Algorithms for Computing Convex Hulls Using Linear Programming

Bo Mortensen, 20073241

Master's Thesis, Computer Science

February 2015

Advisor: Peyman Afshani

Advisor: Gerth Stølting Brodal

AARHUS UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

ABSTRACT

The computation of convex hulls is a major field of study in computational geometry, with applications in many areas: from mathematical optimization, to pattern matching and image processing. Therefore, a large amount of research has gone into developing more efficient algorithms for solving the convex hull problem.

This thesis tests and compares different methods of computing the convex hull of a set of points, both in 2D and 3D. It further shows if using linear programming techniques can help improve the running times of the theoretically fastest of these algorithms. It also presents a method for increasing the efficiency of multiple linear programming queries on the same constraint set.

In 2D, the convex hull algorithms include an incremental approach, an intuitive gift wrapping algorithm, and an advanced algorithm using a variant of the divide-and-conquer approach called marriage-before-conquest. The thesis also compares the effect of substituting one of the more time-consuming subroutines of the marriage-before-conquest algorithm, with linear programming.

An extension of the gift wrapping algorithm into 3D is presented and tested, along with a description of an algorithm for using linear programming to compute convex hulls in 3D.

Results of various tests are presented and compared with theoretical predictions. These results show that in some cases, the marriage-before-conquest algorithm is faster than the other algorithms, but when the hull size is a large part of the input, the incremental convex hull algorithm is faster. Additionally, the tests are used to demonstrate that some of the algorithms are output-sensitive.

ACKNOWLEDGEMENTS

First of all, I would like to thank my thesis advisor Peyman Afshani, for his immense patience in trying to explain different concept to me, and for helping me through writing the thesis. I would also like to thank Johan Sigfred Abildskov for helping with form and proofreading of my thesis. Finally, I would like to thank my beautiful girlfriend for her support throughout this process.

CONTENTS

1 INTRODUCTION . . . 1
2 RELATED WORK . . . 2
3 THEORY . . . 3
  3.1 2D Algorithms . . . 3
    3.1.1 Naive Algorithm . . . 3
    3.1.2 Incremental Algorithm . . . 4
    3.1.3 Gift Wrapping Algorithm . . . 6
    3.1.4 Kirkpatrick-Seidel Algorithm . . . 6
    3.1.5 Kirkpatrick-Seidel Algorithm Using Linear Programming . . . 10
  3.2 3D Algorithms . . . 10
    3.2.1 Gift Wrapping Algorithm . . . 10
    3.2.2 3D Linear Programming Algorithm . . . 11
  3.3 Linear Programming . . . 11
    3.3.1 General . . . 11
    3.3.2 Duality . . . 12
    3.3.3 Incremental Algorithm . . . 12
    3.3.4 Randomized Algorithm . . . 14
    3.3.5 Solving Linear Programs in 2D and 3D . . . 15
  3.4 Multiple Linear Programming . . . 19
    3.4.1 General . . . 19
    3.4.2 Partitioning . . . 19
    3.4.3 Algorithm . . . 20
    3.4.4 Running Time Analysis . . . 20
    3.4.5 Termination Condition . . . 21
  3.5 Pruning . . . 24
4 IMPLEMENTATION . . . 25
  4.1 General . . . 25
  4.2 Convex Hull Algorithms . . . 26
    4.2.1 Naive Algorithm . . . 26
    4.2.2 Incremental Algorithm . . . 26
    4.2.3 Gift Wrapping Algorithms . . . 26
    4.2.4 Kirkpatrick-Seidel Algorithms . . . 27
  4.3 Linear Programming . . . 27
  4.4 Tests . . . 27
  4.5 Visualiser . . . 28
5 ANALYSIS . . . 29
  5.1 2D Distributions . . . 29
    5.1.1 Uniform Square distribution . . . 29
    5.1.2 Square Uniform Bounding triangle . . . 30
    5.1.3 Uniform Circle distribution . . . 30
    5.1.4 Donut . . . 32

5.1.5 Polynomial Parabola Distribution 32
5.1.6 Hull Sizes 33
5.1.7 Section Summary 33
5.2 2D Algorithms 34
5.2.1 Incremental Algorithm 34
5.2.2 Gift Wrapping Algorithm 34
5.2.3 Kirkpatrick-Seidel Algorithm 35
5.2.4 Kirkpatrick-Seidel Algorithm using Linear Programming 36
5.2.5 Running Time Ratios 36
5.3 3D Algorithms 38
5.3.1 Sphere 39
5.3.2 Uniform Box 39
5.3.3 Uniform Bounding Box 39
5.3.4 Paraboloid 39
5.3.5 Results 40
5.4 Linear Programming 40
6 CONCLUSION 44
6.1 Future Work 45

1 INTRODUCTION

The computation of convex hulls is one of the most fundamental, and one of the first studied problems, in computational geometry.

The convex hull $CH(S)$ of a set of points S is defined as the minimum set forming a convex shape containing all points in S. A set is convex if any line segment $\vec{pq}$ between any pair of points $p, q \in S$ is completely contained in S.

A convex hull in the plane can be intuitively described by comparing the set of points to nails hammered into a board. The convex hull can then be visualized as a rubber band stretched around the nails, clasping the outer-most nails.

Mathematically, the convex hull of a finite set of points S in n dimensions is defined by the following:

$$CH(S) \equiv \left\{ \sum_{i=1}^{n} \lambda_i p_i : \lambda_i \geq 0 \text{ for all } i \text{ and } \sum_{i=1}^{n} \lambda_i = 1 \right\}.$$

Convex hulls are used in many other areas, such as image processing, path finding and robot motion planning, pattern matching, and many more. For instance, in situations where the performance of a ray tracing renderer is important, as in computer games, convex hulls can help decrease the number of ray-triangle checks, by building convex hulls of concave models. If a ray does not intersect the convex hull, there is no need to check the, usually much more complex, model inside of the hull.

Linear programming is a technique for optimizing equations subject to linear constraints, and dates as far back as Fourier. In linear programming, the goal is to optimize a linear equation, subject to a number of linear constraints. The constraints can be visualized as a set of halfspaces, each cutting off a part of the valid space. The optimization is then finding the furthest allowed point in a given direction.

2

RELATED WORK

Berg, et al. [2] described a simple, naive algorithm for computing convex hulls in the plane. Graham [5] described an incremental algorithm running in $\mathcal{O}(n \log n)$ time, later modified by Andrew [1]. Jarvis [7] described an intuitive approach to computing convex hulls based on the idea of wrapping a gift. Kirkpatrick and Seidel [8] used a variant of the divide-and-conquer approach to compute convex hulls in $\mathcal{O}(n \log h)$ time.

Megiddo [10] described both a deterministic and a randomized approach to solving linear programs in $\mathcal{O}(n)$ time, later summarized by Berg, et al. [2]. Chan[3] later presented a technique for solving multiple linear programming queries simultaneously in $\mathcal{O}(n \log k)$ time, using a recursive partitioning method to prune a large amount of the unused points.

3 THEORY

This chapter introduces the algorithms for computing convex hulls, which are implemented and tested later. Additionally, the theory used for the more advanced algorithms is presented. First, the algorithms for computing convex hulls in 2D are described, which include an algorithm with a naive approach, and a more efficient incremental algorithm. Two advanced algorithms are also given: a gift wrapping algorithm, and a marriage-before-conquest algorithm based on a variant of the divide-and-conquer approach. Afterwards, the algorithms for 3D convex hulls are described, starting with a 3D gift wrapping algorithm. Finally, algorithms for solving linear programs effectively are presented, starting with an incremental algorithm, then a randomized algorithm, and ending with an advanced algorithm for solving multiple linear programs using the randomized algorithm.

In all of the following convex hull algorithms, it is important to be aware of and handle degenerate cases correctly. In 2D, these include handling vertical lines, and a hull line intersecting more than two points. If a vertical line is created through two points, it can, in most cases, be treated as a special case, where only the highest point is used, and the line dropped. In a more general situation, the entire point set can be rotated so vertical lines can not occur. In this thesis, if a hull line intersects more than two points, all the middle points are ignored and are not part of the hull. In 3D, these cases persist, but as a hull in 3D consists of faces, there may be several points on the plane defining the face. In such a case, it may be necessary to find the convex hull of these points using a 2D algorithm.

3.1 2D ALGORITHMS

In this section, the algorithms for computing convex hulls in two dimensions are detailed, starting out with the simplest algorithm and moving up in complexity.

3.1.1 *Naive Algorithm*

Knowing that for each pair of points, all other points must lie to only one side of the line through these two points. Therefore, the simplest

way of solving the convex hull problem is making a list of all possible $(n^2 - n)$ edges from all pairs of points p, q and testing whether all other $(n-2)$ points k lie on the right side of this edge, in which case the edge is part of the hull. This algorithm runs in time:

$$\sum_{i=1}^{n^2-n} \mathcal{O}(n-2) = \mathcal{O}((n-2)(n-1))n = \mathcal{O}(n^3 - 3n^2 + 2n) = \mathcal{O}(n^3).$$

As the algorithm can never terminate early, and always checks all other points, this is both an upper and a lower bound:

$$\Theta(n^3).$$

Pseudocode for this algorithm can be seen in algorithm 1, which is based on [2]. A special check to see if more than two points were on a hull line, as only the maximum and minimum of these points were to be added to the hull.

Algorithm 1 NaiveHull(S)

1: $sort(S)$
2: $a \leftarrow \emptyset$
3: $L_{upper} = \{p_1, p_2\}$
4: **for** All ordered pairs $(p, q) \in P \times P$ with $p \neq q$ **do**
5:     $valid \leftarrow true$
6:     **for** All points $r \in P$, where $r \neq p$ or q **do**
7:         **if** r lies to the left of the directed line $\vec{pq}$ **then**
8:             $valid \leftarrow false$
9:         **end if**
10:     **end for**
11:     **if** $valid$ **then**
12:         Add the directed edge $\vec{pq}$ to E
13:     **end if**
14: **end for**
15: Construct list L of vertices from E, sorted in clockwise order

3.1.2 *Incremental Algorithm*

Algorithm 2 describes an incremental approach to the convex hull problem, which is a variant of Graham's algorithm [5], modified by Andrew [1]. This algorithm divides the problem into computing the top and bottom parts of the hull separately. The idea is to iterate through the points, from the left-most vertex, to the right-most vertex, adding them to the hull, and checking backwards for concavities. This is illustrated in figure 1. If there are concavities present, the next-to-last vertex is removed from the hull. When both the top and

bottom hulls are computed, they are simply concatenated, removing identical endpoints, if present.

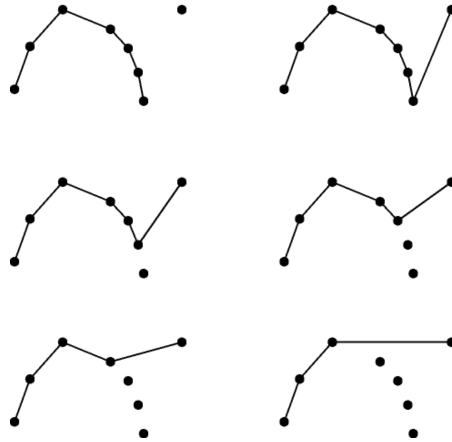

Figure 1: Adding a point

Source: www.ics.uci.edu

With a sorted list, each point is added to the hull once, and each point can be pruned at most once. This means that the algorithm runs in linear time on an already sorted list. Thus, the running time of the incremental algorithm is dominated by the running time of the sorting algorithm used. One of the fastest sorting algorithms is QuickSort [6], which runs in expected $\mathcal{O}(n \log n)$ time. Thus, the incremental algorithm runs in time:

$$\mathcal{O}(n \log n) + \mathcal{O}(n) + \mathcal{O}(n) = \mathcal{O}(n \log n).$$

If the vertices are only sorted by their x-coordinates, and there are several minimum points, the order of them matter. If a lower point is added first, the upper point will not fail, and a lower point will be part of the upper hull. This can be solved either by sorting by y secondarily, or do a special case test. An equivalent case is possible for multiple maxima, with the same solution.

Algorithm 2 IncHull(S)

1: $sort(S)$ by x-coordinate
2: $a \leftarrow median(x)$
3: $L_{upper} = \{p_1, p_2\}$
4: **for** $i \leftarrow 3$ **to** n **do**
5: Add p_i to L_{upper}
6: **while** $size(L_{upper}) > 2$ and last three points of L_{upper} do not make right turn **do**
7: Delete next-to-last point of L_{upper}
8: **end while**
9: **end for**

3.1.3 *Gift Wrapping Algorithm*

Algorithm 3 describes the gift wrapping algorithm, also referred to as Jarvis March. It involves starting from one end of the point set and "folding" a line around the points, as one would wrap a gift [7]. To give more details, the algorithm starts by finding the left-most vertex. Starting out with a vertical line through this vertex, it then checks all other points, to find the point it would hit when rotating this line around the first vertex, i.e. the one with the smallest angle. This point is then added to the convex hull, and the process is repeated until it hits the starting vertex again. An example of running the gift wrapping algorithm can be seen in figure 2.

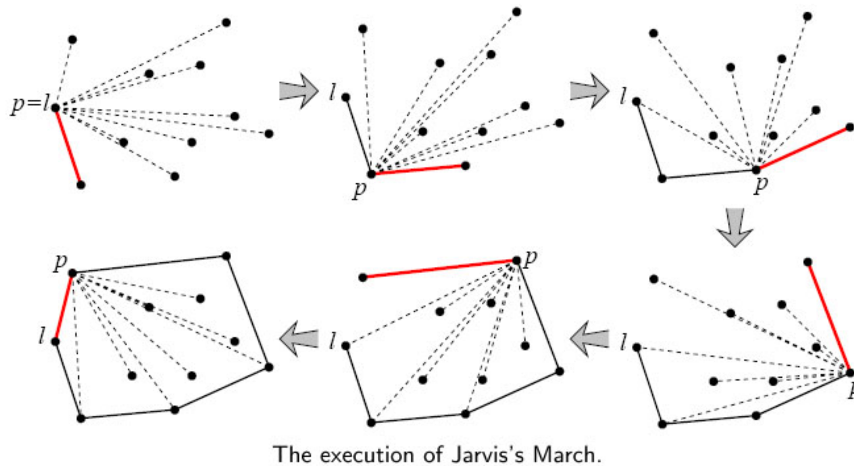

Figure 2: Gift Wrapping

Source: www.tcs.fudan.edu.cn

Each time the algorithm looks for a new edge it checks all the vertices, and it ends when hitting the initial vertex. Adding the time it takes to find the initial vertex, the running time is:

$$\mathcal{O}(n) + \sum_{i=1}^{h} \mathcal{O}(n) = \mathcal{O}(nh).$$

Where h is the size of the convex hull.

3.1.4 *Kirkpatrick-Seidel Algorithm*

Kirkpatrick and Seidel [8] presented an algorithm that reverses the idea of divide-and-conquer, into a concept they dubbed marriage-before-conquest. Normally, a divide-and-conquer algorithm would split the input, recursively perform a set of operations on each part, and merge the parts. The marriage-before-conquest algorithm splits the input, and instead of just recursing on each set, the algorithm first

Algorithm 3 GWHull(S)

```
L = ∅
min ← min(S)
L.add(min)
direction ← (0,1)
while next ≠ L.last do
  nextPoint = ∅
  smallestAngle = ∞
  for all p : points do
    angle ← angle(direction, p)
    if angle < smallestAngle then
      smallestAngle ← angle
      nextPoint = p
    end if
  end for
  L.add(nextPoint)
end while
```

figures out how the parts should be merged, and then recurses on the subsets. This algorithm only focuses on finding the upper part of the convex hull, but the lower hull can be found by flipping or rotating all points. These hulls can then be concatenated to form the complete convex hull.

The algorithm works by finding a vertical line x_m through the median x-coordinate of the input points S. It then finds the *bridge* through x_m, that is, the edge of the upper hull crossing x_m, seen in figure 3, where x_m is called L. As per the definition of a convex hull, all vertices beneath this bridge can not be a part of the upper hull, and they are pruned from the set. The remaining points are split into two sets, S_{left} and S_{right}, containing the points to the left and right of this bridge. The points in the bridge are added to the hull and the algorithm is called recursively on each subset.

3.1.4.1 *Finding the bridge*

The main idea when looking for a bridge is pruning points from the set S, until only the two points forming the bridge remain. To do this, it divides S into $\lfloor S/2 \rfloor$ pairs, and calculates the slopes of the lines intersecting the points from each pair. It then finds the median K of these slopes, approximated by picking the median of one or more slopes at random, and separates all the pairs into three groups: those with a higher slope than K, those with a lower slope, and those with the same slope. The line with slope K is then moved up, to where it hits the top points, maximizing $p_y - Kp_x$, while keeping track of the minimum and maximum point on this line. If there are more than one point on the line, and the points are on different sides of

Algorithm 4 KSHull(S)

$x_m \leftarrow median(x)$
$(i, j) \leftarrow Bridge(S, x_m)$
$S_{left} \leftarrow p_i \cup \{p \in S | x(p) < x(p_i)\}$
$S_{right} \leftarrow p_j \cup \{p \in S | x(p) > x(p_j)\}$
if $i = min(S)$ **then**
  print i
else
  $KSHull(S_{left})$
end if
if $j = max(S)$ **then**
  print j
else
  $KSHull(S_{right})$
end if

x_m, the minimum and maximum points form the bridge, and they are returned. If there is only one point on the line, it lies either to the left, on, or to the right of x_m. Based on the relation between the sets, K, and on which side of x_m the points lie, up to one point from each pair may be pruned. This runs recursively until only one point on each side of x_m remains. Pseudocode for this subroutine can be seen in algorithm 5.

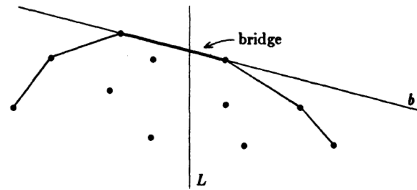

Figure 3: Bridge

Source: Kirkpatrick, Seidel [8]

This algorithm is output-sensitive, in that the running time depends on both the size of the input and the size of the computed hull.

For finding the median, randomized selection is used, as it will give a reasonable approximation of the real median. It is also assumed that the bridge finding algorithm runs on $\mathcal{O}(n)$ time. The running time of the function is determined by $f(|S|, h)$ where f must satisfy the following recurrence relation:

$$f(n, h) \leq \begin{cases} cn & \text{if } h = 2, \\ cn + \max_{h_l + h_r = h} \{f(\frac{n}{2}, h_l), f(\frac{n}{2}, h_r)\} & \text{if } h > 2, \end{cases}.$$

where c is a positive constant and $1 < h \leq n$.

Algorithm 5 *Bridge*(S, x_m)

```
candidates ← ∅
if |S| = 2 then
    return (i, j), where S = {p_i, p_j} and x(p_i) < x(p_j).
end if
pairs ← ⌊|S|/2⌋ disjoint pairs of points from S, ordered by x-coordinate.
if |S| is uneven then
    Add unpaired point to pairs.
end if
Calculate slope for all pairs.
for all pairs in pairs do
    if Any slopes are vertical then
        Add point with highest y-coordinate to candidates.
        Remove pair from pairs.
    end if
end for
K ← median of slopes
small ← pairs with slope < K.
equal ← pairs with slope = K.
large ← pairs with slope > K.
max ← the set of points on the line maximizing y(p_i) − x(p_i)K
p_k ← the point in max with minimum x-coordinate.
p_m ← the point in max with maximum x-coordinate.
if x(p_k) ≤ x_m and x(p_m) > x_m then
    (p_k, p_m) is the bridge, return (k, m).
end if
if x(p_m) ≤ x_m then
    Insert right point from all pairs in large ∪ equal into candidates
    Insert both points from all pairs in small into candidates
end if
if x(p_k) > x_m then
    Insert left point from all pairs in small ∪ equal into candidates
    Insert both points from all pairs in large into candidates
end if
return Bridge(candidates, x_m)
```

To prove the running time of the algorithm, it is initially assumed that the running time is $\mathcal{O}(n \log h)$. Then $f(n,h) = cn \log h$ must satisfy the recurrence relation. For $h = 2$ this is trivial, but for $h > 2$, substituting $f(n,h)$ with this assumed running time yields:

$$\begin{aligned} f(n,h) &\leq cn + \max_{h_l+h_r=h} \{c\tfrac{n}{2} \log h_l + c\tfrac{n}{2} \log h_r\} \\ &= cn + \tfrac{1}{2}cn \max_{h_l+h_r=h} \{\log(h_l h_r)\}\,. \end{aligned}$$

The maximum of $\log(h_l h_r)$ occurs when $h_l = h_r = n/2$, therefore, it is substituted into the equation:

$$\begin{aligned} f(n,h) &\leq cn + c\tfrac{n}{2} \log(h_l h_r)^2 = cn + cn \log \tfrac{h}{2} \\ &= cn + cn \log h - cn = cn \log h. \end{aligned}$$

Thus resulting in the wanted $\mathcal{O}(n \log h)$ bound.

3.1.5 *Kirkpatrick-Seidel Algorithm Using Linear Programming*

As later tests will prove, the normal Kirkpatrick-Seidel algorithm has a large running time constant hidden in the O-notation, partly caused by the bridge finding algorithm, resulting in it being much slower than the $\mathcal{O}(n \log n)$ incremental algorithm, in almost all cases. Another issue is that the method of comparing the slopes of pairs of points and pruning based on the slopes only works in 2D. Another technique exists for finding the bridge that can be extended to higher dimensions. This technique is based on linear programming, which a later chapter will focus on.

3.2 3D ALGORITHMS

In this section, the 3D gift wrapping algorithm is presented, along with a short description of how to use linear programming to find a convex hull in 3D.

3.2.1 *Gift Wrapping Algorithm*

The gift wrapping algorithm for computing convex hulls in 3D includes and expands on the 2D algorithm variant. Instead of folding a line around the point set, it folds a plane. Initially, the algorithm uses the 2D algorithm on the point set, projected to two dimensions, to find an initial edge. It then folds a plane around this edge, until it hits a set of points. Using these points, it forms a polygon, which is used as a base for the complete hull. The algorithm then recursively

picks an edge, uses a plane through the polygon to fold around the chosen edge until it hits a new set of points, and forms a new polygon from the edge and this new set of points.

Similar to the 2D algorithm, this algorithm finds an initial face and, for each face, tries to find a new face, based on newly added edges. Finding the initial extreme takes $\mathcal{O}(n)$ time. For each point or edge in the hull, finding new points takes $\mathcal{O}(n)$ time. Thus, the total running time of the algorithm is the same as the 2D algorithm:

$$\sum_{i=1}^{h} \mathcal{O}(n) = \mathcal{O}(nh).$$

3.2.2 *3D Linear Programming Algorithm*

Extending the bridge finding subroutine from the Kirkpatrick-Seidel algorithm to 3D would be difficult, but when using linear programming instead, this is very simple, as it only involves adding another dimension to the linear program. In 2D, the algorithm looked for the edge of the upper hull above a given vertical line x_m. In 3D, this extends to finding a face of the upper hull above a given vertical line (x_m, y_m).

3.3 LINEAR PROGRAMMING

In this section, the concept of linear programming is introduced. A description of duality explains how to use a point set as a series of constraints. An algorithm for solving linear programs is presented, its running time is analysed, and an addition to the algorithm that greatly reduces its time bound is described. Finally, an in-depth explanation of how to use this algorithm to solve linear programs in both 2D and 3D is presented.

3.3.1 *General*

Linear Programming(LP) is a method for optimizing a linear function, a maximization or minimization problem, subject to a number of linear constraints. When represented in standard form, the program has the following structure:

$$\begin{aligned} \text{Minimize} \quad & c_1x_1 + \cdots + c_dx_d \\ \text{Subject to} \quad & a_{1,1}x_1 + \cdots + a_{1,d}x_d \geq b_1 \\ & \vdots \\ & a_{n,1}x_1 + \cdots + a_{n,d}x_d \geq b_n \end{aligned}$$

Where c_i, $a_{i,j}$, and b_i are real numbers, which form the input to the problem. The function to be maximized or minimized is the *objective*

function, and is together with the constraints called a *linear program*. The number of variables d is called the dimension of the linear program. Each constraint can be interpreted as a halfspace dividing the d-dimensional space in two, and the intersection of all the half-spaces is called the feasible region, which is the set of all points satisfying all the constraints. Points inside the feasible region are called feasible points, and points outside are called infeasible. If the feasible region is empty, the linear program itself is infeasible.

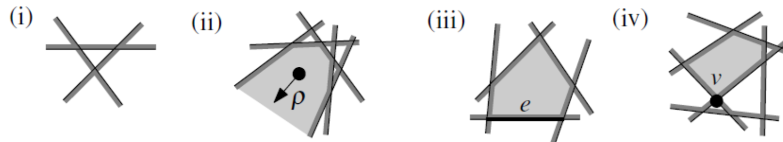

Figure 4: Linear Program Solutions

Source: Berg et al.[2]

When solving a linear program, the solution is not always unique. The constraints can cause any possible solutions to be invalid, rendering the whole linear program infeasible. The solution could also be completely unbounded in the direction of the objective function. Even when bounded and valid, the solution v may still not be unique, as it could be a line instead of a points. These possible solutions are illustrated in figure 4. For the rest of this thesis, it is assumed that if a solution exists, it is always a point.

3.3.2 *Duality*

In 2D, both a point p and a line ℓ are represented by two variables, $p := (p_x, p_y)$ and $\ell := (y = ax + b)$. Therefore, a set of points can be mapped to a set of lines, and a set of lines can be mapped to a set of points. A mapping like this is called a *duality transform*, and in one such mapping, a point $p := (p_x, p_y)$ is mapped to a line $p^* := (y = p_x x - p_y)$, and a line $\ell := (y = ax + b)$ is mapped to a point $\ell^* := (a, -b)$. This mapping retains the following properties: $p \in \ell$ if and only if $\ell^* \in p^*$, and p is above ℓ if and only if ℓ^* lies above p^*[4]. This is illustrated in figures 5.

As vertical lines can not be represented in the form of $y = ax + b$, they must either be handled separately as special cases, or the entire scene must be rotated so no vertical lines remain.

3.3.3 *Incremental Algorithm*

As the point set used to compute a convex hull is not a series of constraints, duality can be used to interpret them as constraints. This

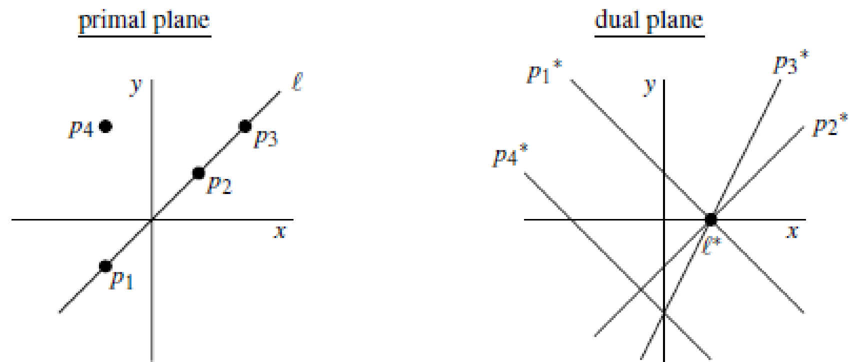

Figure 5: Duality

Source: Berg et al.[2]

changes the problem into the solving of the following LP:

$$
\begin{aligned}
\text{find} \quad & a, b \\
\text{min.} \quad & ax_m + b \\
\text{s.t.} \quad & ax_1 \ + b \geq y_1 \\
& \qquad \vdots \\
& ax_i \ + b \geq y_i \\
& \qquad \vdots \\
& ax_n \ + b \geq y_n
\end{aligned}
$$

As the linear program contains a large number of input points, equating to constraints, and only a few dimensions, the algorithms described by Timothy M. Chan [3] and Berg et al. [2], first described by Megiddo [10], can be used. Very simplified pseudocode for this approach can be seen in algorithm 6.

Algorithm 6 Linear Program

```
if y_{i+1} ≤ ax_{i+1} + b then
    go to step i + 1
else
    find a, b
    min. ax + b
    s.t. ax_i + b ≥ y_i s
end if
```

Algorithm 7 keeps track of the current optimal vertex v_i for each step i. In each step $i+1$, it uses constraint c_{i+1} from the set of constraints $C := \{c_1, \dots, c_n\}$, and checks whether v_i is still valid, that is, if $v_i \in \{c_1, \dots, c_{i+1}\}$. If this is true, then $v_{i+1} \leftarrow v_i$, but if not, the new optimal vertex must lie somewhere on the constraint c_{i+1}. This point can be found by reducing the problem to solving a 1D linear program

on c_{i+1} using the same constraints $c_1, \ldots, c_i$:

$$\begin{array}{ll} \text{find} & a \\ \text{min.} & ax_m \\ \text{s.t.} & ax_1 \geq y_1 \\ & \vdots \\ & ax_i \geq y_i \end{array}$$

Algorithm 7 IncrementalLP(H, $\vec{c}$)

for $i \leftarrow 1$ **to** n **do**
  if $v_{i-1} \in h_i$ **then**
    $v_i \leftarrow v_{i-1}$
  else
    $v_i \leftarrow$ the point p on l_i that maximizes $f_c(p)$, subject to the constraints in H_{i-1}.
    if p does not exist **then**
      Report that the linear program is infeasible and quit.
    end if
  end if
end for
return v_n

The running time of this algorithm is dominated by the $\mathcal{O}(i)$ time of solving the 1D linear program, and as a result, the algorithm has the following time bound:

$$\sum_{i=1}^{n} \mathcal{O}(i) = \mathcal{O}(n^2).$$

As this bound is worse than many other techniques, other methods must be considered.

3.3.4 *Randomized Algorithm*

When adding the constraints one by one, there is a probability that the added constraint will not change the previous optimal solution. In this case, a new solution does not need to be found. As such, a new linear program need only be solved when a newly added constraint fails, rendering the previous solution invalid. Because of this, the algorithm is dependent on the order of the constraints, as there is both a possibility of finding the optimal solution first, and only needing to solve a single LP, but also the possibility of finding the worst possible solution each time, and needing to solve n linear programs. In situations where a lot of similar LPs have to be solved under the same constraints, a bad order would give the algorithm a very poor running time for each LP. Using a randomized algorithm, the probability that the constraints are in a "bad" order, that is, an order where

many constraints fail and forces the algorithm to find a new solution, is decreased. Algorithm 8 is very simple, as it simply shuffles the list of constraints and calls algorithm 7.

Algorithm 8 RandomizedLP(H, $\vec{c}$)

```
Shuffle(H)
IncrementalLP(H, c⃗)
```

3.3.5 *Solving Linear Programs in 2D and 3D*

In all dimensions, the solution to an LP is a point, with each constraint being a hyperplane. Adding the constraints one by one, every time a constraint fails, the new solution is a point on the most recently added hyperplane. Because of this, the problem can be reduced to linear program of dimension $d-1$ instead. In two dimensions, the linear program has the form:

$$\begin{array}{ll} \text{find} & a, b \\ \text{min.} & ax_m + b \\ \text{s.t.} & ax_1 + b \geq y_1 \\ & \vdots \\ & ax_i + b \geq y_i \\ & \vdots \\ & ax_n + b \geq y_n \end{array}$$

When one of the constraints fail, the new solution must lie somewhere on the newly added constraint line. Finding the point on this line requires solving a 1D linear program. Assuming that the constraints c_i is violated at step i, the constraint is an equality instead of a halfspace:

$$ax_i + b = y_i.$$

To find the variables a and b, b is first isolated:

$$b = y_i - ax_i.$$

Substituted into the objective function:

$$ax_m + y_i - ax_i.$$

And arranged into a more fitting form:

$$a(x_m - x_i) + y_i.$$

Also inserted into the constraints $c_1, \ldots, c_j$

$$\begin{aligned} ax_j + y_i - ax_i &\geq y_j \\ a(x_j - x_i) &\geq y_j - y_i. \end{aligned}$$

Which can then be used to solve a 1D linear program:

$$\begin{array}{ll} \text{find} & a \\ \text{min.} & a(x_m - x_i) \\ \text{s.t.} & a(x_1 - x_i) \quad \geq y_1 \\ & \qquad \vdots \\ & a(x_j - x_i) \quad \geq y_j \\ & \qquad \vdots \\ & a(x_{i-1} - x_i) \geq y_{i-1} \end{array}$$

This method can be extended to higher dimensions. In 3D, the linear program to solve is the following:

$$\begin{array}{llll} \text{find} & a, b, c & & \\ \text{min.} & ax_m & + by_m + c & \\ \text{s.t.} & ax_1 & + by_1 & + c \geq z_1 \\ & & \vdots & \\ & ax_n & + by_n & + c \geq z_n \end{array}$$

Like in 2D, when adding a constraint c_i that invalidates the current solution (a, b, c), the new optimal point lies somewhere on c_i, which in 3D is a plane instead of a line. To find this, it is necessary to use c_i, solve for c, and substitute c into all of the previous constraints. Because the new optimum is on the plane, the inequality of the half-space can be set to an equality instead:

$$ax_i + by_i + c = z_i.$$

Solve for c:

$$c = -ax_i - by_i + z_i.$$

Substitute back into the objective function:

$$ax_m + by_m - ax_i - by_i + z_i.$$

Arrange it into the correct form, but ignore the z_i term:

$$a(x_m - x_i) + b(y_m - y_i).$$

Do the same for the constraints $c_1, \ldots, c_j, \ldots, c_{i-1}$:

$$\begin{aligned} ax_j + by_j - ax_i - by_i + z_i &\geq z_j \\ a(x_j - x_i) + b(y_j - y_i) &\geq (z_j - z_i). \end{aligned}$$

As the objective function and the constraints can be rearranged into this form, the 2D linear program can be solved as any other:

$$\begin{array}{lllll} \text{find} & a, b & & & \\ \text{min.} & a(x_m - x_i) & + b(y_m - y_i) & & \\ \text{s.t.} & a(x_1 - x_i) & + b(y_1 - y_i) & \geq (z_1 - z_i) \\ & & \vdots & & \\ & a(x_j - x_i) & + b(y_j - y_i) & \geq (z_j - z_i) \\ & & \vdots & & \\ & a(x_{i-1} - x_i) & + b(y_{i-1} - y_i) & \geq (z_{i-1} - z_i) \end{array}$$

Similar to 2D, where the LP involved finding the optimal point along a line, this LP also involves finding the optimal along a line, but a line in 3D space instead of the 2D plane. Like in 3D, where the LP could be reduced to a 2D LP, this 2D LP can be reduced to a 1D LP. To simplify notation a bit, and to show a more general situation, $x_m = x_m - x_i$, $x_i = x_j - x_i$ will be used, and similarly for y and z, in the following LP:

$$ax_i + by_i \geq z_i.$$

Isolate b:

$$b = \frac{z_i - ax_i}{y_i}.$$

Substitute this into the objective function and the constraints $c_1, \ldots, c_{i-1}$:

$$\begin{gathered} ax_m + \frac{z_i - ax_i}{y_i} y_m \\ a(x_m - \frac{x_i y_m}{y_i}) + \frac{y_m z_i}{y_i}. \end{gathered}$$

Disregarding the products not including a:

$$a(x_m - \frac{x_i y_m}{y_i}).$$

Also inserting b into the constraints:

$$\begin{aligned} ax_j + \frac{z_i - ax_i}{y_i} y_j &\geq z_j \\ a(x_j - \frac{x_i y_j}{y_i}) &\geq z_j - \frac{y_j z_i}{y_i}. \end{aligned}$$

Resulting in the following linear program:

$$
\begin{array}{lll}
\text{find} & a & \\
\text{min.} & a(x_m - \frac{x_i y_m}{y_i}) & \\
\text{s.t.} & a(x_1 - \frac{x_i y_1}{y_i}) & \geq z_1 - \frac{y_1 z_i}{y_i} \\
& \dots & \\
& a(x_j - \frac{x_i y_j}{y_i}) & \geq z_j - \frac{y_j z_i}{y_i} \\
& \dots & \\
& a(x_{i-1} - \frac{x_i y_{i-1}}{y_i}) & \geq z_{i-1} - \frac{y_{i-1} z_i}{y_i}
\end{array}
$$

Solving this is trivial, and the result can be inserted into the 2D LP, where the result from that can be inserted into the 3D LP, yielding the correct result.

3.3.5.1 *Running Time Analysis*

By randomizing the LP algorithm its running time becomes significantly better as shown here. When adding a constraint that does not change v_i, it takes $\mathcal{O}(1)$ time. Therefore, only the constraints that do change v_i matter, as they each take $\mathcal{O}(i)$ time.

Define X_i to be a random variable, with value 1 if $v_{i-1} \notin h_i$, and 0 if $v_{i-1} \in h_i$. The time spent adding the constraints that change v_i amounts to:

$$\sum_{i=1}^{n} \mathcal{O}(i) \cdot X_i.$$

According to *linearity of expectation*, the sum of random variables is the sum of the expected values of the variables. Therefore, the total time to solve all the 1D linear programs is:

$$E\left[\sum_{i=1}^{n} \mathcal{O}(i) \cdot X_i\right] = \sum_{i=1}^{n} \mathcal{O}(i) \cdot E\left[X_i\right].$$

To find $E[X_i]$, *backwards analysis* is used: assuming that the algorithm has already finished, and analysing backwards from that point. The optimal vertex v_n must lie on at least one constraint from C. In step $n-1$, for the optimal to have changed, at least one of the constraints on which v_n lies, must be removed. If v_n lies on more than two constraints, removing one constraint may not be enough to allow v_n to change. Therefore, the chance of removing a constraint that changes the optimum v_i is at most $2/n$. This means, that for each step i, the probability of adding a constraint that changes the optimum is $2/i$ and therefore $E[X_i] \leq 2/i$. Thus, the total expected time bound on solving all the 1D linear programs is:

$$\sum_{i=1}^{n} \mathcal{O}(i) \cdot \frac{2}{i} = \mathcal{O}(n).$$

The same bound holds for the algorithm in 3D with a similar proof. The only difference is that the solution is bounded by at least three constraints instead.

In situations where the input is in a "bad" order, the order will persist in each $d-1$-dimensional linear program when using the incremental method, but when shuffling the input for each new linear program, the same "bad" order is unlikely to occur again.

3.4 multiple linear programming

When using linear programming to find the bridges in the Kirkpatrick-Seidel algorithm, it becomes evident that for all of the linear programs, the same constraints are used. This chapter describes an algorithm for solving multiple linear programs on the same constraints.

3.4.1 *General*

The algorithm for solving multiple linear programs works by recursively dividing the point set into subsets, solving linear programs on one point from each subset, and removing subsets lying underneath these bounds. When a sufficient amount of subsets have been removed, normal linear programming is used to find the optimal points. This algorithm achieves a running time of $\mathcal{O}(n \log k)$, where k is the number of objective functions, instead of the $\mathcal{O}(nk)$ running time of solving the linear programs sequentially.

3.4.2 *Partitioning*

The point set is recursively partitioned into four subsets based on a vertical line, and a single non-vertical line cutting each of these subsets in two as follows: Divide S into two subsets P_l and P_r of size $\frac{|S|}{2}$ by a vertical line x_m. Find a single non-vertical line $\ell := (y = ax + b)$ dividing both P_l and P_r into two equal sized subsets. Recurse on the dataset until the size of the subset is less than $\frac{n}{r}$ where r is an appropriately chosen constant. The result will be r polygons covering the plane. Pseudocode for this algorithm can be seen in algorithm 9 and algorithm 10.

To find x_m, it is possible to use a median finding algorithm with an $\mathcal{O}(n)$ running time[1]. To find the non-vertical cutting line, an $\mathcal{O}(n)$ time algorithm exists for solving the ham sandwich theorem, which is applicable in this situation[9]. Approximating the median can also be done in constant time by picking a few random points of the set and finding the median of these. Likewise, solving the ham sandwich theorem can also be done by picking an uneven number of points

1 en.cppreference.com/w/cpp/algorithm/nth_element

Algorithm 9 *partition*(P, r)

```
B, E
size = 0, crossingNumber = 0
B.push(P)
while |B| > 0 do
  S ← B.pop()
  if |S| < n/r then
    E.push(S)
  else
    B.push(quarter(S))
    size+ = 4
    crossingNumber+ = 3
  end if
end while
return E, size and crossingNumber
```

Algorithm 10 *quarter*(P)

Divide P into two subsets P_ℓ and P_r of size $|S|/2$ around a vertical line x_m
Divide P_ℓ and P_r into two subsets of size $|S|/4$ around a single non-vertical line ℓ.
return The four subsets

from each set, and finding the line through two points, dividing the samples into two equal sized sets. Formal proofs of the validity of these methods are beyond the scope of this thesis, but in the next section it is assumed that this is equivalent to using a deterministic approach. This partitioning algorithm takes $\mathcal{O}(n \log r)$ time.

3.4.3 *Algorithm*

The algorithm works by subdividing the input point set into r subsets. It then takes a single point from each of these subsets, and calls the algorithm recursively with this subset of points as a parameter. The result of this recursive call is k lines. These lines are tested against the subsets, and if a subset is not intersected by any lines, the points contained in that subset are pruned from the point set. The algorithm is then recursively called with the point set. When the algorithm has run this step until a set amount of points are pruned, the linear programs are solved sequentially on the remaining points.

3.4.4 *Running Time Analysis*

In this algorithm there are two paths to take: One involves simply solving the linear programs one by one. Using the previously de-

scribed randomized algorithm, this can be done in $O(nk)$ time. The other path is more complicated: First, partitioning the set P into r subsets $P_1, \ldots, P_r$ will take $O(n \log r)$ time. Finding the lines $\ell_1, \ldots, \ell_k$ is a recursive call. Define the *crossing number* Cr to be the maximum number of partitions intersected by a single line. When pruning partitions, this is the number of partitions that will not be pruned by a given line. The crossing number follows the following recursion in the partitioning algorithm:

$$f(n) = 3f\left(\frac{n}{4}\right) + \mathcal{O}(n).$$

Pruning partitions involves checking all r partitions against all k lines. These partitions each have size $\mathcal{O}(\log r)$, as they are created by a recursive algorithm that cuts each partition in four pieces with a vertical and a non-vertical line. Each such recursion can add only a constant number of vertices to the boundary, so their worst-case size will be $\mathcal{O}(\log r)$. This results in a running time of $\mathcal{O}(rk \log r)$ for the pruning step. Any partition not intersecting any of the lines $\ell_1, \ldots, \ell_k$ can be pruned, as this means that the partition is entirely below all the lines $\ell_1, \ldots, \ell_k$, and none of its vertices can be part of any bridge, for any of the k queries. This leaves kCr partitions, each with $\frac{n}{r}$ points. These remaining points are used for a recursive call. These elements result in the following two running times, depending on the path chosen:

$$f(n) = \mathcal{O}(nk)$$
$$f(n) = \mathcal{O}(n \log r) + f(r) + O(rk \log r) + f(\tfrac{n}{r}kCr).$$

Depending on the relationship of some of these variables, either of the paths may be the fastest. For each recursion step, n gets smaller, but the other variables are constant. To maximize running speed, it is preferable to hit the recursive part of the algorithm many times, bringing n down as much as possible, before sequentially solving the LPs. Using the pruning path also assumes that some points get pruned. If no points are pruned, or if only a few points are pruned, this path will be a waste of time.

3.4.5 *Termination Condition*

It needs to be decided when to end the recursion and solve the linear programs sequentially. Assuming that the set is always recursively divided into four equal sized sets, the crossing number is:

$$Cr = cr^{\log_4 3}.$$

Where c is a constant. Inserting this into the recursive path of the recurrence relation results in:

$$f\left(\frac{n}{r}kcr^{\log_4 3}\right) = f\left(\frac{n}{r^{1-\log_4 3}}ck\right) = f\left(\frac{n}{r^{0.2075}}ck\right)$$

Which will be bounded by the following approximation:

$$\mathcal{O}\left(\frac{n}{r^{1/5}}ck\right)$$

$$f(n) = \mathcal{O}(n\log r) + f(r) + \mathcal{O}(rk\log r) + \mathcal{O}\left(\frac{n}{r^{1/5}}ck\right).$$

Isolate r. Assuming that $\frac{n}{3}$ points are pruned at each level of recursion will give a simple and reasonable bound. This can be ensured by picking r appropriately:

$$\frac{n}{r^{1/5}}ck \leq \frac{n}{3}$$

$$(3ck)^5 \leq r.$$

The running time of the recursive path can be simplified if $r = (3ck)^5$ is chosen. This makes an analysis easier:

$$\mathcal{O}(rk\log r) \Rightarrow \mathcal{O}(k^6\log k).$$

It is assumed that $k \geq 3(3c)^5$. If $n \geq k^6$ then $r \leq \frac{n}{3}$. This means that to get the desired amount of pruning, there must be at least $n \geq k^6$ input points.

Under these assumptions, the running time can be simplified:

$$f(n) = \mathcal{O}(n\log r) + f(r) + \mathcal{O}(k^6\log k) + \mathcal{O}\left(\frac{n}{r^{1/5}}k\right)$$

$$\Downarrow$$

$$f(n) = \mathcal{O}(n\log r) + 2f\left(\frac{n}{3}\right).$$

As there must be at least $n \geq k^6$ input points to get the desired amount of pruning, this path can only be taken if this condition holds.

Algorithm 11 *multiLP*(P, k)

```
if n ≥ k^6 then
  Solve LPs sequentially
else
  P_1, ..., P_r ← r-partition(P)
  P' ← one point from each subset/partition
  Lines ℓ_1, ..., ℓ_k ← multiLP(P', k)
  Prune any partition that does not cross any of the lines
  Solve multiLP(P, k)
end if
```

When moving from step i to step $i+1$, the algorithm must evaluate if the cost of another level of recursion is less than the cost of solving the linear programs sequentially. In figure 6 the recursion tree for the algorithm is shown, along with the running time costs of each level. In this tree, each node is a recursion call and each leaf is solving the LPs sequentially. Pseudocode for this algorithm can be seen in algorithm 11.

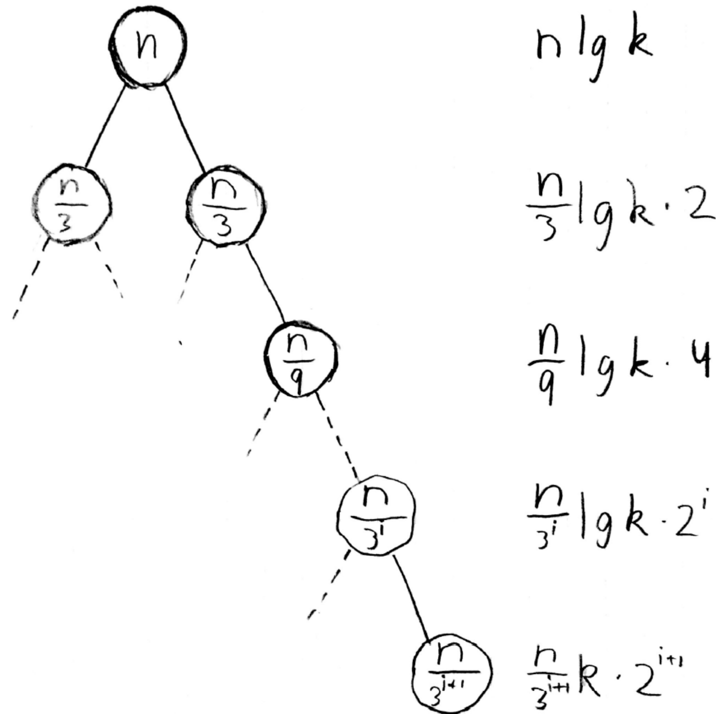

Figure 6: Recursion Tree

Consider an example: Assuming $n = k^9$, the cost of each node on level i is:

$$\frac{k^9}{3^i} = k^6.$$

Then the cost of level $i+1$, if it was chosen to be a leaf, is:

$$i = \log_3 k^3$$

$$\frac{n}{3^{i+1}} 2^{i+1} k \approx \frac{2}{3} n k^{-0.11}.$$

Assuming $n = k^8$ instead, to show how this size affects the node cost:

$$i = \log_3 k^2$$

$$\frac{n}{3^{i+1}} 2^{i+1} k \approx \frac{2}{3} n k^{0.26}.$$

As seen in these examples, when the input size relative to the size of k gets larger, the more of a gain can be achieved by performing a pruning step. This means that a larger part of the input gets pruned, which, in the end, decreases the cost of solving the k linear programs on the final set. In the example for $n = k^9$, the cost of the leaf step is less than the $\mathcal{O}(n \log k)$ price of the pruning steps. This means, that when $n \geq k^9$, with this algorithm, k linear programs can be solved simultaneously on the same n constraints in time $\mathcal{O}(n \log k)$.

3.5 PRUNING

A technique for improving the running time of any convex hull algorithm is *pruning*, that is, removing points from the point set that can never be part of the hull. Define p and q as the vertices with the smallest x-coordinate and the largest x-coordinate, respectively. Assuming the convex hull algorithm is searching for the upper hull, if two or more vertices share the highest or lowest x-value, the point with the highest y-value is chosen. Drawing a line ℓ through two points p and q, a point beneath this line can never be part of the upper hull. As the hull must start at p and end at q, by the definition of a convex hull, no points below this line can be part of the hull. If a point v beneath this line were to be considered as part of the hull and a line l through v and p, then the point q would lie above the line, thus rendering the solution invalid. The same goes for a line going through v and q. This technique can also work in a divide-and-conquer algorithm, where additional points would be pruned with each recursion.

4

IMPLEMENTATION

In this chapter, the implementation of the different algorithms are described. A general explanation of design choices, as well as some of the problems encountered along the way, are described first, followed by similar discussions for the individual convex hull algorthms and the linear programming algorithms.

4.1 GENERAL

All algorithms and tests were implemented in C++ 11 using Microsoft Visual Studio and JetBrains CLion EAP. C++ was the programming language of choice, as it is run directly on the CPU, instead of being run through a virtual machine, making it more reliable for time measuring. The code was designed with an object-oriented approach, to make it more transparent and easier to manage. All convex hull algorithms implemented the same interface, and similarly for all the point distribution generators. This made it easier to test a list of algorithms against a list of distributions.

Both points and lines were represented by a `Vector3D` class, with points being represented as their coordinates (x, y, z), and lines being represented as (a, b, c). Each variable was defined as a double precision floating points number, amounting to a total size of 24 bytes.

An `LPSolver` class was created to handle the solving of linear programs in d dimensions. Internally, it uses a `Halfspace` struct, which is similar to the `Vector3D` class, but with an added `Bound` enum, expressing which side the halfspace cuts.

In the optimization of K in the Kirkpatrick-Seidel algorithm, it had to be tested whether a point was on the line, in rare circumstances a problem occured. As computers work with finite precision arithmetic, small errors could cause a point on the line not to intersect the line. In this situation, the check was made with a small measure of error instead. As finite precision only caused a problem under rare circumstances in the bridge finding subroutine of the standard Kirkpatrick-Seidel algorithm, and as this would be replaced with linear programming, this was deemed an acceptable approximation.

4.2 CONVEX HULL ALGORITHMS

4.2.1 *Naive Algorithm*

The naive algorithm was simple to implement, as it was possible to strictly follow the pseudocode. A double for-loop was used for all possible pairs of points, where $p \neq q$, with an additional for-loop to test if all other points were on, or to the right, of the line defined by the two points. Finally, the pairs of points were sorted clockwise, and the points put in the `vector` to be returned.

4.2.2 *Incremental Algorithm*

The incremental algorithm was even simpler to implement, as it simply involved sorting the points, iterating through the points, and, in some cases, looping back to remove points. Sorting was done using the C++ standard `sort()` which has a guaranteed running time of $\mathcal{O}(n \log n)$[1]. No significant issues arose in implementing or testing this algorithm.

4.2.3 *Gift Wrapping Algorithms*

The 2D gift wrapping algorithm starts out by finding the point with the lowest x, and adds it to the hull list, represented by a `vector<Vector3D>`. Starting out with the direction straight east $(0, 1)$, it searches all the points to find the point which, together with the previous point, forms a line with the smallest angle from the current direction. This point is added to the hull list, and the previous step is repeated until the initial point is hit again.

The 3D gift wrapping algorithm is more complex, given the added layer of complexity of an additional dimension. It starts out by using the 2D algorithm to find the first edge on a projection of the data points simply by disregarding the z-coordinate. It then tests all planes spanned by the the first edge and the remaining points, and findes the one with the lowest angle from the starting position. This was very complicated to implement, as it was necessary to create a local coordinate space when rotating the plane, to make it easier to test the angle. This was created by using the reverse of the unit vector from the first point to the second points. Then, projecting all the points to a plane perpendicular to the vector. This reduced the angle finding to the same problem in 2D. As far as special cases, more attention had to be given paid to situations where the rotated plane hit several points, as this added a new edge and face for each extra point.

1 en.cppreference.com/w/cpp/algorithm/sort

4.2.4 *Kirkpatrick-Seidel Algorithms*

The Kirkpatrick-Seidel algorithm was by far the most time-consuming algorithm to implement. Not because of the complexity of the implementation, as the pseudocode could be followed very closely, but simply because of the sheer amount of code needed. One of the only deviations from the pseudocode in the article, was the return value of a `vector` containing the indices of the hull points in the input vector. This helped ensuring correctness both by manually verifying the hull for smaller inputs, but also by comparing the result to the results from the other algorithms.

In the first iteration of the algorithm no pointers were used, which caused massive memory usage. Because of the recursive nature of the algorithm, and because C++ by default copies all parameters in function calls, the entire point set would be copied each time a new level of recursion would be added. This was eventually fixed by having a single vector with the points, and all access to it was restricted to references and indices.

4.3 LINEAR PROGRAMMING

The linear programming solver was fairly complicated to write, but eventually worked.

Because of time constraints, the 3D linear program solver remains unfinished. Many problems were encountered when trying to implement this. E.g. in situations where the solutions were unbounded, the temporary maximum was a point an infinite distance along a constraint hyperplane. Assigning the values of ∞ or $-\infty$ to the (x, y, z) coordinates would not retain useful info about the constraint, so additional information had to be saved about the unbounded constraint hyperplane.

4.4 TESTS

To make it easier to run all the algorithms on a number of different point set distributions, polymorphism was used to have all the convex hull algorithms implement a single convex hull interface. This made it easier to have a list of function pointers to the point distribution generation functions, and test it against a list of convex hull algorithms.

The input distributions were all generated using one of the new pseudo-random number generators from the C++11 STL: the 64-bit Mersenne Twister, with both a uniform real number distribution, and a normal distribution.

4.5 VISUALISER

For easier debugging, a very simple program was written in OpenGL to visualize the large amount of points, edges and polygons, when working with convex hulls. It only has the very basic functionalities of rendering a single point set, a single line loop, and a single set of faces. A screenshot of this application is shown in figure 7.

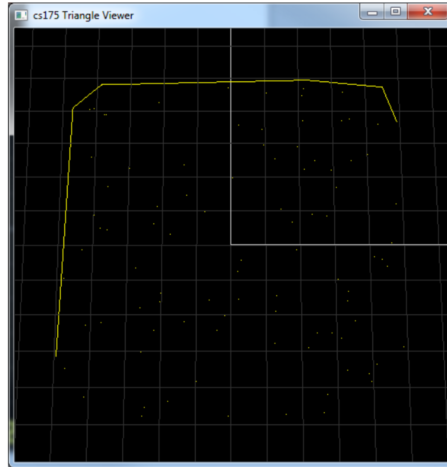

Figure 7: The OpenGL Visualiser

5

ANALYSIS

In this chapter, the tests conducted, and the results obtained from these tests are presented and discussed. All of the implemented algorithms have been run with a series of different point distributions as input, designed to showcase specific strengths and weaknesses of the different algorithms. The test data is presented in two ways: in graphs comparing all the distributions for one algorithm, and in graphs comparing all algorithms for a specific input distribution.

All tests were conducted on a Windows PC with an i5 3570K CPU and 24GB RAM. The C++ implementation was tested using a timer based on the clock from the `<chrono>` header of the C++11 STL. All the tests are the averages of 10.000 iterations of the tests.

5.1 2D DISTRIBUTIONS

In this section the different point distributions will be described, and the test data will be analysed, with accompanying graphs illustrating the results.

The previously described naive algorithm has been omitted from these tests, both because the algorithm performed so poorly that its results made the graphs much harder to read, and because the running time increased very rapidly with even a fairly small input size on the test setup.

5.1.1 *Uniform Square distribution*

A uniform square distribution test is chosen to be a baseline test. As the points in this distribution are distributed uniformly across a square plane, the hull won't be too large, as the majority of the points will be inside the hull. Pruning will also be reasonably effective, as it will probably remove about half of the points on average.

Figure 8 shows that the Kirkpatrick-Seidel algorithm performs as slow as the gift wrapping algorithm, even though its theoretical bound is better. This is most likely caused by a very large constant factor. All the algorithms seem to run in very close to $\mathcal{O}(n)$ time in this situation. Up to around an input of size $n = 2^{10}$ the very simple incremental algorithm is fastest, as the Kirkpatrick-Seidel algorithm has a large

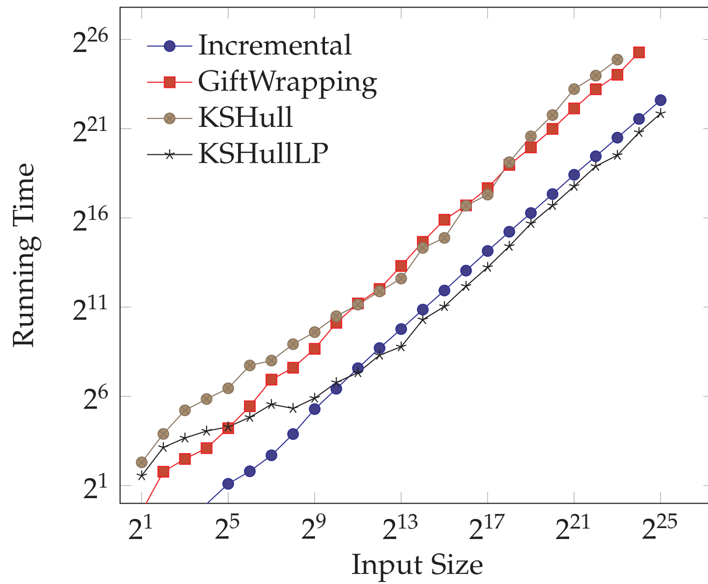

Figure 8: Uniform Square Distribution 2D

overhead. After this, the Kirkpatrick-Seidel algorithm using linear programming takes over, because of its very fast bridge finding algorithm. Comparing the two Kirkpatrick-Seidel algorithms shows very clearly, that using linear programming instead of the standard bridge finding algorithm, gives us a massive reduction in running time.

5.1.2 *Square Uniform Bounding triangle*

This bounding box test is similar to the square distribution test, except for the addition of a three-vertex bounding triangle encasing all the other points. This will help demonstrate the effectiveness of the output-sensitive algorithms, such as the gift wrapping algorithm and the Kirkpatrick-Seidel algorithm, as the hull is always of a constant size $h = 3$.

Figure 9 shows how some of the output-sensitive algorithms overtake the incremental algorithm as the fastest, when the input size gets larger. The only algorithm remaining on top of the incremental is the Kirkpatrick-Seidel standard algorithm, whose slow bridge finding subroutine holds it back. The algorithm using linear programming starts being the fastest at around $n = 2^9$, and the gift wrapping algorithm overtakes the incremental algorithm at around $n = 2^{21}$.

5.1.3 *Uniform Circle distribution*

A uniform distribution inside of a circle, almost similar to the square distribution is also used. This should ensure that the hull contains more points than the square distribution, but still contain most of

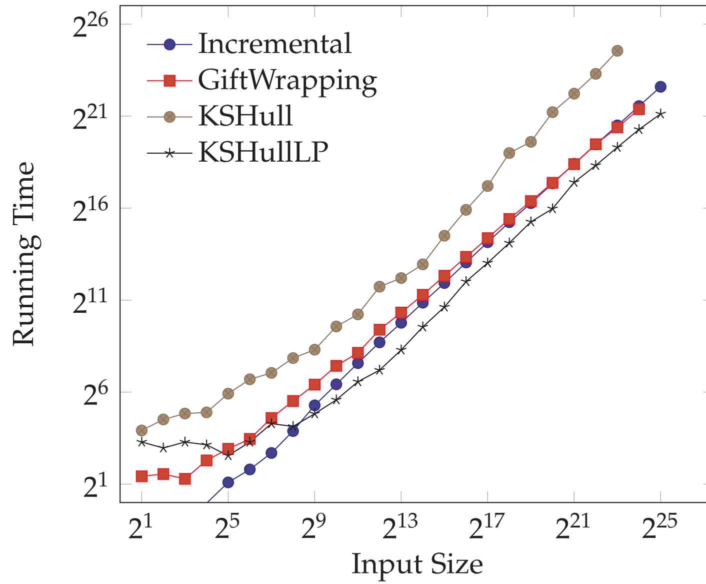

Figure 9: Uniform Distribution 2D w/ Minimal Hull

the points on the inside. The pruning step will have an easier time using a line through the middle of the point set, as the min and max point will be further towards the center of the y-axis, compared to the square distribution.

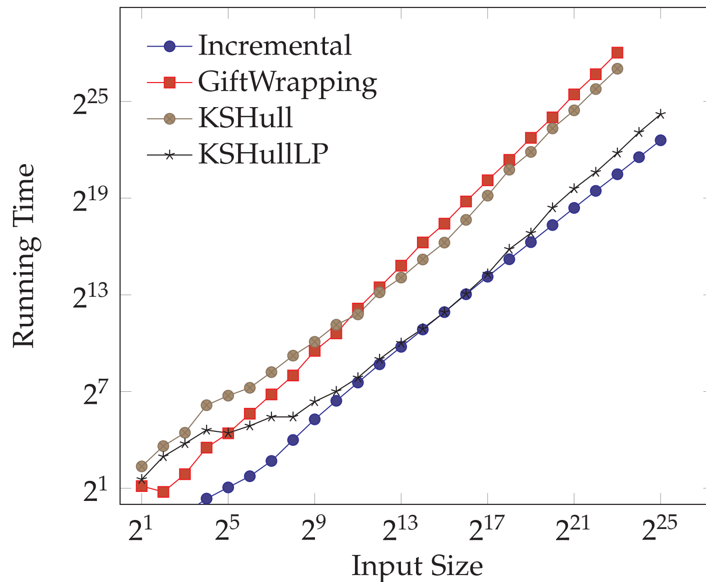

Figure 10: Circle Distribution 2D

In figure 10, the incremental algorithm has overtaken the Kirkpatrick-Seidel algorithm a bit. This is caused by the slight increase in the number of points in the hull, as the incremental algorithm is not affected by it, but the Kirkpatrick-Seidel algorithm is.

5.1.4 *Donut*

The donut distribution puts more points in the proximity of the hull, but not on the hull. As with the circle distribution, the angle is uniformly distributed, but the radius is generated with a normal distribution with mean 3 and standard deviation $\frac{1}{5}$. This would make the work of pruning algorithms sensitive to points' distance to the hull less effective. None of the methods presented in this thesis have this property. Therefore, this test will act as an additional test for input-sensitivity and general performance.

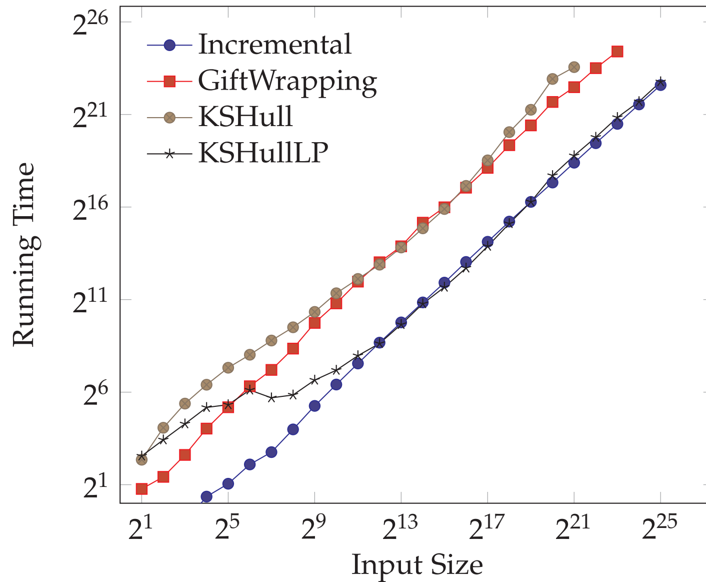

Figure 11: Donut Distribution 2D

In figure 11, the incremental algorithm and the Kirkpatrick-Seidel algorithm achieve quite similar performance in the upper part of the scale, indicating that this hull size could be the dividing line between which algorithm is the fastest.

5.1.5 *Polynomial Parabola Distribution*

In this polynomial distribution, the random points are distributed along the function $y = x^2$. This parabola shape ensures that every single point is part of the convex hull, thus putting the output-sensitive algorithms under the worst conditions. By forcing the size of the hull to be equal to the input size $h = n$, the output-sensitive algorithms will have the worst worst-case running times, e.g. the $\mathcal{O}(nh)$ gift wrapping algorithm will be $\mathcal{O}(n^2)$ instead, and this algorithm is expected to be one of the worst performing algorithms under these circumstances. Depending on whether the algorithm is looking for the upper or lower hull, pruning in this case will either remove no

points at all from the input, or be reducing n to 2, making the subsequent algorithm find the hull in constant time.

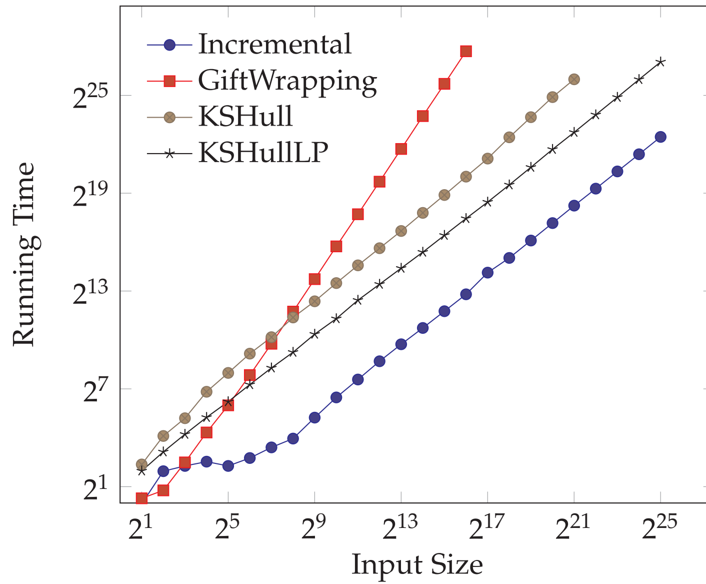

Figure 12: Polynomial $y = x^2$ Distribution 2D

Not surprisingly, figure 12 shows that the output-sensitive algorithms lose out completely to the incremental algorithm, with the gift wrapping algorithm reaching a catastrophic running time.

5.1.6 *Hull Sizes*

In figure 13 the tested average hull sizes of the 2D point distributions is shown. These are tested to give a more clear idea of the correlation between hull size and running time, in analysing the output-sensitive convex hull algorithms. The results of these tests are not surprising and corresponds to the previous predictions. An interesting point is that from input size 2^{19} and up, the uniform distribution and the donut distribution have roughly the same hull sizes. But in the tests for these two distributions, the Kirkpatrick-Seidel algorithm was only fastest with the uniform distribution. This could both because of few points being pruned for each recursion step, but also caused by the number of points near the hull, equating to a larger number of close constraints for the linear program.

5.1.7 *Section Summary*

A general trend in many of these tests is that the incremental algorithm, because of its simplicity and low overhead, and in spite of its running time not being output-sensitive, it is still one of the best in most situations. The Kirkpatrick-Seidel algorithm is fastest until the

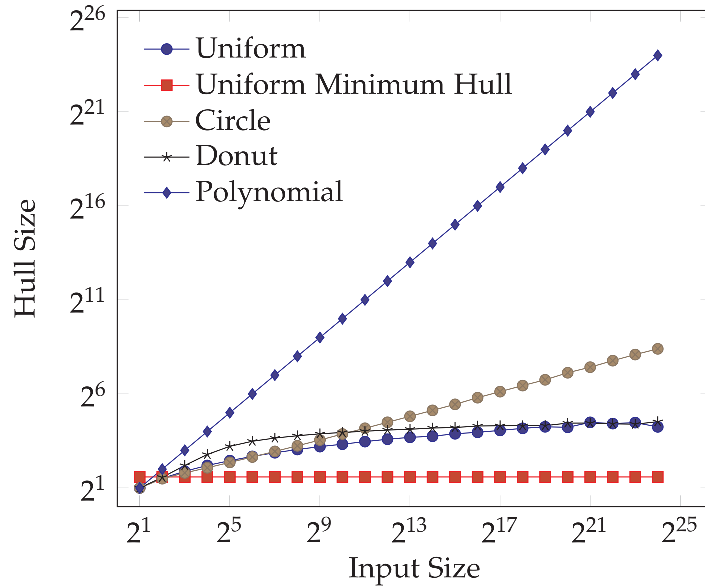

Figure 13: Average 2D Hull Sizes

hull size, compared to the input size, reaches a certain point, where the minimal overhead of the incremental algorithm gives it an advantage.

5.2 2D ALGORITHMS

In this section, the results of the tests run on the 2D algorithms are presented and discussed.

5.2.1 *Incremental Algorithm*

Figure 14 shows the results of running the incremental algorithm on the five different point distributions. As observed, the performance of the incremental algorithm does not change much based on the distribution, as the running time is dominated by the sorting algorithm, which does not care about distribution. The circle distribution runs marginally slower than the other distributions, and that may be attributed to the fact that more points are clustered towards the center, and this forces the algorithm to do more recalculations.

5.2.2 *Gift Wrapping Algorithm*

Figure 15 shows that the gift wrapping algorithm is more sensitive to the point distribution. It also demonstrates the output-sensitivity, by placing the uniform distribution with minimal hull much lower than the other distributions. The polynomial distribution forces every single input point to be part of the final hull, thus reducing the running

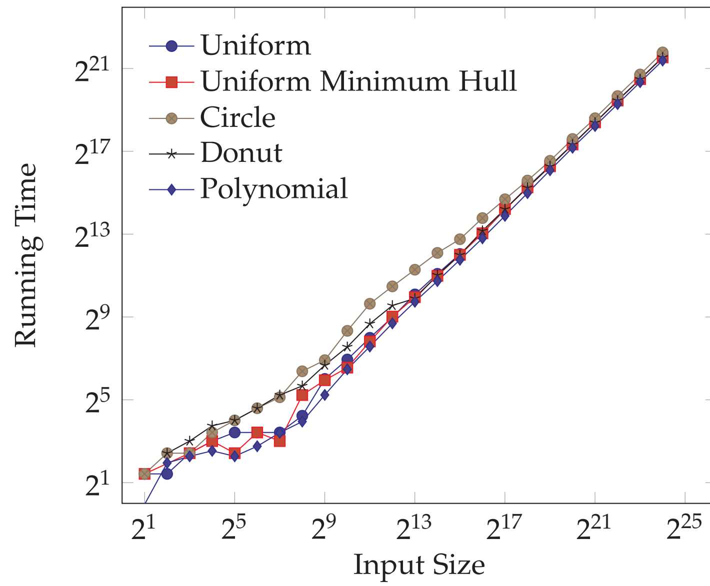

Figure 14: Incremental 2D Algorithm

time of the gift wrapping algorithm to $\mathcal{O}(n^2)$, and that clearly shows. The other distributions correspond perfectly to the average hull sizes.

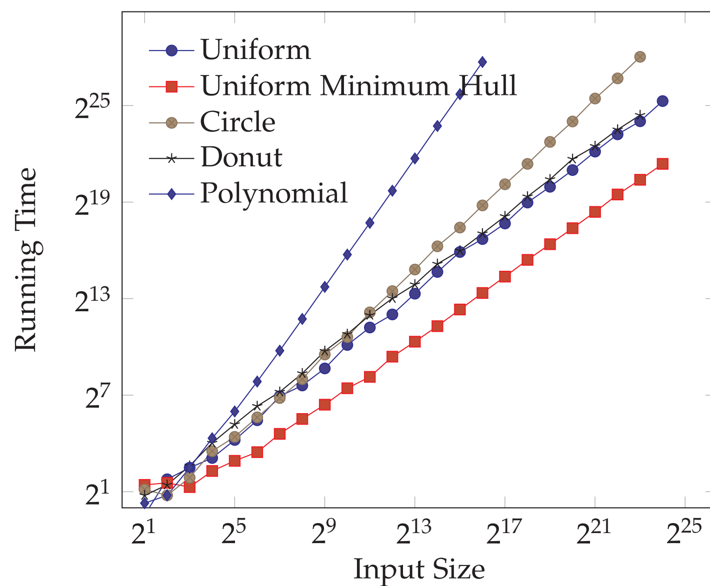

Figure 15: Gift Wrapping 2D Algorithm

5.2.3 *Kirkpatrick-Seidel Algorithm*

As seen in figure 16, the polynomial distribution is the slowest, which is to be expected from the output-sensitive Kirkpatrick-Seidel algorithm, as $h = n$ for this distribution. In this situation, the asymptotic running time will be $\mathcal{O}(n \log n)$ like the incremental algorithm, but

due to the complexity of the Kirkpatrick-Seidel algorithm, it has a much higher running time constant. The circle and donut distributions are second-worst, with the circle distribution being fastest for the smallest half of the graph, before being overtaken by the donut distribution. This makes perfect sense when compared to the graph for average hull sizes, as it is around the same place where the hull size of the donut distribution becomes smaller than the the hull size of the circle distribution. After this it is the uniform distributions, both of which demonstrates the output-sensitivity of the algorithm.

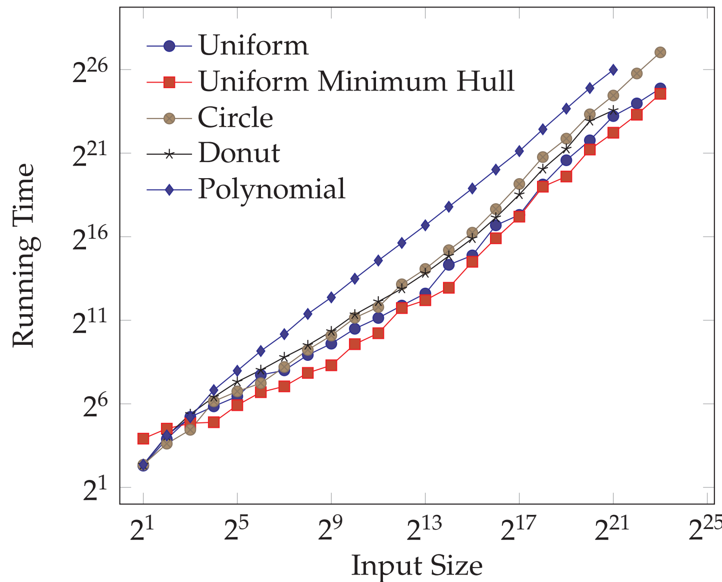

Figure 16: Kirkpatrick-Seidel 2D Algorithm

5.2.4 *Kirkpatrick-Seidel Algorithm using Linear Programming*

The graph in figure 17 shows the exact same relative performance as in the previous algorithm, which makes sense, as the only thing changed is a running time constant. The main difference is that the algorithm runs significantly faster across all distributions.

5.2.5 *Running Time Ratios*

The ratio of tested running time and theoretical running time is also interesting to investigate. The graphs 18, 19, 20 and 21 show this ratio for the 2D algorithms. If the measured running times follow the theoretical running times, they are expected to have a horizontal line through the graph, and the further up in the graph they are, the larger the running time constant. For the output-sensitive algorithms, the theoretical running time is calculated based on the measured average hull sizes.

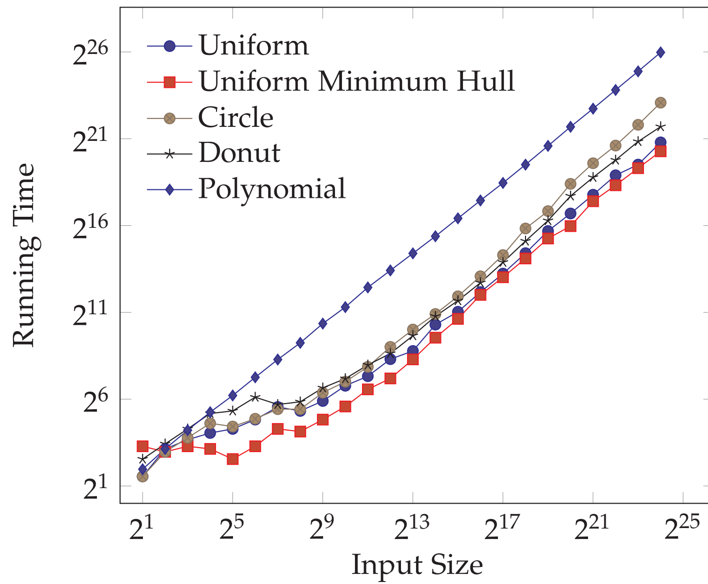

Figure 17: Kirkpatrick-Seidel 2D Algorithm w/ Linear Programming

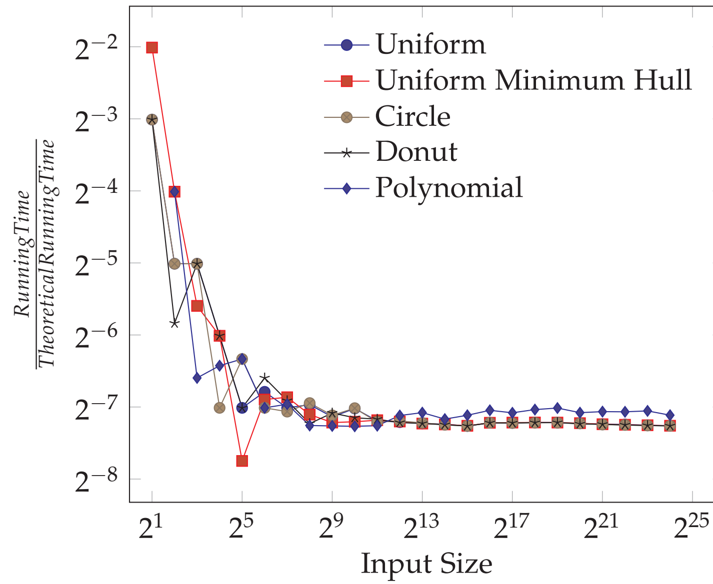

Figure 18: Incremental 2D Algorithm Ratio

Figure 18 shows that the incremental algorithm is initially very divergent from the theoretical $\mathcal{O}(n \log n)$ running time, but as the input gets larger, it settles nicely with a small running time constant.

In figure 19 the two extreme running times have the lowest running time constants. This could be attributed to such low-level optimizations as branch predictions, as the algorithm always picks the same path.

The results of the Kirkpatrick-Seidel algorithms in figures 20 and 21 show wildly varying running time constants. This is likely caused by the very large overhead of the algorithms.

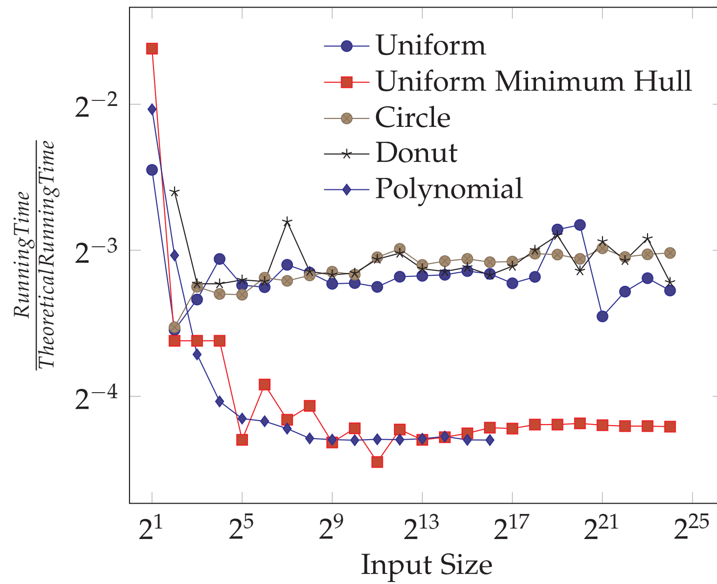

Figure 19: Gift Wrapping 2D Algorithm Ratio

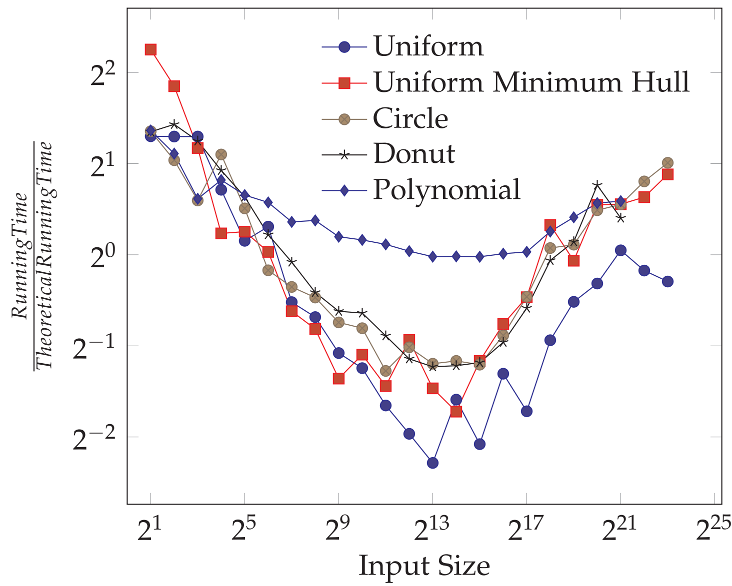

Figure 20: Kirkpatrick-Seidel 2D Algorithm Ratio

5.3 3D ALGORITHMS

In this chapter, tests on the 3D convex hull algorithms are described and discussed. Since the only 3D algorithm implemented is the gift wrapping algorithm, there are no other algorithms to compare to. Thus, this chapter will focus solely on relating the test results to the theoretical bounds. The point distributions on which the algorithm is tested, are similar to the distributions for 2D algorithms, but extended into 3D.

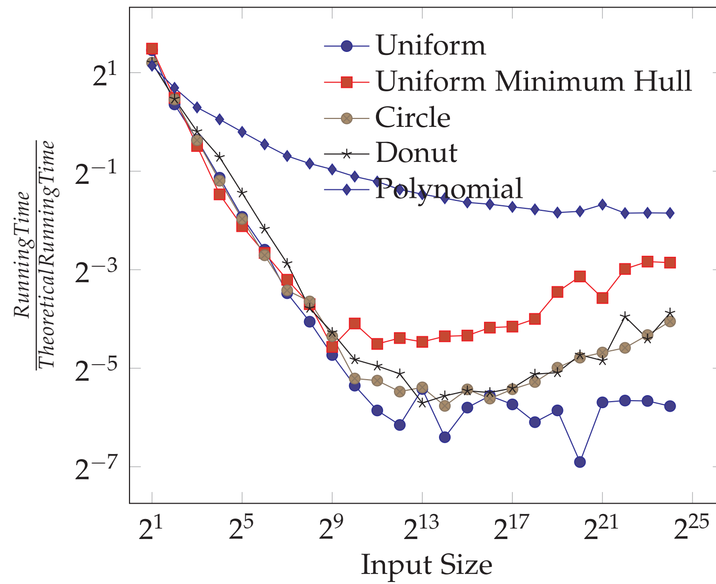

Figure 21: Kirkpatrick-Seidel 2D Algorithm w/ Linear Programming Ratio

5.3.1 *Sphere*

A simple distribution with points uniformly distributed inside of a sphere with radius 1, using uniformly distributed polar coordinates, is used. This acts as a baseline test, as the hull size shouldn't be a too large part of the input set.

5.3.2 *Uniform Box*

A simple distribution with points uniformly distributed on the interval $[-1;1]$ in all three dimensions, is also used. With this distribution, the hull should be a little smaller than with the sphere distribution.

5.3.3 *Uniform Bounding Box*

In this distribution, the random points are encased in a cube, giving us a hull of constant size, no matter the input. As with the bounding triangle test in 2D, this ensures optimal performance of the output-sensitive algorithms. This always adds eight extra points to the distribution.

5.3.4 *Paraboloid*

The points in this polynomial distribution is distributed along a paraboloid with function $Z = -X^2 - Y^2$. Equivalent to the parabola distribution

in 2D, the purpose of this distribution is to examine the worst-case performance of the output-sensitive algorithms.

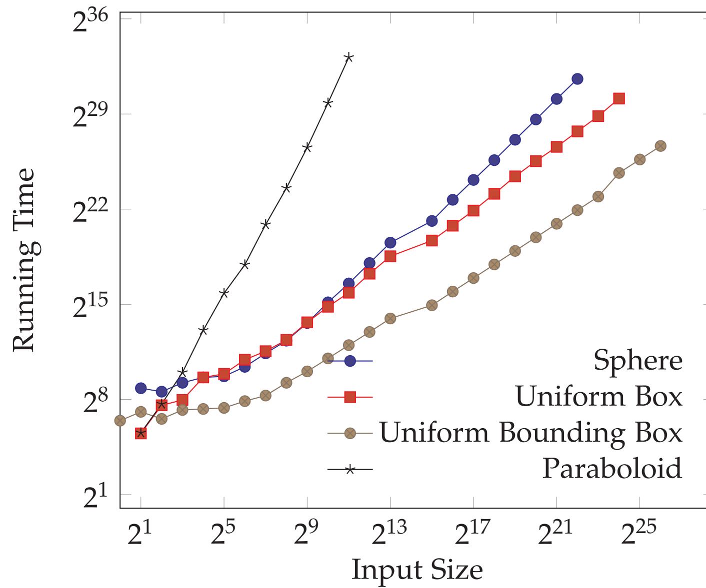

Figure 22: Gift Wrapping 3D Algorithm

5.3.5 *Results*

The results of running the gift wrapping algorithm on these distributions can be seen in figure 22. As was expected, the paraboloid distribution increases the size of the hull to $h = n$, making the algorithm run in $\mathcal{O}(n^2)$ time. The bounding box distribution is far faster than the other distributions, demonstrating the output-sensitivity, as this distribution effectively makes the algorithm run in $\mathcal{O}(n)$ time. At $n = 2^1$, the bounding box distribution is slightly slower than some of the other distributions. This is caused by the adding of the bounding box, no matter the size of the input. The sphere and box distributions are very close in performance, with the box distribution being a bit faster. This is also due to the box distribution having slightly fewer points on the hull. Incidentally, both the sphere and box distributions are much closer in performance to the best-case bounding box distribution, than the worst-case paraboloid distribution.

5.4 LINEAR PROGRAMMING

In this section, the results of tests comparing the incremental method of solving linear programs to the randomized method, will be presented. The results will show that the average performance of the randomized algorithm lies extremely close to the best-case performance of the incremental algorithm.

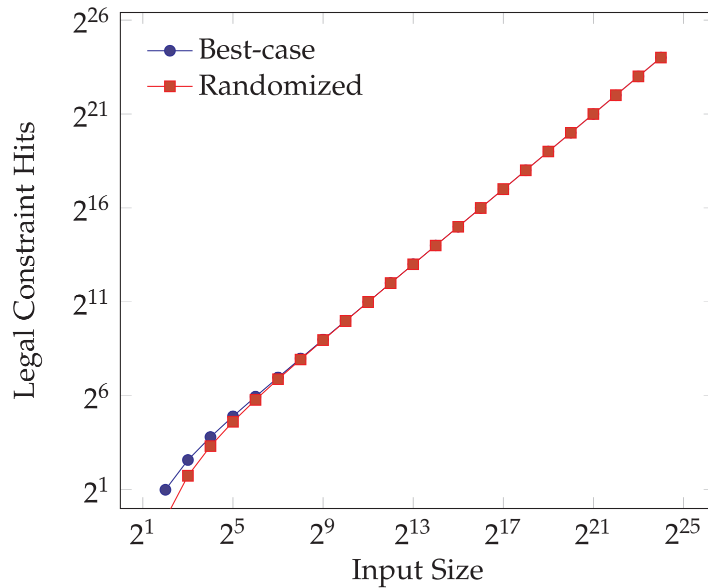

Figure 23: Legal Constraint Hits

When a new constraint is added, and it does not change v_i, this is counted as a *legal* constraint. Similarly a constraint that changes v_i is counted as an *illegal* constraint. As illegal constraints causes the algorithm to solve a new 1D algorithm, the number of illegal constraints should be minimized. In figure 23 and 24 it is observed that the number of legal constraints in the best-case input is $n - 2$, and two illegal constraints for all n. The worst case input has n illegal constraints and 0 legal constraints, so it does not show up in the graph for legal constraints, as it has logarithmic axes. It shows that for the randomized algorithm, the number of illegal constraints are very low, as expected as per the theory for the likelyhood of hitting an illegal constraint. Figure 25 shows the ratio between the legal and the illegal constraints. Again, the number of legal constraints are very close to the best-case.

Figure 26 shows the running times of best- and worst-case input to an incremental algorithm, along with the running time of the randomized algorithm. The figure shows that the worst-case running time fits the theorical bound proven earlier, along with the randomized algorithm only being slower than the best-case by a constant factor, showing the expected $\mathcal{O}(n)$ running time.

Figure 27 shows the percentage of the added constraints where the constraint does not fail. As seen, the randomized approach follows the best-case percentage by a small factor, illustrating that the randomized approach is close to the best-case in legal hits.

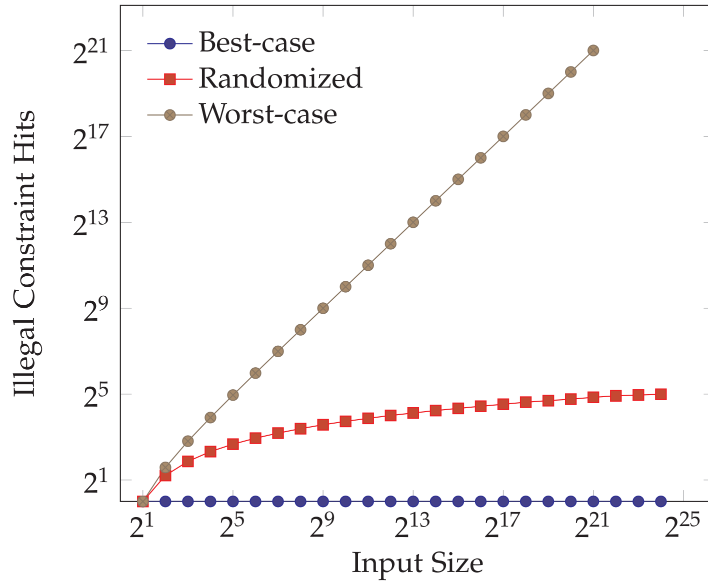

Figure 24: Illegal Constraint Hits

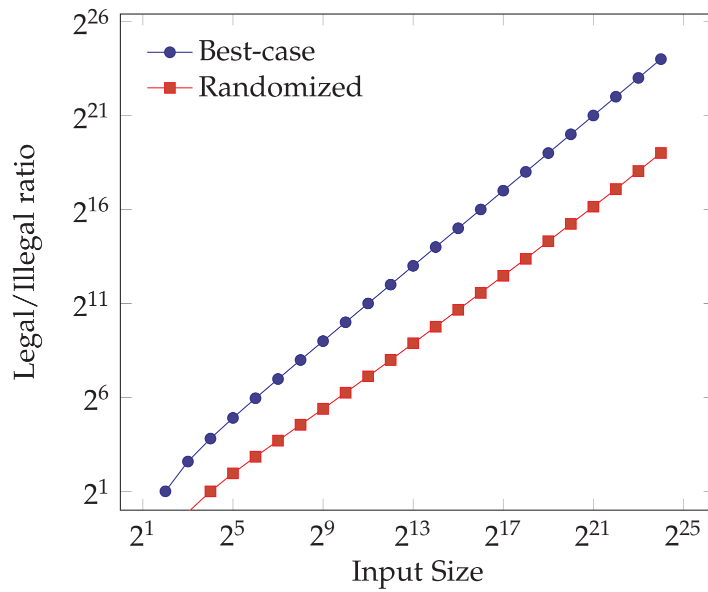

Figure 25: Randomized LP Legal/Illegal ratio

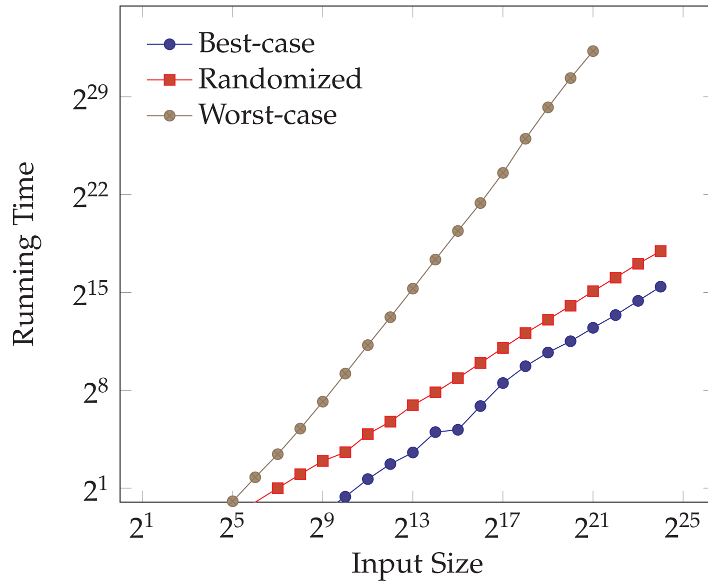

Figure 26: Incremental vs. Randomized LP Running Time

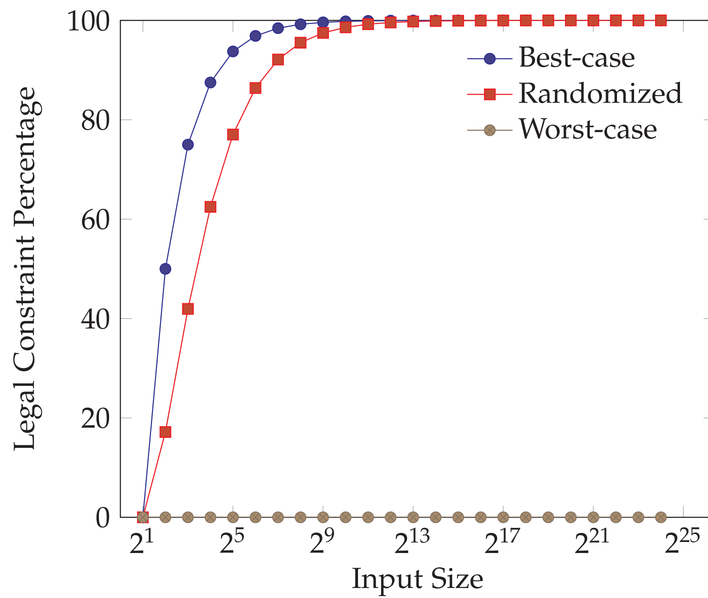

Figure 27: Randomized LP Legal/Illegal Percent

6

CONCLUSION

In this thesis, the problem of computing the convex hull of a set of points have been described, and a number of algorithms for computing convex hulls have been presented, primarily for two dimensions, but also for three dimensions. The majority of these algorithms have been implemented and tested, and the results of the tests have been presented, analysed and discussed. The results have also been compared with theoretical analyses of the algorithms.

The concept of linear programming has been introduced, and a deterministic algorithm, as well as a randomized algorithm, has been presented for solving these linear programs. These algorithms have been analysed, implemented and tested, and the results compared to the analysis and discussed. An algorithm was presented for solving multiple linear programs on the same constraints at the same time, faster than solving them one by one.

Based on the test results of the 2D convex hull algorithms, the fastest algorithms are the Kirkpatrick-Seidel algorithm with linear program, and the incremental algorithm. Based on the relative size of the hull, either of the algorithms performed better than the other. Although the Kirkpatrick-Seidel algorithm was the fastest in some cases, additional work should go into optimizing it before it will be a clearly superior algorithm to the incremental algorithm, given its heavy layer of complexity.

The test results of the incremental versus randomized linear programming algorithm showed that simply shuffling the input randomly can give a performance boost to a worst-case input.

6.1 future work

The next step in extending these algorithms for convex hulls could be to prove that the randomized partitioning scheme is expected to as good as the deterministic approach.

Another possibility could be to extend the the multiple linear programming algorithm into 3D. This would involve extending the partitioning algorithm into 3D as well, where it would work by cutting the eight sets instead of four by using cutting planes instead of cutting lines. It would involve cutting the set into two by the median of the x-values, cutting the sets into two subsets using the ham sandwich approach twice, to achieve the eight subsets.

Further effort could be applied to implementing and testing both the 2D and 3D approach to multiple linear programming, and other algorithms in 3D. This would first involve extending the `LPSolver` with the capability of solving linear programs in 3D.

Dealing with the problem of finite precision arithmetic could also be an interesting area of research, as this could potentially be an issue in all the situations where dealing with mathematical optimizations and line-point tests is crucial.

BIBLIOGRAPHY

[1] Andrew, A. M. Another efficient algorithm for convex hulls in two dimensions. *Inf. Process. Lett.* (1979), 216 – 219.

[2] Berg, M. d., Cheong, O., Kreveld, M. v., and Overmars, M. *Computational Geometry: Algorithms and Applications*, 3rd ed. ed. Springer-Verlag TELOS, Santa Clara, CA, USA, 2008.

[3] Chan, T. M. Fixed-dimensional linear programming queries made easy. In *Proceedings of the Twelfth Annual Symposium on Computational Geometry* (New York, NY, USA, 1996), SCG '96, ACM, pp. 284–290.

[4] Chazelle, B., Guibas, L. J., and Lee, D. T. The power of geometric duality. *BIT 25*, 1 (June 1985), 76–90.

[5] Graham, R. L. An efficient algorithm for determining the convex hull of a finite planar set. *Inf. Process. Lett. 1* (1972), 132 – 133.

[6] Hoare, C. A. R. Algorithm 64: Quicksort. *Commun. ACM 4*, 7 (July 1961), 321–.

[7] Jarvis, R. A. On the identification of the convex hull of a finite set of points in the plane. *Inf. Process. Lett. 2*, 1 (1973), 18–21.

[8] Kirkpatrick, D. G., and Seidel, R. The ultimate planar convex hull algorithm. *SIAM J. Comput. 15*, 1 (Feb. 1986), 287–299.

[9] Lo, C.-Y., Matouek, J., and Steiger, W. Algorithms for ham-sandwich cuts. *Discrete & Computational Geometry 11*, 1 (1994), 433–452.

[10] Megiddo, N. Linear programming in linear time when the dimension is fixed. *J. ACM* (1984), 114 – 127.