Quickhull - Parallel and Approximation Algorithms

Rachelle David and Juan Neri

Department of Electrical and Computer Engineering
University of Texas at Austin,
Austin, TX 78712
{rachelle.david, juanjoneri}@utexas.edu

Abstract

In this project, we examine the Quickhull algorithm for computing the convex hull of a finite set of points in a plane. We explore the sequential version of the algorithm, as well as its parallel and approximate counterparts. We compare and analyze their performance on data sets of various shapes and sizes.

Introduction

The Convex Hull of a set of points is the smallest possible convex set that contains all the points in the set. This interesting problem is frequently explored for solution algorithms, especially pertaining to its many applications in a wide variety of fields, to include computer graphics, game theory, and computational algorithms [RS16]. In order to better understand Convex Hull, we will briefly explore a few of these applications.

The first application we will explore is image processing, particularly in x-rays of objects. In an x-ray CT situation, the information about the object being scanned is limited to only the angle of the x-ray source. Additionally, the data is typically entry and exit points based on the density of that material. However, in order to estimate the object's exterior boundary, technicians can use the x-ray data to construct Convex Hull of that object [Tam89]. This process is very similar to rendering computer graphics, in which the objects being rendered are observed through a single point, called the camera. The camera can view points or vertices on the objects and render the surface of that object that is viewable from the camera using its Convex Hull.

Another application is in gift wrapping. It is a rather simple application of the problem in which a person has an item to give as a gift and needs to determine if they have enough paper to wrap the present. Since the Convex Hull of the item or set of items can provide the smallest boundary that contains all the items, this person can determine the amount of paper they need.

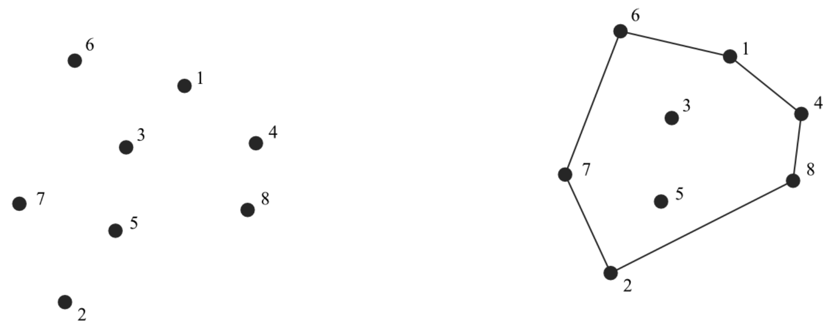
The final application we will discuss here is clustering in a graph in statistics problems. Since clusters are groupings of similar objects, we want to minimize the spread of each cluster. That is, minimize the maximum the distance between any two points in a cluster, or the diameter of the cluster. Through some analysis, it can be determined that the diameter of a set is equal to the diameter of its Convex Hull. So, given a partition of a scatter plot graph, we minimize the diameter of each cluster by finding the minimum Convex Hull that still provides a meaningful grouping of similar objects [Pre85].

In this paper, we will explore the Quickhull algorithm for solving the Convex Hull problem and both a parallel and approximation algorithm variations of it. Then, we will analyze the performance of these algorithms for various data sets.

Convex Hull Problem

Formally, given a finite set of size n of ordered pairs (points) in the Cartesian plane $S \equiv \{p_i \in \mathbb{R}^2 \mid 0 \leq i < n\}$, the convex hull problem requires finding the set of segments that form the smallest convex set (also known as envelope or hull) that contains S. The convention is for these segments to be given as a linked list containing the vertices in clockwise order.

Fig. 1: The convex hull of a set of size 7



(a) Initial set of points in the plane

(b) The solution is a list of edges represented by [7, 6, 1, 4, 8, 2]

Convex Hull Solution Algorithms

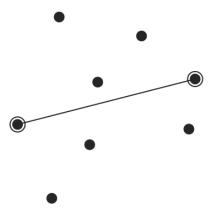
Quickhull Algorithm

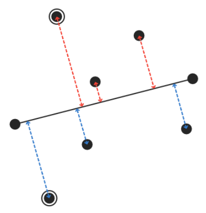
The Quickhull algorithm takes inspiration (and its name) from the well known sorting algorithm Quicksort. Recall that in the original Quicksort method [Hoa62] we have an array of N numbers which are portioned into a left and right sub-arrays such that all elements in the left sub-array are smaller than the elements in the right sub-array. The partition is obtained by selecting a "bound" value (b) and swapping elements using a lower (L) and upper (U) pointer whenever $A[L] > b$ and $A[U] < b$. After the partition is obtained the bound is "fixed", and the same technique can be applied separately to each side.

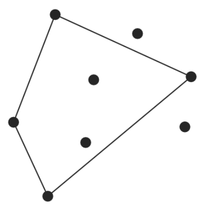
In a similar way, the Quickhull algorithm selects a vertex as a bound, and partitions the remaining vertices geometrically around that bound. After each step the bound becomes fixed, and the same technique is applied to each generated sub-problem.

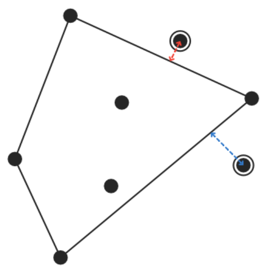
The initial partition is determined by the edge (l, r) with l and r having the smallest and largest value of x respectively. Each successive step operates on the resulting half planes in the following manner [Pre85]: determine a point h in the given half plane such that the triangle (l, h, r) has maximum area. h now becomes the new bound, and we iterate on the half planes generated on the left of the edges (l, h) and (h, r). We are done once there are no more points on the resulting half planes.

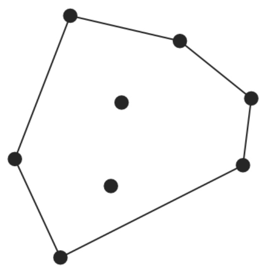
Fig. 2: Quickhull steps on a small set


(a) Initial partition


(b) Find h on each half plane


(c) Draw the edges (l, h) and (h, r)


(d) Iterate on each new bound


(e) Done once there are no more points on the new half planes

Below we provide a less informal description of the algorithm. Here S is assumed to contain at least two points and + denotes linked list concatenation.

Using the master theorem [Cor01] we can derive the expected and worst case complexity of this algorithm. Similarly to quickshort, if we assume that each

Algorithm 1 Quickhull(S)

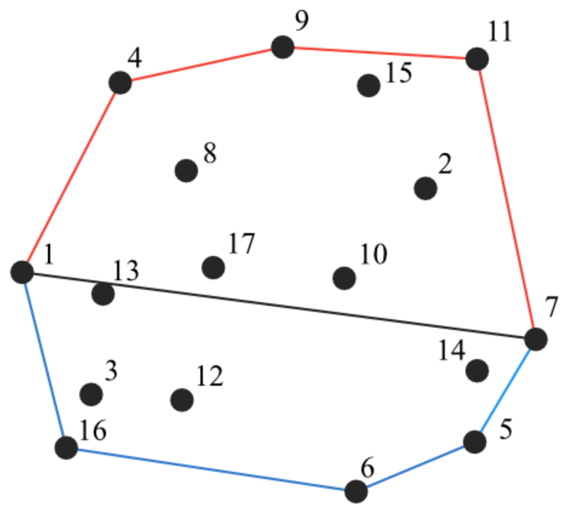
```
function FURTHESTLEFT(S, l, r)
    if no elements in S exist left of $\overrightarrow{lr}$ then
        return $NULL$
    end if
    $h \leftarrow$ element in S with largest perpendicular distance left of $\overrightarrow{lr}$
    return $h$
end function

function QUICKHULLREC(S, l, r)
    $h \leftarrow$ FURTHESTLEFT$(S, l, r)$
    if $h = NULL$ then
        return $(l, r)$ a single convex hull directed edge
    end if
    return QUICKHULLREC$(S, l, h)$ + QUICKHULLREC$(S, h, r)$
end function

$l, r \leftarrow$ elements of S with smallest and largest abscissae
return QUICKHULLREC$(S, l, r)$ + QUICKHULLREC$(S, r, l)$
```

iteration reduces the search space by 1/2, and because finding a new bound h and merging solutions are both linear operations, we arrive at the following recurrence relation: $T(n) = 2 \times T(n/2) + n$ and thus an expected runtime complexity of $O(n \times log(n))$. However, just like in quickshort, the assumption that each iteration reduces the search space by a constant factor does not hold in the worst case, resulting in an $O(n^2)$ running time.

Quickhull Parallelized Algorithm

The quickhull algorithm has the nice feature that each time it subdivides the problem, each resulting sub problem can be solved independently, and therefore simultaneously. Additionally, the merge step consists of a simple linked list concatenation of the results. Unfortunately, although very simple, parallel quickhull does not solve the main issue faced by quickhull, which is that each sub problem may have an arbitrary size. As a result, quickhull does not benefit of a parallel environment in the worst case with a complexity of $O(n^2)$. However as we will see in the performance comparison section, we did observe significant performance gains in practice.

Fig. 3: Merging sub results [1, 4, 9, 11, 7] and [7, 5, 6, 16, 1] consists of a simple concatenation resulting in [1, 4, 9, 11, 7, 5, 6, 16]

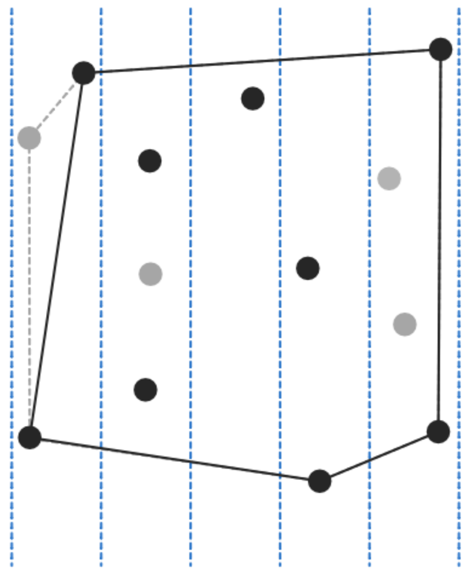


Quickhull Approximation Algorithm

Another method of improving on the Quickhull algorithm is to introduce a pre-processing algorithm that would reduce the size of the data set while providing and approximation of the Convex Hull. The main concept is to divide the initial set of points into equally sized vertical slices based on their x-coordinates. For each slice, we have a bucket of points only and track the points with the highest and lowest y-coordinate. We can remove all the points that do not have the max or min y-coordinate and are left with a subset of the original set S, called S'. Then, we run the simple Quickhull algorithm on S'.

Since we run Quickhull on only a subset of points, we can improve the performance, especially in large data sets. The only additional time complexity comes from the pre-processing. While we do see an improvement on performance in comparison to the simple Quickhull, the solution did not always match the expected deterministic solution. Since this method is only an approximation of the solution, there is a tradeoff of increased time performance for possible loss of accuracy.

Fig. 4: Example of a convex hull using the approximation scheme



Alternatives Considered

The Convex Hull problem has many existing solution algorithms, often coming about from the various applications of it. Two algorithms that we considered to explore are the simple Merge-Hull algorithm and the Graham Scan algorithm.

The Merge-Hull algorithm takes a divide-and-conquer approach. First, it divides the initial set of points into two halves, a right subset and a left subset. Then it finds the convex hull of those subsets recursively. Finally, it merges those subsets together and produces the overall convex hull.

The Graham Scan algorithm leverages polar coordinates and orientation to determine which points are part of the convex hull. This algorithm executes in two phases. First, it sorts the points to find which point has the smallest y value. Then, it accepts or rejects points based on the angle formed from the previous point added to the hull. In this way, it traverses through the path of the hull and produces the convex hull.

Although Merge-Hull and Graham Scan take very different approaches to the problem, both have a time complexity of $O(n \times log(n))$. While both of these algorithms are valid solutions algorithms, we decided to pursue the Quickhull algorithm since we wanted to explore different variations and improvements of it.

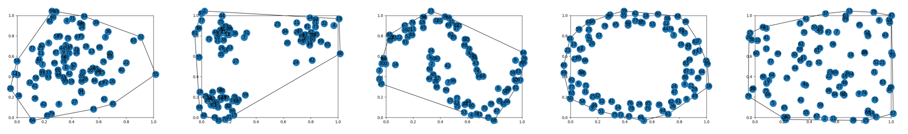
Performance Comparison

The following experimental results were obtained on a virtual machine running on Google Cloud Platform. The virtual machine was configured in the following way:

- 16 vCPU
- 64 GB memory
- 10GB balanced persistent disk
- Ubuntu 18.04 LTS
- Python 3.8.10

The data sets were generated pragmatically using scikit-learn [Ped+11] python library. We used the five distinct data set "shapes" shown below:

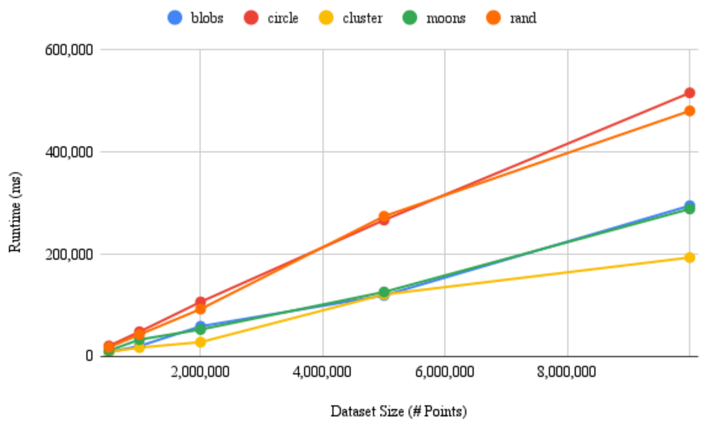
Fig. 5: Dataset "shapes"



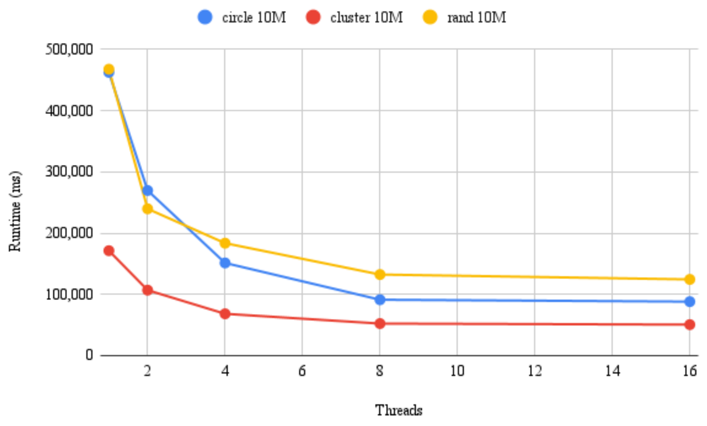
(a) Cluster (b) Blobs (c) Moons (d) Circle (e) Rand

The implementations as well as instructions for running each algorithm can be found at `https://github.com/juanjoneri/TermProject-Fa21-ECE382V`

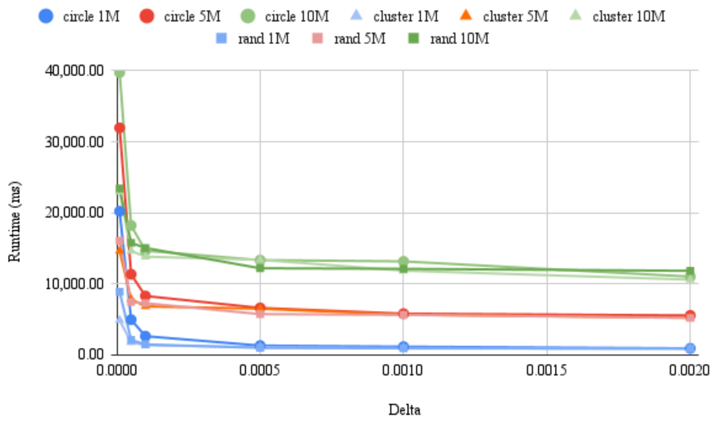
Fig. 6: Experimental Results



(a) Quickhull runtime



(b) Parallel quickhull runtime



(c) Approximation quickhull runtime

Discussion

In this section we will analyze the results of the performance comparison and how the counterparts of Quickhull compare to the simple algorithm.

As indicated in graph Figure 6(a), the simple Quickhull algorithm performs better on datasets of certain shapes. In fact, Quickhull runs slowest on the circle shape because it cannot discard many points on each iteration. The simple algorithm also runs slowly on the random dataset where edges might not be at the center. Since there is no circular symmetry, it many iterations do not benefit from the divide and conquer property of this algorithm.

While the simple Quickhull algorithm performs slowly for the circle and random shapes, it does perform well for very particular shapes. One such shape is the cluster dataset shape because it uses a Gaussian distribution and most points are close to the center and can be discarded early.

As the parallelized algorithm were assessed in Figure 6(b), the results from the parallel quickhull with one thread and one core are identical to regular quickhull performance. This is expected since a parallel algorithm with one thread executes the same as the simple quickhull algorithm and verifies our implementation. Additionally, the circle data set benefits the most from parallelization. This is likely for the same reasons that the simple quickhull algorithm performed poorly. When parallelized, the algorithm can iterate over the many points that remain in the set, rather than being discarded in parallel. In this way, the parallelized algorithms provides a significant improvement on the simple algorithm. Furthermore, this result suggests that for datasets that are non-ideal for the simple algorithm, the parallelized algorithm can be used for performance improvement.

In Figure 6(c) is the approximation algorithm's performance and it is noticeable that the performance plateaus at a delta of 0.0005. At this point, the run time is focusing primarily on the pre-processing itself, that contributes to the approximation, rather than the quickhull algorithm.

In the future, there is a possibility to parallelize the Quickhull approximation algorithm and further improve the efficiency of this algorithm. Further work would be needed to evaluate how much that combination improves the efficiency. Another future approach, is applying these algorithms to real-world datasets.

Conclusion

The Convex Hull problem has many existing solutions tailored for the various applications of it. In this paper we explored one such algorithm, Quickhull, and counterparts of it.

While Quickhull provides a reasonable generalized algorithm, it is not best suited for certain dataset shapes. However, the parallel and approximation algorithms make a noticeable improvement. That improvement is especially apparent with the datasets that impact the simple Quickhull algorithm the most. Those that are are most inefficient with the simple deterministic method benefit the most from one of the improved approaches.

References

[Cor01] Cormen, T. H. "Introduction to algorithms". In: The MIT Press, 2001, p. 73. ISBN: 0262032937.

[Hoa62] Hoare, C. A. R. "Quicksort". In: 5 (1962), pp. 10–16.

[Ped+11] Pedregosa, F. et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[Pre85] Preparata, F. P. "Computational Geometry". In: Springer-Verlag, 1985, pp. 112–180. ISBN: 0-387-96131-3.

[RS16] Ramesh, J. and Suresha, S. "Convex Hull - Parallel and Distributed Algorithms". In: *Stanford University Project Report* 1 (2016), pp. 1–6.

[Tam89] Tam, K. C. "The Application of Convex Hull in Industrial X-Ray Computerized Tomography". In: *The Aircraft Engine Business Group of General Electric Company* 1 (1989), pp. 389–397.