QuickHull Technical report

Darren Gichuru Gitagama

Abstract

Quickhull provides an efficient solution for computing convex hulls in high dimensional space. It is designed for efficiency and speed, implementing a divide and conquer approach to recursively add points into the convex hull. Owing to its uses in computational geometry, it provides a suitable solution for purposes such as robotics, distributed systems, path planning, and more. This report further delves into the key principles, aims, limitations, complexities and applications of the algorithm.

I certify that all material in this dissertation which is not my own work has been identified.

Signed: Darren Gitagama

candidate Number: 710038483

1 Algorithm Principles

Quickhull provides effective means to construct convex hulls, Delaunay triangulations, halfspace intersections, Voronoi diagrams, furthest-site Delaunay triangulations, and furthest-site Voronoi diagrams [1]. The Qhull manual [2] describes the various geometric tasks addressable with quick hull: (i) It is designed to operate effectively in high dimensional space (with dimensions greater than 2) (ii). It is output-sensitive, (iii) capable of mitigating most of the rounding errors formed from floating-point arithmetic, (iv) minimizes space requirements.

The QuickHull algorithm is used to calculate the convex hull of a set of points [3], referring to the smallest convex shape containing all points. This is achieved by implementing a divide and conquer approach that recursively selects points forming part of the convex hull, while also disregarding points that do not contribute to the outcome. The principles of the algorithm are described as follows: (i) Extreme point identification - referring to the selection of minimum and maximum x coordinates (i.e., points P and Q) occurring at the set's extremities for the initial simplex. As such, these points are included in the convex hull following observations stated in lemma 2.1, asserting that extreme points will always be included in the convex hull irrespective of the segment it aligns to. (ii) Segmentation - A line is drawn connecting extreme points (i.e., denoted as PQ), partitioning the set of points into two subsets occurring on opposite sides of PQ. (iii) Lemma 3.1 dictates that a point furthest away from the line joining two points within the convex hull will belong to the convex hull also. Therefore, the next steps involve recursively integrating such points into the convex hull, resulting in the formation of a triangle that may envelop other points, of which will never form part of the convex set and can, therefore, be disregarded.

2 Pseudocode

Algorithm 1 Quick Hull Algorithm

```
1: function QUICK_HULL(set)
2:     convex_hull ← []                ▷ List containing all points in the convex hull, initially empty
3:                                     ▷ If multiple values exist for left/right, pick any
4:     left ← minimum(set)
5:     right ← maximum(set)
6:     convex_hull ← left, right  ▷ Initial points selected are extremities and therefore automatically
   included in convex hull
7:                                     ▷ Split set into two subsets
8:     above, below ← divide_points(set, left, right)
9:     FIND_HULL(above, left, right, convex_hull, 0)
10:    FIND_HULL(below, left, right, convex_hull, 1)
11:    return convex_hull
```

Algorithm 2 Find Hull Function

```
function POINT_DISTANCE(left, right, point)
    return |(point[1] − left[1]) · (right[0] − left[0]) − (right[1] − left[1]) · (point[0] − left[0])|

function FIND_HULL(S, left, right, convex_hull, side)
    max_point_dist ← −∞
    max_point ← (0, 0)
    for s ∈ S do
        distance ← POINT_DISTANCE(left, right, s)
        if distance > max_point_dist then
            max_point_dist ← distance
            max_point ← s
    first_divide ← DIVIDE_POINTS(S, left, max_point)
    sec_divide ← DIVIDE_POINTS(S, max_point, right)
    if side = 0 then                    ▷ required as orderings change depending on the segment
        FIND_HULL(first_divide[0], left, max_point, convex_hull, 0)
        FIND_HULL(sec_divide[0], max_point, right, convex_hull, 0)
    else
        FIND_HULL(first_divide[1], left, max_point, convex_hull, 1)
        FIND_HULL(sec_divide[1], max_point, right, convex_hull, 1)
    if side ≠ 0 then                    ▷ if no points left beside the boundary
        if left ∉ convex_hull then
            append left to convex_hull
    if right ∉ convex_hull then
        append right to convex_hull
```

Algorithm 3 Divide Points Function

```
function DIVIDE_POINTS(S, p1, p2)
    above_segment ← []
    below_segment ← []
    for s ∈ S do
        segment ← (s[1] − p1[1]) · (p2[0] − p1[0]) − (p2[1] − p1[1]) · (s[0] − p1[0])
        if segment > 0 then
            above_segment ← s
        else if segment < 0 then
            below_segment ← s
    return above_segment, below_segment
```

3 Complexity Analysis

The key costs within the quickhull algorithm are associated to the distance computations and recursive partitioning of points within a face. The former occurs in linear O(n) time, as in the worst case it involves iterating through all points within the segment to find the furthest point. The latter however is dependent on the distribution of points in partitions determined by the extreme points chosen, where in the best case there is a balanced distribution of points, denoted by the recurrence relation T(n) = 2T(n/2) + O(n) whose solution is O(n log n). 2T(n/2) is representative of the recursive segmentation of points into two subproblems of n/2 size, and O(n) accounting for the time taken for finding extreme points. Conversely the worst case $0(n^2)$ occurs where points distributions in partitions are extremely unbalanced, occurring in circumstances where almost all points form part of the convex hull. In this case, each recursive call only eliminates or disregards one point, and each partition almost always generates an empty call to find_hull() - referring to cases where no points lie to the right of the defined baseline. Consequently, O(n) calls to find_hull are required, each of which requires an O(n) computation, on average, resulting in an O(n^2) complexity.

4 Limitations

As dicussed, the largest costs are attributed to distance computations and defining the points existing within a face. Several optimizations are implemented in quickhull to address this: (i) by selecting the furthest points we ensure that the initial facet envelopes a large portion of the convex hull, thereby reducing the amount of iterations required. (ii) Skipping computations of points guaranteed to exist within convex. Chan's algorithm [4] offers an intelligent approach to overcome the worst case O(n^2) limitations of quick hull by combining the Jarvis March and Graham scan convex hull algorithms [5]. Graham scan is used to calculate the convex hull of the subsets, having a complexity of O(n log n), then using Jarvis March to compute the merged convex hull, allowing it to achieve the optimal O(n logn n) solution. Whilst Jarvis March typically has a complexity of O(n^2), modifications involving the use of binary search to find single tangents allow it to achieve O(n/m log m) time, where log m represents search time for n/m instances.

Quickhull is primarily designed for computing convex hulls in 2-8 dimensions, though this can be extended at large expense, with convex hull algorithms having an exponential time complexity in higher dimensional space - O(n^d/2), with n referring to number of datapoints and d denoting dimensionality. A 16-d convex hull of 1000 points will have on the order of 1,000,000,000,000,000,000,000,000,000 facets [2], highlighting its computational infeasibility as dimensions grow. As such, study [6] addresses this limitation using a randomized approximation algorithm capable of: (i) treating memory complexity efficiently, (ii) precisely identifying points existing in the real convex hull, (iii) capable of being applied in high dimensions efficiently. This is notably done by providing efficient means to calculating distances from a point to the current convex hull in high dimensions by using approximated distances based on 2 × d (dimensions) vertices which are nearest neighbors to the newly found vertex.

5 Applications

Quickhull has many applications, some of which include robotics - for purposes of computer vision, path planning, collision detection amongst many others. study [7] proposes an enhanced convex hull approach that enables human operators to accurately control industrial robots using hand gestures. Prior to the use of convex hull, retrieved web cam footage must undergo several preprocessing stages to ensure hand gestures are isolated from background noise and other obstruction using skin detection approaches. Quickhull can then be implemented to detect the finger count, where each finger tip is

considered to be a point in the convex hull and other regions in the hand can be disregarded. This information is then transmitted via serial communication to the robot controller, signaling specific actions to be performed. In comparison to well established machine learning techniques (i.e, K-NN), hand gesture representations are vastly limited. However this approach eliminates the requirement of large (labeled) datasets, which in itself can be expensive to acquire and train.

Research conducted [8] explores the use of Quick hull in distributed systems, with uses in distributed robotics, cloud computing, sensor networks and generally any distributed computing scenario where the dataset is evenly distributed among multiple machines. Quickhull proves to be an appropriate solution, greatly reducing costs of communication, as these costs scale only with the number of edges in the convex hull and not with the number of points under consideration. In this system a centralized driver receives local max/min points from each dataset, from which global max/min data points are computed and broadcasted back to sender machines. Returned points are then added to individual convex hulls, defining the manner in which communication occurs within the distributed system

References

[1] J. Fisher, "Visualizing the connection among convex hull, voronoi diagram and delaunay triangulation," in *37th Midwest instruction and computing symposium.* Citeseer, 2004.

[2] C. B. Barber and H. Huhdanpaa, "Qhull manual." [Online]. Available: http://www.qhull.org/html/index.htmwhen

[3] C. B. Barber, D. P. Dobkin, and H. Huhdanpaa, "The quickhull algorithm for convex hulls," *ACM Transactions on Mathematical Software (TOMS)*, vol. 22, no. 4, pp. 469–483, 1996.

[4] T. M. Chan, "Optimal output-sensitive convex hull algorithms in two and three dimensions," *Discrete & Computational Geometry*, vol. 16, no. 4, pp. 361–368, 1996.

[5] X. Kong, H. Everett, and G. Toussaint, "The graham scan triangulates simple polygons," *Pattern Recognition Letters*, vol. 11, no. 11, pp. 713–716, 1990.

[6] A. Ruano, H. R. Khosravani, and P. M. Ferreira, "A randomized approximation convex hull algorithm for high dimensions," *ifac-papersonline*, vol. 48, no. 10, pp. 123–128, 2015.

[7] S. Ganapathyraju, "Hand gesture recognition using convexity hull defects to control an industrial robot," in *2013 3rd International Conference on Instrumentation Control and Automation (ICA)*, 2013, pp. 63–67.

[8] J. Ramesh and S. Suresha, "Convex hull-parallel and distributed algorithms," Technical Report, Standford University, Tech. Rep.