Variations Of Quick Hull And Their Effects On Time Complexities

Ioannis Nearchou, Jonah Kubath, Western Michigan University

Abstract—Convex hull is a general technique for finding external points in **R**^{2} and **R**^{3} graphs and drawing a polygon that surrounds all points in such graphs, determined by the orientation of the external points of the graphs. One variation of it is known as QuickHull, which uses a quicksort-like algorithm for computing convex hulls. This paper explores whether it could be optimized in similar ways to quicksort, to create stable partitions of subproblems in execution. Three techniques are applied: randomized pivot choice, locating the partition point based on a modification to median-of-medians, and the use of two pivots in a partition execution. Each implementation is timed and averaged over 5 runs, performing hull computations on randomized n x n array representations for varying $10 \leq n \leq 500$ at steps of 10. At n = 10, computations are printed to standard output to evaluate the effectiveness of the application of each partition adapted to compute hulls. Collected times are compared to theoretical calculations over a worst-case performance of O(n^2).

Keywords—hull, partition, pivot, subproblem

1 Introduction

Convex hull is the general name of a technique for drawing polygons around external points in an **R**^{2} or **R**^{3} graph to encapsulate the points that exist within the space of the graph. This is useful for pattern recognition in graphics or for solving other geometric problems, according to [1]. There are several different implementations of convex hull in different spaces, including an algorithm that sorts adjacent points to a point on the forming hull and compares them to identify the point with the smallest y-coordinate value known as Graham's Scan. Another implementation is based on the standard quicksort algorithm, known as QuickHull [2]. QuickHull, due to its similarity to the standard quicksort, uses recursion to break the main problem of an array of points representing a graph into subarrays. Using a partition, the external graph points are found and added to the resulting hull. It is known that in [2] the partition of the subproblem determines the overall performance of a quicksort and so does the same for QuickHull. If not partitions properly, then for a problem with n number of points on a graph, the overall breakdown of the array is by one element at a time, such that n-1 breakdowns occur for the n sized array. Thus, the worst case asymptotic performance of a quicksort and QuickHull can be O(n^2). If a good partition is chosen, such that it lies somewhere in the middle of the array, then the sub arrays for finding hull points are roughly even in size with each recursive call and the average performance would be of O(n log(n)).

Algorithms for quicksort have been designed to reduce the performance time of solving subproblems by making stable partitions of sub arrays to find hull points. Several are known in [2], including RQuickSort, which chooses a random element for comparing with other elements and swapping based on values that order them in nondecreasing order, known as a pivot. Another algorithm called median-of-medians, finds the element of the most median value and uses its index as a partition [2]. There is another technique discussed in [5], which is a nonrandomized version of RQuickSort, which uses a partition that has two pivots used for ordering comparisons. This paper explores implementing QuickHull, analyzing if adaptations these partitioning techniques could be adapted to QuickHull's partition and affect its performance such that its time complexity behaves in the average case. Development of QuickHull is performed in the Java language and uses the principle of object-oriented programming to create implementations with variations on the three above techniques for performing QuickHull and designed with methods for handling subtasks defined in [2], [4], [5]. Each variation is timed over an average of 5 runs on randomized graph points, for 2-D plane graphs of varied row/column sizes n from 10 to 500 in steps of 10, to study performance behavior at lower and middle data size ranges. Resulting times are compared to the asymptotic average QuickHull performance of O(n log(n)), to measure the effectiveness of the adaptations. Results of the comparisons are discussed in the results section of the paper.

2 Various Methods

2.1 QuickHull

QuickHull operates according to [2] by finding the extreme lowest and highest x-coordinate valued points in a given graph and creates a line segment from them. With the graph now separated into two halves, arrays are created to hold points that lie outside either side of the line segment. These arrays both bookended by the endpoints of

- *I. Nearchou is a student under the Department of Computer Science, Western Michigan University, Kalamazoo, MI 49008. E-mail: ioannis.n.nearchou@wmich.edu*

- *J. Kubath is a student under the Department of Computer Science, Western Michgan University, Kalamazoo, MI 49008. E-mail: jonah.kubath@wmich.edu.*

this line segment. Breaking down these arrays into subproblem arrays, each are partitioned by a point that forms a triangle with the line segment previously mentioned, such that the triangle has a maximum area over all triangles formed by the other external points in that subproblem. When the partitions are found, those points are included to the set of points that make up the hull of the input graph. Those partitions break their respective arrays into sub arrays, where each sub array is a grouping of external points to the line segments formed by the endpoints of the arrays to be divided, with the partition points. At each recursive step, the sub arrays are broken down until they consist only of two points, which form a line segment. Upon the return to the main QuickHull method, an array that contains all previously calculated hull points from each subproblem is returned.

There are possibly conflicts in this method if there are multiple points that are considered leftmost and rightmost in x-coordinate in value. In this case, [2] suggests breaking ties by choosing two points from each list of extreme points, finding those leftmost points with lowest and highest y-coordinate values, and doing the same for the list of rightmost points. The leftmost and rightmost points of lowest y-coordinate value then form a line segment and the other two points for another. Then hull is recursively called on the points left of the first line segment and points to the right of the second line segment. To find the orientation of points to these line segments, a method is used to calculate the determinant of a matrix made of the line segment points and the point to compare. The details of determinant will be discussed later. All other points are not considered because they are considered interior points to the hull to be computed. The pseudocode for QuickHull given by descriptions in [2] are given below:

```
QuickHull(X){

    Find two points, one p1 with leftmost x-
    coordinate and the second p2 with the rightmost
    x-coordinate

    if more than one extreme point on either
    side of the graph exists then

        choose points with lowest and
        highest y-coordinates from each
        list of extremes. For the leftmost x-
        coordinate list, call these points p1 and
        p11. For the right list, call them 2 and
        p22.

        create an array x1 for all points to the left
        of the line segment formed by p1 and p2,
        including those points.

        create an array x2 for all points to the
        right of the line segment formed by p11
        and p22, including those points.

        Hull(0, length(x1) - 1, x1);
        Hull(0, length(x2) -1, x2);
    Else

        create array x1 of points to the left of the
        line segment p1 to p2, including those
        points.

        create array x2 containing points to the
        right of segment p1 to p2, including
        those points.

        Hull(0, length(x1) - 1, x1);
        Hull(0, length(x2) -1, x2);
    }

    Return computed hull from recursive hull steps
}
```

Here, the amount of time taken by QuickHull is determined by the two Hull calls, the time taken to find the extreme points of the graph and the time to break down the graph into subgraphs with external points to the line segment(s) formed by the extreme points. Let n be the number of points in X. To find the extreme points, it will take time looking through all m elements to determine the extremes. For ties, a little more complexity will be taken to resolve them, but no more than an extra O(n) maximum, in case if all points are aligned to the extreme right or left of the graph. Another O(n) is taken to search the input array a again to compare each point to the extreme points to decide whether they are to the left, right or colinear with the line segments formed by the extreme points. Then, because hull is based on a recursive quicksort, a recurrence relation can be used to describe it. Such a recurrence relation is:

$$T(m) = T(m1) +T(m2) + O(m),$$

as given in [2], where m is the number of points in a given subarray to be processed, m1 is the number of points in the first subproblem of Hull and m2 being the number of points in the second subproblem of Hull. The extra O(m) is for finding points of division for the subproblems. Recalling the discussion of quicksort in the introduction, if the partitioning of subproblems is at one side or the other, then the T(m1) + T(m2) expression becomes equivalent to roughly T(m-1). This implies that m searches are performed m-1 times, yielding a performance $O(m^2)$. Then if this occurs with the initial n elements in QuickHull, the total complexity would be $O(n^2)$. If the partitioning is even though, as with the quicksort average case, solving the recurrence gives O(n log(n)) for n input points, because each subproblem is broken evenly.

2.2 Hull

According to [2] QuickHull requires recursive calls of a helper function Hull, which subdivide its incoming array of graph points into sub arrays to divide the problem into subproblems and then compute hull points. To perform the division, much like in quicksort, a partition is chosen to break the array into halves, of sizes based on the number of external points to the line segments formed by breaking the current line segment being analyzed to two segments using the partition as an endpoint, as previously discussed. Hull is recursively called upon itself until the subproblem passed to it contains only the two endpoints that consist the line segment to be analyzed. When this base case occurs, no partitioning is performed, no hull point is added, and the method ends. The pseudocode for Hull defined in [2] is as follows:

```
Hull(X, p, q){

        Find partition point of the array X

        Add the partition point to the structure
        maintaining calculated hull points

        call Hull on points to left of line segment of p
        and partition point

        call Hull on points to left of line segment of
        partition point and q
}
```

Here, if the size of X is m elements, the partition call will be O(m) because all but two elements (the endpoints of the line segment of X) will be compared to find a partition. Then the assignment of the partition element to the internally calculated hull can be considered O(1). The heart of the time complexity are the recursive calls of Hull, based on the partition result. A described earlier, if the partition is inefficient, the total complexity of recursive calls from the top level with n elements will asymptotically be $O(n^2)$ at worst-case. If evenly partitioned the subproblems solved by the recursive calls will be of order O(n log(n)).

2.3 Partition

Partition is a method that generally takes an array for input, and two indices that are the beginning and end of the array. In general, the partition finds an element to split he array into parts and returns the integer index value of that element to the caller. In quick sort, during the partition step, if its partition is determined to be at the beginning or end of its subarray, then the subproblems for sorting the values in the array are divided into one subproblem size of 1, being the partition, and the other subproblem containing the rest of the elements in the subarray [2]. For a problem of asymptotic n number of values in that subarray, if this kind of partitioning continues, then n-1 steps of breaking down the subproblem will be taken. The resulting asymptotic worst-case performance is then $O(n^2)$. Contrarily, if the partitioning of the sub array is roughly even with each step, then average case asymptotic performance of O(n log(n)) is encountered, since the sub arrays are approximately broken in half with each step until the base case of subproblem array size is reached. Similarly, the average and worst-case asymptotic performances of QuickHull are defined the same, since the solution is based on this technique. The pseudocode for QuickHull partition is described in [2] is as follows:

```
partition(X, p1, p2){

        Find point p in X where the absolute triangle
        formed by p,p1 and p2 is maximum

        if a there is a tie for point p
                Find point with largest angle to p1,p2
                And make that point p

        return index of p;

}
```

The index returned is at the behest of X's point ordering. If the partition is at any end, then the worst-case of breaking down the subarrays of this array will occur, just as in quicksort. Due to searching the entire array for a partition, if the number in the points in the array is n, the time complexity will be O(n).

Now, several optimizations are known to exist for attempting to improve the chances of stable partitioning with each subproblem breakdown in quicksort. One optimization in [2] utilizes the random picking of an element in the array to be sorted and swap it with the beginning element, such that in the partition algorithm the swapped element is chosen as the pivot by which all other elements are compared and sorted in order. This element is chosen as the partition when swapped at the end of the algorithm. The performances of this RQuickSort variant are the same as in the basic quicksort case, but for different reasons. The average case performance is based on the probability space of choosing a random element that will have a value that is roughly the median of the other values, whereas the standard quicksort average case is based on the value of the chosen pivot, In the beginning of the array. RQuickSort in [2] assumes array elements contain comparable values, such as numerical values. Since each element in the array that aren't line segment points must have their areas with the line segment points compared, the conditional statements are somewhat different. Implementation of this adaptation will be given later in the implementations section.

Another optimization discussed for finding pivots is median-of-medians [2]. The original description of this algorithm requires the selection of a value for a term r, which is used to subdivide an array of size n to n/r groups. The elements with median comparable values are chosen from the groups and the median of those medians is taken as the partition, such that the successive breakdown of the array into subarray subproblems is expected to be even. Due to a maximum triangular area being required for the partition element, this technique will not find the right element. Instead it can be adapted to work by finding the elements with maximum triangular areas within the subgroup. This adaptation is made, and its implementation will be discussed in the next section.

Another method considered for partition is choosing two elements as pivots, as in [5], when implementing for quicksort. Here the algorithm in [5] is adapted to the context of this partition specification is as follows:

```
partition(X, m, p){

    if X[p] is less than X[m] then
        interchange(X, m, p);
    lt := m+1; gt:= p+1; i:= m+1
    while i ≤ gt do
        if X[i] < X[m]
            interchange(X, i++, lt++);
        else if X[p] < X[i]
            interchange(X, i, gt--);
        else then i++;
    interchange(X, m, --lt);
    interchange(X, p, ++gt);

    return i;
}
```

Even with modifications to fit the specification, this algorithm isn’t appropriate for QuickHull. The conditionals must compare points by their triangular areas and the pivots cannot swap with the endpoints at m and p, because these are the endpoints that consist of the problem's line segment. This method will be significantly adapted in its algorithm for application to problem at hand and that adapted algorithm will be given in the next section. Without loss of comparison for complexities, the adaptations will not include loops more complicated than this, so even if multiple instances of loops occur, because they are singular loops, the additive complexities will still be of order O(n) for the case of n entries in X.

Other partition schemes in quicksort that could be hypothesized for adaptation to QuickHull are described in [6]. These include Lomuto's partition, which takes the last element of X as the pivot and uses indices i and j to compare elements that are smaller in value to the pivot and swaps when i isn't equal to j [6]. This could have been chosen, over one of the previously discussed three, but was not chosen over assessments on its stability of partitioning arrays. No matter the partition though, conflicts may arise again when the partition is chosen. There could be ties for greatest triangular values for possible partition points. Traditionally, as described in [2], the point with the greatest angle to the line segment is chosen. How angles are determined is covered next in the discussion of the determinant. Another tie breaking method is by measuring point distance to the line, as in [3]. The furthest point is taken as the partition. For this paper, distance via [3] will be used as the tie breaker, but both methods have issues in point resolution, when two points are very close to each other and the result of the calculation is an integer. Taking 3 points p_1 with coordinates (x_1, y_1), p_2 with (x_2, y_2) and p with (x_3, y_3), [3] calculates the distance as follows:

$$\text{Dist} = ((y_3 - y_1) * (x_2 - x_1)) - ((y_2 - y_1) * (x_3 - x_1)).$$

These complications will be discussed later in the results and error summary.

2.4 Determinant

The textbook [2] gives a geometric prior used for determining hull points that are used for partitioning subproblems. One can note that from [2], three points, call them p1, p2 and p, can have their x and y coordinates applied in a 3x3 matrix and the determinant can be calculated. The resulting number calculated can allude to the degree of the angle formed between one of these points to the other two. A negative determinant for the orientation p1p2p means the angle between p and the line segment of p1 and p2 is less than or equal to 180°, with 0° being colinear to p1, p2. Such a result means p is a right turn from p1,p2 [2]. If the determinant for that orientation is positive, it means the angle of p is a left turn from the segment and the angle is greater than or equal to 180 degrees. The general determinant algorithm is taken from [3] and [4] such that it is as follows:

```
Determinant(p, q, r){

    Create 3x3 array called a.
    Fill first column with x-coordinates of p,q,r
    Fill second column with y-coordinates of p,q,r
    Fill third column with 1's

    Return (a[0,0]*((a[1,1]*a[2,2]) - (a[2,1]*a[1,2]))) -
            (a[0,1]*((a[1,0]*a[2,2]) - (a[2,0]*a[1,2]))) +
            (a[0,2]*((a[1,0]*a[2,1])-(a[2,0]*a[1,1])));
}
```

Here the operations in the return statement calculate the determinant. For a matrix {{a,d,1}, {b,e,1}, {c,f,1}} the determinant is calculated a*(e-f) - d*(a-c) + (b*f - c*e). This is equivalently done in the return statement for points p, q and r. Because this method is a matter of assignments statements and a few operations, the overall time complexity is O(1). It can be further noted that the absolute area of the triangle formed by p, q and r can be found by taking the absolute value of their determinant and dividing it by 2. This is useful for finding the partition point in partition().

2.5 Interchange

The partition() implementations for quicksort in [2] and [6], require swapping elements at certain positions to orient them in a defined ordering scheme. To perform these swaps, the interchange method is used. It takes an array a and two integers i and j for arguments. An algorithm in [2] is applied:

```
Interchange(a, i, j){
    p := a[i];
    a[i] := a[j]; a[j] := p;
}
```

Here, the elements in a at i and j are swapped. Because there are only three assignment operations, the time-complexity is expected to be O(1) for asymptotic worst-case.

3 Design

To implement these methods by their variations and adaptations to QuickHull, three Java classes are designed to encapsulate the functionality of a QuickHull computation with the three designated partition strategies of interest. These classes are called RHullCreator, MMHullCreator and DPHullCreator. The methods of these classes are called within a main Application class, which creates random graphs and variables to hold times before and after QuickHull executions.
The actual layout of these classes and their algorithms/pseudocode will be given in the next section. As a note, since the actual order of the elements in an array (beyond the points that represent a line segment) does not affect the sizes of the partitioned subproblems, no interchange is applied in MMHullCreator, so the method does not exist in this class.

4 Implementations

4.1 RHullCreator

There are three written classes which contain implementations of the named methods in the previous section, based on each partition technique. RHullCreator is the Java class that contains the method implementations related to the pivot randomization technique. It contains three attributes on top of these methods: hullList, subListLeft and subListRight, along with setter and getter methods used change or retrieve the attribute values while limiting their visibility to the level of the class itself. The hullList attribute is a dynamic list used to internally store computed hull points with each partition call within a call of this class' Hull implementation, which will be used to create the array of computed hull points to be returned in the quickHull() method. The subListLeft attribute is a dynamic list used for keeping track of points to the left of the first new line segment formed after partitioning in Hull, while subListRight is used to keep track of points to the left of the second new line segment formed after partitioning.

4.1.1 quickHull

This quickHull implementation takes one parameter, being a 2-D integer array a, representing the list of 2-tuples that are plotted points on a 2-D n x n graph. It returns a reference to a 2-D integer array, representing the calculated hull of a. If the parameter references a null value or points to an array of size less than 3, then the parameter reference is immediately returned. If such a condition is not violated, the quickHull class performs subroutines and assignments congruent to performing the steps reported in the pseudocode for QuickHull given in the various methods section. At the end of this algorithm, after the return of the second Hull, before the hull is returned, this method creates a 2-D integer array locally to the size of hullList and fills the array with all points in hullList. The return statement in QuickHull refers to the return of the array of points in hullList. Given that the implementation's time-complexity relies on the recursive local hull calls, the time-complexity is expected to be O(n^2) at asymptotic worst-case for n points. The average complexity is expected to be O(n log(n)).

4.1.2 rHull

rHull is the implementation of hull() given in the various methods section, related to the technique of choosing a random pivot for the partition step. The algorithm implemented is based on the RQuickSort algorithm given in [2] and is as follows:

```
rHull(p, q, a){

        if q - p + 1 > 2 then
                if q -p > 5 then
                interchange(a, Random(q -p) +2, p+1);
                hullPointIndex := partition(a, p, q);
                add element a[hullPointIndex] to
                internal hull list
                create arrays x1 and x2 to hold the
                points in the two internal subproblem
                lists
                fill each array by the lists of their
                respective subproblems.
                rHull(0, length(x1) - 1, x1)
                rHull(0, length(x2) - 1, x2)
    }

}
```

Here, x1 and x2 are the arrays representing the set of points to the left of the line segment from the element at p in array a and the point at hullPointIndex in a and the points to the left of the line segment between a[hullPointIndex] and element a[q] respectively. Each array contains their related line segment's endpoints as elements too. These arrays are created because there may be other points in array a that are internal two these line segments. These points are dropped from consideration in recursive rHull calls because the algorithm picks outermost graph points for its class' hull. Thus, rHull is called recursively on x1 and x2 to find hull points that are external to the line segments related to each array.

4.1.3 partition

This RHullCreator partition is adapted from the traditional partition given in [2] and is as follows:

```
partition(a, m, p){
        v:= a[m+1]; i := m; j := p;

        find determinant and area of v to a[m] and a[p]
        compute distance of v to line segment m-p

        for k := m + 1 to  p - 1{
                if area of k greater than current partition
```

```
                    j := k
                    set partition determinant, area
                    and distance to that of k
          else if area of k is equal to area of current
          partition
                    if distance of k is greater than
                    distance of current partition
                              j := k
                              set partition
                              determinant, area
                              and distance to that of
                              k
     }

     Set internal lists for creating subarrays
     partitioned by j to new empty lists

     part := j;
     v := a[j];

     add a[m] to subListLeft
     add v to subListRight
     j := p; //reset j to end
     while i < j do
          i := i + 1;
          while determinant(a[m], v, a[i]) > 0
                    and i < p) do
                    add a[i] to sublistLeft
                    i := i + 1;
          j := j -1;
          while determinant(v, a[m], a[j]) > 0
                    and j > m) do
                    if a[j] is not in subListLeft
                              add a[j] to sublistRight
                    j := j - 1;

     add v to subListLeft
     add a[p] to subListRight
     return part; //return proper partition


}
```

Although there are nested while loops, if any of the inner loops goes through all the array elements, the outer loop will exit after the following iteration. The loops could approximate then O(n) for n elements in a. Then, the opening for loop for finding the partition point is also O(n) for n elements. These can be generally O(m) since the size of these arrays get smaller with each recursive Hull call. Multiple O(n) loops add and approximate to an asymptotic worst-case of O(n) for n elements, which isn't worse than the asymptotic case of the quicksort partition it adapted. Unlike the quicksort version in [2], this adaptation does not have interchange calls in the while loops, but instead uses them to load lists for creating the subarrays that are partitioned by v. In practice, these calls created errors with elements in a, so they were removed. The loops are still meaningfully maintained for handling the subproblems.

4.1.4 determinant

The determinant method in this class is implemented identically to the determinant pseudocode given in the various methods section. Because of this, the expected asymptotic worst-case time-complexity is O(1).

4.1.5 Interchange

The interchange method for this class is implemented identically to the algorithm given in the various methods section. This method too can be considered O(1) at worst case, for the reasons described in the various methods section.

4.2 MMHullCreator

This class implements the defined functions in the various methods section, except interchange(), in context of a variation of the median-of-medians partition technique in [2]. The variation focuses on the maximum of all maximum triangle area values, for triangles formed by each other point in the array being partitioned, to the points that are the endpoints of its related line segment. This adaptation is applied because a partition point must be chosen such that it forms the largest triangle with the line segment. Due to this, the median triangular area value of all points cannot partition by an external point. Instead, the maximum is taken. The implementation changes of the partition method in this class, relative to the median-of-medians description in [2], are only related to the area comparison values to each point.

MMHullCreator contains the same attributes as RHullCreator, as well as an additional attribute r, which is used for subdividing arrays in the partition method. This attribute has related setter and getter methods, so the user of the program can set the divisor of the array at the main method level.

4.2.1 quickHull

The algorithm used in this quickHull() implementation is identical to that for the RHullCreator class. The only difference is that the recursive Hull calls are done with the local method mMHull(). Due to this, the expected performances are the same, varying from $O(n^2)$ to O(n log(n)). This is owed to the dependency on the number of points to the left of each line segment formed by the old line segment endpoints to the partition point.

4.2.2 mMHull

This Hull implementation is identical to the rHull implementation, but without a random swapping of elements prior to calling partition. Likewise, it takes a 2-D integer array a and integer indices and q as arguments. Subarrays are created, based on the partitioned points and recursive calls of mMHull() are made for the two subarrays created after the partition, only if the number of elements in a is greater than 2. Similarly to rHull, for asymptotic n elements, the worst-case complexity is expected to be $O(n^2)$, while the average case is expected to be O(n log(n)).

4.2.3 partition

This MMHullCreator partition is adapted from median-of-medians description given in [2] but adapted to find the point with maximum triangular area. is as follows:

```
partition(a, m, p){

    n := p-m +1;
    i := m +1;
    Initialize variable used for finding partition
    Create list to hold list of points with max area

    if n > r then

        numGroups := n /r;
        for j := 0 to numGroups
            if p - i < r
                find max point from i
                to p and add to list of
                max points
            else
                i := m + j*r;
                find max point from i
                to i + r and add to list
                of max points
    else
        find max point from m to p and add
        to list of max points

    find max point area in list of max points and set i
    to its array index

    initialize subListLeft with points left of segment
    from m to i

    initialize subListRight with points left of segment
    from i to p

    return i; //return proper partition


}
```

What is not expressed in detail in the pseudocode is how the maximum points are found. Those lines that express the action consist of the same for loop discussed in the RHullCreator partition, where each element in the array is analyzed for having the largest triangular area to the line segment m-p and ties are handled by taking the point with the greatest distance to the line. The first loop that iterates through subarray groups of the array a, will have an overall asymptotic performance of O(n) for n elements. This is because the combination of loops used in this block are structured to examine each element in the array. The step where the maximum point is found among the other maximums should be less than O(n) because this order implies that all points are superposed at the same location, which cannot occur.

The additions of points to the subproblem lists are grouped in one loop where the entire array a is examined for points left of the new line segments created by the partition point. Due to this, it is an additional complexity of O(n). Adding these complexities together over an asymptotically large n, the total time-complexity is approximately O(n) at worst case.

4.2.4 determinant

The determinant method in this class is implemented identically to the determinant pseudocode given in the various methods section. Because of this, the expected asymptotic worst-case time-complexity is O(1).

4.3 DPHullCreator

The class is related to an implementation of partition that uses a dual pivot algorithm adapted from [5]. The attributes and method specifications of this class are identical to those of the RHullCreator class.

4.3.1 quickHull

The algorithm used in this quickHull() implementation is identical to that for the RHullCreator class. The only difference is that the recursive Hull calls are done with the local method dPHull(). Due to this, the expected performances are the same, varying from $O(n^2)$ to O(n log(n)) due to being dependent on the number of points to the left of each line segment formed by the old line segment's endpoints to the partition point.

4.3.2 dPHull

This Hull implementation is identical to the mMHull. Likewise, it takes a 2-D integer array a and integer indices and q as arguments, creates subarrays based on the partitioned points and makes two recursive calls on itself for the two subarrays created after the partition, only if the number of elements in a is greater than 2. Similarly to rHull and mMHull, for asymptotic n elements, the worst-case complexity is expected to be $O(n^2)$, while the average case is expected to be O(n log(n)).

4.3.3 partition

This DPHullCreator partition is adapted from the dual pivot quicksort in [5], described in the various methods section. The pseudocode of the adaptation is as follows:

```
partition(a, m, p){

    leftPiv := a[m +1]; rightPiv := a[p-1];
    left := m +1; right := p-1; i := left;

    get determinant and area of leftPiv to line
    segment m-p

    get determinant and area of rightPiv to line
    segment m-p

    while i ≤ right do
        partArea := determinant(a[m], a[p], a[i]);
        if partArea < leftPiv's area
            interchange(a, i, left);
            set leftPiv's area to partArea
            left++; i++;
        else If partArea < rightPiv's area
            interchange(a, i, right);
            set rightPiv's area to partArea
            right--;
            i++;
        else i++;

    check if i is maximum point and find maximum
```

```
        if not
        set i to index of maximum point

        add a[m] to subListLeft
        add a[i] to subListRight
        for j := m+1 to p-1
                if determinant(a[m], a[i], a[j]) > 0
                        add a[j] to subListLeft
                else if determinant(a[i], a[p], a[j]) > 0
                        add a[j] to subListRight
        add a[i] to subListLeft
        add a[p] to subListRight

        return i; //return proper partition


}
```

The exchanges of elements in the first while loop is used to order the array elements such that the element(s) with the largest triangular area to the line segment of m-p would be found toward the right end of the array. Due to this, i cannot by chosen to be the pivot. The choice is then given a second O(n) check to make sure it is the correct partition point, with the same tie breaking techniques as used in the other partitions. The last for loop is the same as the one described in MMHullCreator's partition, where external points of the subproblems are added to lists that will be used to construct the arrays that become the subproblems in the recursive dPHull() calls. Due to these combined O(n) loops, the overall total time-complexity will be O(n) at worst-case.

4.3.4 determinant

The determinant method in this class is implemented identically to the determinant pseudocode given in the various methods section. Because of this, the expected asymptotic worst-case time-complexity is O(1).

4.3.5 Interchange

The interchange method for this class is implemented identically to the algorithm given in the various methods section. This method too can be considered O(1) at worst case, for the reasons described in the various methods section.

4.4 Application

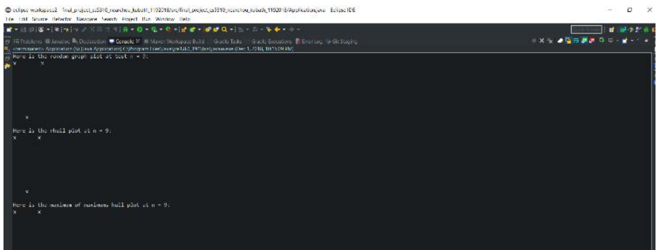
The Application class is the main class of the project. It contains the main method, which executes the tasks for preparing data structures to be used in the hull computation runs, the loops for averaging and varying runs, printing run results, timing the complexities of the runs of the quickHull() method calls and printing the resulting measured times to an output file. The tasks are wrapped in a try-catch block that handles thrown exceptions if there is an error when accessing the output file. Inside, variables are declared for referencing data structures and holding necessary values to the tasks of the project. Then, an if condition is encountered, executing code based on a boolean flag for throwing control to lines used in the project experiment if set to false, or throwing control to lines that execute a static example written for demonstrations and testing. Inside the if block, there is an outer for loop for varying array sizes n from 10 to 500 in steps of 10. Inside that loop, other loops exist. First, two nested for loops are used for generating random entries in an n x n array, which has its coordinate 2-tuple values entered into an array of points.

Following this nested loop is another for loop that loops for the number of runs set at the top of the main method. In this loop, the hull is calculated for the list of points with each instance of a quickHull() implementation. After that loop is a conditional statement that checks if the variable n = 10. If so, then a sequence of nested loops are executed, one for printing the original graph converted into a graph of characters where each point is an "x", one for printing the hull computed through the rHull() in quickHull(), one for printing the hull computed through mMHull() and the last for printing the hull computed through dPHull(). The else block of the outer conditional contains a statically created 9x9 array with three fixed points to form a triangle. The list of these three points will have its hull computed through the three quickHull() calls and just as with the above conditional statement checking for n = 10, a sequence of loops is encountered to print the hulls and graph of the else block.

5 Testing

Test data is constructed through a random number generator that, when applied to a nested for loop, one with index counter i and the other j, that goes through each element of an initialized n x n integer array, chooses randomly a 0 or 1 and enters those values into the current element at [i,j]. Simultaneously, when 1 is chosen, the point [i,j] in the array is entered into a list used to construct the input array that will have its hull computed. A temporary array is used to store this array, so array references passed into each quickHull() will reference a copy before they have their hulls computed.

Optionally, for demonstration purposes, the boolean test flag may be set to true, causing logic flow of the executing program to execute quickHull() calls for a static example graph. A 9x9 graph is created with points [0,0], [0, 8], [8,4]. These 3 points are supposed to form a large triangle and show the ability of the quickHull() calls to calculate the hull of the graph. Figure 1. below shows the resulting output of the quickHull() calls. The original graph is printed first, then the hulls are printed in the order of their discussion in the implementations section of this paper.


eclipse-workspace2 - final_project_cs5310_nearchou_kubath_11192018/src/final_project_cs5310_nearchou_kubath_11192018/Application.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help
Problems Javadoc Declaration Console Maven Workspace Build Gradle Tasks Gradle Executions Error Log Git Staging

```
Here is the random graph plot at test n = 9:
x       x




   x
Here is the rhull plot at n = 9:
x      x




   x
Here is the maximum of maximums hull plot at n = 9:
x      x
```

Figure 1. Test hull computations

6 Results

6.1 Runtime Results

The program analysis is performed on the described implementations of quicksort partition optimizations adapted to QuickHull, where n x n random graphs are built and the performances of these quickHull() calls are measured for increasing n. Given the estimated worst-case and average case costs, the scope of the analysis is output generated for a range of the value of n from 10 <= n <= 500 in steps of 10. As stated earlier, to get accurate empirical results, the elapsed times were accumulated for 5 executions of the loops that execute and time each quickHull(), then averaged over the five runs. This is because background tasks performed by the computer while running the program could affect the call times of each building function or routine, as well as how the algorithms are written. These empirical results will be compared to the theoretical calculations and the implications will be discussed. In Figure 2, the theoretical calculations and experimental measurements are plotted for the performance of the three quickHull() methods on n x n graphs of increasing n. The theoretical calculations are based on a run on n and are compared to the range of n*log(n) equivalent time operations, scaled by a normalization constant. This assumes the average case, due to the array randomization. The normalization constant is chosen to be $1x10^6/(2.5GHz)$ to give a theoretical instruction execution time in microseconds, based on the average clock speed of the CPU used on the testing machine.

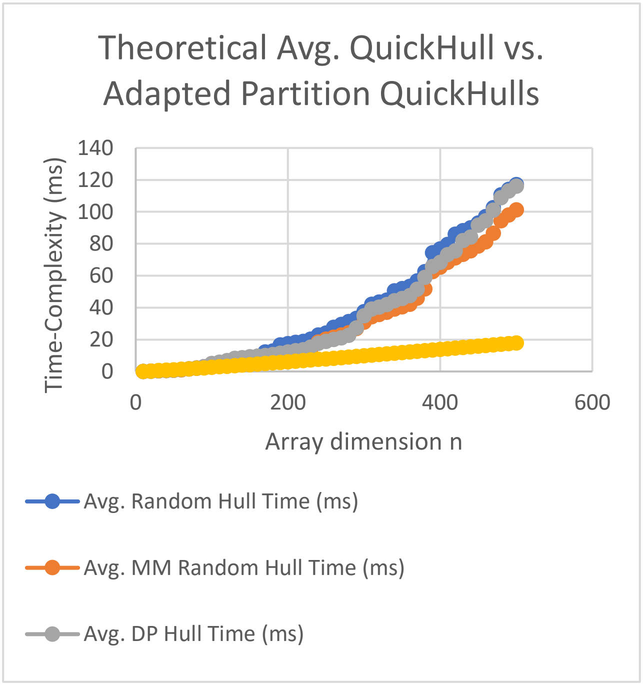


Figure 2: Theoretical v. Experimental QuickHull times

Here in the figure, we see some agreement with the expected Time-complexities to theoretical time-complexities until sometime before n = 200. At this moment, the growth curves of all the experimental time-complexities rapidly grow over the theoretical n*log(n) curve. In fact, the appearance of the curves agrees more toward $O(n^2)$ growth. Each experimental curve also appears to be comparable in growth, with dominations in the curves alternating between them due to background tasks or internal algorithm tasks adding to their time-complexities. Due to the significant divergence we conclude the adaptations did not improve the overall performance of QuickHull. The curves also happen to not be perfectly curved, with periodic decreases and increases in performance from n = 200 to n = 500. The bumps may be due to the algorithms written where conditions don't allow some lines to execute, giving minor decreases in time-complexity, or other costly executions to happen more often, causing local increases in time-complexity.

6.2 Sources of Error

Incompatibilities and general errors in execution may be traced to the level of adaptability possessed by each quicksort partition optimization to each implementation class or to the chosen methods for handling conflicts in partitioning compared to the variety that exist. One source of error noted was discovered during the debugging phase of the project. When ties were being resolved in a partition through determinants to compare angles, two points that were very close, but with one clearly the most external point, would have the same determinants calculated. Of course, this would be expected if their areas are the same. The cause of the tie breaking issue is due to using integer values for comparisons of points that may have a small degree of difference from each other, depending on the level of point clustering from random point generation in the main method. A solution to resolve this issue over determinants or distances from the line, is taking the Euclidean distance of the point to be compared to the left endpoint of the line segment used as a frame of reference. The Euclidean distance of that endpoint to the other one, as well as the external point to the other endpoint can be calculated to get the sides of the triangle they form together. Then these distances can be used in geometric operations to obtain the angle of formed with the external point, with double precision. This allows finer comparison of angles with slight differences to pick the proper partition point.

Another source of difficulty involves the adaptation of the partitions themselves. There is no actual need for ordering array elements, so the swapping loops in the standard quicksort's pivot and the dual pivot given in [5] are not useful. The adaptation of median-of-medians to be oriented toward maximum values still has use for finding the partition but does nothing to improve the balancing of partition because the size of the subproblems broken down after partitioning is dependent upon the number of elements that are points to the left of each new line segment from the partition step. The randomization technique is also not considered useful, because the partition cannot be randomly picked. The randomly chosen pivot also has no impact since ordering isn't itself necessary for finding hull partition points.

7 User Guide

7.1 Compilation

The program file requires any of the recent versions of Java to exist on the user's computer. With this requirement met, the program itself may be run by using an IDE like Eclipse or can be run from the console through Java. For an IDE, the user needs to import the project into their file explorer. Then, open the imported project folder at the src level and click on the application.java file. Once the class file appears in the main window of the IDE that shows files, click on the green triangle execute button on the top toolbar of the IDE screen. The program will compile, then run. If the user wishes to engage in a demonstration instead of calculating time-complexities, set the boolean test flag at the top of the main method to true before running.

In the case of command-line execution, enter the file path of the class file to execute and type run. The program will compile, then run according to the demonstration or timed runs depending on the current value of the test flag in the source code. To change the flag, go to the source code of the project file and replace "true" with "false" or vice versa.

7.2 Input

No user input is necessary to run the program. All data is randomly generated internally to the main method of the program.

References

[1] Convex Hull. Wikipedia contributors. (16 September 2018). Retrieved November 30, 2018, from https://en.wikipedia.org/wiki/Convex_hull

[2] Horowitz, E., Sahni, S., Rajasekeran, S., *Computer Algorithms, 2nd Edition*, Silicon Press, Summit, NJ 07901, 2008.

[3] GeeksForGeeks Contributors. (n.d.) - GeeksForGeeks. Retrieved November 30, 2018, from https://www.geeksforgeeks.org/quickhull-algorithm-convex-hull/

[4] Geometric Algorithms. João Comba. (n.d.). Retrieved November 30, 2018, from. cse.unl.edu/~ylu/raik283/notes/quickhull.ppt

[5] QuickDualPivot.java from §2.3 Quicksort. (Fri Oct 20 12:50:46 EDT 2017). Retrieved November 30, 2018, from https://algs4.cs.princeton.edu/23quicksort/QuickDualPivot.java.html

[6] Quicksort. Wikipedia contributors. (27 November 2018). Retrieved November 30, 2018, from https://en.wikipedia.org/wiki/Quicksort