Abstract – (FCS’15) The convex hull of a set of points, \( Q \), denoted by \( \text{CH}(Q) \), is the smallest convex polygon that encloses all the points of \( P \). Each point in the convex hull is either on the boundary of \( P \) or in its interior. Traditional algorithms such as Graham’s Scan, Jarvis March, and QuickHull use all the points in \( Q \) as input to obtain the \( \text{CH}(Q) \). This research developed a pre-processing “sieve” algorithm that reduces the number of points necessary to compute the \( \text{CH}(Q) \). Given a set \( Q \) with integer coordinates, an \( O(n) \) sieve algorithm “filters” interior collinear points from \( Q \) into a reduced set of sieve points \( SP \). This reduces the number of points necessary to compute the \( \text{CH}(Q) \) and improves the overall performance of the traditional convex hull algorithms. The sieve points remain a superset of the hull points; \( SP \supset \text{CH}(Q) \). A program has been created to test the correctness of the sieve algorithm[4][8][9]. Using a uniform distribution of points, the program has shown that \( |SP| \ll |Q| \).

Keywords: Computational geometry, convex hull, sieve algorithms.

1 Introduction

Computing a convex hull of a set of points is a well-studied research topic in computational geometry. A typical course in analysis of algorithms often covers traditional convex hull algorithms such as Graham Scan, Jarvis March, and QuickHull [4][8][9]. It is widely known that the complexity of Graham Scan is \( O(n \lg n) \) since the points must be sorted radially about the lowest-leftmost point. Over the years, other algorithms and improvements to existing algorithms have been developed.

1.1 Research Overview

Figure 1: Data flow of points through the sieving algorithm followed by computing the convex hull.

Other improvements to existing convex hull involve heuristics such as the Akl-Toussaint heuristic that finds upper, lower, left-most, and right-most extreme points to form an irregular convex quadrilateral. Points within this convex quadrilateral can safely be eliminated, thus reducing the number of points for subsequent convex hull algorithms. Many of these improvements involve comparing two or more points together to satisfy some criterion. It is the number of point-to-point comparisons that is used to measure the complexity of a convex hull algorithm.
using a traditional algorithm. As shown in Figure 1, the output from this sieve is a set of points that are a superset of the convex hull points.

The sieving algorithm uses a form of “bucket sort” that sort/organizes points with corresponding coordinates in the first dimension, then uses a second “bucket sort” in the second dimension to arrange the points in ascending order. The sieving algorithm does not compare any point with any other point; i.e. no comparisons. The complexity of the algorithm is $O(n)$ and will be further explained later in this paper.

1.2 Organization

The organization of this paper is the following. Section 2 will present some background information on sieving algorithms, traditional convex hull algorithms, and sorting in linear time using bucket sort. Section 3 will present and explain our algorithm and discuss the program used to test its correctness and gather timing results. Section 4 will present results from our algorithm testing for varying number of points and point densities. Section 5 will present some conclusions and future work for this research.

2 Background

Numerous algorithms and algorithmic techniques were researched and explored. The major components of this research are the concepts of sieving algorithms, traditional convex hull algorithms, and sorting in linear time using bucket sort.

2.1 Sieving Algorithms

A sieve, in the mathematical sense, is an algorithmic technique to filter or eliminate non-essential data to achieve a desired result. The classic sieving algorithm is the Sieve of Eratosthenes used to find all prime numbers less than or equal to some number $n$ [4][6]. Named after the Greek mathematician, this prime number sieve algorithm repeatedly eliminates multiples of prime numbers ($2p, 2p, 4p, ...$) until the value of $np$ is less than the $\sqrt{n}$. The overall complexity of the Sieve of Eratosthenes is $O(\lg \lg n)$ [4].

A characteristic of sieve algorithms is that they often do not have to compare data values with other data values: no comparisons and few (if any) data dependencies.

2.2 Traditional Convex Hull Algorithms

There are many algorithms that compute the convex hull of a set of points. Three classic algorithms are the Jarvis March (gift-wrapping) algorithm, Quickhull, and Graham Scan. The Jarvis March algorithm was developed in 1970 by Chand and Kapur and also independently in 1973 by R. Jarvis. [7] This algorithm begins with a known point on the hull, $p_0$, and compares polar angles of other points. It selects the next point on the hull that has the smallest polar coordinate and then the process repeats “wrapping” around the points to form the convex hull. The complexity of Jarvis March is $O(nh)$ where $n$ is the number of points, and $h$ is the number of hull points.

The Quickhull algorithm was discovered by Eddy 1977 and also independently by Bykat in 1978. [1][9] It is analogous to the quicksort algorithm. The Quickhull algorithm finds two points with the minimum and maximum x coordinates to create a dividing line through the set of points creating an upper set and lower set of points. It next finds a point in P that is a maximum distance from the dividing line. All points lying in this triangle of points are excluded from the convex hull. Then the process repeats until no other hull points are discovered. The same procedure repeats for finding the lower convex hull. Finally the two sets of points are combined. The complexity of Quickhull is $O(n \ln n)$ in the average case and $O(n^2)$ in the worst case.

The Graham scan algorithm first finds the lowest right-most point $p_0$. This point is on the convex hull. Next, all points are sorted radially about $p_0$. Next, the first two points are pushed on a stack. If the top two points on the stack and the next point considered make a counter-clockwise turn, then the new point is pushed on the stack. If the points make a clockwise turn, then the top point on the stack is popped off. This process repeats until the three points considered make a counter-clockwise turn. The complexity of Graham scan is bounded by sorting and has an overall complexity of $O(n \lg n)$.[4][5][9][11]
2.3 Sorting in Linear Time Using Bucket Sort

Bucket sort is also known as bin sorting.[4] It can sort a set of data by partitioning the input array into buckets. Each bucket can then be sorted using a traditional sorting algorithm or recursively calling bucket sort again. The complexity of bucket sort depends on the number of buckets. If \( n \) is the input data size and \( M \) is the number of buckets, then the complexity of bucket sort is \( O(n+M) \).

3 Methods

This section will present an overview of the convex hull sieve describing major stages of the algorithm. This section will also present a discussion of the full algorithm and implementation.

3.1 Sieve Algorithm Overview

For purposes of this discussion, the following example uses a set of 50 points. All input points to this algorithm have integer coordinates and are in the first quadrant. Figure 2 shows the initial set of points. Notice the x-collinearities (which are points that share a common x-coordinate) and y-collinearities (which are points that share a common y-coordinate). The set of input points are in no particular order (random permutation of the list of points).

![Initial Set of Points](image1)

**Figure 2: Initial set of 50 points.**

The algorithm begins by first identifying y-collinearities. This is accomplished by traversing the list of points and placing each point in a common bucket if they share a common y-coordinate. Again, because the initial list of points is random, the points in each “y-bucket” are also randomized (unsorted). The points in each y-bucket are passed to a bucket sort again and lexicographically ordered by x-coordinate. Any point between the first point and last point in each y-bucket is removed because it is an interior point and not on the convex hull. Figure 3 illustrates the y-collinear points in each bucket.

The next stage of the algorithm then eliminates the x-collinearities. The remaining points from the previous step are stored in respective “x-buckets” in lexicographic order. Any point between the first point and last point in each x-bucket is removed because it is also an interior point and not on the convex hull. Figure 4 illustrates the x-collinear points in each bucket.

The output of the sieving algorithm is a set of candidate points that are on the outer boundaries of the initial set of points. These points are either maxima or minima in each x and y directions. Figure 5 shows the resultant set of points with the x-collinearities removed. The output points are used as input points into a traditional convex hull algorithm and the convex hull points are determined. The final convex hull is also shown in Figure 5.

![Eliminating Interior Y Collinearities](image2)

**Figure 3: Identifying and eliminating y-collinearities.**
3.2 Detailed Sieve Algorithm

Using these major steps, a detailed algorithm was developed and shown in Figure 6.

Algorithm SievePoints(Q) returns SP
1. Let y be a list of lists with magnitude defined by the Range(Q)
2. Let x be a list of lists with magnitude defined by Domain(Q)
3. For each point p in Q Do:
   4.   y[p.y].append(p)
5. For each bucket b in y Do:
   6.   b = sort_bucket(b)
   7.   If b.length > 0 Then
      8.      p = b[0]
   9.   x[p.x].append(p)
10. If b.length >= 2 Then
11.      p = b[-1]
12.      x[p.x].append(p)
13. For each bucket in x Do:
14.      Else If b.length > 0 Then
15.         SP.append(b[0])
16.      If b.length >= 2 Then
17.         SP.append(b[-1])

Figure 6: Detailed sieving algorithm.

The algorithm has input a set of points Q and returns a set of points SP. Steps 1 and 2 set up lists that are used temporarily store the points in buckets. These will be used to sort the points in the “y-direction” and “x-direction” respectively. Step 3 and 4 scan through the list of points and assign them to a bucket based on their y-coordinate. Then, in steps 5 and 6, each y-bucket is sorted using a stable sort such as bucket-sort. Step 6 identifies buckets with size greater than zero. In steps 7, 8, and 9, if a bucket size is found, then the first point is then reassigned to an x-bucket. In steps 10, 11, and 12, if the number points in each y-bucket is greater than or equal to 2, then the last point is also reassigned to an x-bucket. At this point in the algorithm all y-collinearities have been removed.

The second phase of the algorithm is similar to the first stage. In steps 13, 14, 15, and 16, each x-bucket is examined. If the number of points in an x-bucket is greater than 0, then the first point is copied to the sieve point list. If the number of points in an x-bucket is greater than 2, then the last point in the x-bucket is copied to the sieve points list.

3.3 Complexity of the Sieve Algorithm

In terms of space complexity, the sieving algorithm clearly uses O(n) space where n is the number of points in Q. In terms of time complexity, steps 3 and 4 can be computed in O(n) time. Each bucket is sorted using the sort_bucket() algorithm. This sorting algorithm uses bucket-sort that is a form of radix-sort. This is a stable-sort that can execute in O(n) time. Also noted is that the input points to the sort are only those that are y-collinear or x-collinear and only analyzed once per bucket. This reduces the time complexity significantly to ||y|| * O(||y.b||) where ||y|| is the number of buckets and ||y.b|| is the number of points in a given bucket. The value ||y|| is
a constant determined at runtime. Therefore, steps 5 through 12 run in $O(n)$ time. Steps 13 through 17 run in $O(n)$ time. Overall the runtime complexity of the sieving

4 Results and Analysis

This section will present the results and analysis of an implementation of the convex hull sieve algorithm when combined with an execution of the Graham Scan algorithm to compute the actual convex hull.

4.1 Test Suite

As a proof of concept, this convex hull sieve algorithm has been implemented in Python. Several sets of input points were used. The data sets were a uniformly distributed set of points that varied number the domain space of input points as well as the density of points in the domain. For example, for a domain space of 1000 x 1000, the total number possible points is 1 million points. However, the density would then reduce the number maximum number of points. With a density of 0.4, then a total of 400,000 points would be uniformly spaced in a domain of 1000 x 1000. The Python Timeit module was used to record the execution times.

4.2 Results for 10 x 10 Space

Figure 7 is a plot of computation time in seconds for different point densities. For this set the total number of unique points was 100. The algorithm was tested for the density range of .1 to .8. The series in orange displays the time it takes to sieve plus the time it takes to compute the convex hull using one of the conventional convex hull algorithms. From here, it is observed that at a density of 0.25 (25 points), the addition of a preprocessing step surpasses the performance of computing the convex hull of the entire set of points $G$.

4.3 Results for 1000 x 1000 Space

Table 1 shows sample data gathered for a 1000 x 1000 space. Time(sieve) is the time it took to transform $P$ to $SP$. Time(CH(SP)) is the time it took to compute the convex hull of the sieved points. Time(S+CH(SP)) is the sum of the first two columns. Time CH($G$) is the time it took to sort the same points using Graham Scan without the sieving step. $G'$ is the number of points after the sieve. $G$ is the original number of input points.

Figure 6: Results from a 10 x 10 space with varying point densities.

Figure 7: Sample execution times for a 1000 x 1000 space with varying densities.

The graph in Figure 7 is similar to Figure 6 above and shows the time it takes to compute the convex hull using Graham’s Scan algorithm ($O(n \log n)$). This Figure also displays the time it takes to compute the convex hull using Graham’s Scan with the addition of a preprocessing step. Each series was tested with the same set of unique points starting at 100,000 points and increasing in increments of 100,000 until the total reaches 800,000 (increasing densities from 0.1 to 0.8). The addition of a preprocessing step drastically reduces the amount of time needed to compute the convex hull. At 800,000 points the conventional algorithm took nearly 90 seconds to complete, while the algorithm in addition to the preprocessing step took under 1 second to compute the convex hull for the same set of points $G$. 

Figure 7: Results from a 1000 x 1000 space with varying point densities.

Figure 8 verifies empirically that the convex hull sieving algorithm is a linear function with respect to the number of points: O(n).

Figure 8: Algorithmic performance of the convex hull sieve algorithm.

Figure 9 is a plot showing the efficiency of adding the convex hull sieve algorithm. The efficiency of executing Graham Scan improves as the density of points increases. As the density increases, so does the likelihood that it will be removed using the sieving algorithm.

Figure 9: Efficiency of adding the preprocessing step.

5 Conclusions and Future Work

The research presented a technique to increase the performance of traditional convex hull algorithms. This is accomplished by a preprocessing sieving algorithm that filters the points prior to the execution of a convex hull algorithm. The sieving algorithm repeatedly uses bucket-sort, which has complexity O(n), in the “x” and “y” directions, to remove interior collinear points.

For the case of a 10 x 10 space, it is shown that the benefits of using the convex hull sieving algorithm begins at roughly 25 points. For the case of 1000 x 1000, the performance improvement of using this convex hull sieving algorithm is significant. In the case of 800000 points the execution time of the sieve algorithm and Graham’s Scan was approximately 1 second as compared to 90 seconds using Graham’s Scan alone.

Future directions for this research are numerous. First, the test program is going to be converted from Python to C++ for more native and accurate timing analysis. This may lead to a more accurate timing model for this algorithm. Second, a variant of the algorithm is being developed for 3-dimensional convex hulls. This should be a straightforward extension, but will require significant testing and verification. Finally, the sieving algorithm has opportunities for parallelism. Perhaps using threads, the buckets could independently be populated from the original list of points. Using a portable thread library such as pthreads, would increase the portability of this algorithms to include hyper-threading processors and hardware accelerators.

6 References


