Algorithms That Changed the World:

Quickhull

Candidate Number: 239106
Student Number: 700018351
Word Count: 1490

December 8, 2022

Abstract

The quickhull algorithm is used to find the convex hull of a set of points in 2 or 3 dimensional space. This report will explore the principles of the algorithm, how to implement the algorithm, the complexity of algorithm, possible limitations and constraints and finally the applications of the algorithm in the real world.

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

Signature: awalliss

1 Algorithm Principles

A set of points S is described as a convex hull if, $S \subseteq P$, where P is the set of all points on a graph and no other two points in P can be on a straight line segment that intersects a segment formed by the contiguous points in S. The quickhull algorithm is used to calculate the points on a convex hull given a set of coordinates. It takes heavy inspiration from the sorting algorithm quicksort having a divide and conquer nature. The algorithm begins by declaring a set of points and an empty set for the points on the convex hull. The algorithm then finds the two coordinates in the set of points, A and B, that have the largest and smallest X-coordinate. After this, two new sets of points are formed using the segment (A,B) to divide them into there separate sets. The initialisation steps are then complete and the recursive steps of the algorithm begin. Two function calls are made to the recursive function passing in the parameters: A and B, and also one of the sets of points either side of the segment. The first step in the recursive function is to check if set is empty or only contains one element; in this scenario the element will be added to the convex hull and the function will return. If the set has more than one point, the point with the greatest perpendicular distance from the segment will be selected. This point will be removed form the original set and added to the convex hull. After this, the algorithm creates two more sets, formed from the points outside of the triangle (A, B, furthestPoint). One set of points will be on the outside of the line (A, furthestPoint) and the other set will be on the outside of the line (B, furthestPoint). The recursive function is then called again with the new set of points and the line segment(either (A, furthestPoint) or (B, furthestPoint)). The algorithm will continue recursively until all the points on the convex hull have been added to the set.

2 Pesudocode

Algorithm 1 quickhull()

```
1: let points be the list of points that we want to find the convex hull of
2: let hull be an empty list of points on the convex hull
3: minPoint ← point with smallest x-coordinate
4: maxPoint ← point with largest x-coordinate
5: add(hull, minPoint)
6: add(hull, maxPoint)
7: remove(points, minPoint)
8: remove(points, maxPoint)
9: let right be the list of points that are to the right of the segment created by (minPoint, maxPoint)
10: let left be the list of points that are to the left of the segment created by (minPoint, maxPoint)
11: findHull(minPoint, maxPoint, right, hull)
12: findHull(maxPoint, minPoint, left, hull)
13: return hull
```

Algorithm 2 findHull(a, b, set, hull)

```
1: insertPosition ← index of b in hull
2: if length(set) == 0 then
3:     return
4: else if length(set) == 1 then
5:     add(hull, sets[0])
6:     return
7: end if
8: furthestPoint ← the furthest point in the set from segment (a, b)
9: remove(set, furthestPoint)
10: insert(hull, insertPosition, furthestPoint)
11: leftSetAP ← set of points on the side of line (a, furthestPoint) which does not contain b
12: leftSetBP ← set of points on the side of line (b, furthestPoint) which does not contain a
13: findHull(a, furthestPoint, leftSetAP, hull)
14: findHull(furthestPoint, b, leftSetPB, hull)
```

3 Complexity Analysis

3.1 Time Complexity:

3.1.1 Best Case:

The best case time complexity is:

$$O(nlogn)$$

The reason for this is that the first 8 lines of **Algorithm 1** where the the segment points are selected and added to the convex hull can be computed in $O(n)$. The part of the algorithm where all the points are partitioned into the two sets also be computed in $O(n)$ as the program will simply loop through all points and add them to one of the sets. The recurrence relation for the best case in a divide and conquer algorithm is $2T(n/2)$ as the number of points for the function to process will have halfed however the function will also need to be run twice, once for both halves. Lines 1 to 10 in **Algorithm 2** can be computed in $O(n)$ due the most expensive operations involving looping through the list such as removing items and finding the index of items. The partition in **Algorithm 2** can also be computed in $O(n)$, the same as in **Algorithm 1**. Therefore the final best case recurrence relation of the

algorithm can be written as

$$T(n) = 2T(\frac{n}{2}) + n$$

From here, we can prove this is equivalent to $O(nlogn)$ as follows:

For recursion depth 1:

$$T(n) = 2T(\frac{n}{2}) + n$$

For recursion depth 2:

$$T(\frac{n}{2}) = 2T(\frac{n}{2^2}) + \frac{n}{2}$$

$$T(n) = 2(2T(\frac{n}{2^2}) + \frac{n}{2}) + n$$

$$T(n) = 2^2T(\frac{n}{2^2}) + 2n$$

For recursion depth 3:

$$T(\frac{n}{2^2}) = 2T(\frac{n}{2^3}) + \frac{n}{2^2}$$

$$T(n) = 2^2(2T(\frac{n}{2^3}) + \frac{n}{n^2}) + 2n$$

$$T(n) = 2^3T(\frac{n}{2^3}) + 3n$$

For recursion depth k (base case):

$$T(n) = 2^kT(\frac{2}{2^k}) + kn$$

Assume:

$$T(\frac{n}{2^k}) = T(1)$$

$$\therefore \frac{n}{2^k} = 1$$

$$\therefore n = 2^k$$

$$\therefore k = logn$$

Final proof:

$$T(n) = 2^kT(1) + kn$$

$$T(n) = 2^k * 1 + kn$$

$$T(n) = n * 1 + kn$$

$$T(n) = n + nlogn \approx nlogn$$

$$O(n) = nlogn$$

3.1.2 Worse Case:

The worst case time complexity is:

$$O(n^2)$$

The reason for this is the same as best case other than during the recursive step. The recursive relation will be $T(n-1)$ as all points in the set other than the furthest point will be added to the same set in the partition. Therefore the final recurrence relation is:

$$T(n) = T(n-1) + n$$

From here, we can prove this is equivalent to $O(n^2)$ as follows:

For recursion depth 1:

$$T(n) = T(n-1) + n$$

For recursion depth 2:

$$T(n-1) = T((n-1)-1) + (n-1)$$

$$T(n-1) = T(n-2) + n - 1$$

$$T(n) = T(n-2) + (n-1) + n$$

For recursion depth 3:

$$T(n-2) = T((n-2)-1) + (n-2)$$

$$T(n-2) = T(n-3) + n - 2$$

$$T(n) = T(n-3) + (n-2) + (n-1) + n$$

For recursion depth k:

$$T(n) = T(n-(k)) + (n-(k-1)) + (n-(k-2)) + ... + (n-1) + n$$

Assume

$$n - k = 0$$

$$\therefore n = k$$

Final Proof:

$$T(n) = T(n-(n)) + (n-(n-1)) + (n-(n-2)) + ... + (n-1) + n$$

$$T(n) = T(n-n) + (n-n+1) + (n-n+2) + ... + (n-1) + n$$

$$T(n) = T(0) + (1) + (2) + 3 + ... + (n-1) + n$$

$$T(n) = 1 + \frac{n(n+1)}{2}$$

$$T(n) = 1 + \frac{n^2+1}{2}$$

$$T(n) = 1.5 + \frac{n^2}{2}$$

$$\approx O(n^2)$$

3.2 Space Complexity:

The space complexity for all cases of the algorithm is:

$$O(n)$$

The reason for this is that the algorithm is always required to store all points from the original set. For example, in **Algorithm 1** there is an instance where both points on the segment are being stored and all other points either side of the segment are stored in one of two other sets. For the auxiliary space in the worse case scenario, the complexity is also $O(n)$ due to the maximum recursion depth being the same number as that of the points. This is because each level of recursion is being called with one less point until only one coordinate is passed to the function. At this point, the recursion will collapse and the algorithm will complete.

4 Limitations and constraints:

4.1 Worse Case Time Complexity

One clear limitation of the algorithm is that the worst case scenario is $O(n^2)$. The worse case scenario is that all points in the original set are on on the convex hull. When running the algorithm with a large set of points, the time required for computation will be infeasible. If a problem using a convex hull algorithm has a tendency to have a large proportion of points on the convex hull, it may be beneficial to choose a different convex hull algorithm with a better worse case scenario. Algorithms such as Grahams Scan have a better worse case scenario with $O(nlog(n))$ for all cases.

4.2 Stack Overflow

Due to the recursive nature of the program, the algorithm is susceptible to stack overflow for very large data sets with nonoptimal solutions leading to many recursive calls. Depending on the size of the stack on the hardware running the program, large data sets will lead to the program crashing and exceeding the hardware stack size. For problems of this nature, it may be easier to use a different convex hull algorithm with an iterative approach such as, Graham Scan or Gift Wrapping. Iterative approaches are also much more efficient than recursive approaches due to them not having to switch between different function calls.

5 Applications

5.1 Collision Prevension

One use of quickhull and other convex hull algorithms is object modelling in virtual environments [1]. It is difficult to accurately model real world objects in 2 or 3 dimensional space due to the fine details in their design. The solution to this is to wrap the surfaces of models in a convex hull reducing the detail and computation power required whilst still retaining the general shape of the object. This results in the detailed model being fully encapsulated by the convex hull. This means that when two objects are trying to avoid each other in virtual environments such as those used by self driving cars, they are only required to avoid the convex hull of the object and do not have to calculate whether they are likely to collide with the fine details of a model. Figure 1 highlights the effect of reducing details of virtual objects using quickhull.

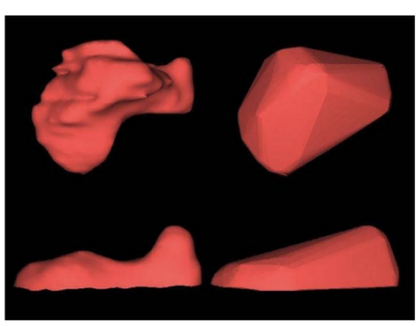

Figure 1: An example of model simplification using convex hull algorithms [1]

5.2 Onion Peeling

Onion peeling is the process of iteratively removing the outer points from a set of data. To do this, programs find the points on the convex hull of a data set and remove them. The process is repeated until there are no points left within the convex hull and all points belong to a defined layer. One use of an onion peeling algorithm is outlier detection [2]. Many outlier detection methods already exist however onion peeling methods can be much simpler and more efficient. The principle behind the method is to use onion peeling to find the centre of data and then use a distance method to measure how far all the other points are from the centre. The points further than a defined distance can be classified as outliers and discarded from the data set.

References

[1] C. Fares, "Convex envelope generation using a mix of gift wrap and quickhull algorithms," 2012.

[2] A. Harsh, J. E. Ball, and P. Wei, "Onion-peeling outlier detection in 2-d data sets," *arXiv preprint arXiv:1803.04964*, 2018.