the same potential function, and the upper bound follows easily since the potential function is nearly identical to the classic potential of [FSST86]. The variants differ in how decrease-key is implemented. The first variant uses a very simple implementation of decrease-key, similar in spirit to classic pairing heaps. Unfortunately, the authors were not able to analyze this variant, beyond the fact that insertion takes constant time, and deletion takes logarithmic amortized time, if no decrease-key operation is ever performed. The second variant is mentioned only in passing: it uses a relatively heavy-weight implementation of decrease-key. In this case, they were able to show that decrease-key takes at most logarithmic amortized time (although this result was merely sketched). Both variants are robust against the sorts of tricks we used to establish the lower bounds of the preceding potential functions. Conjecture: there exists a sequence of insert, decrease-key, and delete-min operations that raise the potential of this heap to something more than linear, and most likely all the way to the upper bound of $O(n \log n)$.

Amr Elmasry's variants: The potential in [Elm09a] is $O(n \log n)$ by inspection. It appears to be even harder to prove this bound is tight than in Stasko and Vitter's variant, for similar reasons.

In the case of [Elm09b], an upper bound of $O(n \log n)$ is also easy to see by inspection. However obtaining a good lower bound is harder. The amortization argument is somewhat complex, using a combination of potential, credits, and debits. The credits are mostly just terms in the potential function by another name: some operations create credits and later operations use them. The potential functions, as written, would cause insertion into the pairing heap to take more than constant time. (And likewise for melding.) The purpose of the debits is to get around this problem, by making later deletions subsidize the earlier insertions. Rephrasing this in terms of potential functions is not as natural as in the case of credits, but should be doable. Once this is done, it is by no means clear how to construct a heap with large potential.

John Iacono's variant: sort heaps [IÖ14]. Here the upper and lower bounds are again easy. The heap potential is the sum of the node potentials, and the potential of a node is at most $\lg \lg n$. This is tight: insert n nodes in increasing order, and one third of them will be right heavy, and thus have a potential of $\lg \lg n$.

Binomial Heaps [Vui78]**:** This heap barely qualifies for inclusion into the table, since it performs all operations in worst-case logarithmic time, and we are only interested in amortized heaps. But, in fact, the amortized time of insertion is $O(1)$, and the proof is so simple that we will sketch it here. A binomial heap is an ordered forest of rooted trees, each satisfying the heap-order property (that each child's key exceeds that of its parent). The size of each tree is a power of two, and no two trees have the same size. (The forest is sorted by tree size.) This set of constraints is already enough to allow us to deduce the structure of the heap given only its size, n: simply write n in binary and if, say, the eights place has a 1, then the binomial heap will have a tree of size 8, otherwise it won't. If, in the course of performing an operation, the unique size condition is violated, the two trees with the same size are merged into one. (This is analogous to the pairing operation in pairing heaps, but in the context of binomial heaps is called merging instead.)

Thus, insertion is performed by adding a tree of size 1, and then performing as many merges as needed to restore the unique size condition. If we let the potential be the number of trees in the forest, then it is trivial to show that insertion takes $O(1)$ amortized time: any merge takes one unit of work and releases one unit of potential, so they are effectively free. The range of the potential function is clearly logarithmic, since there are no more than $\lg(n+1)$ trees in the heap (and usually less, unless n is one less than a power of two).

Fibonacci heaps: [FT84]. A delete-min in a Fibonacci heap may take linear time, so the potential range must be at least linear or else the potential function can't even be used to prove that a Fibonacci heap can sort n items in $O(n \log n)$ time. The potential function for Fibonacci heaps is almost as simple as that for binomial heaps: the number of trees in the forest, plus twice the number of marked nodes. (The potential for marked nodes is only important for proving that decrease-key takes constant amortized time.) Clearly the range is never more than linear, since one can't do worse than marking every node and placing it in its own tree.

Rank-pairing heaps: [HST09]. Again, the potential must be at least linear. The paper analyzes two variants of rank-pairing heaps, and in both cases the analysis is involved. Here we make a few superficial observations. The potential function depends only on the current state of the structure, rather than knowing the past or the future. Of the three primary operations we are concerned with, delete-min in general releases potential. Likewise, decrease-key in general needs to release potential, since in this heap, as in Fibonacci heaps, the worst-case time is linear. Thus, the only operation that allows us to build up potential is insertion, and its amortized cost is constant. Thus we should expect a typical rank-pairing heap to have linear potential. Unfortunately, in the worst case, this kind of argument proves nothing. For instance, even though a typical decrease-key should release potential, there could be a sequence of cleverly chosen decrease-key operations such that each of them costs a unit of potential (but no more than that, decrease-key takes constant amortized time). Given a long enough such sequence, the heap potential would eventually rise to more than linear. Proving such a sequence does not exist requires looking at the potential function in detail. Without that, at best we can say it seems implausible, since such a sequence will be applying decrease-key to the same nodes repeatedly, which is not going to change the heap structure, since they are already roots, unless some delete-min operations are mixed in. Neither can we rule out a clever sequence of delete-min operations, each raising the potential by $O(\log n)$; such a sequence can't be very long, since eventually the heap will be empty and have a potential of zero. However, we again recall that the reason we care about this at all is that we wish to amortize over short sequences. Such tricks, even if they were to work, are nearly impossible to pull off with a short sequence. (But not quite impossible: imagine a sequence of $n/\sqrt{\lg n}$ deletions, each raising the potential by $\lg n$. At the end, we would have a heap of size $n - o(n)$ with potential $n\sqrt{\lg n}$.)

The analysis of pairing heaps presented in Chapter 4 of this thesis: In Lemma 4.1 we prove present a new potential function for pairing heaps and prove that its range is linear. This is optimal, because we must release linear potential during a delete-min in the worst case, or else the amortized time of delete-min would equal its actual time, which is linear, whereas a good heap should have logarithmic amortized time for delete-min.

Part I

Geometric Optimization

Chapter 2

Covering Most Points With a V-shape of Minimum Width

2.1 Introduction

Motivation. The motivation for this problem comes from curve reconstruction: given a set of points sampled from a curve in the plane, find a shape approximating the original curve. It has been suggested in [AD11] that in an area where the curve makes a sharp turn, it makes sense to model the curve by a *V-shape*; see below for a formal definition. The authors remark that it would be natural to investigate a variant that can handle a small number of outliers, to accommodate a few bad data points. We investigate that variant here. The problem is an instance of a large class of problems known as *geometric optimization* or *fitting* questions; see [AS98] for a survey.

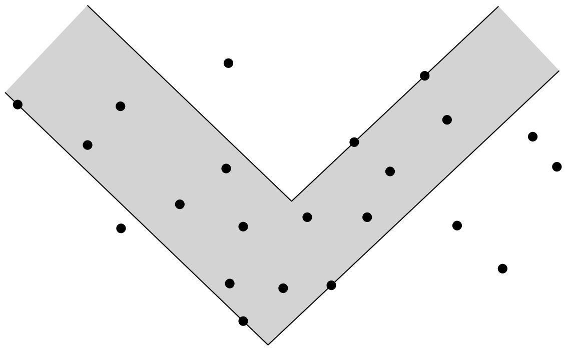


Figure 2.1: A V-shape with six outliers.

Previous work. In [AD11], the authors develop an algorithm for covering a point set in general position with a V-shape of minimum width (allowing no outliers) that runs in $O(n^2 \log n)$ time and uses quadratic space. They also find a constant-factor approximation algorithm with running time $O(n \log n)$, and a $(1+\varepsilon)$-approximation algorithm with a running time of $O((n/\varepsilon)\log n + (n/\varepsilon^{3/2})\log^2(1/\varepsilon))$, which is $O(n \log n)$ for a constant $\varepsilon > 0$.

Our Result. Given a point set P of n points in the plane in general position (no vertical line goes through two points, no three points are collinear, and no lines defined by pairs of points are parallel), and an integer $k > 0$, we show how to find the minimum-width V-shape enclosing all but k of the points with probability $1 - 1/n^c$ for any $c > 0$ in $O(cn^2k^4 \log n(\log n \log\log n + k))$ expected time and quadratic space. The last expression is $\tilde{O}(cn^2k^5)$, where $\tilde{O}(\cdot)$ is meant to suppress polylogarithmic terms.

Definitions and notation. A V-shape is an unbounded polygonal region delimited by two pairs of parallel rays emanating from two vertices; see Figure 2.1. The rays on the region's convex hull are the *outer rays*. The remaining two rays are the *inner rays*.

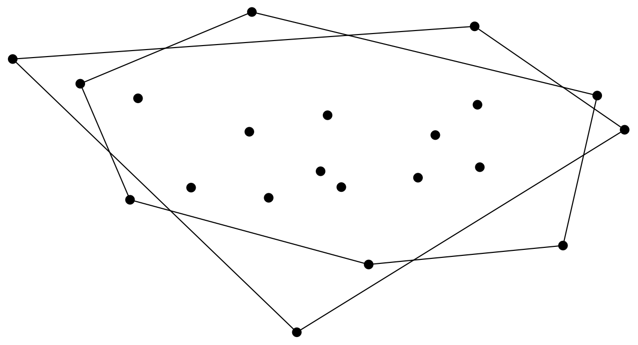


Figure 2.2: The edges at level 1 of a point set.

The directed line segment connecting the vertices of the outer and inner rays separates the V-shape V into its *left arm* and *right arm*. The *width* of an arm is the distance between its two delimiting rays. The *width* of V is the width of its wider arm. An *outlier* of V is a point of P not contained in V. Each arm has an associated *strip*, defined by the pair of directed parallel lines supporting its boundary rays. The left and right strips together uniquely determine a V-shape. This is in fact how our algorithm works: by trying to find a pair of strips determining the thinnest V-shape with the prescribed number of outliers.

Given a point set P, a *k-edge* (see Figure 2.2) of P is a directed edge between two points in the set such that exactly k points of P lie to the left of the directed line through the edge (so if P is in general position, there are $n - k - 2$ points to the right). For example, a 0-edge is a directed edge of the convex hull. A k-edge is also said to be *an edge at level k*. Let $L(e, P)$ denote the level of edge e in point set P. The set $H(k, P)$ of edges at level k or less are known as the at-most-k-edges, or more concisely, the $(\leq k)$-edges.

It will also be useful to talk about levels in an arrangement $\mathcal{A}$ of non-vertical lines (see Figure 2.3). We use the following definition: an edge of $\mathcal{A}$ is on the k-level whenever there are exactly k lines of $\mathcal{A}$ strictly above it.

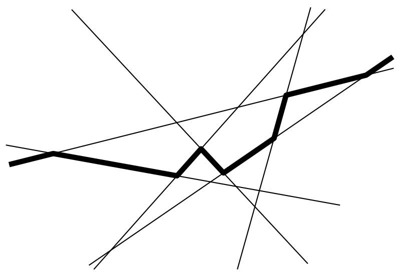


Figure 2.3: The 2-level of a line arrangement with six lines.

2.2 The algorithm

2.2.1 Overview

We need only consider locally optimal V-shapes. A V-shape is *locally optimal* if and only if any slight perturbation either increases the width of an arm or decreases the number of covered points. In [AD11], it was shown that there are three types of locally optimal configurations (which they referred to as canonical, instead of locally optimal), which they called both-outer, inner-outer, and both-inner (see Figure 2.4). In a *both-outer* V-shape, both outer rays have two points on them. In a *both-inner* V-shape, both inner rays have two points on them. In a *inner-outer* V-shape, one of the outer rays has two points and one of the inner rays also has two points. Normally, if one ray of an arm has two points on it, the other ray will have one point, so that, for instance, the inner rays of a both outer V-shape have one point each. But even for point sets in general position, one of the arms may have its two rays partially overlap, and thus the V-shape will have only five distinct points on its boundary instead of six. In this case, the inner ray might have two points on it, instead of just one. A V-shape with k outliers is called a k-outlier V-shape of the point set. Our algorithm has the same structure as theirs does: find the minimum-width V-shape of each type, and return the one that has the smallest width of all three. (Except that they find one with no outliers, while ours has k outliers.)

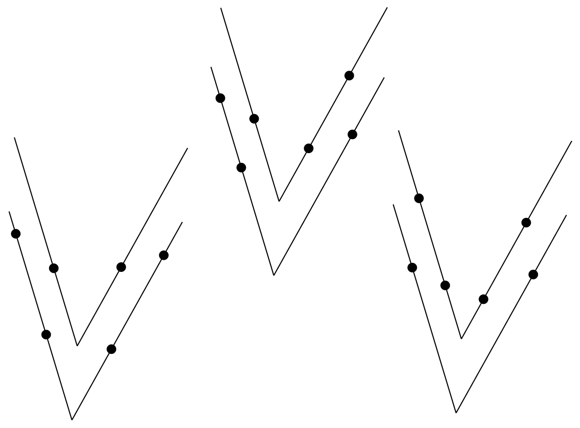


Figure 2.4: Both-outer, inner-outer, and both-inner V-shapes.

We (very) briefly discuss how the authors of [AD11] handled all three cases. They handled the both-outer case by explicitly enumerating all pairs of convex hull edges of the point set (which determines the outer rays), and for each pair they found the best inner rays in logarithmic time. They handled the inner-outer case by explicitly enumerating all choices for the first strip, and then finding the best matching second strip in logarithmic time. Finally, they handled the both-inner case by considering all pairs of points as candidates for one of the inner rays, and for each one found the best matching V-shape in amortized logarithmic time. For some of these steps, they use fairly sophisticated tools to achieve such fast running times, such as a fully persistent data structure for semi-dynamic convex hull.

Our approach for the both-outer case and the inner-outer case was inspired by the approach of [AD11] for the inner-outer case, except instead of enumerating all choices for the first strip, we use binary search to find the inner ray. Finally, our approach for the both-outer case uses their both-outer algorithm as a black box, by running it on random subsets of the point set.

We handle both-outer V-shapes and inner-outer V-shapes in almost the same way (see Figures 2.5 and 2.6). We begin by enumerating the $(\leq k)$-edges of the point set. Each such j-edge e is considered in turn as a candidate

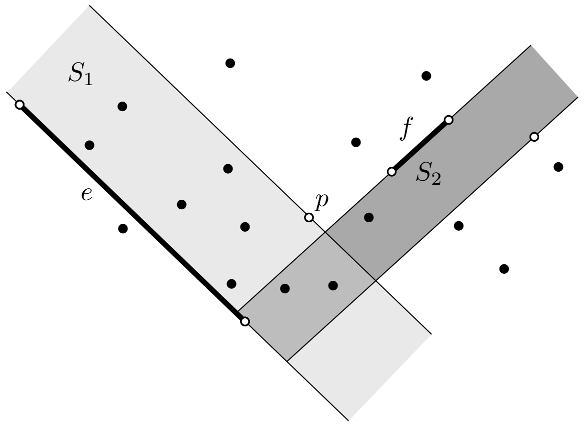


Figure 2.5: A snapshot of the inner-outer algorithm.

for supporting an outer ray, with $j \leq k$ outliers already accounted for. For a fixed e, we do a binary search among remaining points of P, ordered by perpendicular distance from e; this distance is the width of the first candidate strip. For each point of the search we find the second strip that has the smallest possible width and still covers the remaining points, except for $k - j$ outliers. If the second strip is wider than the first, the binary search widens the first strip so that the second strip has fewer points, otherwise it narrows it. To find the second strip, we again enumerate the edges at levels 0 through $k - j$ of the remaining points. The precise definition of “remaining” here is the key difference between the both-outer and the inner-outer algorithm, as discussed below. By now we have chosen three rays, and have no freedom for the fourth: it is dictated by how many more outliers we need. The running time is $O(n^2(k+1)^2 \log^2 n)$; see Lemma 2.3 for details of the algorithm and its analysis.

To find the minimum-width both-inner k-outlier V-shape, we use a randomized algorithm that takes random samples of the given point set. For each sample, it enumerates *all* both-inner 0-outlier V-shapes using the algorithm from [AD11]. We show that with enough samples, the minimum-width both-inner k-outlier V-shape will be one of the V-shapes enumerated with

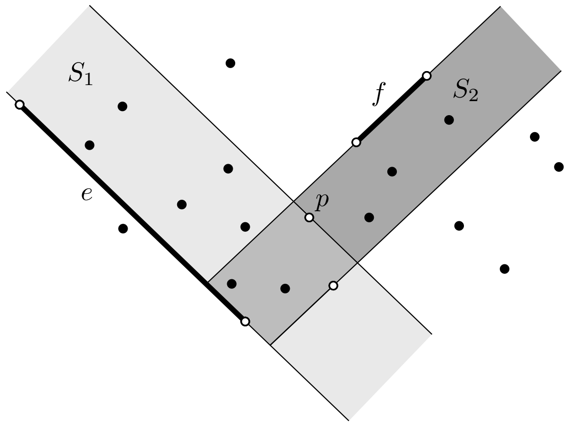


Figure 2.6: A snapshot of the both-outer algorithm.

probability at least $1 - 1/n^c$ for any real number $c > 0$ (given as an input parameter). The V-shapes we enumerate might have more than k outliers, so we use a range searching data structure from [CY84] to detect and discard such V-shapes. The expected running time of the both-inner case is $O(cn^2k^4 \log n(\log n \log\log n + k))$, which dominates the running time of the other two cases.

In the following sections, we fill in the details of the algorithms for all three cases.

2.2.2 Both-outer and inner-outer

The following two algorithms find the thinnest inner-outer k-outlier V-shape and the thinnest both-outer k-outlier V-shape. The algorithms have the same structure: for each candidate first strip S_1, find the thinnest possible second strip S_2. In Procedure 2.1, we show how to find S_1.

It remains to explain how to find the thinnest S_2 and this depends on whether the k-outlier V-shape we seek is inner-outer or both-outer. First we define a function *find-line* (described in Lemma 2.2) that takes a directed edge e, an integer i, and point set P, and finds a line parallel to e; specifically the $(i + 1)$st furthest line from e going through a point in P right of e. We

Procedure 2.1 Finding S_1

```
Input: integer $k$, point set $P$, and a procedure strip-two() for finding $S_2$.
For each directed edge $e \in H(k, P)$:
    Let $P'$ = points to the right of $e$, and the two points on $e$.
    Sort $P'$ by distance from the line through $e$.
    // Perform binary search on $P'$.
    For each point $p$ of the binary search:
        Let $k' = k - L(e)$.
        Let $S_1$ = strip defined by $e$ and $p$.
        Let $S_2$ = strip-two($k', P', S_1$).
        The binary search is guided by which strip is thicker:
            If $S_1$ is thicker, move $p$ closer to $e$, else further.
```

will slightly abuse notation and use S_1 to refer both to a strip and the points within it.

* The two cases are similar, so the steps that differ are marked with an asterisk (like the first line of this paragraph). It may help to refer to Figures 2.5 and 2.6. To find a both-outer k-outlier V-shape, we pass Procedure 2.2 as the third parameter to Procedure 2.1, and to find an inner-outer k-outlier V-shape, we pass Procedure 2.3 instead.

Procedure 2.2 Finding S_2 for a both-outer V-shape

```
  Input: integer $k'$, point set $P'$, and strip $S_1$.
* Preprocess $P' - S_1$ for find-line to work.
* For each edge $f \in H(k', P')$:
*     $j = k' - L(f, P')$. // number of outliers still needed
*     Let $\ell$ = find-line($f, j, P' - S_1$). // $\ell$ may not exist; see main text
      Let $S_2$ = the strip formed by $f$ and $\ell$.
      Record the thinnest $S_2$ found so far.
  Return thinnest $S_2$ found.
```

It is possible that ℓ from Procedure 2.2 does not exist, because $P' - S_1$ has less than j points. If so, then the strip determined by S_1 is too wide, and we can proceed to the next p. (Notice that a different choice of f will not help.) In Procedure 2.3, ℓ will certainly exist.

Procedure 2.3 Finding S_2 for an inner-outer V-shape

```
  Input: integer $k'$, point set $P'$, and strip $S_1$.
* Preprocess $P'$ for find-line to work.
* For each edge $f \in H(k', P' - S_1)$:
*         $j = k' - L(f, P' - S_1)$. // number of outliers still needed
*         Let $\ell = \text{find-line}(f, j, P')$.
          Let $S_2 =$ the strip formed by $f$ and $\ell$.
          Record the thinnest $S_2$ found so far.
  Return thinnest $S_2$ found.
```

We now address the correctness and running time of these two algorithms.

Lemma 2.1. *The above algorithms are correct.*

Proof. These algorithms are optimization-by-enumeration algorithms and there are only two ways such an algorithm can fail. The first way is that the enumeration skips over the optimal solution. This can not happen for an exhaustive enumeration algorithm, but ours is not quite exhaustive, because of the binary search; we must establish that the binary search never discards an optimal solution. The second way that an enumeration algorithm can fail is by accidentally enumerating a non-solution; that is, it might generate a solution that seems very good, but fails to satisfy one of the problem constraints and is thus not a solution at all. There are two kinds of non-solutions we need to worry about. First, two arbitrary strips may not form a V-shape. Second, even if they do, the V-shape may have the wrong number of outliers. Below, we rule out these possibilities.

Two arbitrary directed strips may form a shape that looks like an X or a T instead of a V (see Figure 2.7). More formally, we want to avoid the arm corresponding to S_1 having points on *both* sides of the arm corresponding to S_2. The points covered by S_1 might indeed be split by S_2, but this can only happen when the points that were split off were among the k outliers. This is because only the outer ray of S_2 can split off points from S_1, and it only splits off points near the convex hull: the outliers.

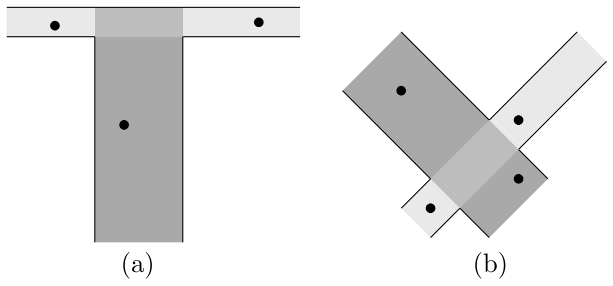


Figure 2.7: Two strips forming a T (a), or an X (b), instead of a V.

The algorithms never create more than k outliers, because they keep track of how many are needed and at each step never create more than that. Do they ever create less than k? This can only happen if the algorithm counts some outlier more than once. The algorithms choose outliers three times: first when they choose e, then when choosing f, and finally when choosing ℓ. The outliers created by e (that is, the i points to its left) are never double-counted, because they are invisible to the rest of the algorithm, which works with P' instead of P. The outliers created by f and those created by ℓ are on opposite sides of f, so they can not be counted twice either.

Lastly, can the thinnest both-outer or inner-outer V-shape be overlooked? For both-outer and inner-outer V-shapes, there is at least one outer ray defined by two points, and we consider all edges e that could possibly define it. For a fixed choice of e and p, we examine all feasible choices of f. For a fixed choice of e, p and f, we have no freedom in choosing ℓ, so no wrong choice is possible. The only place where we do not look at all possibilities is in choosing p, where we do binary search. Moving p farther from e when S_1 is thicker than S_2 would result in a V-shape thicker than the current one. Therefore we may legitimately discard all such choices of p. Likewise for moving p closer when S_2 is thicker. $\square$

Before analyzing the running time of the algorithms, we show how to implement find-line efficiently.

Lemma 2.2. *Given a point set of size n and an integer $0 \leq k < n$, after $O(kn \log n)$ preprocessing, we can answer queries of the following form: given an integer $0 \leq i \leq k$ and a directed line e, find the $(i+1)$st furthest point from e among points to the right of e, in $O(\log n)$ time.*

Proof. Finding the desired point is equivalent to finding the line ℓ parallel to e which goes through the point in P such that there are i points in P right of ℓ. (This is the line that find-line returns.) To do this, we go to the dual; let P^* be the dual of P, and let $\mathcal{A}$ be the line arrangement induced by P^*, where ℓ dualizes to a point ℓ^*. The requirement in the primal that there be k points right of ℓ means that ℓ^* must lie on an edge in the k-level or the $(|P|-k-1)$-level of $\mathcal{A}$, and the fact that ℓ must lie to the right of e eliminates one of these two possibilities. Using [EW86], we can compute the i-levels, and the $(|P|-1-i)$-levels, for all $i \leq k$, in sorted order by x-coordinate, in time $O(kn \log n)$. This completes the preprocessing.

We now perform the query. Since we know the x-coordinate of ℓ^* (it is given by the slope of e in the primal), we can do binary search on the i-level to identify the two vertices that ℓ^* lies between. These two vertices lie on a line of P^*, which corresponds to a point of P in the primal. This is the desired point. □

Lemma 2.3. *The running time of the above algorithms is $O(n^2k^2 \log^2 n)$.*

Proof. The algorithms are structured as a triply nested loop, so it suffices to count the number of iterations of each loop. It is well known that the set of $(\leq k)$-edges has size $O(kn)$ [GP84, AG86], so the loops for e and f both iterate at most that many times. The binary search for p iterates $O(\log n)$ times. We can enumerate the j-edges, for all $j \leq k$ in $O(kn \log n)$ time using the algorithm in [EW86, pages 272–278]. (The algorithm of [EW86] depends on a data structure for dynamic convex hull. At the time, the best available such structure was that of [OvL81, pages 169–181]. Using the one described in [Jac02] instead gives the claimed running time.) By Lemma 2.2

we can implement find-line to run in $O(\log n)$ time. The claimed running time follows. $\square$

2.2.3 Both-inner

Procedure 2.4 finds the thinnest both-inner k-outlier V-shape with high probability.

Procedure 2.4 Find a min-width both-inner k-outlier V-shape for P (w.h.p.)

```
Input: integer $k > 0$, point set $P$, real number $c > 0$
Let $n = |P|$, and let $K = k + 1$
Repeat $K^6 ce \ln n$ times: // $e \approx 2.7$ is the base of the natural logarithm
    Initialize $R$ to the empty set
    For each point in $P$, add it to $R$ with probability $1/K$
    $W$ = Find-empty-V-shapes($R$) // [AD11, page 66]
    Remove V-shapes with more than $k$ outliers from $W$
Return the thinnest V-shape seen.
```

Lemma 2.4. *Given $k > 0$ and $c > 0$, Procedure 2.4 finds the thinnest both-inner k-outlier V-shape with probability at least $1 - 1/n^c$, in expected time $O(cn^2k^4 \log n(\log n \log\log n + k))$ and $O(n^2)$ space.*

Proof. Denote the thinnest both-inner k-outlier V-shape by V. Clearly, V is defined by (at most) six points of P [AD11, pages 63–64]. Consider a subset R of P, which contains the six points defining V but does not contain the k outliers. V is a valid both-inner 0-outlier V-shape for R, though perhaps not the thinnest one. The algorithm simply samples P over and over, in the hopes of eventually picking such a subset R. For each sample R, it enumerates *all* both-inner 0-outlier V-shapes using the algorithm from [AD11], and checks whether they end up having at most k outliers in P. Note that if all the V-shapes we consider end up resulting in more than k outliers, our algorithm fails to find *any* valid V-shape. However, we show that this is very unlikely: the probability that the algorithm fails to find the *optimal* V-shape is less than $1/n^c$, where c is the given positive constant.

Each point in P is independently chosen to be part of R with probability $1/K$. Thus, R has expected size n/K. Now, what is the probability that V is one of the valid both-inner 0-outlier V-shapes for R? The probability of having the required six defining points is $1/K^6$, and the probability of avoiding the k outliers is $(1-1/K)^k = (1-1/(k+1))^k > 1/e$, since $(1-1/(k+1))^k$ converges to $1/e$ from above. So, the probability of our random sample containing the six points we need and not containing the k points we should avoid is at least $p = 1/(eK^6)$. If we call this the probability of success, then the probability of failure is at most $1-p$. If instead of taking just one such random sample, we take $m = K^6$ samples, the probability of them all failing is at most $(1-p)^m$. Now using the fact that, for any x, we have $1-x \leq e^{-x}$, we conclude that the probability q of all m samples failing to contain the optimum both-inner V-shape is at most $(1-p)^m < e^{-pm}$. Since $pm = \frac{1}{eK^6} \cdot K^6 = 1/e$, we have $q < e^{-pm} = e^{-1/e} \approx 69\%$. If we increase the number of samples from m to $mce \ln n$, then the probability of failure q reduces to at most $e^{-pmce \ln n} = e^{-(ce \ln n)/e} = (e^{\ln n})^{-c} = 1/n^c$, which concludes the high-level description of the algorithm, and the proof that its probability of failure is at most $1/n^c$.

The crucial operation in the above algorithm is to check that the V-shapes returned by the algorithm from [AD11] do not have too many outliers. This can be done using range searching with wedges, which is a special case of simplex range searching, for which there are a variety of data structures with various space/time trade-offs. (A *wedge* is simply the convex region bounded by two rays with a common vertex.) We use a data structure that takes $O(n^2)$ space (really $o(n^2)$ space) and gives $O(\log n \log\log n + k)$ query time [CY84, pages 41–45]. The time taken by the algorithm from [AD11] to enumerate all both-inner 0-outlier V-shapes of a point set with $O(n/K)$ points is $O((n^2/K^2)\log n)$. The subset may have as many as $O(n^2/K^2)$ V-shapes, each of which take $O(\log n \log\log n + k)$ time to check to make sure the number of outliers is not too high, for a total time of $O((n^2/K^2)(\log n \log\log n + k))$

per random sample.

We have glossed over a statistical subtlety here. If the expected value of a random variable X is $E[X]$, then in general $E[X^2]$ may not be $O(E[X]^2)$, or indeed, it might not even be finite. In this case, how do we know, just because the expected size of R is $O(n/k)$, that the expected number of V-shapes is $O(n^2/k^2)$? What is true for all random variables X from distributions with finite mean and variance is that $E[X^2] = E[X]^2 + \text{Variance}[X]$. The size of R follows the binomial distribution with mean n/K and variance $n(1/K)(1-1/K)$, so we have $E(|R|^2) = n^2/K^2 + n(1/K)(1-1/K) < n^2/K^2 + n/K$, which is $O(n^2/K^2)$.

Since we are taking $O(cK^6 \log n)$ random samples, finding the best both-inner k-outlier V-shape takes time $O(cn^2k^4 \log n(\log n \log\log n + k))$ in expectation. $\square$

Remark. We have calculated how many samples we need in order to find a particular both-inner k-outlier V-shape with high probability (specifically, the thinnest one). A natural question to ask is how many samples we would need to find *all* both-inner k-outlier V-shapes. If the probability of failing to find an arbitrary both-inner V-shape is q, then the probability of there being at least one both-inner V-shape we fail to find is at most q times the number of both-inner V-shapes present in the point set. Clearly, regardless of the value of k, this number is at most $6!n^6 = 720n^6$, and we already computed $q < 1/n^c$. By choosing $c > 7$, we have $q < 1/n^7$, and thus our probability of failure is less than $720/n$.

2.3 Wrap-up

Theorem 2.5. *Given n points, an integer $0 \le k \le n-4$ denoting the desired number of outliers, and a real number $c > 0$, our algorithm finds the minimum-width k-outlier V-shape for the points with probability $1 - 1/n^c$, in* *expected time* $O(cn^2k^4 \log n(\log n \log\log n + k))$ *and using* $O(n^2)$ *space.*

Proof. We find the thinnest k-outlier V-shape of each of the three types separately and return the thinnest of the three. The both-inner algorithm dominates the running time. The running time and correctness of the algorithms handling the three cases is established in Lemmas 2.1, 2.3, and 2.4. □

Acknowledgments

We would like to thank Sariel Har-Peled for the randomized algorithm, and Muriel Dulieu for participating in discussions of a simpler version of the problem.

Chapter 3

Placing a Point to Maximize Angles

3.1 Introduction

This chapter describes several algorithms for solving the following problem: Given a set P of n points in the plane, and a set C of constraints with endpoints in P, find a point q within the interior of the convex hull of P, such that the constrained Delaunay triangulation of the combined point set has the largest possible minimum angle.

The plan We begin in Section 3.2 by describing some background material that will be useful in understanding all four algorithms. Then, in Section 3.3, we proceed to explain the original algorithm of [AAF07]. We then present in Section 3.4 an algorithm based on LP-type problems. It involves changing just one step in the original algorithm, and then fixing a technicality that comes up. In Section 3.5, we present the simplest algorithm we know of for this problem, which is (relatively) fast if there are very few constraints. Finally, in Section 3.6, we present an algorithm which is fast even if there are a linear number of constraints. In Section 3.7 we wrap things up, and mention an open problem or two.

Summary For convenience, we summarize the runtime of all the known algorithms below:

- The Original Algorithm: $\tilde{O}(n^4)$
- The LP Algorithm: expected $\tilde{O}(n^3)$
- The Envelope Algorithm
  - $\tilde{O}(n^2)$ without constraints
  - $\tilde{O}(|C|n^2)$ with constraints, where C is the set of constraints.
- The Feasible Region Algorithm: expected $\tilde{O}(n^2)$
- scattered throughout: output-sensitive analysis.

3.2 Preliminaries

Notation The constrained Delaunay triangulation of P and C is T. The constrained Delaunay triangulation of $P \cup \{q\}$ and C is T_q.

Problem statement Given a set P of n points in the plane in general position (see next paragraph), and a set C of constraints with endpoints in P, find a point q within the interior of the convex hull of P, such that T_q has the largest possible minimum angle. Placing q on a constraint is not allowed. It should be possible to adapt our algorithms to allow placing points outside the hull with only minimal changes, but hereafter, we assume q must be placed within the hull.

General position All the algorithms we discuss assume general position in one way or another, but not all of them make the exact same set of general position assumptions. Also, some general position assumptions come from the analysis rather than the algorithm. That is, the algorithm would still work, and maybe it would still be fast, but our proof of its speed would no longer hold.

On occasion we will call attention to specific general position assumptions as we make them, but on the whole we will sweep the problem under the rug by making essentially the strongest possible assumption: that the input points are algebraically independent over the integers. That is, we assume that no subset of two or more input coordinates satisfies any (nontrivial) polynomial equation with integer coordinates. (We'll give three examples. First, the fact that no two points lie on a horizontal line implies that no two points a, b satisfy $a_x - b_x = 0$. Second, the fact that no three points are collinear means that no triangle spanned by three points has zero area. The area of $\triangle abc$ is easy to compute given the coordinates of a, b, and c: it is $\frac{1}{2}|(b-a) \times (c-a)|$. That is, we simply take the half the magnitude of the cross product of the vectors defined by two sides of the triangle. Finally, checking that no four points lie on the same circle is similar: we lift each point (x, y) to $(x, y, x^2 + y^2)$, and then ensure that no four points are coplanar. To test if four points are coplanar we test if the pyramid they span has zero volume, which is likewise easy given the coordinates of the points; we omit the formula, but merely mention that it is a four-by-four determinant involving the coordinates of the points; refer to [O'R98] for details.)

Here, we merely state two general position assumptions made by all the algorithms. First, no three points of P lie on a common line, as is typical when dealing with triangulations. Second, no four points lie on a common circle, because otherwise the Delaunay triangulation is not unique.

Definitions We first review some (mostly) standard terminology. A triangle in a (constrained) Delaunay triangulation is a *(constrained) Delaunay triangle.* The circumcircle of a Delaunay triangle is a *Delaunay circle.* The interior of a Delaunay circle is an (open) *Delaunay disk.* A disk containing no points of P is *empty.* If two Delaunay triangles share an edge, the intersection of the two corresponding disks is the *Delaunay lune* of that edge, or simply *lune*, for short. Every edge of a Delaunay triangulation has an associated Delaunay lune; we will not need any lunes for hull edges since we only consider placements of q inside the hull. Lastly, we say that two points a and b can *see* each other if the line segment ab does not cross any constraint of C. Note that the two endpoints of a constraint do see each other. A set of points can all see each other if every pair of points can see each other.

3.2.1 Characterizations

We now review the characterization of Delaunay triangulations. Much of this information can be found in textbooks such as [BCKO08, Chapter 9]. A triangle belongs to the Delaunay triangulation T of P if and only if the triangle's three vertices are in P and the triangle's circumdisk is empty; that is, if there are no points of P in its interior. We can now derive a useful corollary. First, observe that an edge $e = rs$ belongs to T precisely when it belongs to some triangle which is in T. In fact, unless e is a hull edge, it is part of two triangles. Now let T_q be the Delaunay triangulation after a new point q is added. If q is placed within e's lune, then neither triangle that e belonged to has an empty disk anymore, and hence neither triangle is in T_q and thus neither is e. Conversely, if q is not placed in e's lune, then e does appear in T_q.

The characterization of constrained Delaunay triangulations is more complicated than that of unconstrained ones. A triangle is a constrained Delaunay triangle if and only if the following two conditions are met. First, all three vertices of the triangle must see each other. Second, the triangle's circumdisk must contain no points of P that can be seen from any point in the interior of the triangle. There are several other equivalent definitions; see, for instance, [LL86].

Let D be a constrained Delaunay disk, and consider the locus L of points in D that can be seen from some point in the interior of D's triangle. We

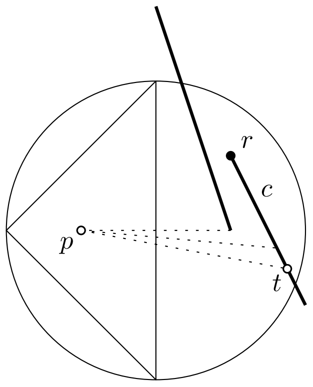


Figure 3.1: The constraint labeled c can not exist.

call L the *visible region* of D. By definition of a constrained Delaunay triangulation, L contains no points of P, but what does L look like? If there are no constraints, then L is simply D. If any edge of the triangle itself is constrained, then clearly nothing beyond that edge is visible from inside the triangle. In addition, any constraint that does not intersect D obviously does not affect L, while those that do intersect D might make L smaller. The ones that make L smaller are the ones visible from the triangle.

We claim that a constraint with an endpoint in D must not be visible to the triangle. Indeed, suppose for the sake of contradiction that there is a point p in the triangle that sees part of some constraint c with an endpoint r in D, and let t be a point on c that p sees; see Figure 3.1. By the empty circle property, p does not see r, so if we gradually move t along c towards r, eventually p will stop seeing t. At the moment that p stops seeing t, visibility between p and t must be blocked, necessarily by some constraint endpoint. But this means that there is a clear line of sight from p to that endpoint, violating the empty circle property and contradicting our assumption that c exists. Thus the visible region L of Delaunay disk D looks like D with some pieces clipped off; see Figure 3.2. Slightly more formally, each constrained edge has two associated half-planes. If the edge intersects D but has no endpoints in D, then one of the two half-planes will contain D's triangle.

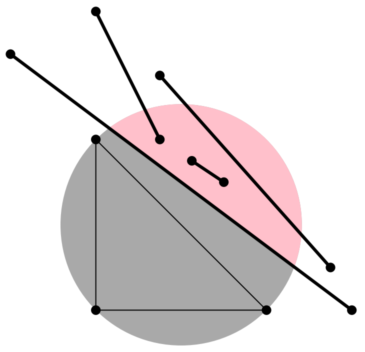


Figure 3.2: The visible region of the disk is dark gray, the rest is light pink.

Each constraint "clips off" the part of D that lies in the other half-plane.

We argued earlier that in an unconstrained Delaunay triangulation T, placing a point into the Delaunay lune of some edge causes that edge to disappear from T, because the relevant disks are no longer empty, and that this is the only way an edge can disappear. Similarly, in the constrained case, if an edge is not itself constrained, then adding a point to the intersection of the visible regions of its two Delaunay disks causes the edge to disappear from the constrained Delaunay triangulation. We omit the entirely analogous proof.

Unsurprisingly, we will find it useful to know how T changes when q is added to P. The answer becomes obvious once we look at the situation backwards: starting from T_q, remove q to obtain T. Clearly, all edges (and triangles) which were incident to q are gone, leaving a star-shaped polygonal hole. (A star-shaped polygon is simply a polygon whose interior contains at least one point that can see every vertex of the polygon.) The certificate that it is star-shaped comes from q itself, since of course q was able to see all of its neighbors! Edges that were not incident to q remain, since if they were Delaunay edges before, they were associated with some constrained Delaunay disks, and removing q can not cause the previously empty visible region of a disk to suddenly become non-empty. Thus, in the forward direction, inserting q makes some edges go away, and the resulting star-shaped hole is triangulated by connecting q to all vertices of the hole.

3.3 The Original Algorithm

Our presentation will differ slightly from that of [AAF07], partly for idiosyncratic reasons, and partly to make the parallels to the faster algorithms (that we describe later) more apparent. We first present the algorithm in the context of unconstrained triangulations, and then show the (nearly trivial) modifications needed to get constraints to work. Consider the arrangement $\mathcal{D}$ consisting of all the Delaunay circles of T. Now imagine placing q within some cell of this arrangement, yielding T_q. Observe that the topology of T_q does not change as long as we keep q within the same cell of $\mathcal{D}$. This immediately suggests an idea for an algorithm: construct $\mathcal{D}$, and for each cell of $\mathcal{D}$, we "magically" find the best placement of q within that cell, and then return the best placement found. Constructing $\mathcal{D}$ can be done by using fairly standard tools such as randomized incremental construction [SA95, Theorem 6.20], so the only question is how to work the magic within each cell.

If the polygonal hole formed after inserting q has h vertices, then triangulating the hole together with q gives us a triangulation with exactly h triangles, all incident to q. Each of those triangles has (surprise!) three angles, for a total of $3h$ angles. The key observation is this: if $q = (x, y)$, then the measure of each of those $3h$ angles is simply a function of x and y. Making the smallest angle as big as possible corresponds to finding the x and y coordinate of the highest point on the lower envelope of all these functions. Thus, our problem is reduced to computing the lower envelope.

The lower envelope of n surfaces can be computed in $O(n^{2+\varepsilon})$ time (with the implied constant depending on ε) using standard tools [SA95, Section 7.3.4], provided that two conditions hold (see [SA95, page 188]). First, the intersections of the graphs of the functions must be sufficiently well-behaved, and second, the functions must have sufficiently simple domains.

The precise technical conditions for "well-behaved" and "simple" are complicated, but the following suffices: first, the functions must be algebraic. That is, if the function is $z = f(x, y)$, it must be possible to rewrite the formula as $g(x, y, z) = 0$, where g is a bounded-degree polynomial in x, y, and z. Second, it must be possible to express the domain of the function as a bounded size Boolean formula, where each literal is an inequality of the form $h(x, y) \leq 0$, where h is a bounded-degree polynomial.

We start by handling the second condition. The domain is a cell of $\mathcal{D}$, and the boundary of a cell may have arbitrarily high complexity even in an arrangement of lines, let alone curves. We can fix this simply by refining $\mathcal{D}$; if we replace $\mathcal{D}$ by its trapezoidal decomposition [BCKO08, Section 6.1], then each cell is bounded by $O(1)$ line segments and circular arcs, so each function has a simple domain. We now return to the first condition: are the functions that defining the angles algebraic (and of bounded degree)? They aren't, but their cosines are, as we will show using the law of cosines. Indeed, let rs be an edge of the polygonal hole created in T by inserting q. Using $|\cdot|$ to denote edge length, the law of cosines gives $|qr|^2 + |sq|^2 - |rs|^2 = -2|qr||sq| \cos\angle sqr$. If we square both sides to eliminate the implied radicals in $|qr||sq|$, we have an equation of the form $f(q) = g(q, \cos\angle sqr)$, where $q = (x, y)$, f is a polynomial in x and y, and g is a polynomial in x, y, and $\cos\angle sqr$. The degree of f is 4, and the degree of g is 5, and thus the equation can be rewritten as $h(q, \cos\angle sqr) = 0$, where h is of degree 5. (Note that $|qr|^2$ is a term of degree 2, while $|rs|^2$ is a term of degree 0, since r and s are points of P and thus the algorithm can consider their coordinates as known constants, because they are part of the input.) For $\angle srq$ the law of cosines gives us $|qr|^2 + |rs|^2 - |sq|^2 = -2|qr||rs| \cos\angle srq$. After squaring both sides, the left hand side again has degree four, but this time the right hand side has degree only three, since $|rs|$ is a constant. We handle $\angle rsq$ symmetrically.

Using cosines instead of the angles is safe, since cosine is a one-to-one function for angles between 0 and 180 degrees; the only side effect is that since cosine is monotonically decreasing, we must switch to upper envelopes instead of lower envelopes. Squaring both sides of the equation is not so safe, since two numbers whose squares are equal may not be equal themselves, if one of them is negative. Fortunately, most of the elements of the formula are segment lengths, and those are never negative. But the cosine of an angle is sometimes negative, which leads to the following concern: might there be some very obtuse angle, whose cosine is so negative, that after squaring it appears on the upper envelope, thus confusing our algorithm? This can't happen, because if there is some large angle $z > 90°$, then at least one of the other two angles in that triangle has measure no more than $\frac{180°-z}{2} = 90° - \frac{z}{2}$. So the event we're worried about can't happen unless $\cos^2 z > \cos^2(90° - \frac{z}{2})$; i.e., unless $|\cos z| > |\cos(90° - \frac{z}{2})|$. Letting $z' = z - 90°$, we have $|\cos(90° + z')| = |\cos(90° - z')| > |\cos(90° - \frac{90°+z'}{2})|$. But this means that $|\sin z'| > |\sin \frac{z'+90°}{2}|$. Recalling that the angle range we're investigating is $z \in [90°, 180°]$, and thus $z' \in [0, 90°]$, we see that we can drop the absolute value signs. In fact, we can also drop the sin, since on both sides of the inequality, the angle is between 0 and 90 degrees. This leaves us with $z' > \frac{z'+90°}{2}$, which solves to $z' > 90°$, which implies $z > 180°$, which is impossible since no angle in a triangle can be that big. Thus, squaring is safe, and thus we have algebraic functions, and we can use the standard envelope machinery. This completes our description of the unconstrained version of the algorithm.

In the presence of constraints, we can build $\mathcal{D}$ just as easily as before, but knowing the cell of $\mathcal{D}$ that q is placed in is no longer enough information to compute the topology of T_q. The solution chosen in [AAF07] is to refine $\mathcal{D}$ by including not only the constrained Delaunay circles of T, but also all the edges of T itself. In fact, this is overkill, and it suffices to include only those edges present in C, but in the worst case the efficiency is the same either way. The authors of [AAF07] note that this algorithm can easily be adapted to find the best place to insert q so that the smallest angle in the triangulation of a simple polygon is as big as possible: the boundary of the polygon is C, and we simply refrain from checking the cells of $\mathcal{D}$ that lie outside of the polygon.

We finally analyze the runtime. For each cell of $\mathcal{D}$, we find the lower envelope of $O(n)$ different algebraic functions, each of which has bounded degree and a simple domain. This takes time $O(n^{2+\varepsilon})$ per cell, for any $\varepsilon > 0$, with an implied constant depending on ε [SA95, Theorem 7.16]. Since there are $O(n^2)$ cells in the arrangement, this leads to a worst-case running time of $O(n^{4+\varepsilon})$ [AAF07, Theorem 1]. This worst case is achievable: start with n points of a regular polygon, and then perturb slightly to establish general position. Since the Delaunay circles are all nearly identical, every pair intersects, so the complexity of the arrangement is indeed quadratic, and further, a linear number of functions are defined on a linear number of cells, so the average depth is also linear.

Nevertheless, we can in some sense obtain a tighter bound, by being more specific. Let k be the complexity of $\mathcal{D}$, and let d be its maximum *depth*. The depth of a point in $\mathcal{D}$ is simply the number of circles from $\mathcal{D}$ whose interior it lies in, and the maximum depth of $\mathcal{D}$ is the depth of the deepest point. It takes $O(n \log n)$ time to build T [Che87]. We can then build $\mathcal{D}$ in expected time $O(k + n \log n)$ using a standard randomized incremental construction [SA95, Theorem 6.20]. For each of the k cells of $\mathcal{D}$, we find the lower envelope of at most d surfaces, which gives us:

Theorem 3.1. *Given a set P of n points in the plane in general position, and a set C of non-crossing segments with endpoints in P, the Original Algorithm finds a point q in the interior of the convex hull of P, such that the constrained Delaunay triangulation of $P \cup \{q\}$ and C has the largest possible minimum angle. The point q avoids both C and P. The algorithm runs in time $O(n \log n + kd^{2+\varepsilon})$ on any input.*

If the point set is fairly uniformly distributed (as an extreme case, take a square grid), then k is $O(n)$. Intuitively, one might expect a square grid, or

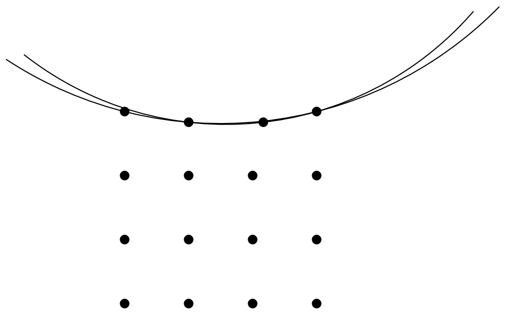


Figure 3.3: A grid with high depth

even a perturbation of a square grid (for general position) to have constant depth, in which case the running time would be $O(n \log n)$, which is essentially the best one could hope for, since it takes that long just to triangulate P. However, a perturbation of a $\sqrt{n}$-by-$\sqrt{n}$ grid (see Figure 3.3) may have cells with depth as high as $O(\sqrt{n})$, leading to a running time of $O(n^{2+\varepsilon})$. To see this, let p be a point infinitesimally below the midpoint of the upper right and upper left grid points. Now consider a circle that goes through p as well as the upper right and upper left grid points. By nudging all points on the top row of the grid down so that they are very near this circle (or on it if we don't care about general position), we will create a point set with $\sqrt{n} - 2$ mutually overlapping Delaunay circles. These overlapping circles contribute $O(n)$ grid cells of high depth. Plugging in to the above formula, we get $n \cdot (\sqrt{n})^{2+\varepsilon} = n \cdot n^{1+\varepsilon} = n^{2+\varepsilon}$.

3.4 The LP Algorithm

The algorithm we describe here has appeared as [AY13a].

Aronov, Asano, and Funke considered, and surprisingly rejected the approach we describe next [AAF07, page 96]. We again present the algorithm first in the context of no constraints, and then patch it later. We use essentially the same algorithm as before, except instead of optimizing the placement of

q using lower envelopes, we use something else. Recall that a star-shaped polygon is one in which there exists a point in the interior that can see the entire polygon boundary. The *kernel* of a star-shaped polygon is the locus of such points. Specifically, in [MSW96], it was shown that the following is an LP-type problem: given a star-shaped polygon H, find the point q in its kernel that maximizes the smallest angle in the triangulation that results from connecting q to all vertices of H. Being an LP-type problem, it can be solved in expected time linear in the number of vertices of H, while the approach from [AAF07], based on explicitly computing lower envelopes of bivariate functions, takes time roughly quadratic in their number. However, this LP-type problem is not quite the problem we actually wish to solve, as we need the optimal placement of q within a given cell of $\mathcal{D}$, rather than anywhere within H's kernel, which may be a much larger area. Otfried Cheong (personal communication [Che13]) optimistically conjectured that this in fact can not cause trouble: if the optimal point within the kernel is not within the arrangement cell, then the globally optimal solution must not have been in that cell. We do not know how to prove this conjecture, so we must make sure our algorithm is correct even if it's false. Fortunately, there is a conceptually simple fix. In the *region search* stage of our procedure, for each cell of $\mathcal{D}$, we run the algorithm from [MSW96], discarding the result if the returned optimum lies outside the cell. A simple argument (see lemma below) shows that if the solution to the LP-type problem results in a point not in the cell, then the optimum within the cell must lie on the cell boundary. So in a separate *boundary search* step detailed below, we find the best placement of q on any cell boundary. Combining the results from the two steps we obtain the globally optimal placement for q.

Lemma 3.2. *If the optimal solution to the LP-type problem corresponding to cell c is not in c, then the optimal solution for c lies on its boundary.*

Proof. Consider the locus $R(z)$ of points q such that every angle in the new triangulation of H has measure at least z. It was shown in [MSW96] that

$R(z)$ is convex; observe that it varies continuously with z when non-empty. If $y < z$, then clearly $R(z) \subset R(y)$. As z decreases from its optimum value, $R(z)$ will gradually grow from a single point outside c and eventually intersect c; as it is connected and changes continuously with z, the first intersection must occur along the boundary of c. The point of contact corresponds to the maximum smallest angle achievable by placing q in c. $\square$

It remains to find the best placement for q on each cell boundary. A cell boundary has two sides, and we process each separately. First consider an edge of $\mathcal{D}$. (Since we are still dealing with the unconstrained case, this edge must be a circular arc.) For a fixed side of a fixed edge e, we know which cell of $\mathcal{D}$ we are in, and thus the hole H. If H has h vertices, the triangulation has $3h$ angles. The measure of each of these angles is a univariate function of the position of q along the edge. To maximize the smallest of these functions, we find the maximum of their lower envelope by computing the envelope explicitly. We show below that the graphs of any pair of these functions intersect at most 16 times. A well-known result from the theory of Davenport-Schinzel sequences immediately implies that the maximum complexity $E(h)$ of the lower envelope is $\lambda_{16}(h)$, where $\lambda_s(n)$ is the maximum length of a DS(s, n) sequence [SA95, Section 1.2]. The maximum length of a DS sequence grows slowly as a function of n when s is constant: it is $o(n \log^* n)$ for any constant s.

Lemma 3.3. *The complexity of the lower envelope of n univariate angle functions is at most $\lambda_{16}(n)$.*

Proof. There are two kinds of angles to consider: angles at the boundary of H, and angles at the new point q. We consider first angles at q. Let $q = (x, y)$, and let r and s be two consecutive vertices of H; the coordinates of r and s are fixed. We are interested in the angle $\angle rqs$ at which q sees the segment rs. Let t, u be another pair of consecutive vertices. The angle at which q sees the segment tu is $\angle tqu$. Consider the locus of points q specified by the equation $\angle rqs = \angle tqu$; a point q satisfying this equation will see rs and tu at the same angle. This locus of points is a curve in the plane, and an intersection between this curve and an edge of $\mathcal{D}$ corresponds precisely to an intersection of the graphs of two angle functions. We aim to prove that there are at most 16 such intersections. For convenience, we will equate the cosines of the angles instead of the angles themselves. (It is possible for two angles to have equal cosines while being unequal, but this does not happen in the angle range between 0 and 180 degrees.) Using $|\cdot|$ to denote segment length, the law of cosines gives $|rs|^2 = |qr|^2 + |sq|^2 - 2|qr||sq|\cos\angle rqs$. Solving for $\cos\angle qpr$ gives

$$\cos\angle rqs = \frac{|qr|^2 + |sq|^2 - |rs|^2}{2|qr||sq|}.$$

Setting $\cos\angle rqs$ equal to $\cos\angle tqu$ produces

$$\frac{|qr|^2 + |sq|^2 - |rs|^2}{|qr||sq|} = \frac{|qt|^2 + |qu|^2 - |tu|^2}{|qt||qu|}.$$

After reshuffling, we obtain

$$(|qr|^2 + |sq|^2 - |rs|^2)|qt||qu| = (|qt|^2 + |qu|^2 - |tu|^2)|qr||sq|.$$

Some quantities in this equation, such as $|rs|$, are simply constants. Others, such as $|qr|$, are functions of x and y. Ideally, we would like polynomial functions, since these are easier to work with, but $|qr|$ is not a polynomial. Luckily, $|qr|^2$ is, so we can make the whole equation polynomial just by squaring both sides. (This has the risk of introducing extra solutions, but this in turn merely means that our bound of 16 may not be tight.) Squaring both sides yields

$$(|qr|^2 + |sq|^2 - |rs|^2)^2|qt|^2|qu|^2 = (|qt|^2 + |qu|^2 - |tu|^2)^2|qr|^2|sq|^2.$$

Now, as can be seen by inspection, each side of the equation is a polynomial in x and y of total degree eight, which means that the locus of points q with $\angle rqs = \angle tqu$ is a curve of degree eight. How many times can such a curve intersect an edge of $\mathcal{D}$? An edge is an arc of a circle, which is a curve of degree two. According to Bézout's theorem [B1779], the number of proper intersection points is at most the product of the degrees, so there can be at most 16 intersection points. A similar argument is needed for $\angle qrs = \angle tqu$ and also $\angle qrs = \angle qtu$, but they also result in polynomial equations of degree at most eight; we omit the entirely analogous calculation. (In some cases, the degree is only two, but since we are concerned with the worst case, this is little comfort.) So, the complexity of the envelope is $O(\lambda_{16}(n))$, and we are done. $\square$

If the worst-case complexity of the lower envelope of h functions from some class is $E(h)$, then we can compute the lower envelope of n functions from that class in $O(E(n) \log n)$ time using a simple divide-and-conquer algorithm [SA95, Theorem 6.1]. This gives us a running time of $O(\lambda_{16}(n) \log n)$ per arc, which would then make our total running time slightly super-cubic if there are a quadratic number of arcs. However, we are duplicating much work: if we follow a Delaunay circle as it crosses another circle, very little changes when we cross: either one triangle of T ceases to be valid, or else one triangle becomes valid. (This assumes that we only cross one circle at time. At a point of P, we may cross many circles at once, so the total change is large, but it is still true that each circle we cross does only one triangle's worth of damage.) Suppose that a triangle of T becomes valid when we cross (the other case is symmetric). Then H loses a boundary vertex, and our triangulation of H loses two old triangles and gains one new one, which means our set of angle functions gains 3 new angles and loses 6 old ones. The other angle functions remain unchanged. Instead of restricting the domain of the angle functions to a single arrangement edge, we allow them to be defined wherever the corresponding angle itself exists (so the domain becomes one or more arcs of a Delaunay circle). On a given arc, there are at most $3n$ functions. If there are m Delaunay circles, then the boundary of a fixed circle can only have $2(m-1) < 2m$ intersections with other circles, and for each of those intersections, at most 6 new functions appear. The number of circles equals the number of triangles, which is less than $2n$. Thus for the entire circle, there are at most a linear number of functions ($2n \times 2 \times 6 + 3n \leq 27n$). It is still the case that any pair of function graphs intersect at most 16 times, but because each is not defined over the entire circle, but only a contiguous arc of it, the complexity of the lower envelope can increase slightly, up to $\lambda_{18}(n)$ [SA95]. This is the complexity of an envelope associated with a single circle, and there are $m = O(n)$ circles. We explicitly compute the m envelopes and find the m associated maxima, $u_1 \ldots u_m$. We also do the region search from the beginning of the section. The algorithm's final answer is either the best value that the region search found, or the biggest u_i, whichever is larger. Thus, the running time of the boundary search stage is $O(n\lambda_{18}(n) \log n)$ and the total running time of our algorithm is dominated by the $O(n^3)$ region search time. This concludes our description and worst-case analysis of the unconstrained version of the LP Algorithm.

3.4.1 Handling constraints

We now describe the changes needed for this algorithm to handle constraints. The first obvious change is that instead of starting with the Delaunay triangulation, we start with the constrained Delaunay triangulation, which can be computed in $O(n \log n)$ time [Che87].

The second change to the algorithm is that the arrangement $\mathcal{D}$ must include not only the constrained Delaunay circles, but also the constrained edges. With this change, knowing the cell containing q gives enough information to determine the star-shaped polygonal hole formed by the invalidated triangles, as discussed in Section 3.3. However, it is important to actually compute this information quickly. (Before, it sufficed to do $O(n)$ in-circle tests, which tell you which triangles to eliminate.) However, if we are willing to spend

$O(n \log n)$ time, we can simply insert an artificial point in the cell, compute the constrained Delaunay triangulation from scratch, and then compare the resulting triangulation to the triangulation T to determine which triangles were invalidated. Thus, the region search takes $O(n^3 \log n)$ time. (By the end of this chapter, this approach will become a bottleneck, and we will see that there is a faster one.)

It remains to handle the boundary search. We argued above that, as we trace along a circle, crossing another circle does only one triangle's worth of damage. Unfortunately, when crossing a constrained edge, it is possible that due to changed visibility, many triangles disappear or reappear. A simple-minded analysis is as follows: at each constrained edge crossing, at most $m < 2n$ triangles appear or disappear. A circle can only cross $e < 3n$ constrained edges. Thus, there are at most $6n^2$ triangle "state changes" along the whole circle boundary. This analysis is only a constant factor away from being tight: start with $n/3$ nearly co-circular points, and then add $n/3$ constrained segments that each clip the circles. Thus there is no longer any sense in doing "whole-circle-at-once" processing, and the running time of the boundary search is $O(n^2 \lambda_{16}(n) \log n)$ by computing envelopes on each arc of the arrangement of Delaunay circles. This dominates the total running time of the whole algorithm. (To figure out which angle functions to take the envelope of, we again simply insert an artificial point on the arc in question and re-triangulate from scratch.) The boundary search can be sped up slightly to $O(n\lambda_{15}(n) \log n)$ by using the algorithm of Hershberger [Her89], yielding:

Theorem 3.4. *Given a set P of n points in the plane in general position, and a set C of non-crossing segments with endpoints in P, the algorithm described finds a point q in the interior of the convex hull of P, such that the smallest angle in T_q is as big as possible. The point q avoids both C and P. The running time of our algorithm is $O(n\lambda_{15}(n) \log n)$ on any input.*

3.4.2 Well-behaved inputs

We now analyze the running time of the LP Algorithm in terms of the complexity k of $\mathcal{D}$ and its depth d. We start by computing the constrained Delaunay triangulation, which can be done in $O(n \log n)$ time. We then compute the arrangement of circles in $O(k + n \log n)$ time using a standard randomized incremental construction [SA95, Theorem 6.20]. In the Original Algorithm, the analog of the region search runs in time $O(kd^{2+\varepsilon})$. We analyze the region search and the boundary search stages of the algorithm separately. The region search runs in expected time $O(kd)$, as its bottleneck is solving k LP-type problems of size at most d each. (Note that this requires that we quickly determine the set of constraints that correspond to a cell. This is easy to arrange if we traverse the arrangement going from a cell to its immediate neighbor. Occasionally we will have to cross a constraint to get to a neighboring cell and compute things from scratch, but if we are careful this will only happen a linear number of times.)

We now turn our attention to the boundary search. Our analysis here needs stronger general position assumptions than the algorithm itself does. In particular, we require that if two Delaunay circles intersect in some point not in P, no third circle passes through that point.

A Delaunay circle that intersects several different constraints is split into multiple pieces (arcs). We perform the boundary search on each piece separately. Let f_i denote the number of functions along circular piece A_i. If A_i intersects x_i other pieces, and the deepest cell adjacent to A_i has depth d_i, then by refining our previous analysis we obtain $f_i \leq 3(d_i + 2) + 6 \cdot 2x_i$. (If the new point is at depth d_i, then the star-shaped hole is composed of d_i triangles and has $d_i + 2$ vertices, and the new triangulation will therefore have $d_i + 2$ new triangles, and three times as many new angles. Each time a circle is crossed, six new angles may appear, and each circle is crossed at most twice.) Note that since these are parts of Delaunay circles, no disk fully contains another. Hence, any circle adjacent to a cell of large depth must intersect many other circles. In particular, $x_i \geq d_i - 1$. Thus, we have $f_i \leq 3(d_i + 2) + 12x_i \leq 3(x_i + 3) + 12x_i = 15x_i + 9$, which is $O(x_i)$.

We now show that the sum of x_i over all pieces is at most proportional to the arrangement complexity k. Let X_i denote the number of Delaunay circles crossed by circle i. Observe that the sum of x_i over all pieces is at most twice the sum of X_i over all circles. This latter sum is simply twice the number of pairs of intersecting circles. Our approach will thus be to show that most pairs of intersecting circles contribute a vertex of degree four to the arrangement $\mathcal{D}$, that is, a vertex that no third circle passes through. Indeed, consider a pair of intersecting circles such that both intersection points, call them r and s, have degree at least six (in a circle arrangement, all vertices have even degree). By our stronger general position assumption, both r and s are from the original point set P. We now have a pair of points with two Delaunay circles passing through it: hence rs must be a Delaunay edge! But there are only a linear number of such edges, so we are done: all but $O(n)$ pairs of intersecting circles contribute a vertex of degree four to the arrangement, and each pair that contributes a vertex of degree four can be charged to the vertex.

Finally, let m be the number of circles, X be the number of pairs of intersecting circles, u be the number of vertices of degree four, and e be the number of edges of the Delaunay triangulation. We now bound the sum of X_i over all circles: $\sum_{i=1}^{m} X_i = 2X \leq 2(u + e) = 2u + 2e < 2k + 2e \leq 2k + 2(n + m - 2) \leq 2k + 2(m + 2 + m - 2) = 2k + 4m < 2k + 4k = 6k$, which is $O(k)$.

Lastly, the total running time of the boundary search is at most proportional to $\sum_{i=1}^{m} \lambda_{18}(X_i) \log X_i \leq \sum_{i=1}^{m} \lambda_{18}(X_i) \log m = \log m \cdot \sum_{i=1}^{m} \lambda_{18}(X_i) \leq \log m \cdot \lambda_{18}(\sum_{i=1}^{m} X_i) \leq \lambda_{18}(6k) \log m$, which is $O(\lambda_{18}(k) \log n)$. Since this is dominated by the time taken by the region search, we have:

Theorem 3.5. *Given a set P of n points in the plane in general position, and a set C of non-crossing segments with endpoints in P, the LP Algorithm finds* *a point q in the interior of the convex hull of P, such that the constrained Delaunay triangulation of $P \cup \{q\}$ and C has the largest possible minimum angle. The point q avoids both C and P. The algorithm runs in time $O(kd)$ on any input.*

We can slightly refine the above analysis in another direction: recall that we defined d to be the *maximum* depth of the arrangement $\mathcal{D}$. If we let $\bar{d}$ be the *average* depth, over all the cells, the expected running time of the region search can then be bounded by $O(k\bar{d})$, while the running time of the analogous part of the Original Algorithm is $O(\sum_{c \in \mathcal{D}} d_c^{2+\varepsilon})$, where d_c is the depth of cell c; the latter runtime is, roughly, k times the average *squared* depth. The running time of the boundary search is not easily expressed in terms of $\bar{d}$, but it is less likely to dominate the running time of the algorithm.

3.5 The Envelope Algorithm

Both of the previous algorithms partition the plane into k pieces, where k is the complexity of $\mathcal{D}$, and then solve the problem on each piece separately. It turns out that this is overkill: it suffices to partition the plane into $1 + O(|C|)$ pieces. We again first explain the algorithm with no constraints, where the plane need not be partitioned at all. Later, we show that, with very little conceptual difficulty, we can handle constraints at the cost of a drastic slowdown.

We bring back our old friend the angle function, but with a small generalization or two. We actually will have two kinds of angle functions. For each angle $\angle rst$ of T we define a partial function f_{rst} as follows. Its domain is the entire plane minus the locus of placements of q which would cause either edge rs or edge st to not show up in T_q. In other words, its domain is all placements of q such that both of the angle's bounding edges survive in T_q. Wherever f_{rst} is defined, its value is simply the measure of $\angle rst$ in T. We will call these the *old* angle functions, because they come from angles in the old triangulation T. (In the next section, we will discuss the Feasible Region Algorithm, which makes use of so-called good regions. These are precisely the complements of the domains of the old angle functions.)

For each edge rs of T, we define three partial functions: $f_{rs,r}$, $f_{rs,s}$, and $f_{rs,q}$. They all have the same domain: those placements of q such that triangle qrs would appear in T_q. The value of $f_{rs,r}(x, y)$ is the measure that $\angle qrs$ would have if q were placed at (x, y). Likewise, $f_{rs,s}$ measures $\angle qsr$, and $f_{rs,q}$ measures $\angle rqs$. We call these the *new* angle functions, since they come from angles in the new triangulation T_q. (The Feasible Region Algorithm makes use of so-called existence regions, which are precisely the domains of the new angle functions.)

Given T, each function can be constructed explicitly in constant time. Let E be the lower envelope of these functions. Observe that $E(x, y)$ is precisely the measure of the smallest angle in the Delaunay triangulation of $P \cup \{(x, y)\}$. Actually, this is not as obvious as it may sound. This would be trivial if the domain of each angle function corresponded precisely to the existence of the angle in T_q. For the new angle functions, this is true: they are defined if and only if the corresponding new angle is present in T_q. For the old angle functions, the implication only holds in one direction: if an old angle survives a given placement of q, then its angle function is defined there. If it does not survive, its angle function might be defined anyway. In particular, the function remains defined if the angle was split in two by a new edge, but both of its bounding segments remain in T_q. However, in that case, there are two new angles, and they both have measure less than that of the old angle, and hence the old angle is blocked from appearing on the lower envelope.

As the angle functions are well-behaved, the complexity of their lower envelope is $O(n^{2+\varepsilon})$ and we can compute it in that time, and then use standard tools to find its maximum, which we then use as the optimum placement for q.

Theorem 3.6. *Given a set P of n points in the plane in general position,* *the Envelope Algorithm finds a point q in the interior of the convex hull of P, such that the Delaunay triangulation of $P \cup \{q\}$ has the largest possible minimum angle. The expected running time of our algorithm is $O(n^{2+\varepsilon})$ on any input.*

It seems difficult to analyze this algorithm in terms of k and d, since it never computes $\mathcal{D}$ explicitly.

We now generalize the algorithm to allow constraints. First we must compute the constrained Delaunay triangulation, rather than the unconstrained one. Second, the domains of the angle functions must be adjusted to reflect the fact that some edges never go away, even if they have a nonempty lune. Lastly, the domains of the angle functions must be adjusted yet again, to account for the fact that they are clipped by the constraints. This last change causes some difficulty: our angle functions no longer have constant complexity domains, because the natural domain may be clipped by a linear number of constrained segments. Having simple domains is a crucial condition for envelope machinery we've been invoking to work, or at least work quickly. Thus, we must do what the previous two algorithms did: partition the plane into pieces so that the angle functions on each piece are either totally defined, or totally undefined, or at the very least have simple domains, and then solve the problem separately on each piece. One such partition is provided by T itself. However, this will slow the running time down by a factor of n, even if there are only one or two constraints. Instead, we can use a trapezoidal decomposition of the plane based on the constraints in C. The gives us:

Theorem 3.7. *Given a set P of n points in the plane in general position, and a set C of non-crossing segments with endpoints in P, the Envelope Algorithm finds a point q in the interior of the convex hull of P, such that T_q has the largest possible minimum angle. The point q avoids both C and P. Its running time is $O((|C|+1)n^{2+\varepsilon})$ on any input, where k is the complexity of $\mathcal{D}$, the arrangement of the constrained Delaunay circles of P together with the constraints C.*

3.6 The Feasible Region Algorithm

The Envelope Algorithm is simple, but it is only fast in the absence of constraints. Why do constraints slow it down so badly, and can anything be done about it? One way of looking at the problem is that the algorithm explicitly computes the lower envelope of all the angle functions, which is expensive if there are many constraints. In some sense, this is wasteful, because we do not want the entire lower envelope, but only its topmost point. Suppose we have a magical black box (which we will call the *decision procedure*) that takes an angle measurement z, and returns a point $q = (x, y)$ such that (x, y, z) is below the lower envelope, if such a q exists, or says "no" if it doesn't. This still doesn't quite solve our problem, but we are getting close. In particular, since we know the optimum z must be between zero and sixty degrees, this is enough to perform a binary search. We would not get the exact answer, but we could get arbitrarily close. If in addition, we could get our hands on a list of critical z values that includes the exact answer, we could perform a binary search on that list and get the exact answer. This is, in fact, almost exactly the plan we will follow. We will first describe the decision procedure (Subsection 3.6.1), and then we will discuss how to use the decision procedure in our *search procedure* (Subsection 3.6.2) to get the critical z values we need.

3.6.1 Decision procedure

Our decision procedure works by explicitly constructing the locus V_z of all placements of q such that all angles in T_q have measure at least z, and then checking if V_z is empty. It does this by computing two types of regions: "bad" regions, and "good" regions. The good and bad regions are defined in such a way that all angles in T_q have measure at least z if and only if q is in all of the good regions and none of the bad regions. It computes V_z by constructing the good and bad regions, building the arrangement induced by their boundaries,

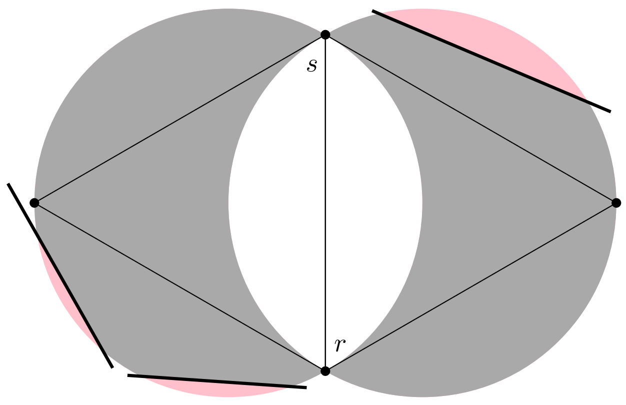


Figure 3.4: E_{rs}, clipped by three constraints

and then traversing this arrangement to label each arrangement feature (face, edge, vertex) as "inside V_z" or "not in V_z," in quadratic time. We now fill in the details of this plan.

Bad regions

Each edge $e = rs$ of T is associated with an *existence region*. This is the locus E_{rs} of points such that placing the new point q there will result in $\triangle qrs$ appearing in T_q. In the absence of constrained edges, E_{rs} would be the symmetric difference of e's two Delaunay disks (that is, their union minus their intersection). To see this, note that for $\triangle qrs$ to be in T_q, two conditions must be satisfied. First, edge rs must be in T_q. Thus, q must not be in e's lune. Second, the circumdisk of $\triangle qrs$ must be empty. Thus, q must be in at least one of e's two Delaunay disks.

In the presence of constraints, E_{rs} is formed, not directly from Delaunay disks, but rather from the clipped visible regions of those disks; see Figure 3.4. If e itself is constrained, then E_{rs} is formed using a union instead of a symmetric difference of the two visible regions, since rs cannot disappear from T_q. Note that, due to clipping, the worst-case complexity of an existence region is linear in n.

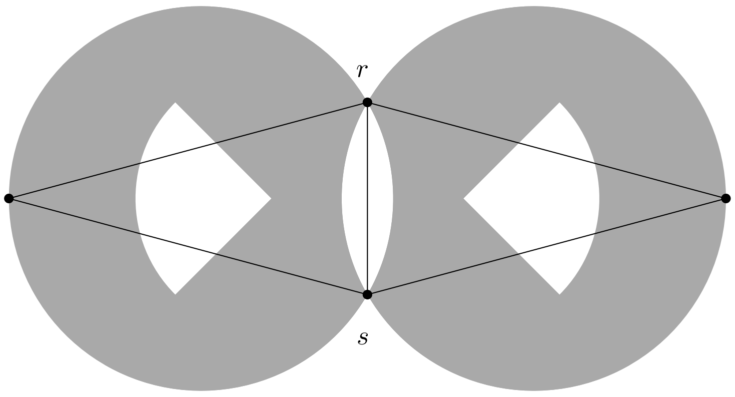


Figure 3.5: B_{rs} (unclipped), when $z = 45°$

For a fixed value of z, we define the *bad region* of rs, denoted by $B_{rs} = B_{rs}(z)$, as the subset of E_{rs} where some angle of $\triangle qrs$ has measure less than z; see Figure 3.5. Our definition is not particularly visual or geometric, so we now describe its consequences for how to draw B_{rs}.

Consider the locus $L_{rs} = L_{rs}(z)$ of points q such that $\angle rqs$ has measure at least z. Since z is less than 90 degrees (indeed, at most 60 degrees), it is the union of two congruent disks, both of whose boundaries pass through r and s. (If z were more than 90 degrees, L_{rs} would look like an American football; that is, it would be the intersection of two disks.) Define $L'_{rs} = L'_{rs}(z)$ to be the locus of points q such that the measure of $\angle qrs$ is at least z; L'_{rs} is the union of two half-planes. Define $L''_{rs} = L''_{rs}(z)$ analogously for $\angle qsr$. Then $B_{rs} = E_{rs} - (L_{rs} \cap L'_{rs} \cap L''_{rs})$ is the bad region associated with rs: if q is in B_{rs} then $\triangle qrs$ appears in T_q and at least one of its three angles has measure less than z.

Note that we need not analyze the effect of constraints on a bad region, since it is simply a subset of its existence region. A bad region can have linear complexity due to clipping by constraints, just like an existence region.

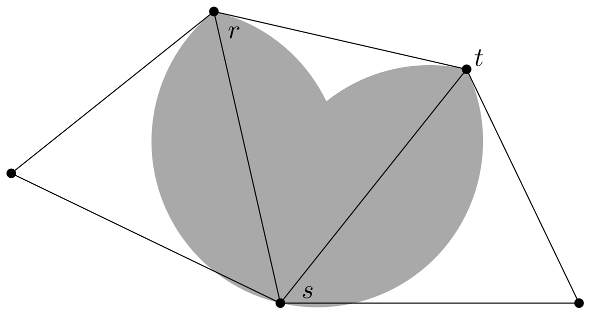


Figure 3.6: $G_{\angle rst}$ (unclipped), when z is greater than the measure of $\angle rst$

Good regions

Intuitively, bad regions are our mechanism to avoid placements of q that create new small angles. Good regions, on the other hand, are meant to ensure that we eliminate all pre-existing small angles. To that end, each angle α of T is associated with a *good region* $G_\alpha = G_\alpha(z)$. If the measure of α is at least z, then G_α is the entire plane. Otherwise, it is the locus of points such that placing q there will eliminate at least one of α's two bounding edges from T_q; see Figure 3.6. (Intuitively, if α is small, we want to merge it with a neighboring angle.) In the absence of constraints, if α measures less than z, then G_α is the union of the Delaunay lunes of α's bounding segments. If one of those segments is constrained, then G_α is simply the lune of the other segment. If both segments are constrained, then G_α is the empty set. Good regions are affected by clipping in the same way as existence regions; that is, in the presence of clipping, a good region is the union of two clipped lunes.

Wrap-up

To conclude, in order to have all angles of T_q larger than z, we must ensure that q lies in no bad region and in every good region. Letting $\alpha(T)$ denote the set of angles of T, we have:

Lemma 3.8. *No angle of T_q has measure smaller than z if and only if $q \in V_z$, where $V_z = \bigcap_{\beta\in\alpha(T)} G_\beta - \bigcup_{e\in T} B_e$.*

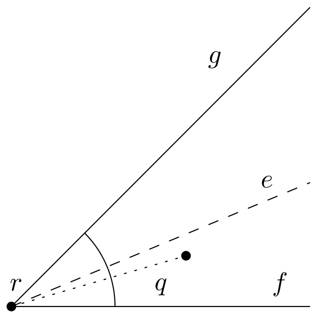


Figure 3.7: Merging two tiny angles does not allow q to sneak into V_z.

Proof. One direction is immediate: if $q \notin V_z$, then there will be some angle smaller than z in T_q, for if q fails to be in some good region, there is a preexisting small angle that remains, and if q ends up in a bad region, a new small angle is created.

It remains to show the other direction: if $q \in V_z$, can there be small angles? There are two possible sources of small angles. The first source is a new small angle created by q: either $\angle qrs$ or maybe $\angle rqs$, for some $rs \in T$. Then q is in B_{rs} and thus not in V_z.

The other source is a small angle already present in T. This also can't happen, because then q fails to be in that angle's good region. This last statement seemingly sweeps a bit under the rug. Suppose two adjacent angles in T are both so small that their sum has measure less than z. The good region requirement is satisfied if we merely merge the two angles by eliminating their shared edge, but we are still left with an angle whose measure is less than z. Luckily, we are saved by the bad regions.

To see how, let one of the small angles be bounded by edges e and f, and the other by e and g; see Figure 3.7. Now suppose insertion of q eliminates e, but f and g remain. By assumption, f and g bound an angle α measuring less than z. Clearly, f and g are two consecutive edges of the star-shaped hole created by q, and thus T_q has an edge connecting q to the common endpoint r of f and g. But qr splits α into two smaller angles! These are both *new*

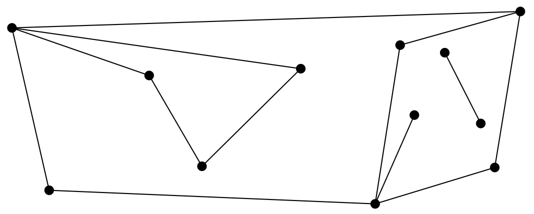


Figure 3.8: Three rooms

angles, and thus are protected from being too small by the bad regions of f and g. (The case of merging three or more consecutive small angles is handled identically.) □

Building V_z

Let the *universe* U be the portion of the plane strictly inside the convex hull of P, minus all constraints of C and all points of P. In general, U consists of several connected components we call *rooms*, separated from each other by the constraints (which can be viewed as walls); see Figure 3.8.

Lemma 3.9. *V_z is a closed set, for $z > 0$.*

Proof. Note that if q stays within a single connected component of U, then the measures of the angles of T_q change continuously with the position of q as long as the combinatorics of T_q don't change. (That is, if T_q is viewed as a graph, then the angles change continuously as long as the graph does not change.) When the combinatorics do change, they change only by edge flips. Such an edge flip does not change the measure of the smallest angle, and thus the smallest angle of T_q changes continuously with the position of q. Therefore, V_z, which we defined as the portion of the plane where all angles have measure at least z, is closed relative to U.

Observe also that the smallest angle of T_q approaches zero if q approaches the boundary of U, and thus V_z never includes the region of the plane near C and P, and therefore V_z is closed, not just relative to U, but also as a subset of the plane. □

We will compute V_z by constructing the overlay of all the good and bad regions. Let $\mathcal{A}_z$ be the arrangement of the boundaries of the good and bad regions before clipping, together with the segments of C. More precisely, $\mathcal{A}_z$ is the part of that arrangement that lies within U. If we were content with roughly cubic running time, we could simply build the arrangement $\mathcal{A}_z$, and then observe that, by construction, each face of the arrangement is either entirely contained in V_z, or entirely outside of it. Likewise, each edge and vertex of $\mathcal{A}_z$ is either entirely contained in V_z, or entirely outside of it. Thus, for each arrangement feature, we generate a point q within it, and build T_q from scratch. If the resulting triangulation has no angles with measure less than z, then that arrangement feature is part of V_z, otherwise it isn't. Computing a constrained Delaunay triangulation from scratch takes $O(n \log n)$ time, so this gives us a decision procedure that runs in $O(n^3 \log n)$ time. We show in this section that we can do much better. In particular, quadratic time is possible.

Observe that V_z is the union of various features (edges, faces, and vertices) of $\mathcal{A}_z$ and, being closed, their boundaries as well. We will compute V_z by computing its part within each room of U separately.

To that end, consider the following version of the adjacency graph G of $\mathcal{A}_z$: there is a node for every feature of $\mathcal{A}_z$ (vertex, edge, and face), and each edge-node is connected to the two face-nodes corresponding to the edge's adjacent faces and also to the (at most) two vertex-nodes corresponding to its adjacent vertices (recall that some vertices and edges are considered removed from $\mathcal{A}_z$ as they are not in U). Each room of U has a connected component of G associated with it.

We sketch in Procedure 3.1 how to build V_z, and in the next few pages go into more detail. The algorithm uses two arrays of bits, GoodFlags and BadFlags; GoodFlags is indexed by the good regions, while BadFlags is indexed by the bad regions. The algorithm will set a flag precisely when the current placement of q is in the corresponding region.

Procedure 3.1 How to build V_z

```
Build $\mathcal{A}_z$ and $G$.
Build a spanning forest $F$ of $G$ in which vertex-nodes appear as leaves.
Allocate and initialize the GOOD- and BADFLAGS arrays.
Annotate $\mathcal{A}_z$ so that it encodes the clipped region boundaries.
For each tree of $F$:
    Pick a face-node corresponding to some face $f$.
    Place $q$ somewhere in face $f$, and compute $T_q$.
    For each good/bad region, check whether $q$ is in it by inspecting $T_q$.
    Initialize the GOOD- and BADFLAGS arrays based on this.
    Mark $f$ as "in $V_z$" or "not in $V_z$."
    Traverse $F$ in depth-first order, starting from $f$.
    For each edge traversed:
        Imagine moving $q$ from the edge's source to its sink.
        Update the flags arrays appropriately.
        Mark the feature as "in $V_z$" or "not in $V_z$."
```

Initialization We build $\mathcal{A}_z$ in expected quadratic time using a randomized incremental construction. In fact, the randomized incremental construction is output-sensitive: if the *complexity* of $\mathcal{A}_z$ is k_z, it runs in expected time $O(k_z + n \log n)$ [SA95, Theorem 6.20]. Notice that k_z is at least linear in n. We need not build G explicitly, since it is implicitly encoded in the adjacency structure of $\mathcal{A}_z$. Forest F is easy to build using ordinary depth-first search.

Maintaining the flag arrays Finally, we must initialize GOODFLAGS and BADFLAGS. Since some good regions cover the whole plane, we do not maintain a flag for those regions, since they would be redundant. Since we haven't placed q anywhere yet, all flags are initially unset, to indicate that q is not in that region. When we enter a new room, we want to quickly reset our flag arrays to this pristine state without having to initialize them from scratch. Thus, on the side, we maintain a list of regions that q is currently in. When we enter a new room, we use that list to quickly see which flags need resetting. This takes time proportional to the number of regions containing q, rather than proportional to n.

Annotating the arrangement After building $\mathcal{A}_z$, we would like to augment it with information related to the good/bad region boundaries that will be useful for computing V_z. In particular, for each edge of the arrangement, we wish to know which good/bad region boundaries it is part of, if any. Let R be a fixed good or bad region, let B denote its unclipped boundary (or more precisely, B is the boundary of R before R gets clipped), and finally let ∂R denote the boundary of R relative to U. (If R has been clipped by constraints, we do not count those constraints as part of ∂R.) We now explain how to trace ∂R through $\mathcal{A}_z$ in time proportional to the number of vertices of $\mathcal{A}_z$ that B passes through. In the absence of constraints, $\partial R = B$ consists of a constant number of line segments and circular arcs and hence is a union of $O(n)$ edges of $\mathcal{A}_z$. We start with a point on ∂R in $\mathcal{A}_z$ and just walk along the edges of $\mathcal{A}_z$ tracing out the boundary, in time proportional to the number of edges and vertices of $\mathcal{A}_z$ visited. (There is a subtlety here: as we saw in Figure 3.5, bad regions can have holes, and thus their boundary can have multiple connected components, and so this procedure needs to be repeated for each component.)

With constraints present, we need a mechanism for clipping R by the constraints. We first identify a point of ∂R that does not lie on a constraint, by picking a point on B and adding constraints one by one. Each constraint either clips off the chosen point or not. If it does, there is an unclipped point adjacent to the new constraint. Once all constraints have been processed, we have a point on the boundary that has not been clipped off. We can optimize this slightly by ignoring constraints that don't cross R. That is, instead of using all constraints of C, we walk along B, and collect all constraints that cross it, and then use those to find an unclipped point. With this optimization, finding an unclipped point takes time proportional to how many constraints B intersects. Since each such intersection contributes to k_z and is only shared by $O(1)$ regions, finding an unclipped point on each region boundary takes $O(k_z)$ time overall.

We then proceed as before, tracing B until it crosses a constraint c, necessarily at a vertex v of $\mathcal{A}_z$. At v, the unclipped B stops coinciding with the clipped ∂R, and will rejoin it when B reaches its next intersection with c, at another vertex w of $\mathcal{A}_z$. Since we know the arcs defining B and we know c, we can compute w in constant time. When we reach v, we set a flag indicating we have left ∂R, and continue tracing B. When we reach w, we unset the flag. As long as the flag is unset, all edges and vertices of ∂R that we trace through $\mathcal{A}_z$ will be marked as "this vertex/edge lies on ∂R (and hence in R if R is a good region and not in R if R is a bad region)," and for edges, we also record which side of the edge lies in R and which outside of it. Note that the segment vw lies on ∂R, but we do not mark it as such since we are computing the boundary of R relative to U. Indeed, we can't afford to mark vw, as it may pass through $\Omega(n)$ vertices of $\mathcal{A}_z$, and B may be clipped by a linear number of such segments, leading to quadratic time to trace ∂R, and thus cubic time to trace all region boundaries.

We repeat this process for all good and bad regions and obtain the arrangement $\mathcal{A}_z$ decorated with $O(k_z)$ pieces of information marking the boundary of each region, relative to U. This takes $O(k_z)$ time, since we spend constant time per feature of $\mathcal{A}_z$ and each feature appears on the boundary of $O(1)$ regions.

Labeling the features We are now ready for the last step: for each feature of $\mathcal{A}_z$, we label it as "in V_z," or "not in V_z". We first build a spanning forest F of G, in which every vertex of $\mathcal{A}_z$ appears as a leaf. We traverse each tree of F in depth-first order, starting at a face-node, as follows.

First, we generate a generic point q in a face f of a room. By construction, either f is completely in V_z or completely outside of it. By running a standard constrained Delaunay triangulation algorithm on $P \cup \{q\}$ and C, we could construct T_q in $O(n \log n)$ time. Instead, by taking advantage of the fact that the old and new point sets are almost identical and the constraints are unchanged, we obtain T_q from T in linear time [FP92]. In fact, it takes time at most proportional to the number of Delaunay triangles in the room, so the total cost of this step over all rooms is $O(n)$. We then compare the angles of T_q against z to decide whether $f \subset V_z$ or not. In fact, we learn more: we can determine if f is contained in or disjoint from every good/bad region and we record this information in two arrays GoodFlags and BadFlags, indexed by the regions. We also keep track of the number of good and the number of bad regions containing f: recall that f is in V_z if and only if it is contained in all good regions and none of the bad ones. Note that we need not measure all the angles of T_q: it suffices to measure the angles in the current room, since angles elsewhere are the same as they were in T.

An edge of F corresponds to either a face/edge incidence or an edge/vertex incidence in $\mathcal{A}_z$. At each face/edge incidence, we have recorded what good/bad regions contain the edge on the boundary and whether the region lies to the left or the right of the edge, which is sufficient to modify the good/bad region counts and the Flags arrays when transitioning between a face and an incident edge, in either direction.

As for an edge/vertex incidence, since vertices of $\mathcal{A}_z$ appear as leaves in F, we will enter a vertex v of $\mathcal{A}_z$ from an adjacent edge e and immediately back out of it, again by e, so we have to explain only how to handle an "edge *to* vertex" traversal, since the next step is just reversing the previous one and can be implemented by recording the changes made and explicitly undoing them. Since a vertex is decorated with the list of good and bad regions whose boundary it is on, we simply use this information to update the Flags arrays when visiting the vertex, and then to undo the update when leaving.

To summarize, for every room of U, we initialize T_q and then traverse the corresponding component of F. For each feature of $\mathcal{A}_z$ encountered in that component, we compute whether it is part of V_z or not.

Lemma 3.10. *Given a set P of n points, a set C of non-crossing edges with endpoints in P, and a number z representing an angle measurement, the decision procedure correctly computes V_z.*

Running Time We define the *complexity* of a room to be the number of features and decorations in the room. Notice that each component of F is traversed at cost proportional to the complexity of the corresponding room. The combined complexity of all rooms is $O(k_z)$, and thus the running time of the decision procedure is $O(k_z)$. However, bounding the running time using k_z is inconvenient, because the decision procedure will be run several times, each with a different value for z and thus of k_z. We'd like to bound the time it takes to build V_z in terms of quantities that don't depend on z. Clearly, k_z is $O(n^2)$, but as we will discuss in the sub-subsection entitled "Well-behaved inputs" immediately following Theorem 3.14, this may be far from tight if the input is "well behaved."

Let $\mathcal{D}$ be the arrangement of the Delaunay circles of T and the constrained edges C; denote its complexity by k. We will show that k_z is $O(k)$. First, observe that k_z is $O(j_z)$, where j_z is the number of pairs of intersecting curves in $\mathcal{A}_z$ (where a curve is either a constraint or the unclipped boundary of a region). Thus, if we wish to upper bound k_z, it suffices to upper bound j_z. We first show that j_z is asymptotically no greater than the number j of pairs of intersecting circles and segments in $\mathcal{D}$, and then show that j is $O(k)$.

In this paragraph we treat all region boundaries as unclipped. There are several contributers to j_z: two (unclipped) good region boundaries intersecting, two (unclipped) bad region boundaries intersecting, a good and a bad region boundary intersecting (again, unclipped), or an unclipped region boundary intersecting a constraint. The good regions that are the whole plane do not have a boundary, and hence don't contribute to j_z. We claim that, for every pair of good regions whose boundaries cross, we can find a pair of Delaunay circles to charge. In fact this is obvious, since the boundary of a good region is composed of arcs from at most three circles. Thus, each pair of intersections in $\mathcal{A}_z$ between two good region boundaries can be charged to the corresponding intersection in $\mathcal{D}$, which will only be charged $O(1)$ times, since a fixed Delaunay circle may appear on the boundary of at most nine good regions (three for its own triangle, and two for each neighboring triangle). Now note that two bad regions intersect only if the corresponding existence regions do, and thus two bad region boundaries can cross only if the boundaries of their existence regions also cross. For existence regions, we apply the same logic as for good regions: there are $O(1)$ circles in $\mathcal{D}$ to charge. An intersection between a good and a bad region boundary can be handled similarly, as can an intersection between a constraint and a region boundary. This establishes that j_z is $O(j)$, and thus k_z is $O(j)$.

It remains to show that j is $O(k)$. In fact, we proved this for the case of no constraints when discussing Algorithm LP in Section 3.4; we rephrase the proof slightly here to handle constraints. Segments of C don't cross each other (though they may share endpoints), so we need only show that most pairs of intersecting circles contribute at least one vertex of degree exactly four to the arrangement, and also that a circle crosses a constraint at a vertex of degree four. Since we assume general position, the fact that a circle crosses a constraint at a vertex of degree four is immediate. Now consider a pair of circles intersecting at r and s, such that both r and s have degree exceeding four in $\mathcal{D}$. By general position of P, we can conclude that r and s both belong to P, and thus rs is a Delaunay edge! Since there are only $O(n)$ such edges, we have $j - O(n) = k$, and thus $j = k + O(n) = O(k)$. This completes the proof that k_z is $O(k)$, and therefore:

Lemma 3.11. *Given a set P of n points, a set C of non-crossing edges with endpoints in P, and a number z representing an angle measurement, the decision procedure computes V_z in $O(k + n \log n)$ expected time, where k is the complexity of $\mathcal{D}$.*

Note that, for a general point set, there is no guarantee that there exists a placement of q such that the smallest angle in T_q exceeds the smallest angle in T. In fact, since we forbid placing q outside the hull, there is no guarantee that we can avoid making things worse. However, we can, if we wish, quickly detect if this is the case by running the decision procedure with z equal to the measure of the smallest angle in T:

Lemma 3.12. *Given a set P of n points and a set C of non-crossing edges with endpoints in P, we can determine whether adding a single point can increase the smallest angle in $O(k + n \log n)$ expected time, where k is the complexity of $\mathcal{D}$.*

3.6.2 Search procedure

We already have the tools needed to numerically approximate the best achievable angle. Since it must be between zero and sixty degrees, we simply perform a binary search on that range using the decision procedure. We now show how to find the exact answer.

We seek the highest value z^* of z such that V_z is not empty. Recall that this can be phrased as seeking a topmost point on the lower envelope of the surfaces defined by the graphs of the various angle functions from the Envelope Algorithm of Section 3.5.

Before we continue, a slight shift in perspective will be helpful. Instead of speaking of the lower envelope of various functions, we will speak of the arrangement $\mathcal{A}$ formed by various surfaces. In particular, consider for instance the bad region B_{rs}. Recall that this region is implicitly parameterized by a z value, and we can make this explicit by writing $B_{rs}(z)$. We can view B_{rs} as a set in three dimensions and $B_{rs}(z)$ as a horizontal slice of this set. We can perform a similar transformation to get three-dimensional good regions. We can form the three-dimensional set V by taking the intersection of all good regions and subtracting the union of all the bad ones, just as in two dimensions. Observe that we seek a topmost point of the set V. Computing

V explicitly is prohibitively expensive, but fortunately not needed. Observe that a topmost point of V must be a topmost point of some cell of $\mathcal{A}$. We now discuss how to refine $\mathcal{A}$ to simplify our analysis (and the algorithm itself).

A two-dimensional good/bad region is bounded by $O(1)$ circular arcs and line segments, and possibly $O(n)$ constraints. In three dimensions, the constraints become vertical walls (since they do not change with z), and the circular arcs and line segments become curved surfaces. We now make some transformations to the surfaces of $\mathcal{A}$, to make our life simpler. First, many surfaces come from clipped regions, which means the surface has high complexity. We can instead use unclipped surfaces, and avoid losing information by inserting into the arrangement the vertical constraint walls. We can simplify further. For instance, a good region in two dimensions is either the whole plane or the union of two lunes. Thus its boundary in two dimensions consists of arcs of the three circles that define the lunes. Therefore in three dimensions, we end up with a half-infinite cylinder whose base is two lunes, and a horizontal plane corresponding to the measure of the good region's angle. We can replace a cylinder whose base is the boundary of two clipped lunes with a cylinder whose base is the boundary of two unclipped lunes. (Recall that we've already added the constraints to our arrangement.) We then further decompose that into two separate cylinders of one lune each.

In addition to decomposing into simpler surfaces, we can also expand a surface to simplify it. For instance, a half-plane can be specified as an equation for a plane plus an inequality specifying where to cut the plane in half. It is simpler to expand a half-plane into a whole plane. In the same vein, a lune is the intersection of two circles, so instead of two cylinders with a lune base (not to be confused with a lunar base; ask NASA about those), we make three circular cylinders. (Not four, since the two lunes shared a circle.) These transformations introduce extra intersections and vertices into $\mathcal{A}$, but its asymptotic complexity remains the same, and this make the analysis below simpler. (In fact, undoing the clipping makes the individual surfaces asymptotically simpler, without which the algorithm would not work at all.) We continue in this fashion, making each cylinder fully infinite instead of half-infinite, and doing similar decomposition and simplification for the bad regions, until each surface of $\mathcal{A}$ can be written as a single polynomial equation. (More precisely, as a single irreducible polynomial equation. That is, it must not be possible factor it. In other words, if the surface is given by $f(x, y, z) = 0 = g(x, y, z)h(x, y, z)$ where g and h are non-constant polynomials, then f is reducible. If we allowed reducible polynomials, then a single equation could easily represent the union of two cylinders, and we wish to avoid such things.) Finally, we gather all these surfaces into a set S.

As promised, the new arrangement $\mathcal{A}$ is a refinement of the old one: some cells are now divided into smaller pieces, but nothing else has changed. We've already observed that a topmost point of V is also a topmost point of some cell of $\mathcal{A}$, and this is still true after refinement. This means a topmost point of V must either be at a local maximum of a surface of S, or at local maximum of the intersection curve of two surfaces of S, or at a vertex resulting from the intersection of three surfaces of S. (Not quite true, stay tuned for another paragraph or two.)

We now have the tools to achieve a running time of $O(n^3 \log n)$: compute all vertices and local maxima by brute force, and then sort them by z coordinate. After that, perform binary search using the decision procedure. In fact, we can speed this up to cubic, using linear-time median-find in order to avoid sorting: find the median z value, invoke the decision procedure, and eliminate half the candidate z values. Repeat $O(\log n)$ times.

Our set S contains $O(n)$ vertical planes, which we inserted when decomposing the clipped surfaces. In fact, these are redundant and we can remove them from S, because placing q near a constraint results in a small z value. Thus we now redefine S to exclude the constraints.

An annoying technicality

We've been acting as if it's obvious that the algorithm above need only consider triples of surfaces, rather than quadruples or more. Indeed, part of the reason for simplifying the surfaces is to make it even more obvious. We wish to show that this is justified. In fact, as phrased, it is not; looking only at local maxima is not enough, we need to look at some other points. At a local maximum of a surface the normal vector is vertical, but although our surfaces have bounded algebraic degree, they may not be as simple as planes or vertical cylinders, and in particular they might have so-called singular points, where the normal to the surface is not well defined, and we must check those as well. For instance, $z^2 = x^2 + y^2$ describes an infinite double-cone, and at the apex the normal vector is not defined. If our arrangement consists of just this one surface, it has three cells, and the apex of the cone is the topmost point of one of the cells; the apex is a singular point but not a local maximum of the surface. Likewise, the intersection of two surfaces may consist of a number of curves, and we need to check both local maxima (where the tangent is horizontal), and singular points where the tangent is not well defined (because the curve self-intersects or makes a sharp turn or does something else exceptional). In fact even this might not be enough: a curve might be singular at a point despite having a well-defined tangent, and we should check those points too. (For instance, consider the two-dimensional curve defined by $x^3 = y^2$. The horizontal line $y = 0$ is tangent to it at the origin, but the curve has a cusp there, and indeed the origin is an extremum for the curve; namely its leftmost point.)

To ensure that we do not miss any points that we need to check, we now state precisely several things that we've been hand-waving up till now, such as the representation of a surface, and the definition of a singularity. A *surface* is the locus of points (x, y, z) with $f(x, y, z) = 0$, where f is a polynomial function in x, y, and z. Without loss of generality, we assume the polynomials are square-free and irreducible. (A polynomial f is *square-free* if there does not exist a non-constant polynomial g such that g is divisible by g^2. Notice that if such a g exists, then f and f/g define the same surface.)

A *singular point* of a surface defined by f is a point (x, y, z) on the surface where the gradient vector is zero; that is where $f(x, y, z) = f_x(x, y, z) = f_y(x, y, z) = f_z(x, y, z) = 0$; subscripts denote partial derivatives. Roughly, at a singular point the surface defined by f does not have a well-defined normal. The *singular locus* of a surface is the set of its singular points. The singular locus of a square-free polynomial surface is at most one-dimensional.

A point of intersection of two surfaces defined by polynomials f and g is *singular* if the gradients to f and g at the point are linearly dependent. We will also need the singular points of the singular locus of a surface. That is, if the locus is one-dimensional, where are its cusps, self-intersections, and other singularities? Consider the four surfaces $f(x, y, z) = 0$, $f_x(x, y, z) = 0$, $f_y(x, y, z) = 0$, and $f_z(x, y, z) = 0$. If we find the singular points of intersection of each pair of these surfaces, we end up with six sets of singular points. The singular points of the singular locus of f are those points common to all six sets [Eis95] (personal communication, Guillaume Moroz [Mor16]).

We need a technical lemma which, roughly, says that the point we are looking for, or more precisely its z-coordinate, can be obtained by examining single surfaces, pairs of surfaces, or triples of surfaces. Intuitively, this seems almost obvious, but is somewhat more subtle than it appears, because we have not analyzed the surfaces we are working with in detail and there exist polynomials which define surfaces which do strange and surprising things. A classic example is Whitney's umbrella, defined by $x^2 = y^2 z$. Its singular locus is the entire z axis (defined by $x = y = 0$). The negative z axis is called the handle of the umbrella, since the "surface" there is in fact one-dimensional. Thus, the origin is certainly a special point for this surface, since it is where it transitions from being one-dimensional to two-dimensional. In the "arrangement" of just one Whitney umbrella, the origin is a local minimum of a cell. Therefore if our set of surfaces includes Whitney's umbrella, we must make sure to detect the origin as a special point. Unfortunately, the analysis technique we use is not powerful enough to do so. Luckily, it seems that our set of surfaces does not contain things like Whitney's umbrella, and we will take advantage of this to simplify our proof. In particular, the lemma below will assume that all surfaces are everywhere two-dimensional.

Our proof works by enumerating all possible locations for the topmost point of a cell. The list of cases is an artifact of our proof technique. Some of the cases may not correspond to a possible location for a topmost point; this does not affect the algorithm's worst-case asymptotic running time. Also, the cases are not all mutually exclusive; for instance, Case 5 is a special case of Case 7, but we feel that the exposition is clearer this way.

Lemma 3.13. *Let v be a topmost point of some bounded cell c of the arrangement formed by a set of irreducible surfaces in three-space represented by square-free polynomials, with each surface everywhere two-dimensional. The z-coordinate of v must be the z-coordinate of one of the following:*

1. *a local maximum of an individual surface;*
2. *an isolated singular point of an individual surface;*
3. *a local maximum of the singular locus of an individual surface;*
4. *an isolated singular point of the singular locus of an individual surface;*
5. *an isolated intersection point of a pair of surfaces;*
6. *a local maximum of a curve of intersection of a pair of surfaces;*
7. *an isolated singular point of a curve of intersection of a pair of surfaces;*
8. *an isolated intersection point of one surface and the singular locus of another surface; or*
9. *an isolated intersection point of a triple of surfaces.*

Proof. The point v must lie on at least one of the surfaces, since otherwise it could be moved higher within c along a vertical line. Call this surface s. If v is a topmost point of s (that is, a local maximum), then Case 1 applies. Note that in this case, v need not be an isolated topmost point. Indeed, if s is a horizontal plane, then all points of s share the same z value. If the locus of local maxima of s near v is one-dimensional (for instance, if s is a horizontal cylinder), then again all points on the locus share the same z value. Both of these situations fall under Case 1, because we have found the z-coordinate of v by examining local maxima of an individual surface.

Let $\sigma(s)$ denote the set of singular points of s. It consists of zero or more isolated points ($\sigma_0(s)$) and zero or more one-dimensional components ($\sigma_1(s)$). (Since s is defined by a square-free polynomial, $\sigma(s)$ can not contain two-dimensional components.) If v is in $\sigma_0(s)$, then Case 2 applies.

Suppose v is in $\sigma_1(s)$. If v is a local maximum or a singularity of $\sigma_1(s)$ then either Case 3 or Case 4 applies. (Again, if v is not an isolated local maximum, then all nearby local maxima share the same z-coordinate, and the same cases apply.)

Otherwise, v is neither a local maximum nor a singular point of $\sigma_1(s)$. Thus $\sigma_1(s)$ is a smooth curve near v. Since v is topmost in c, it must lie on another surface t which does not contain $\sigma_1(s)$ near v, for otherwise v could be moved higher along $\sigma_1(s)$. Thus v must be an isolated point of intersection between t and $\sigma_1(s)$, and Case 8 applies.

We have seemingly omitted the case where $\sigma_1(s)$ intersects another sheet (call it s_1) of s at v, instead of a new surface t. Since s is everywhere two-dimensional, we can conclude that $\sigma_1(s)$ is the intersection of two sheets s_2 and s_3 of s. The intersection of s_1 and s_2 is contained in $\sigma(s)$, and so is the intersection of s_1 and s_3. Therefore v is the intersection of three branches of $\sigma(s)$, and thus is a singular point of it, which was in fact covered by Case 4.

If none of the preceding cases apply, then v must be a regular point of s. Since it is not a local maximum of s, it must lie on another surface t.

(Otherwise, there are points on the boundary of c higher than v, near v, on s.) As the surfaces are irreducible, s and t can not overlap in a two-dimensional set (for this would imply $s = t$). Therefore s and t intersect near v in an at most one-dimensional set.

If v is an isolated point of $s \cap t$, then Case 5 applies. Otherwise, v is contained in one (or more) branches of the one-dimensional locus of $s \cap t$. If v is a local maximum or a singular point of $s \cap t$, then Case 6 or Case 7 applies. (A non-isolated local maximum is handled as above.)

Finally, if none of the preceding cases apply, then v must be a regular point of $s \cap t$. Therefore, since v is not a local maximum of $s \cap t$, it must lie on another surface r which does not contain $s \cap t$ near v, for otherwise v could be moved higher along $s \cap t$. So, v must be an isolated point of intersection between r and $s \cap t$. Thus, Case 9 applies.

This completes the proof. □

Back to the description of the Search Procedure

We've now justified our claim that, examining every surface in S, every pair of surfaces in S, and every triple of surfaces in S, suffices to find the topmost of V is $\mathcal{A}$, and there is no need to examine quadruples.

To speed up our cubic-time algorithm, we must avoid enumerating all vertices of $\mathcal{A}$, which means we can't afford to find the intersections of all triples of surfaces; that is, those points of $\mathcal{A}$ covered by Case 9 of the lemma. (We can afford to find all points covered by Cases 1–8, since there are only $O(n^2)$ of those and thus brute force is fast enough.) Instead, we wish to perform a binary-like search on the z-values that we would obtain from the enumeration of triples of surfaces of S, without actually listing them all. We present a high-level description of our algorithm in Procedure 3.2, and then explain the steps in more detail.

The first step is to pick a random sample of triples. In particular, we sample each triple independently with probability $p = 1/n$, so our sample

Procedure 3.2 Search Procedure

```
Draw a random sample of triples of surfaces.
Compute the vertices corresponding to each triple in the sample.
Collect the $z$-values of those vertices into set $Z$.
Perform binary search on $Z$ to identify a slab containing the optimal point $v^*$.
Find all vertices in that slab.
Perform a binary search within the slab to obtain the final answer.
```

has $O(n^2)$ triples in expectation. (Incidentally, any value of p between $1/n$ and $1/n^2$ gives similar results.) We must be careful: sampling each triple independently seemingly forces us to perform a cubic number of coin flips, which dooms us to cubic running time. However, since p is small, we will likely reject the first triple, and the second. Let X be a random variable representing how many triples we reject. That is, we reject the first X triples and accept the $(X+1)$st triple. The alert reader will observe that $X+1$ is a random variable drawn from a geometric distribution. Thus, if we can generate such a random variable, it will tell us which triple to pick first. We can repeat this to determine which triple to pick second, third, and so on. We continue until we "run off the end" of our implicit list of triples. It remains to explain how to generate a random variable from a geometric distribution. If W is a random variable uniformly distributed in $(0,1)$, then $\lceil \log_{1-p} W \rceil$ is geometrically distributed with probability parameter p; see [Dev86, Chapter 3, Example 2.2, page 87] for details.

For each triple we pick, we compute the $O(1)$ relevant arrangement vertices and collect their z values. This gives us a set Z with $O(n^2)$ different z values (in expectation). In fact, we could get much fewer than this, because the complexity of the arrangement might be much lower than cubic. To be more precise: each vertex of $\mathcal{A}$ gets picked with probability p, so if there are g vertices in $\mathcal{A}$, then pg vertices get chosen, in expectation. Thus, Z partitions the set of g candidate z values into $pg+1$ intervals. We perform a binary search on Z which narrows it down to a single interval $[z_0, z_1]$ and then explicitly enumerate all vertices within the slab $I = \mathbb{R}^2 \times [z_0, z_1]$. Finally, we perform binary search on the vertices in I using the decision procedure, and we're done.

For this to be efficient, we must show that the expected number of vertices in the slab I is small. To see how small it is, imagine starting at the vertex v^* in I and moving up in the positive z direction. As we move up, we will pass vertices of $\mathcal{A}$, until eventually we pass the vertex whose z coordinate is z_1. Each vertex we pass corresponds to the intersection of some triple of surfaces. For each vertex we pass, we mark it (in our imagination) if this is the first time we've passed a vertex corresponding to that particular triple of surfaces. We now attempt to count how many vertices we mark. Since we sample each triple independently with probability p, we expect to pass $1/p$ distinct triples before hitting z_1. Likewise, if we go in the negative z direction, we will pass $1/p$ distinct triples before hitting z_0. Thus, we expect $2/p$ distinct triples in our slab, and since each triple gives rise to $O(1)$ vertices (some of which don't even lie in the slab), the slab contains $O(1/p)$ vertices in expectation.

We now explain how to enumerate all vertices in I. Fix a surface s from S. Intersecting $s \cap I$ with all other surfaces in S produces a collection of $O(n)$ curves in s forming an arrangement $\mathcal{A}_s$. We find the vertices of $\mathcal{A}_s$ by using a randomized incremental construction, in $O(k_s + n \log n)$ expected time, where k_s is the complexity of $\mathcal{A}_s$. But k_s is at most proportional to n plus the number of pairs of intersecting curves in $I \cap s$. The number of pairs of intersecting curves, when summed across all surfaces s, is at most proportional to the number of triples of surfaces that contribute to a vertex in I, which we've already determined is $O(1/p)$ in expectation. Thus, the expected time to enumerate all vertices of I is $O(n^2 \log n + 1/p) = O(n^2 \log n + n) = O(n^2 \log n)$. Performing a binary search on these vertices (by repeated calls to the decision procedure) identifies the critical value z^* of z in time $O(n^2 \log n)$, as claimed. This finally gives us:

Theorem 3.14. *Given a set P of n points in the plane in general position, and a set C of non-crossing segments with endpoints in P, the Feasible Region*

Algorithm finds a point q in the interior of the convex hull of P, such that the constrained Delaunay triangulation of $P \cup \{q\}$ and C has the largest possible minimum angle. The point q avoids both C and P. The expected running time of our algorithm is $O(n^2 \log n)$ on any input.

Well-behaved inputs

We now show how to modify the search procedure so that our algorithm's running time is better than the worst case if the input is "well behaved," without harming the worst-case behavior. Recall that the decision procedure takes time proportional to $k + n \log n$, where k is the complexity of $\mathcal{D}$. We would like to modify the search procedure so that the expected running time of our algorithm is not much worse than $O(k + n \log n)$.

For the purposes of this modification, we will need to partially undo the simplification we applied to the surfaces of S. We again start with the two-dimensional good and bad regions, and again extend them into three dimensions. We then again undo the clipping by constraints. However, we do not decompose the region boundaries into irreducible surfaces. We abuse our notation and from now on refer to this set of surfaces as S. Observe that we still seek the highest point in some cell of the arrangement $\mathcal{A}$ induced by these surfaces. Since that point is defined by at most three irreducible surfaces, it must also be defined by at most three "bundled" surfaces. Intersecting three bundled surfaces takes constant time, since each is composed of $O(1)$ patches of irreducible surfaces. (That is, for each surface σ bounding a three-dimensional good/bad region, there exist a set Σ of $O(1)$ irreducible surfaces, such that $\sigma \subset \bigcup_{s \in \Sigma} s$.)

Notice that we can still afford to enumerate all the local maxima (and other special points) of surfaces in S, since there are only $O(n)$ of them. We can't afford to enumerate all the local maxima corresponding to the intersections of two surfaces because of a technicality with the good region surfaces. Recall that they consist of an infinite cylinder and a horizontal plane. The horizontal planes intersect with all the cylinders, creating too many intersections. However, all these intersections contribute only one z value per surface: namely, the measurement of an angle of T. Thus, if we add the angles of T to our set of candidates Z, we can then eliminate the horizontal planes from our arrangement. We will abuse our notation and continue to refer to the arrangement of the remaining surfaces as $\mathcal{A}$. Now we can afford to enumerate all pairs of intersecting surfaces, because of the following two facts:

Lemma 3.15. *(1) There are $O(k)$ pairs of intersecting surfaces. (2) They can all be found in $O(k)$ time.*

Proof. (1) If two surfaces in space intersect, then so do their projections in the plane. Thus, if we wish to bound the number of pairs of intersecting surfaces in space, it suffices to bound the number of pairs of projections that overlap. Let $f(s)$ denote the projection of surface s. If s is the surface bounding some 3D region R, then $f(s)$ is $R(60°)$. That is, if s bounds a 3D good region corresponding to some angle α, then $f(s) = G_\alpha(60°)$ is the union of two Delaunay lunes, or possibly just one or zero, depending on the presence of constraints. (Recall that we removed the horizontal plane-like portions of the good surfaces.) Similarly, if s bounds a 3D bad region corresponding to edge rs, then $f(s) = B_{rs}(60°) = E_{rs}$ is simply the existence region of rs. Since good regions and existence regions don't nest, the only way that two regions can overlap is if their boundaries cross. As we already know from the analysis of the decision procedure (see page 76), the number of pairs of regions whose boundaries cross each other is $O(k)$.

(2) To get a list of all surfaces that might intersect s, we simply traverse $\mathcal{D}$ to see which region boundaries intersect that of $f(s)$, which takes $O(k)$ time over all surfaces. □

We now show how to sample the vertices of $\mathcal{A}$ to get the running time we want. It turns out that simply tweaking p will not work. To see this, let p

denote the probability of sampling a triple. We would spend $\Theta(n^3)p$ expected time collecting a sample, and as we saw previously, the expected number of vertices in the slab containing z^* is $\Theta(1/p)$. The total expected running time would then be at least $\Omega(n^3p+1/p)$, and thus the best we could hope for with this approach is an expected running time of $\Omega(n\sqrt{n})$, by setting $p = 1/n^{1.5}$, even if $\mathcal{A}$ had far fewer than n^3 vertices.

The key is to bias our sampling of triples so that we are more likely to pick those that intersect. For each surface s, we can get a conservative list L_s of the surfaces it might intersect, by using the method in the preceding lemma. We can use these lists to bias our sampling of triples of surfaces. Namely, we sample triples of the form (s, t_1, t_2) with some probability p to be fixed later, where t_1 and t_2 are distinct surfaces from L_s.

There is a small glitch in the above sampling scheme which causes no difficulty in implementation, but makes analysis slightly more awkward. The sampling algorithm described above might sample the same triple of surfaces up to six times: once for each permutation. Triples get sampled only once if we assign a number to each surface and only output triples whose surface numbers are in increasing order.

Our algorithm now samples each triple that may contribute a vertex to $\mathcal{A}$ with probability p, just like the brute force algorithm did. It may sample some triples which contribute no vertices, which is harmless aside from wasting time. If we let c_s be the size of L_s and let $K = \sum_s c_s^2$, then the expected sampling time is $O(pK)$, as is the expected size of Z. As before, the expected number of vertices in the slab that contains v^* is $O(1/p)$.

To bound the total running time, we bound the time taken by the individual steps in Procedure 3.2. Sampling takes time $O(pK)$. Both binary searches take time $O((k+n \log n) \log n)$. We expect $O(1/p)$ vertices in the slab, so gathering the vertices in it takes at least that long, but there is also some overhead, since we need to construct several 2D arrangements. The time to construct the arrangement for the surface s is proportional to $c_s \log c_s + k_s \leq O(c_s \log n + k_s)$, since there are c_s different curves in the arrangement. (Recall that k_s is the complexity of the portion of the arrangement that lies within the slab.) Summing over all s, we have $O(\sum_s c_s \log n + k_s)$. The sum of c_s over all surfaces s is simply the number of pairs of surfaces that intersect, which we've already seen (see Lemma 3.15) is bounded by $O(k)$, so the sum simplifies to $O(k \log n + \sum_s k_s)$. To bound the sum of k_s over all surfaces s, observe that $k_s = c_s + v_s$, where v_s is the number of vertices in $\mathcal{A}_s$ within the slab I. So our sum simplifies to $O(k \log n + \sum_s c_s + v_s) = O(k \log n + k + \sum_s v_s) = O(k \log n + \sum_s v_s) = O(k \log n + 1/p)$, which is the time needed to build all the arrangements. The $k \log n$ is a lower-order term, since it is dominated by the time for the binary searches. The total time is thus $O(pK + (k + n \log n) \log n + 1/p)$. It remains to set p so as to minimize the running time. Setting $pK = 1/p$ gives us $p = 1/\sqrt{K}$. Our running time is then $O(\sqrt{K} + (k + n \log n) \log n)$. Can we say anything about which of these two terms is larger? Indeed we can: the second term must dominate, since k is $\Theta(\sum_s c_s)$, while $K = \sum_s c_s^2$, and thus K is $O(k^2)$, and therefore $\sqrt{K}$ is $O(k)$. Our main theorem follows:

Theorem 3.16. *Given a set P of n points in the plane in general position, and a set C of non-crossing segments with endpoints in P, the Feasible Region Algorithm finds a point q in the interior of the convex hull of P, such that T_q has the largest possible minimum angle. The point q avoids both C and P. The expected running time of our algorithm is $O((k + n \log n) \log n)$ on any input, where k (which is at most quadratic) is the complexity of $\mathcal{D}$, the arrangement of the constrained Delaunay circles of P together with the constraints C.*

3.7 Concluding remarks and open problems

It would be nice to know which, if any, of the algorithms presented here can be extended to handle two or more points, and if this would lead to a speed-up relative to the method of [AAF07]. Unfortunately, even if it would, the running time would still be of the form $n^{O(m)}$, where m is the number of points to add. (We have a smaller exponent though.) Since even an exponent of ten or twelve would lead to a running time that is for all practical purposes infinite even for a fairly small point set, it would be nice to find a way to reduce the dependence on m.

Is there a significantly faster approximation algorithm for our problem? John Iacono conjectured (personal communication [Iac14]) that if q is restricted to lie within the convex hull of P, then the smallest angle in T_q has measure at most twice that of the smallest angle in T. If this were true, then in some sense not adding any point would yield a 2-approximation for this problem. This turns out to be false, as shown in Figure 3.9. However, it is likely that the statement is true for a larger absolute constant; we know of no example that achieves anywhere near eight, even if the added point is not constrained to lie within the hull.

Finally, we know no non-trivial lower bound for this problem. Is it 3SUM-hard to determine whether V_z is empty for a fixed value of z? Need we construct $\mathcal{D}$ or an analogous object to solve the problem?

Acknowledgments

We would like to thank Sylvain Lazard, Xavier Goaoc, and Guillaume Moroz. Their help was vital in getting the proof of correctness for the search procedure onto a sounder footing.

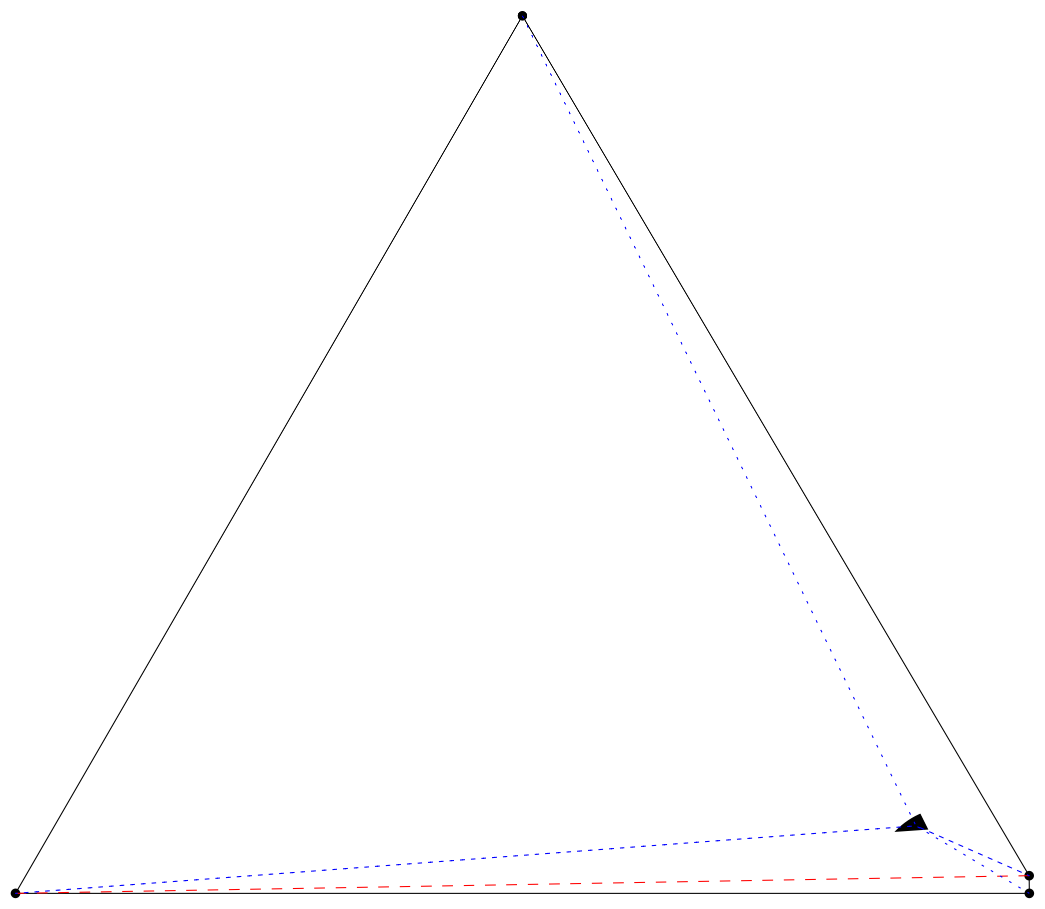


Figure 3.9: The original point set P is indicated by the black dots. The smallest angle in the Delaunay triangulation of P is $1°$. The solid black region is $V_{4°}$. The dashed red edge is removed in the new triangulation and the dotted blue edges are added.

Part II

Data Structures

Chapter 4

Paring Down Your Potential: Pairing Heaps on a Diet

We will first briefly review some facts about pairing heaps, then develop our potential function via a series of attempts that don't quite work. Finally, we will present the actual potential function formally and analyze all the heap operations.

A *pairing heap* [FSST86] is a heap-ordered general rooted ordered tree. That is, each node has zero or more children, which are listed from left to right, and a child's key value is always larger than its parent's. The basic operation on a pairing heap is the *pairing* operation, which combines two pairing heaps into one by attaching the root with the larger key value to the other root as its leftmost child. For the purposes of implementation, pairing heaps are stored as a binary tree using the leftmost-child, right-sibling correspondence. That is, a node's left child in the binary tree corresponds to its leftmost child in the general tree, and its right child in the binary tree corresponds to its right sibling in the general tree. In order to support decrease-key, there is also a parent pointer which points to the node's parent in the binary representation. Priority queue operations are implemented in a pairing heap as follows:

make-heap(): return null

get-min(H): return H.val

insert(H, x): create new node containing x; pair it with root H and update root; return pointer to the newly created node.

decrease-key(p, y): sets the value of the key at p to y, and if p is not the root, detaches p from its parent and pairs it with the root

delete-min(H): removes the root, and then pairs the remaining trees in the resultant forest in groups of two. Then, the remaining trees from right to left are incrementally paired from left to right. (Actually, if the root has an even number of children, this step could just as well be performed from right to left and there would be no difference.) Finally, the new root is returned. See Figure 1.1 for an example of a delete-min executing on a pairing heap. (Readers familiar with splay trees may notice that in the binary view, a delete-min resembles a splay operation.)

All pairing heap operations take constant actual time, except delete-min, which takes time linear in the number of children of the root.

4.1 Towards a functional potential function

Our goal is to construct a potential function which is always between zero and $O(n)$, that gives us logarithmic amortized time for delete-min, where n is the current size of the heap. Since the classic potential function already achieves the second goal, it is natural to attempt to tweak it somehow to achieve the first one without breaking anything. The classic potential function assigns each node x a potential of $\lg |x|$, where $|x|$ is the size of the subtree rooted at x in the binary view. Suppose we simply try to scale down: the classic potential is too large by a logarithmic factor, so we assign each node a potential of $\frac{\lg |x|}{\lg n}$ instead. Unfortunately, this means that a delete-min releases only $O(n/\log n)$ potential, leading to a linear amortized cost. Another idea, if we don't mind randomization, is to flip a biased coin each time a node is inserted into the heap, and, if it lands on tails, then that node will have the classic potential, and otherwise none at all. But if the coin is biased enough to get the potential low enough, then most nodes in the heap will have no potential at all and the amortized cost of delete-min will again be too high.

Let's try something else. Some nodes have very small subtrees, and thus don't contribute much to the potential anyway. Suppose we assign a potential of zero to a node if its subtree size is less than $\lg n$. Pairing two such small nodes certainly won't release any potential, but neither is it likely to cost any potential. Thus, if we have to pair $2s$ small nodes during a delete-min, we will have to pay for those pairings out-of-pocket; that is, they will contribute a cost of roughly s to the cost of delete-min. However, the root of the heap can't possibly have more than $\lg n$ small nodes as children (in the general view), since if a node is small, it must have less than $\lg n$ right siblings. So, our next attempt is to use the classic potential for nodes that are not small, and zero potential for those that are. Then we get the desired run time for delete-min, but the potential can be too large. In particular, inserting nodes in sorted order (reverse sorted also works) yields a heap that looks like a path. Most nodes will use the classic potential, and thus the total potential will be $\Omega(n \log n)$. However, we are making progress. If we could get the binary view of the heap into the shape of a balanced binary tree, the potential would indeed be linear. But instead we ended up with a long path.

Suppose we insisted that both the left and the right subtrees of a node must have size at least $\lg n$, and otherwise the node gets no potential. This certainly solves the problem with long paths having too much potential, but it goes too far: now long paths have no potential at all, and we may well encounter them when performing a delete-min. We need something in between. We end up with three types of nodes. Small nodes where both the left and right subtrees have sub-logarithmic size have no potential. Large nodes where both subtrees have at least logarithmic size have the classic potential. And finally mixed nodes where exactly one of the subtrees has logarithmic size smoothly interpolate between the two extremes. The potential of a mixed node is simply the classic potential times $s/\lg n$, where s is the size of its smaller subtree. As we will show soon (see Lemma 4.1), this is enough to guarantee a linear range for the potential function. Unfortunately, this does not guarantee that delete-min takes amortized logarithmic time.

There is an annoying issue we've been sweeping under the rug ever since we introduced the idea of thresholds where the potential of a node changes sharply. The classification of many nodes could change simultaneously simply because an insertion into the heap increased n, and thus the cutoff $\lg n$, or because a deletion decreased it, and this could have a large effect on the total potential of the heap. We solve this problem in a similar spirit to how one implements a dynamic array. A dynamic array has a capacity which always moves by large jumps; for instance one can constrain it to be a power of two. Changing to a new capacity takes much time, but there is always enough potential saved up to pay for it. Our analogue of capacity is the sticky size N. Our cutoff will be based on $\lg N$, rather than $\lg n$. The sticky size starts at 1, and is doubled after an insertion if n is at least twice N, and is halved after a deletion if N is at least twice n. This allows us to introduce a size potential for the heap, equal to $9|N - n|$. We will see that this is more than enough to allow us to pay for the changes in potential caused by changed node classifications due to changed N.

Our potential function almost works now, and we are ready to consider a proof strategy for delete-min. As in the classic analysis, we analyze the first pairing pass and the second pairing pass separately. For the second pairing pass, we will be content to show that it causes at most a logarithmic increase in potential. Then we will show that the first pass releases enough potential to pay for all pairings performed in both passes. (Since the actual work done by both passes is the same, we can certainly pay for both passes if we can pay for at least one, simply by doubling the potential of the heap; that is, instead of assigning a potential of $\lg n$ to each node, assign a potential of $2\lg n$. We will return to the issue of constant factors shortly.) Thus we simply show that a pairing between two large nodes (in the first pass) must usually release a unit of potential (with at most a logarithmic number of exceptions), and likewise for a pairing between two mixed nodes, and a pairing between a mixed and a large node, and so on.

We should expect a pairing between two large nodes to give us little trouble, since we should be able to reuse the classic analysis. We can also expect little trouble if at least one of the nodes involved in the pairing is small, since there are so few small nodes involved in a delete-min operation. Thus, we should look for trouble in the cases mixed-mixed and mixed-large. It turns out there is a sub-case of a first-pass mixed-mixed pairing using this potential function where the potential of the heap ends up increasing by $O(1)$, instead of decreasing. In that case just one of the nodes involved becomes small, which opens up the following simple patch to our potential function: small nodes still have a potential of zero, while mixed and large nodes have a potential of $4 + \phi_x$, where ϕ_x is the potential of the node x under the slightly broken potential function we are trying to fix. The constant 4 is somewhat arbitrary, but the idea is that we offset the $O(1)$ potential increase by a $O(1)$ decrease caused by a node going from mixed to small.

Finally, we turn to the mixed-large pairings of the first pass. It turns out that with the potential function we have so far, mixed-large pairings fail to release any potential. However, most such pairings performed during the first pass result in the winner of the pairing being large. (That is, the node with the smaller key value will be parent of the node with the larger key value, and this node with a small key will be a large node at the completion of the pairing.) This does not seem helpful at first, until we realize that if three consecutive mixed-large pairings are performed, this means we have a chain of six nodes, three of which are large. At most two of those large nodes were adjacent before the three pairings, and afterwards all three are consecutive siblings. Thus, if we assign potential to edges (in the binary view) as well as to nodes, we have our final solution. Most edges will have zero potential, except for those that connect two large siblings, which have a slight negative potential. (More precisely, an edge has negative potential if it connects a large parent to its large left child in the binary view.) We then split our analysis into two cases: either there are few mixed-large pairings, in which case they are subsidized by the potential released by the other pairings, or else there are many mixed-large pairings: so many that we can count on many such consecutive triples. It remains to adjust the constants so that everything works out. The potential on the negative edges must be -3 or less, since a single edge may have to pay for three consecutive mixed-large pairings. This is doubled to -6 when we recall that the first pass has to pay for the second pass. And finally, since we must also pay for the mixed-large pairings that didn't occur as part of a triple, we need a bit extra. In the interest of sticking with small integers, we set the edge potential to -7. We now have to also increase the potential of the mixed and large nodes so that if most pairings are not mixed-large, we can pay for the few that are. In the interest of working with nice round numbers, we achieve this by multiplying the potential by one hundred.

4.2 The potential function

Our potential function is the sum of three components. The first is the node potential, which will give a value to each node. (The total node potential is the sum of the values for individual nodes.) The second is the edge potential: each edge will have a potential of either 0 or -7. (The total edge potential is likewise the sum of the values of individual edges.) The third we shall call the size potential. We begin with explaining the concept of the sticky size, since we will need it to define all three components. The size n of a heap is how many elements it currently stores. The sticky size N is initially 1. After every heap update, the sticky size is updated as follows: if $n \geq 2N$, then N is doubled, and if $n \leq N/2$, then N is halved. The sticky size is the only aspect of the potential function which is not computable simply from the current state of the heap but is based on the history of operations.

The size potential is simply $900|N - n|$.

The node potential is slightly more complicated. Given a node x, let $|x|$ be the size of its subtree (in the binary view). Its left child in the binary view is x_L and its right child is x_R. The node potential ϕ_x of x depends on $|x_L|$ and $|x_R|$. Note that $|x| = |x_L| + |x_R| + 1$. Let $\lg = \log_2$. There are three cases:

Large node. If $|x_L| > \lg N$ and $|x_R| > \lg N$, then x is *large* and $\phi_x = 400 + 100 \lg |x|$.

Mixed node. If $|x_L| \leq \lg N < |x_R|$, then x is *mixed* and $\phi_x = 400 + 100\frac{|x_L|}{\lg N} \lg |x|$. The case where $|x_R| \leq \lg N < |x_L|$ is symmetric.

Small node. If $|x_L| \leq \lg N$ and $|x_R| \leq \lg N$, then x is *small* and $\phi_x = 0$.

If the right child of a large node is also large, then the edge potential of the edge connecting them is -7. All other edges have zero edge potential.

We define the actual cost of an operation to be one plus the number of pairings performed. Through most of the analysis, it will seem that the node potential is a hundred times larger than what is needed. Near the end, we will see that we use the excess to pay for mixed-large pairings during a delete-min.

4.3 The analysis

In the proofs below, we assume for convenience that the heap has at least four elements, so we can say things like "the root of the heap is always mixed, since it has no siblings and $n - 1$ descendants in the general representation," which assumes that $n-1 > \lg N$, which may not be true for heaps with less than four elements. If we need stronger assumptions on the size, we will call those out explicitly.

Lemma 4.1. *The potential of a pairing heap is $O(N) = O(n)$.*

Proof. The size potential is linear by definition. The edge potential is slightly negative: most edges have potential zero, the exception being those edges that connect two large nodes. If there are L large nodes, the total edge potential may be as low as $-7(L-1)$. But the node potential is at least $400L$, so the edge potential can never make the heap potential negative.

We now turn to the node potential. The small nodes have potential zero. Observe that the lowest common ancestor (in the binary representation) of two large nodes must itself be large. This immediately implies that if the left subtree (in the binary view) of a large node contains any large nodes, then this subtree contains a unique large node which is the common ancestor of all large nodes in this subtree. (And likewise for the right subtree.) Thus, it makes sense to speak of the tree induced by the large nodes, which is the binary tree obtained by taking the original pairing heap and performing the following two operations: first, erase all small nodes (they have no mixed or large descendants). After this, all mixed nodes have only one child, so they form paths. Second, contract these paths to edges. Now only large nodes remain, and ancestor-descendant relations are preserved. (One thing that is not preserved is the root: in the original tree, the root is a mixed node.) If a large node has less than two children in this shrunken tree, call it a *leaf-ish* large node, and otherwise call it an *internal* large node. In the original tree, a leaf-ish large node x has a left and a right subtree (in the binary representation), at least one of which has no large nodes in it: call that a *barren subtree* of x; see Figure 4.1.

Let x and y be two leaf-ish large nodes, and observe that their barren subtrees are disjoint, even if x is an ancestor of y. We say that a leaf-ish large node *owns* the nodes in its barren subtree. Observe that, in the binary

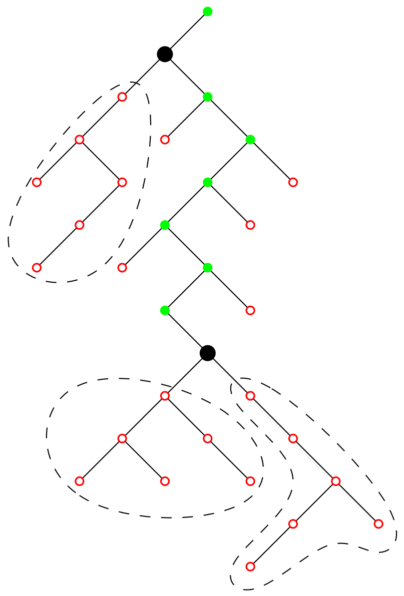


Figure 4.1: The barren subtrees are marked by dashed bubbles; small nodes are marked with hollow red circles, large nodes are marked with bold black disks, and mixed nodes are green.

view, every leaf-ish large node has at least $\lg N$ descendants which are not owned by any other leaf-ish large node, which implies that there are at most $n/\lg N$ leaf-ish large nodes. There are more leaf-ish large nodes than there are internal large nodes (since a binary tree always has more leaf-ish nodes than internal nodes, as can be shown by induction), so there are at most $2n/\lg N$ large nodes in total, each of which has a potential of at most $100\lg n + 400 = 100\lg N + O(1)$, so the total potential of all large nodes is at most $200n + o(n)$.

This leaves the mixed nodes. Every mixed node has a *heavy subtree* with more than $\lg N$ nodes in the binary view, and a possibly empty *light subtree* with at most $\lg N$ nodes. Since the light subtree contains no mixed nodes, every node in the heap is in the light subtree of at most one mixed node; that is, the light subtrees of different nodes are disjoint. If x is a mixed node and

L_x is the size of its light subtree, the node potential of x is

$$\begin{aligned}
\phi_x &= 400 + 100\tfrac{L_x}{\lg N}\lg|x| \\
&\le 400 + 100\tfrac{L_x}{\lg N}\lg n \\
&\le 400 + 100\tfrac{L_x}{\lg N}\lg 2N \\
&= 400 + 100\tfrac{L_x}{\lg N}(1+\lg N) \\
&= 400 + 100\tfrac{L_x}{\lg N} + 100L_x \\
&\le 400 + 100\tfrac{L_x}{\lg 4} + 100L_x \\
&= 400 + 150L_x.
\end{aligned}$$

Summing the potential over all mixed nodes, we have $\sum_x(400+150L_x) = 400n + 150\sum_x L_x$. Since all the light subtrees are disjoint, $\sum_x L_x$ is at most the heap size: n. Thus, the combined potential of all mixed nodes is $550n$, and that of all nodes is $750n + o(n)$. □

We now analyze the five heap operations. All operations except delete-min take constant actual time. The get-min operation does not change the heap so its amortized time is also obviously constant. The make-heap operation creates a new heap with a potential of 900, due to the size potential. This leaves insertion, decrease-key, and deletion, which we handle in that order.

Lemma 4.2. *Insertion into a pairing heap takes $O(1)$ amortized time.*

Proof. The actual time is constant, so it suffices to bound the change in potential. We first bound the potential assuming N stays constant during the execution of the operation. Note that we need not worry about edge potentials here, since inserting a new node can not disturb any existing edges, and creates only one new edge, and the new node is never large, since if it becomes the root it will have no siblings in the general view, and if it does not become the root then it will have no children in the general view. (In the binary view this corresponds to having no right child or no left child.)

There are only two nodes whose node potentials change, the new node and the old root. If the old root has a larger key value than the new node, then the new node becomes the root. They both become mixed nodes with a potential of 400. If the new node is bigger than the old root, then the old root still has a potential of 400, and the new node becomes a mixed node, because it has no children in the general view, and thus also has a potential of 400. Thus, if N does not change, the amortized cost is constant.

However, N could increase to the next power of two. If it does, some mixed nodes may become small and some large nodes may become mixed or even small. These are decreases, so we can ignore them. Also, all mixed nodes that remain mixed will have their potential decrease. What we have to worry about is the edge potential. However, since there are only $O(n/\log n)$ large nodes, the total edge potential can only increase by an amount that is $o(n)$. Meanwhile, if N increases, its new value is the new heap size n, while the old value was $n/2$, so we release $450n$ units of size potential. The increase in potential if N doubles is thus $O(n/\log n) - \Theta(n)$, which is negative for large enough n. How large must n be for this to hold? There were previously at most $\frac{2n}{\lg(n/2)}$ large nodes, and thus at most that many edges with negative edge potential. Thus, we need $\frac{7\cdot 2n}{\lg(n/2)} < 450n$. Dividing both sides by n, we obtain $\frac{14}{\lg(n/2)} < 450$, or $\frac{7}{\lg(n/2)} < 225$. Reshuffling, we get $7 < 225\lg(n/2) = 225(\lg n - 1) = 225\lg n - 225$, or equivalently, $232 < 225\lg n$. Thus, the asymptotic statement actually holds for all $n > 3$. Since a heap that small contains no large nodes at all, the statement holds unconditionally. □

The following observation will be useful when we analyze decrease-key.

Lemma 4.3. *A node's potential is monotone nondecreasing in the size of both of its subtrees (in the binary view) if n is fixed.*

Proof. We must show that increasing the size of the left or right sub-tree of a node never causes its potential to drop. This follows immediately from several simple observations. As long as a node is small, its potential is identically zero and thus monotone. Since the potential is always non-negative, transitioning from small to non-small can only increase the potential. Observe that the formulas for mixed nodes and large nodes can be combined, as follows:

$$400 + 100 \min\left(1, \frac{\min(|x_L|, |x_R|)}{\lg N}\right) \times \lg(|x_L| + |x_R| + 1).$$

This formula is monotone in $|x_L|$ and $|x_R|$, by inspection. □

Lemma 4.4. *Decrease-key in a pairing heap takes $O(\log n)$ amortized time.*

Proof. The actual time is constant, so it suffices to bound the potential change. There are three types of nodes that can change potential: the old root, the decreased node, and the decreased node's ancestors in the binary representation. The ancestors' potential can only go down, since their subtree size is now smaller and potential is a monotone function of subtree size. This leaves just two nodes that might change potential: the root and the node that had its key decreased. But the potential of a node is between zero and $400 + 100 \lg n$ at all times. This leaves the edge potential and the size potential. The size potential does not change. Only $O(1)$ edges were created and destroyed, so the edge potential change due to directly affected edges is negligible. However, it is possible that there is a large indirect effect: some large ancestors of the decreased node might transition from large to mixed, and if those nodes had an incident edge with negative potential, its potential is now zero. Fortunately for us, at most $\lg n + O(1)$ edges can undergo this transition. To see this, let x be the decreased node. The parent of x may transition from large to mixed as a result of losing x, but it can't transition from large to small, because losing x can only affect the size of one of its subtrees, not both. Likewise, x's grandparent may transition from large to mixed, as well as x's great-grandparent, and x's great-great-grandparent = great^2-grandparent, and so on. However, x's great$^{\lg n}$-grandparent still has more than $\lg n$ descendants in both subtrees despite losing x, and so will not undergo this transition. Thus, the change in edge potential is $O(\log n)$. □

Lemma 4.5. *Delete-min in a pairing heap takes $O(\log n)$ amortized time.*

Proof. We break the analysis into two parts. Delete-min changes n, which means it may change N, which may affect the size potential, and the node potential, and the edge potential. The first part of our analysis bounds the change in potential due to changing N, and the second part deals with the delete-min and associated pairings.

Changing N may change small nodes into mixed or even large, and likewise it may change mixed nodes into large. In the case of the edge potentials, this works in our favor, since the edge potentials can only go down when then the number of large nodes increase.

If the new value of N is n and the old value was $2n$, then we release $900n$ units of size potential, while in Lemma 4.1 we showed that the sum of the mixed and large potentials is at most $750n+o(n)$. In fact, we can redo the calculation of that lemma slightly more precisely now, by taking advantage of the fact that we now have $N = n$. There are at most $\frac{2n}{\lg n}$ large nodes, each with a potential of at most $400+100\lg n$, so the total potential of all large nodes is $200n+\frac{800n}{\lg n}$. We then add the mixed nodes for a total of $750n + \frac{800n}{\lg n}$. Thus, we release enough size potential if $900 - 750 = 150 > \frac{800}{\lg n}$. If the heap contains at least 64 items, this inequality holds, since $150{\cdot}6 = 900 > 800$. Could it be that for small heaps, we do not release enough size potential to pay for changing N? If the heap has size 8 or less, then there are no large nodes, and we are then also guaranteed to release enough size potential. Since the size of the heap must be a power of two when N changes, this leaves the heap sizes of 16 and 32 in question. A heap of size 16 has at most one large node, so the relevant inequality is $900 \cdot 16 > 550 \cdot 16 + 400 + 100 \lg 16$. Reshuffling: $350 \cdot 16 > 400 + 100 \cdot 4 = 800$, which clearly holds. Finally, a heap of size 32 has at most three large nodes, so we need $350 \cdot 32 > 3(400 + 100 \lg 32)$. Dividing both sides by 100, we get $3.5{\cdot}32 = 7{\cdot}16 = 7{\cdot}8{\cdot}2 = 56{\cdot}2 = 112 > 3(4+\lg 32) = 12+3{\cdot}5 = 12+15 = 27$, which also clearly holds. It remains to analyze the cost of the pairings performed.

If the root has $c > 0$ children, then a delete-min performs $c - 1$ pairings and thus takes c units of actual time. (If the root has $c = 0$ children, we are doing delete-min on a heap of size 1, which is trivial.) The loss of the root causes a potential drop of 400. Notice first that when two nodes are paired, this does not affect the subtree sizes of any other nodes. There are several cases to consider, depending on the sizes of nodes that get paired, and also depending on whether it is the first or second pairing pass. To avoid confusion: whenever we use the notation $|x|$ to refer to the size of a node x, if x changed size as a result of the pairing we are analyzing, we mean its initial size. Finally, when we pair two nodes, the node that becomes the parent of the other is said to have won the pairing, while the other is said to have lost it.

Let k be the number of pairings performed in the first pass. The number of pairings performed in the second pass is either k or $k - 1$. We will show that the second pass increases the potential by $O(\log n)$ and that the first pass increases the potential by $O(\log n) - 2k$, and thus the amortized cost of delete-min is $O(\log n)$.

We first establish some vocabulary we will use throughout the analysis. Every pairing performed during the delete-min will be between two adjacent siblings (in the general view) x and y, where x is left of y; see Figure 4.2. (In the binary view, y is the right child of x.) We use x_L to denote x's left subtree (in the binary view), y_L for y's left subtree, and y_R for y's right subtree (which is the subtree containing the siblings right of y in the general view).

Finally, let k_{ML} denote the number of pairings performed in the first pass of a delete-min that involve one mixed node and one large node, and let $\Delta\phi_{ML}$ denote the increase in potential as a result of those pairings. We also define k_{LL}, k_{MM} and so on, analogously.

Large-large. We start with a large-large pairing. In this case, the potential function is the same as the classic potential of [FSST86], and the analysis is

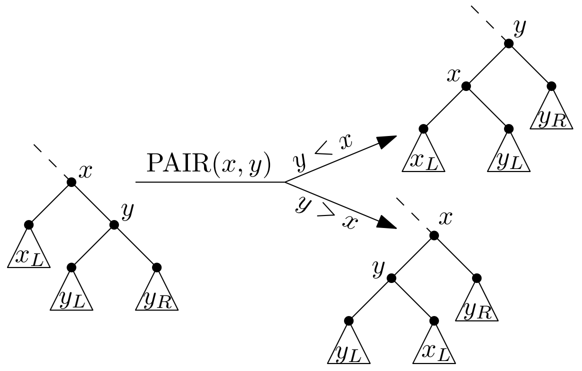


Figure 4.2: A step of the first pairing pass in the binary view.

nearly identical.

First pass. We show that $\Delta\phi_{LL} \le -393k_{LL} + O(\log n)$. We will use the fact that $ab \le \frac{1}{4}(a+b)^2$. (Proof: $ab \le \frac{1}{4}(a+b)^2 \iff 4ab \le (a+b)^2 = a^2+b^2+2ab \iff 0 \le a^2+b^2-2ab = (a-b)^2$, and the square of a real number is never negative.)

We know that $|y_R| > 0$, for otherwise y could not be large (see Figure 4.2). The initial potential of x is $400+100\lg|x| = 400+100\lg(|x_L|+2+|y_L|+|y_R|)$, and the initial potential of y is $400+100\lg|y| = 400+100\lg(|y_L|+1+|y_R|)$. The potential of x and y after the pairing depends on which of them won the pairing, but the sum of their potentials is the same in either case: $800 + 100\lg(|x_L|+|y_L|+1) + 100\lg(|x_L|+|y_L|+2+|y_R|)$. The change P in node potential is

$$\begin{aligned}
P &= 100\lg(|x_L| + |y_L| + 1) - 100\lg(|y_L| + |y_R| + 1)\\
&< 100\lg(|x_L| + |y_L| + 1) - 100\lg|y_R|\\
&= 100[\lg(|x_L| + |y_L| + 1) + \lg|y_R|] + 100[-\lg|y_R| - \lg|y_R|]\\
&= 100\lg[(|x_L| + |y_L| + 1)|y_R|] - 200\lg|y_R|\\
&\le 100\lg\tfrac{1}{4}(|x_L| + |y_L| + 1 + |y_R|)^2 - 200\lg|y_R|\\
&< 100\lg\tfrac{1}{4}|x|^2 - 200\lg|y_R|\\
&= 200\lg\tfrac{1}{4}|x| - 200\lg|y_R|\\
&= 200(\lg|x| - \lg 4) - 200\lg|y_R|\\
&= 200\lg|x| - 200\lg 4 - 200\lg|y_R|\\
&= 200\lg|x| - 400 - 200\lg|y_R|.
\end{aligned}$$

We now sum the node potential change over all large-large pairings done in the first pass. Denote the nodes linked by large-large pairings during this pass as $x_1, \ldots, x_{2k}$, with x_i being left of x_{i+1}. As a notational convenience, let $L_i = 200\lg|x_i|$. Also, let x'_i denote the right subtree of x_i. Note that for odd i, we have $x'_i = x_{i+1}$, but for even i, we don't, since there is no guarantee that all large nodes are adjacent to each other. If we also define $L'_i = 200\lg|x'_i|$, then by the calculation above, the ith pairing raises the potential by at most $L_{2i-1} - 400 - L'_{2i}$. If all large pairings done in the first pass were adjacent, we'd have $L'_{2i} = L_{2i+1}$. Since they need not be, we have $L'_{2i} \ge L_{2i+1}$. Thus, we have a telescoping sum:

$$\begin{aligned}
P &\leq \textstyle\sum_{i=1}^{k}(L_{2i-1} - 400 - L'_{2i}) \\
&\leq \textstyle\sum_{i=1}^{k}(L_{2i-1} - 400 - L_{2i+1}) \\
&= -400k + \textstyle\sum_{i=1}^{k}(L_{2i-1} - L_{2i+1}) \\
&= -400k + \textstyle\sum_{i=1}^{k} L_{2i-1} - \sum_{i=1}^{k} L_{2i+1} \\
&= -400k + (L_1 + \textstyle\sum_{i=2}^{k} L_{2i-1}) - (L_{2k+1} + \sum_{i=1}^{k-1} L_{2i+1}) \\
&= -400k + L_1 - L_{2k+1} + \textstyle\sum_{i=2}^{k} L_{2i-1} - \sum_{i=1}^{k-1} L_{2i+1} \\
&= -400k + L_1 - L_{2k+1} + \textstyle\sum_{i=1}^{k-1} L_{2i+1} - \sum_{i=1}^{k-1} L_{2i+1} \\
&= -400k + L_1 - L_{2k+1} \\
&\leq -400k + L_1 \\
&\leq -400k + 200 \lg n.
\end{aligned}$$

We should also consider the effects of edge potential. In the course of a pairing, five edges are destroyed and five new edges are created. Out of these five, three connect a node to its right child (in the binary view) and thus may have non-zero potential. In the case of a large-large pairing, if the edge that originally connected x to its parent p had negative potential, then the edge that connects the winner of the pairing to p also has negative potential. Likewise, if the edge that connects y to y_R had negative potential, then the edge that connects the winner to y_R does too. However, the edge that connected x to y had negative potential, while the edge that connects the loser to its new right child might not.

Thus, each pairing releases at least 400 units of node potential and costs at most 7 units of edge potential, and therefore $\Delta\phi_{LL} \leq -393k_{LL} + O(\log n)$.

Second pass. The second pairing pass repeatedly pairs the two rightmost nodes. Therefore, one of them has no right siblings in the general representation, which means in the binary representation its right subtree has size zero, which implies that it is not a large node. Hence, in the second pairing pass there are no large-large pairings.

Mixed-mixed. Since x and y are initially mixed, any incident edge has potential zero, so any edge potential change would be a decrease, and thus we can afford to neglect the edge potentials.

First pass. We will show that $\Delta\phi_{MM} \leq -150k_{MM} + O(\log n)$. There are two cases: (1) x and y both have small left subtrees in the binary view, or (2) either x or y has a large left subtree in the binary view. We first handle the second case: at least one of the two nodes being paired has a large left subtree and hence a small right subtree. The left-heavy node can not be x, because y is contained in the right subtree of x, and y can not be mixed if the right subtree of x is small. Thus we conclude that y's right subtree is small. This can happen, but it can only happen once during the first pass of a delete-min, because in that case all right siblings of y in the general view will be small. An arbitrary pairing costs only $O(\log n)$ potential, so this case can not contribute more than $O(\log n)$ to the cost of the first pass.

We now turn to case (1), where x and y both have small left subtrees. The initial potential of x is $400 + 100\frac{|x_L|}{\lg N}\lg|x|$, and the initial potential of y is $400 + 100\frac{|y_L|}{\lg N}\lg|y|$. Observe that whichever node loses the pairing will have left and right subtrees with sizes $|x_L|$ and $|y_L|$. Thus, both its subtrees will be small, and so the loser becomes a small node with a potential of zero. There are two sub-cases to consider: (a) the winning node remains mixed, or (b) it becomes large. We first consider case (a). Since the winner remains mixed, its new potential is $400 + 100\frac{|x_L|+|y_L|+1}{\lg N}\lg|x|$. We will make use of the fact that $|y_L| + 1 < \lg N$, since the winner is mixed. The increase P in potential is:

$$\begin{aligned}
P &= 400 + 100\tfrac{|x_L|+|y_L|+1}{\lg N} \lg|x| - (400 + 100\tfrac{|y_L|}{\lg N} \lg|y|) - (400 + 100\tfrac{|x_L|}{\lg N} \lg|x|) \\
&= -400 + 100\tfrac{|x_L|+|y_L|+1}{\lg N} \lg|x| - 100\tfrac{|y_L|}{\lg N} \lg|y| - 100\tfrac{|x_L|}{\lg N} \lg|x| \\
&= -400 + 100\tfrac{|y_L|+1}{\lg N} \lg|x| - 100\tfrac{|y_L|}{\lg N} \lg|y| \\
&= -400 + 100\tfrac{|y_L|+1}{\lg N} \lg(|x_L| + 1 + |y|) - 100\tfrac{|y_L|}{\lg N} \lg|y| \\
&\leq -400 + 100\tfrac{|y_L|+1}{\lg N} \lg(\lg N + 1 + |y|) - 100\tfrac{|y_L|}{\lg N} \lg|y| \\
&\leq -400 + 100\tfrac{|y_L|+1}{\lg N} \lg(|y_R| + |y|) - 100\tfrac{|y_L|}{\lg N} \lg|y| \\
&< -400 + 100\tfrac{|y_L|+1}{\lg N} \lg(|y| + |y|) - 100\tfrac{|y_L|}{\lg N} \lg|y| \\
&= -400 + 100\tfrac{|y_L|+1}{\lg N} \lg 2|y| - 100\tfrac{|y_L|}{\lg N} \lg|y| \\
&= -400 + 100\tfrac{|y_L|+1}{\lg N}(1 + \lg|y|) - 100\tfrac{|y_L|}{\lg N} \lg|y| \\
&= -400 + 100\tfrac{|y_L|+1}{\lg N} + 100\tfrac{|y_L|+1}{\lg N} \lg|y| - 100\tfrac{|y_L|}{\lg N} \lg|y| \\
&= -400 + 100\tfrac{|y_L|+1}{\lg N} + 100\tfrac{1}{\lg N} \lg|y| \\
&< -400 + 100 + 100\tfrac{1}{\lg N} \lg n \\
&= -300 + 100\tfrac{1}{\lg N} \lg n \\
&< -300 + 100\tfrac{1}{\lg N} \lg 2N \\
&= -300 + 100\tfrac{1}{\lg N}(\lg N + 1) \\
&= -300 + 100(1 + \tfrac{1}{\lg N}) \\
&= -200 + \tfrac{100}{\lg N} \\
&\leq -200 + \tfrac{100}{\lg 4} \\
&= -200 + \tfrac{100}{2} \\
&= -200 + 50 \\
&= -150.
\end{aligned}$$

Thus, if the node that wins the pairing remains mixed, then at least 150 units of potential are released. We now turn to case (b) where the node that wins the pairing becomes large. This can only happen if $|x_L| + |y_L| + 1 > \lg N$. In that case, the increase in potential is

$$\begin{aligned}
P &= 400 + 100 \lg |x| - 100(4 + \tfrac{|y_L|}{\lg N} \lg |y|) - 100(4 + \tfrac{|x_L|}{\lg N} \lg |x|) \\
&= -400 + 100 \lg |x| - 100\tfrac{|y_L|}{\lg N} \lg |y| - 100\tfrac{|x_L|}{\lg N} \lg |x| \\
&< -400 + 100 \lg |x| - 100\tfrac{|y_L|}{\lg N} \lg |y| - 100\tfrac{|x_L|}{\lg N} \lg |y| \\
&= -400 + 100 \lg |x| - 100\tfrac{|x_L|+|y_L|}{\lg N} \lg |y| \\
&\leq -400 + 100 \lg |x| - 100 \lg |y| \\
&= -400 + 100 \lg(|x_L| + 1 + |y|) - 100 \lg |y| \\
&\leq -400 + 100 \lg 2|y| - 100 \lg |y| \\
&= -400 + 100(\lg |y| + 1) - 100 \lg |y| \\
&= -400 + 100 \lg |y| + 100 - 100 \lg |y| \\
&= -300 + 100 \lg |y| - 100 \lg |y| \\
&= -300.
\end{aligned}$$

and thus at least 300 units of potential are released.

Second pass. In the second pass, we are guaranteed that $|y_R| = 0$. The initial potential of y is thus 400. The initial potential of x depends on which of its subtrees is small. But in fact we know that x is right-heavy, or else y could not be mixed. Thus the initial potential of x is $400 + 100\frac{|x_L|}{\lg N} \lg |x| = 400 + 100\frac{|x_L|}{\lg N} \lg(|x_L| + 2 + |y_L|)$. Whichever node wins the pairing will have a final potential of 400 (because it will have no right siblings). Whichever node loses the pairing will have a final potential of $400 + 100\frac{|x_L|}{\lg N} \lg(|x_L| + 1 + |y_L|)$. Thus, there is no potential gain.

Mixed-small and large-small. These types of pairings can only happen once during the first pass. To see this, observe that, in the general view, all right siblings of a small node are small. Therefore we have $k_{MS} + k_{LS} \leq 2$. An arbitrary pairing costs only $O(\log n)$ potential, so $\Delta\phi_{MS} + \Delta\phi_{LS}$ is $O(\log n)$. (The winner of such a pairing is no longer small, so this type of pairing can happen only once during the second pass as well.)

Small-small. We show that the number k_{SS} of small-small pairings performed in both passes is $O(\log n)$, and that the potential increase $\Delta\phi_{SS}$ caused by said pairings is also $O(\log n)$. In one pass, there are fewer than $\lg N$ small-pairings, because a small node has few right siblings. By the same logic as for mixed-mixed, the loser of a small-small pairing remains small. The winner may remain small, in which case the pairing costs no potential. Or the winner may become mixed. However, that can only happen once per pass. For if the winner is mixed, that means that the combined subtree sizes of the two nodes exceeded $\lg N$, which means none of the left siblings were small. Thus, in the first pass, only the first small-small pairing may have the winner become mixed. Thus, $\Delta\phi_{SS}$ is $O(\log n)$.

Mixed-large. We show that the heap potential can increase by at most $O(\log n)$ as a result of all mixed-large pairings performed. We begin with the second pass this time, to get the easy case out of the way first. The mixed-large pairings of the second pass cause no potential increase at all.

Second pass. In the second pass, we are guaranteed that $|y_R| = 0$. The initial potential of y is thus 400. The initial potential of x is $400 + 100\lg|x| = 400 + 100\lg(|x_L| + 2 + |y_L|)$. Whichever node wins the pairing will have a final potential of 400 (because it will have no right siblings). Whichever node loses the pairing will have a final potential of $400 + 100\lg(|x_L| + 1 + |y_L|)$. Thus, there is no increase in node potential. At first it seems that the edge potential could increase, for even though $|y_L| \geq \lg N$ (and likewise for x_L), there is no guarantee that y_L (or x_L) is a large node. On the other hand, there is also no guarantee that the parent of x in the binary view is not a large node. Thus, it is possible that we lose an edge with negative potential and have nothing to replace it with. However, this can only happen once, because the loser of such a pairing becomes a large node and will play the role of y_L in the next pairing, and once any pairing has a large y_L, all subsequent ones will too.

First pass. We have three cases to consider, depending on which of $|x_L|$, $|y_L|$, or $|y_R|$ is small. The case where $|y_R| \leq \lg N$ is easy to dispense with, as it can only happen once during a delete-min. Observe that in the other two cases, the edge potential can't increase, because the winner of the pairing is large. If $|y_L| \leq \lg N$, the initial potential of x is $400 + 100 \lg |x|$, and the initial potential of y is $400 + 100\frac{|y_L|}{\lg N} \lg |y|$. After the pairing, the loser is a mixed node with potential $400 + 100\frac{|y_L|}{\lg N} \lg(|x_L| + 1 + |y_L|)$. The winner is large with a potential of $400 + 100 \lg |x|$. Thus, the increase P in potential is

$$\begin{aligned}
P &= 100\tfrac{|y_L|}{\lg N} \lg(|x_L| + 1 + |y_L|) - 100\tfrac{|y_L|}{\lg N} \lg |y| \\
&= 100\tfrac{|y_L|}{\lg N}[\lg(|x_L| + 1 + |y_L|) - \lg |y|] \\
&= 100\tfrac{|y_L|}{\lg N}[\lg(|x_L| + 1 + |y_L|) - \lg(|y_L| + 1 + |y_R|)] \\
&\leq 100[\lg(|x_L| + 1 + |y_L|) - \lg(|y_L| + 1 + |y_R|)] \\
&< 100[\lg |x| - \lg(|y_L| + 1 + |y_R|)] \\
&< 100[\lg |x| - \lg |y_R|].
\end{aligned}$$

Observe that when we sum over all such mixed-large pairings, we get a telescoping sum of the same form as the one that arose in the analysis of large-large pairings, and thus the combined potential increase for all such pairings is $O(\log n)$. We now turn to the last case, where $|x_L| \leq \lg N$. The initial potential of x is now $400 + 100\frac{|x_L|}{\lg N} \lg |x|$, and that of y is $400 + 100 \lg |y|$. The potential of the winner of the pairing is $400 + 100 \lg |x|$, and the potential of the loser is $400+100\frac{|x_L|}{\lg N}\lg(|x_L|+|y_L|+1)$, so the increase P in potential is

$$\begin{aligned}P &= 100\lg|x| + 100\tfrac{|x_L|}{\lg N}\lg(|x_L|+|y_L|+1) - 100\lg|y| - 100\tfrac{|x_L|}{\lg N}\lg|x| \\ &< 100\lg|x| - 100\lg|y| \\ &< 100\lg|x| - 100\lg|y_R|.\end{aligned}$$

Summing over all such mixed-large pairings, this sum again telescopes in the same way as in the large-large case. Thus, all mixed-large pairings combined cost only $O(\log n)$ potential. However, unlike small-small, the actual cost can be far greater, and unlike large-large, we may not release enough node potential to pay for it. What saves us is the edge potentials.

Call a mixed-large pairing *normal* if $|y_R| > \lg N$. Observe that during the first pass, at most one mixed-large pairing can be abnormal, because if $|y_R| \leq \lg N$, all its right siblings (in the general view) are small nodes. We show that three consecutive normal mixed-large pairings release at least 7 units of potential: enough to pay for those three pairings, with 4 units left over. First, observe that the winner of a normal mixed-large pairing must be large. Thus, for every three consecutive normal mixed-large pairings, at least two large siblings become adjacent that were not adjacent before, and thus some edge has its potential go from 0 to -7. (We must of course be careful that this is not offset by some other edge nearby undergoing the opposite transition, but indeed, we are safe here, because the winner of the pairing is large.) Thus, even though we don't release enough node potential to pay for all mixed large pairings, there is hope that if there are so many of them that they tend to be consecutive, the edge potentials can pay for them instead, while if there are not so many that they tend to be consecutive, perhaps they do not dominate the cost of the first pass. We will soon see that this is indeed the case.

Bringing it all together. We can now calculate the total amortized runtime of delete-min. The actual work done in the second pass is the same as that of the first pass, and the second pass causes at most a logarithmic increase in potential. Thus, we must show that $\Delta\phi \leq -2k + O(\log n)$, where k is the number of pairings done by the first pass and $\Delta\phi$ is the change in potential due to the first pass. We have $k = k_{LL} + k_{MM} + k_{SS} + k_{ML} + k_{MS} + k_{LS} \leq k_{LL} + k_{MM} + \lg N + k_{ML} + 1 = k_{LL} + k_{MM} + k_{ML} + O(\log n)$. The increase in potential from the first pass is $\Delta\phi = \Delta\phi_{LL} + \Delta\phi_{MM} + \Delta\phi_{SS} + \Delta\phi_{ML} + \Delta\phi_{MS} + \Delta\phi_{LS} \leq O(\log n) - 393k_{LL} - 150k_{MM}$. Summing the actual work with the node potential change, we obtain $O(\log n) - 392k_{LL} - 149k_{MM} + k_{ML} \leq O(\log n) - 149(k_{LL} + k_{MM}) + k_{ML}$. Thus, the question is whether most of those terms cancel, leaving us with a $O(\log n) - k$ amortized cost. There are two cases to consider, depending on how large k_{ML} is. If at most $\frac{74}{75}$ of all pairings done are mixed-large (that is, $k_{ML} < \frac{74}{75}k$, or rather $k_{ML} < \frac{74}{75}(k_{LL}+k_{MM}+k_{ML})$, or equivalently $k_{ML} < 74k_{LL} + 74k_{MM}$), then the amortized cost C of the first pass is at most

$$\begin{aligned}
C &\leq O(\log n) - 149(k_{LL} + k_{MM}) + k_{ML} \\
&\leq O(\log n) - 149(k_{LL} + k_{MM}) + 74(k_{LL} + k_{MM}) \\
&= O(\log n) - 75(k_{LL} + k_{MM}) \\
&= O(\log n) - (74 + 1)(k_{LL} + k_{MM}) \\
&= O(\log n) - 74(k_{LL} + k_{MM}) + k_{LL} + k_{MM} \\
&\leq O(\log n) - k_{ML} + k_{LL} + k_{MM} \\
&\leq O(\log n) - k.
\end{aligned}$$

Since the second pass only increases the potential by $O(\log n)$ and its actual cost is k, the cost for the whole delete-min is $O(\log n) - k + k = O(\log n)$.

That leaves the case where more than $\frac{74}{75}$ of parings are mixed-large ones. In fact, we will use the weaker assumption that at least $\frac{20}{21}$ of the pairings in the first pass are mixed-large. We divide the pairings into groups of three: the first three pairings, the second three, and so on. If a group consists of only mixed-large pairings, then it releases 7 units of potential, for a total amortized cost of -4. There are $k/3$ groups (give or take divisibility by three), and all but $k/21$ of those groups consist entirely of mixed-large pairings. These $k/3 - k/21 = 6k/21 = 2k/7$ groups release $8k/7$ units of spare potential. The remaining $k/21$ groups require $k/7$ units of potential to pay for them, leaving k units of spare potential, which we use to pay for the second pairing pass. $\square$

4.4 Final words

Perhaps a similar potential function can be made to work for splay trees, but the one presented here does not. In particular, a splay where every double-rotation is a zig-zag does not release enough potential if the node x being accessed is large and all nodes on the path to x are mixed, so the amortized cost of the splay would be super-logarithmic.

Bibliography

[AAF07] Boris Aronov, Tetsuo Asano, and Stefan Funke; first appeared as: Optimal triangulation with Steiner points. *18th International Symposium on Algorithms and Computation*, ISAAC, December 2007, pages 681–691.

Later as: Optimal triangulations of points and segments with Steiner points. *International Journal of Computational Geometry & Applications*, Volume 20, Issue 1, February 2010, pages 89–104.

[AD11] Boris Aronov and Muriel Dulieu. How to cover a point set with a V-shape of minimum width; first appeared at *Algorithms and Data Structures, 12th International Symposium*, WADS, August 2011, pages 61–72.

Later in *Computational Geometry: Theory and Applications*, Volume 46, Issue 3, April 2013 pages 298–309.

[AG86] Noga Alon and E. Györi. The number of small semispaces of a finite set of points in the plane. *Journal of Combinatorial Theory, Series A*, Volume 41, Issue 1, January 1986, pages 154–157.

[AIÖY13] Boris Aronov, John Iacono, Özgür Özkan, and Mark Yagnatinsky. How to cover most of a point set with a V-Shape of minimum width. *25th Canadian Conference on Computational Geometry*, CCCG, August 2013, pages 211–215.

[AS98] Pankaj K. Agarwal and Micha Sharir. Efficient algorithms for geometric optimization. *ACM Computing Surveys*, Volume 30, Issue 4, December 1998, pages 412–458.

[AY13a] Boris Aronov and Mark Yagnatinsky. How to place a point to maximize angles; first appeared at the *25th Canadian Conference on Computational Geometry*, CCCG, August 2013, pages 259–263.

Later at http://arxiv.org/abs/1310.6413, Oct. 2013 and Jan. 2014.

[AY13b] Boris Aronov and Mark Yagnatinsky. A simple way to place a point to maximize angles; presented at the *23rd Fall Workshop on Computational Geometry*, October 2013; http://www-cs.engr.ccny.cuny.edu/~peter/fwcg13/abstracts/m_yagnatinski.pdf.

[AY14] Boris Aronov and Mark Yagnatinsky. Quickly placing a point to maximize angles. *26th Canadian Conference on Computational Geometry*, CCCG, August 2014, pages 395–400.

[BCKO08] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational Geometry: Algorithms and Applications*, Third Edition, March 2008. Springer-Verlag.

[B1779] Étienne Bézout. Bézout's theorem. From http://en.wikipedia.org/wiki/B%C3%A9zout%27s_theorem; retrieved May 11, 2013.

[Blo70] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, Volume 13, Issue 7, July 1970, pages 422–426.

[BM72] R. Bayer and E. McCreight, Organization and maintenance of large ordered indices, *Acta Informatica*, Volume 1, Issue 3, September 1972, pages 173–189.

[Che87] L. Paul Chew. Constrained Delaunay triangulations. *Proceedings of the Third Annual Symposium on Computational Geometry*, SoCG, June 1987, pages 215–222.
Later in *Algorithmica*, Volume 4, Issue 1, 1989, pages 97–108.

[Che13] Otfried Cheong. Personal communication. Fall 2013.

[Col00] Richard Cole. On the dynamic finger conjecture for splay trees. Part II: The proof. *SIAM Journal on Computing*, Volume 30, Issue 1, April 2000, pages 44–85.

[CY84] Richard Cole and Chee K. Yap, Geometric retrieval problems, *Information and Control*, Volume 63, Issues 1–2 (October–November 1984), pages 39–57.

[Dev86] Luc Devroye. *Non-Uniform Random Variate Generation.* 1986. Springer-Verlag. Full text at http://luc.devroye.org/rnbookindex.html

[Eis95] David Eisenbud. *Commutative Algebra with a View Toward Algebraic Geometry.* 1995. Springer-Verlag.

[Elm09a] Amr Elmasry. Pairing heaps with $O(\log\log n)$ decrease cost. *Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA, January 2009, pages 471–476.

[Elm09b] Amr Elmasry. Pairing heaps with costless meld, `http://arxiv.org/abs/0903.4130` in April 2009.
Later: *Part I of the Proceedings of the 18th Annual European Symposium Algorithms*, ESA, September 2010, pages 183–193.

[EW86] H. Edelsbrunner and E. Welzl, Constructing belts in two-dimensional arrangements with applications, *SIAM Journal on Computing*, Volume 15, Issue 1, February 1986, pages 271–284.

[FP92] L. De Floriani and E. Puppo. An on-line algorithm for constrained Delaunay triangulation. *CVGIP: Graphical Models and Image Processing*, Volume 54, Issue 4, July 1992, pages 290–300.

[Fre99] Michael L. Fredman. On the efficiency of pairing heaps and related data structures. *Journal of the ACM*, Volume 46, Issue 4, July 1999.

[FSST86] Michael L. Fredman, Robert Sedgewick, Daniel D. Sleator, and Robert E. Tarjan. The pairing heap: A new form of self-adjusting heap. *Algorithmica*, Volume 1, Issue 1, 1986, pages 111–129.

[FT84] Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. First appeared in *Proceedings of the 25th IEEE Symposium on the Foundations of Computer Science*, FOCS, October 1984, pages 338–346.
Later in *Journal of the ACM*, Volume 34, Issue 3, July 1987, pages 596–615.

[GP84] Jacob E. Goodman and Richard Pollack. On the number of k-subsets of a set of n points in the plane. *Journal of Combinatorial Theory, Series A*, Volume 36, Issue 1, January 1984, pages 101–104.

[GS78] Leo J. Guibas and Robert Sedgewick. A dichromatic framework for balanced trees. *Proceedings of the 19th Annual Symposium on Foundations of Computer Science*, FOCS, October 1978, pages 8–21.

[Mor16] Guillaume Moroz via Sylvain Lazard. Personal communication. May 2016.

[Her89] John Hershberger. Finding the upper envelope of n line segments in $O(n\log n)$ time. *Information Processing Letters*, Volume 33, Issue 4, December 1989, pages 169–174.

[HST09] Bernhard Haeupler, Siddhartha Sen, and Robert E. Tarjan. Rank-pairing heaps. First appeared at *Proceedings of the 17th Annual European Symposium on Algorithms*, ESA, September 2009, pages 659–670.
Later: *SIAM Journal on Computing*, Volume 40, Issue 6, 2011, pages 1463–1485.

[Iac00] John Iacono. Improved upper bounds for pairing heaps. *The Seventh Scandinavian Workshop on Algorithm Theory*, SWAT, July 2000, pages 32–45.
Later at http://arxiv.org/abs/1110.4428 in 2011.

[Iac14] John Iacono. Personal Communication. May 1, 2014

[IÖ14] Why some heaps support constant-amortized-time decrease-key operations, and others do not.
First version: John Iacono. http://arxiv.org/abs/1302.6641, Feb. 2013.
Later: John Iacono and Özgür Özkan. *41st International Colloquium on Automata, Languages, and Programming*, ICALP, July 2014, pages 637–649.

[Jac02] Riko Jacob, *Dynamic Planar Convex Hull*, PhD thesis, May 2002, University of Aarhus, Denmark, http://brics.dk/DS/02/3

[LL86] D. T. Lee and A. K. Lin. Generalized Delaunay triangulation for planar graphs. *Discrete and Computational Geometry*, Volume 1, Issue 3, September 1986, pages 201–217.

[LST14] Daniel H. Larkin, Siddhartha Sen, and Robert E. Tarjan. A back-to-basics empirical study of priority queues. Initially in *2014 Proceedings of the Sixteenth Workshop on Algorithm Engineering and Experiments*, ALENEX, January 2014, pages 61–72.
Later at: http://arxiv.org/abs/1403.0252, March 2014.

[MSW96] J. Matoušek, M. Sharir, and E. Welzl. A subexponential bound for linear programming. *Algorithmica*, Volume 16, Issue 4, October 1996, pages 498–516.

[O'R98] Joseph O'Rourke. *Computational Geometry in C*. September 1998. Cambridge University Press.

[OvL81] Mark H. Overmars and Jan van Leeuwen. Maintenance of configurations in the plane. *Journal of Computer and System Sciences*. Volume 23, Issue 2, October 1981, pages 166–204.

[Pet05] Seth Pettie. Towards a final analysis of pairing heaps. In *46th Annual IEEE Symposium on Foundations of Computer Science*, FOCS, October 2005, pages 174–183.

[SA95] Micha Sharir and Pankaj K. Agarwal. *Davenport-Schinzel Sequences and Their Geometric Applications.* 1995. Cambridge University Press.

[ST85] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, Volume 32, Issue 3, July 1985, pages 652–686.

[SV86] John T. Stasko and Jeffrey Scott Vitter. Pairing heaps: Experiments and analysis. *Communications of the ACM*, Volume 30, Isuue 3, March 1987, pages 234–249.

[Vui78] Jean Vuillemin. A data structure for manipulating priority queues. *Communications of the ACM*, Volume 21, Issue 4, April 1978, pages 309–315.

[Wir76] Niklaus Wirth. *Algorithms + Data Structures = Programs*, 1976, Prentice Hall.