ROBERT ALLEN SCOTT AWARD REPORT

Mathematics and Computer Science Department
Rhodes College

Center Regions and Colored Tverberg Points

Michael Siler[1]

Advisor: Dr. Ivaylo Ilinkin[1]

July 31, 2008

[1]{silwm, ilinkin}@rhodes.edu, Department of Mathematics and Computer Science, Rhodes College

1. Preface

The purpose of this summer research project was to learn about computational geometry, which is a branch of computer science that studies algorithms that deal with properties of configurations of objects in space. This purpose was achieved in two ways. The first was by studying the problem of finding the *center region* (defined below) of a set of points in the plane. I studied and applied several important concepts of computational geometry to develop the results presented below. However, I avoided the published results devoted to this problem, so that I could develop the results on my own. As such, the results presented here are original work, but not new.

Secondly, I studied a recent computational geometry publication and tried to improve one of its results on *Colored Tverberg points*. This problem is a specialized version of the center region problem that has not been well studied. This objective was achieved in that several ways of attacking this problem were examined, and some very promising lines of research were discovered.

The remainder of this paper will describe the problems I analyzed, some background information on computational geometry, and my work.

2. Introduction

Let S be a set of n points in $\mathbb{R}^d$. A point $x \in \mathbb{R}^d$ is said to have depth k in S if every closed halfspace containing x contains at least k points of S. Without loss of generality, only the closed halfspaces with x a member of the bounding hyperplane need be considered. The *depth-k* region of S is the set of all points of depth k in S and is denoted $c_k(S)$. In [1], it is shown that if $k \leq \lceil n/(d+1) \rceil$ then $c_k(S)$ is nonempty. Points with depth of at least $\lceil n/(d+1) \rceil$ are called *center points* and $c_{\lceil n/(d+1) \rceil}(S)$ is called the *center region*.

Let R, G, and B be disjoint "red", "green", and "blue" sets of n points in the plane, and let $S = R \cup G \cup B$. A *multicolored triangle* is a set of points $\{r, g, b\}$ where $r \in R$, $g \in G$, and $b \in B$, and a triangle is said to *cover* a point x if x is on the boundary or in the interior of the triangle. A point $x \in \mathbb{R}^2$ is called a *Colored Tverberg point* (CTP) if there exists a pairwise disjoint set of n multicolored triangles which each cover x.

In Section 3 we will described classic concepts in computational geometry that will be used in Section 4 to develop an $O(n^2)$ algorithm for finding center regions. Section 5 will describe the only known polynomial time algorithm for determining if a given point is a CTP [1], and Section 6 attempts to provide a tighter bound for the algorithm in [1] through a more thorough analysis.

3. Arrangements, Duality, and Convex Hulls

Three classic concepts in computational geometry are arrangements, duality, and convex hulls. We will give a brief explanation of each here.

3.1. **Arrangements.**

A set L of n lines in the plane partitions it into vertices (where lines intersect), edges (segments of the lines that run between vertices), and faces (the spaces closed off by edges and vertices). This partition is called the arrangement of L and is denoted $\mathcal{A}(L)$. The arrangement $\mathcal{A}(L)$ can be stored in a data structure called a "doubly-connected edge list" (DCEL). The DCEL contains a record of every edge, face, and vertex in $\mathcal{A}(L)$ and provides a very useful

interface. Given a vertex from $\mathcal{A}(L)$, we can "walk" around it and get an ordered list of the incident edges. Given an edge, we can get the incident vertices and the faces it bounds, and given a face, we can "walk" around it and get the bounding edges. The DCEL of the arrangement $\mathcal{A}(L)$ can be constructed with the following algorithm given in [6]:

Algorithm *ConstructArrangement*
Input: A set L of n lines.
Output: A DCEL for the arrangment $\mathcal{A}(L)$.
1. Construct $\mathcal{A}_0$, a DCEL for an empty arrangement.
2. **for** $i \leftarrow 1$ **to** n
3. **do** $A_i \leftarrow A_{i-1}$
4. Insert line l_i into $\mathcal{A}_i$ as follows:
5. Find an intersection point x between l_i and some line of $\mathcal{A}_i$
6. Walk forward from x along the faces l_i hits, and update $\mathcal{A}_i$
7. Walk backward from x along the faces l_i hits, and update $\mathcal{A}_i$

The Zone Theorem [6] tells us that each line we add will intersect $O(n)$ faces. Thus, we have the following lemma.

Lemma 1. *An arrangement of n lines can be constructed in $O(n^2)$ time.*

3.2. **Duality.** Arrangements are quite useful when studying a set of lines, but the concept of duality allows us to use the obvious and inherent structure of arrangements when studying a set of points as well. Note that a point in $\mathbb{R}^2$ is defined by two real coordinates, and a line in $\mathbb{R}^2$ is defined by two real values, a slope and a y-intercept. Hence, we can map points to unique lines and vice versa. The original space is called the *primal* space, and the transformed space is the *dual* space. There are many ways to form this transformation. The following is perhaps the simplest:

$$(a, b) \Leftrightarrow y = ax - b$$

Figure 1 shows an example of this transformation. Note that this transformation does not allow for vertical lines. This is not a problem, because vertical lines can generally be ignored or treated as a special case. The dual of a point p is denoted p^*; duals of lines are indicated similarly, $l \Leftrightarrow l^*$. Let p be a point in the plane and l be a non-vertical line in the plane. A duality transform must satisfy two requirements

(1) $p \in l$ if and only if $l^* \in p^*$.
(2) p lies above l if and only if l^* lies above p^*.

It is natural to consider what a line segment transforms to. A line segment can be thought of as a set of colinear points. These points transform to an infinite set of lines which all share a common point. Thus, the dual of a line segment forms a double wedge which is bounded by the transformed endpoints of the line segment (see Figure 2).

Note that we do not gain any more information by switching to the dual from the primal space, but a collection of lines has more apparent structure than a collection of points, and it allows us to use the properties of arrangements.

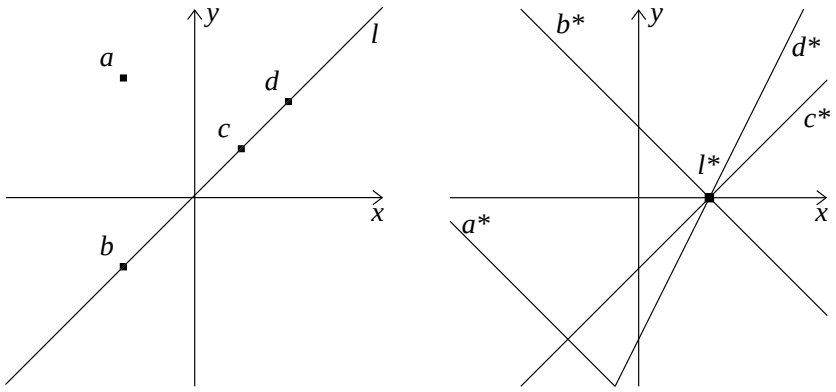

FIGURE 1. Left: Primal plane. Right: Dual plane.

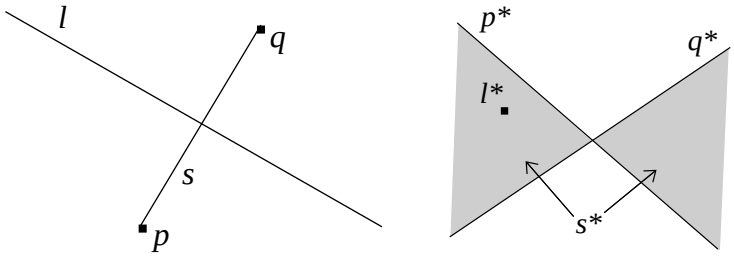

FIGURE 2. Left: Primal plane. Right: Dual plane.

3.3. **Convex Hull.** The final classic computational geometry technique that we discuss is finding convex hulls. A region X of the plane is convex if for every $a, b \in X$, the line segment $\overline{ab}$ is contained in X. The convex hull of a finite set of points C is the unique convex polygon containing C whose vertices are elements of C. There are many $O(n \log n)$ algorithms for computing the convex hull of a set of points in the plane, such as Graham's scan [3] and a divide-and-conquer algorithm given by Preparata and Hong [7]. We present an incremental $O(n \log n)$ algorithm described in [2]. Let C be a set of n points in the plane. Denote by $ch(C)$ the convex hull of C. The convex hull of C can be broken into two halves, upper and lower. When the points of the upper half of $ch(C)$ are examined from left to right, any three consecutive points will turn right. Points from the lower half turn left. Algorithm *ConvexHull* incrementally adds the points of C to $ch(C)$ and removes any point that prevents the last three points in $ch(C)$ from turning the correct direction. It handles the upper and lower halves separately, and then merges them.

Algorithm *ConvexHull*
Input: A set C of n points in the plane.
Output: The convex hull of C.
1. Sort the points by x-coordinate, resulting in a sequence $p_1, \ldots, p_n$
2. Put the points p_1 and p_2 in a list $\mathcal{L}_{upper}$, with p_1 as the first point.
3. **for** $i \leftarrow 3$ **to** n
4. **do** Append p_i to $\mathcal{L}_{upper}$.

5. **while** $\mathcal{L}_{upper}$ contains more than two points **and** the last three points in $\mathcal{L}_{upper}$ do not turn right
6. **do** Delete the middle of the last three points from $\mathcal{L}_{upper}$
7. Compute the lower half $\mathcal{L}_{lower}$ symmetrically, but using the input sequence $p_n, \ldots, p_1$
8. Remove the first and last point from $\mathcal{L}_{lower}$ to avoid duplication of the points where the upper and lower hull meet.
9. Append $\mathcal{L}_{lower}$ to $\mathcal{L}_{upper}$ and call the resulting list $\mathcal{L}$.
10. **return** $\mathcal{L}$

The sorting step can be done in $O(n \log n)$ time, and the remainder runs in $O(n)$ time, giving a total runtime of $O(n \log n)$. The proof of the correctness and runtime of this algorithm can be found in [2].

Lemma 2. *The convex hull of a set of n points in the plane can be found in $O(n \log n)$ time. If the points are given in order, then the sorting step can be skipped, and the convex hull can be found in $O(n)$ time.*

Lemma 3. *Given a point x, it is possible to find a line through x that is tangent to a convex hull S consisting of n vertices in $O(\log n)$ time.*

The tangent line can be found using a sort of binary search. The tangent line must go through some vertex of S. If the tangent line goes through the wrong vertex p, then we can determine which direction the line needs to be moved by examining the two points on either side of p. We can then narrow down on the correct point in $O(\log n)$ time (for a more detailed explanation, see [6]).

4. Center Region Algorithm

Given a set S of n points in the plane, no three of which are colinear, a point x is a center point, if every closed half-plane whose bounding line contains x contains at least $\lceil n/3 \rceil$ points of S. For the sake of brevity, write $k = \lceil n/3 \rceil$. Recall from Section 2, that the center region of S is the set of all center points of S. Note that if a and b are center points of S and c is a point on the line segment between them, then any closed half-plane with c in its bounding line contains either a or b, so it must contain at least k points of S. Thus, the center region is convex. There has been some research on problems related to center points and regions. Matoušek [5] gives an $O(n \log^4 n)$ time algorithm for finding the center region of a set of planar points, and Jadhav, et al. [4] shows that a planar center point can be found in linear time. In the rest of this section, we give the details of the proof of the following theorem.

Theorem 1. *The center region of a set of n points can be found in $O(n^2)$ time.*

Faster algorithms have been produced, but ours is easier to implement and provides a good introduction to computational geometry. The algorithm works by first converting the points to the dual. Thus, it is important to consider what the center region looks like in the dual. The definition of center point implies that a center point p has the property that there are k points of S above and below every line that goes through p. In the dual, the lines that go through p become points on the line p^*. Thus, in the dual, there are k lines from S^* above and below every point on the line p^*. The center region will be the set of all lines with this property.

Definition 1. *The* lower level *of a point is the number of lines on or below it. The* upper level *of a point is the number of lines on or above it.*

Definition 2. *The* k-lower region *is the set of all points with lower level at least* k*. The* k-upper region *is the set of all points with upper level at least* k*. For a set of lines* A*, the* k*-lower region is denoted* $L_k(A)$*, and the* k*-upper region is denoted* $U_k(A)$*, or simply* L_k *and* U_k *if the set of lines is clear.*

The center region in the dual is the set of lines that lie entirely in $L_k(S^*) \cap U_k(S^*)$. We will demonstrate an $O(n^2)$ algorithm for finding the center region in the dual, then convert that region back to the primal.

After transforming S to the dual, we construct its arrangement, $\mathcal{A}(S^*)$. We can determine the levels of each vertex on a line $l \in S^*$ by first finding the level of its leftmost vertex in $O(n)$ time by checking every other line to see if it is above or below this vertex. Then we find the levels for every other vertex on l by walking from left to right across l visiting each of the other vertices. Since no three lines share a vertex, the levels of adjacent vertices can only differ by 1, and that difference can easily be determined by considering the slope of each edge incident to the vertices. Thus, we can find the levels of all vertices in $O(n^2)$ time.

In Figure 3, $L_k(S^*)$ is every point on or above the bold line. Note that the k-lower and k-upper regions will always be bounded by a path that runs through some subset of the vertices of $\mathcal{A}(S^*)$. The next step of our algorithm is to determine those vertices.

Lemma 4. *The vertices bounding the* k*-lower region are those with lower level* $k+1$ *or* k*. The vertices bounding the* k*-upper region are those with upper level* $k+1$ *or* k*.*

The proof of this is clear. As we are finding the levels of the arrangement, if we find a vertex with level k or $k+1$, then we know that two of the edges incident to the vertex consists of points that are of level k or $k+1$. We can walk along these edges in either direction to another vertex of level k or $k+1$. By continuing this walk, we will get all level k or $k+1$ vertices, in order by increasing x-coordinate. Hence, in $O(n^2)$ time we can find the bounding vertices of L_k and U_k in order.

The region $L_k \cap U_k$ is the set of all points that have at least k lines of S^* above and below them. However, we are interested in the set of lines that have this property at each of their points. A line will have at least k lines of S^* below it as long as it stays within L_k. That is, the line cannot cross the boundary of L_k. The set of all lines with this property is the set of all lines that lie on or above $ch(L_k)$, similarly for $ch(U_k)$. Thus, the next step is to compute $ch(L_k)$ and $ch(U_k)$. As shown if Figure 3, L_k (and similarly, U_k) has "tail" rays coming away from the leftmost and right most points. We do not want to include lines that cross these rays, so we must modify our convex hull algorithm. The algorithm will be identical to Algorithm *ConvexHull*, except that we may remove the leftmost or rightmost points if their tail rays cause the hull to turn the wrong direction, in which case we create new tails rays from the new leftmost or rightmost points, parallel to the old rays.

There are $O(n^2)$ vertices in L_k and U_k, and they are already in order. Thus, we can find $ch(L_k)$ and $ch(U_k)$ in $O(n^2)$ time. Furthermore, each line in S^* can contribute at most 2 vertices to $ch(L_k)$ and $ch(U_k)$. Therefore, $ch(L_k)$ and $ch(U_k)$ have linear complexity.

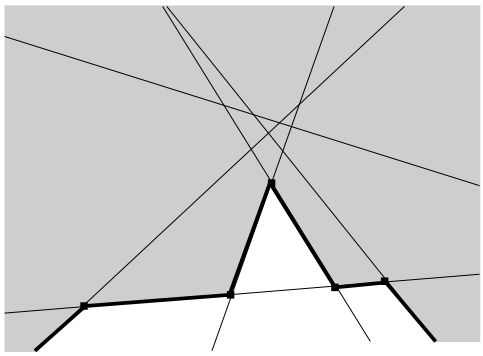

FIGURE 3. The k-lower region is every point on or above the bold line.

We want the final output of our algorithm to be a set of line segments that bound the center region. These line segments have two properties we will make use of.

(1) Points above the line segment and sufficiently close to it are center points while those below it are not, or vice versa.
(2) Every point on the line segment is a center point.

The first property will allow us to identify the line on which the segment lies, and the second property will tell us which section of the line we are interested in. In the dual, the line on which the segment lies is transformed into a point, which, by property 1, will have a "bad" side and a "good" side. Lines that run through the bad side, which is directly above or below the dual point are not center point duals, while lines running on the other side (sufficiently close to the point) may be center point duals. The points in the dual with this property are those that lie on $ch(L_k)$ and $ch(U_k)$. Line segments become wedges in the dual, and the only points on the convex hull that can be the intersection of two distinct lines that do not cross the convex hull boundary are the vertices. Thus, in trying to find the duals of the line segments that bound the center region, we need only consider wedges that go through vertices on $ch(L_k)$ and $ch(U_k)$.

If we determine that a vertex on the convex hull is the dual of a line that bounds the center region, then the next step is to determine which segment of the line bounds the center region. Property 2 tells us that this segment will be the union of every center point on the line. Note that since the center region is convex, this union will form exactly 1 line segment. Let l be a line in the primal plane which contains a line segment $\bar{l}$. In the dual, $\bar{l}^*$ is a set of lines that all go through the same point, l^*. The rightmost point of $\bar{l}$ will have the greatest slope of all the lines in $\bar{l}^*$ and the leftmost point will have the least slope.

The final step in our algorithm is to visit each vertex on $ch(L_k)$ and $ch(U_k)$ and determine the lines with the greatest and least slope that go through the vertex and lie entirely in $ch(L_k) \cap ch(U_k)$. Let $\{p_1, \ldots, p_m\}$ be the vertices of $ch(L_k)$. For each vertex p_i there are 3 possibilities for the bounding lines of the wedge that goes through it.

(1) The boundary lines may be tangent to $ch(U_k)$.
(2) If the first possibility does not work (i.e. every line that runs through p_i and any point in k-upper goes through at least one of the convex hulls),

then there may be a boundary line running parallel to one of the tail rays of k-upper.

(3) The line running through p_i and p_{i-1} or p_i and p_{i+1} may be a boundary line.

For the first possibility, we need to be able to find a line tangent to $ch(U_k)$ and running through the vertex p_i. As we have already seen, this can be done in $O(\log n)$ time.

The second and third possibilities involve checking a fixed number of lines to see if they cross $ch(U_k)$. For each line l, we can create an orthogonal vector u. Then (as described in [6]) we can find the point of $ch(U_k)$ that is extreme in the direction of u and the point extreme in the direction of $-u$. The line l crosses $ch(U_k)$ if and only if these two points are on the opposite side of l. Using the variation of a binary search in [6], we can do this in $O(\log n)$ time. This final step is performed on each point in $ch(L_k) \cup ch(U_k)$. Thus it runs in $O(n \log n)$ time.

If fewer than two lines come from these options, then this vertex is of no interest and we can simply ignore it. Otherwise, find the bounding line with the least slope and the one with the greatest slope. When those lines are converted back to points in the primal space, the line segment that runs between them will be one bound of the center region. A symmetric algorithm can be used for the k-upper vertices.

The basic outline of the center region algorithm is as follows:

$O(n)$	Convert S to dual.
$O(n^2)$	Construct arrangement of S^*.
$O(n^2)$	Compute L_k and U_k.
$O(n^2)$	Compute $ch(L_k)$ and $ch(U_k)$.
$O(n \log n)$	Determine the duals of the bounding line segments as described above.

We can therefore find the center region on n points in the plane in $O(n^2)$ time. Faster algorithms have been produced, but this algorithm has two nice properties: (1) it is fairly easy to implement, whereas some of the faster center region algorithms are difficult to implement and are mainly of theoretical interest; (2) the basic concept of the algorithm is easy to grasp and relies on three classic topics in computational geometry, which makes it a good, didactic introduction to computational geometry.

5. Colored Tverberg Points

In Section 2, we introduced the problem of determining if a given point is a Colored Tverberg point (CTP). Let S be the union of the disjoint sets R, G, $B \subset \mathbb{R}^2$ each with n points, and let x be a point in the plane. Recall from Section 2 that x is a CTP if S can be partitioned into n disjoint multicolored triangles that each cover x. Agarwal, et al. give the first polynomial time algorithm for determining if a given point is a CTP in [1]. What follows is a description of their algorithm.

Notice that we can project the points of S onto a circle C centered at x, and this will not change the status of x as a CTP (see Figure 4). Let C_0 be a closed, half-circle subset of C, and let $C_1 = C - C_0$. Assign each point of $C_0 \cup R$ the label `R+`, and label similarly the points in G and B. The points in C_1 are projected along a straight line through x to C_0 and labelled `R-`, `G-`, or `B-`, depending on their color. The input to the algorithm is the list of labels read clockwise around C_0 (see Figure 4).

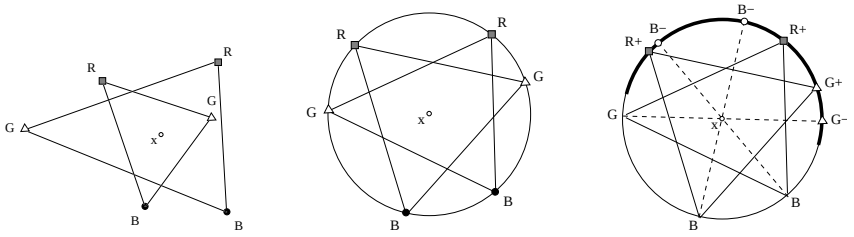

FIGURE 4. Left: x is a CTP. Center: S projected onto a circle centered at x. Right: S produces the labels `R+B-B-R+G+G-`.

`R+G-B+`	`R+B-G+`	`R-G+B-`	`R-B+G-`
`G+R-B+`	`G+B-R+`	`G-R+B-`	`G-B+R-`
`B+R-G+`	`B+G-R+`	`B-R+G-`	`B-G+R-`

TABLE 1. All groupings of labels corresponding to multicolored triangles covering x.

Within the list of point labels, sequences such as `R+G-B+` (not necessarily consecutive) where the signs alternate indicate two points on one side of the circle with a third point between them on the opposite side. Hence this is a triangle covering x. Table 1 gives all possible multicolored triangles covering x. To determine if x is a CTP it is sufficient to show that the list of labels created for S contains n disjoint subsequences from Table 1.

Write $E = (e_1, \ldots, e_{3n})$ for the ordered list of labels produced from S. The algorithm will loop over each e_i creating a set of *configurations* X_i for each prefix of E, $E_i = (e_1, \ldots, e_i)$. A configuration in X_i represents a possible grouping of the labels $e_1, \ldots, e_i$ into prefixes of the multicolored triangles in Table 1. Each point e_i can extend a configuration in X_{i-1} in one of three ways:

(singleton) e_i could be the first label in a new grouping that could lead to a multicolored triangle.

(doubleton) e_i could be grouped with any existing singleton of opposite sign and color.

(triple) e_i could be grouped with an existing doubleton, with which it forms a multicolored triangle.

Therefore, each configuration needs to keep track of the number of available singletons and doubletons. Note, we only need to keep track of the number, not the actual list, since a label can be grouped with any appropriate prefix. There are six possible singletons (a `+` and a `-` for each color), and 12 possible doubletons (each singleton can be completed by two other colors of the opposite sign). Thus, each configuration is stored as an 18-tuple of integers.

When examining e_i, we will construct X_i by looping over every configuration in X_{i-1} and modifying each in up to five ways. For example, suppose $e_i =$ `R+`. Then e_i may create new configurations as follows:

(1) e_i could be a singleton prefix; add 1 to the `R+` component.

(2) e_i could be paired with a `G-` we have already seen; add 1 to the `G-R+` count and subtract 1 from the `G-` count (because one of those points is no longer just a singleton).
(3) e_i could be paired with a `B-` we have already seen; add 1 to the `B-R+` count and subtract 1 from the `B-` count.
(4) e_i could be the last point in the triple `G+B-R+`; subtract 1 from the `G+B-` count.
(5) e_i could be the last point in the triple `B+G-R+`; subtract 1 from the `B+G-` count.

The rules are similar for the other point labels. We only allow configurations with nonnegative components and each X_i contains no repetitions. If a configuration in X_{3n} contains a positive component, then some of the points were not used to form a triangle; hence that configuration does not indicate a CTP. If the zero-tuple is in X_{3n}, then some sequence of configurations divided S entirely into multicolored triangles covering x. Therefore x is a CTP.

Since each component of a configuration is an integer between 0 and n. The number of possible configurations in X_i is $O(n^{18})$. For each point, we must examine every configuration, so the algorithm runs in $O(n^{19})$ time.

Now that we have established a basic algorithm, we can optimize it. Notice that the counts for `R+B-` and `B+R-` are only used when a `G+` is completing a triple. Since both counts must be zero at the end and each can only be decremented when we process a `G+`, it does not matter which prefix we pair `G+` with. Hence, we need not store the counts for `R+B-` and `B+R-` separately; we only need their sum. Thus, we can combine the 12 doubleton prefixes into the following 6:

`(R+B-)` + `(B+R-)`
`(R+G-)` + `(G+R-)`
`(B+G-)` + `(G+B-)`
`(R-B+)` + `(B-R+)`
`(R-B+)` + `(B-R+)`
`(R-B+)` + `(B-R+)`

The configurations are now 12-tuples, where each component has $O(n)$ possible values, so the algorithm runs in $O(n^{13})$ time.

We can optimize this even further. Suppose $\xi \in X_j$, and let $M_\xi(R)$ be the sum of each component in ξ that involves an element of R. Define $M_\xi(G)$ and $M_\xi(B)$ similarly. Let $K_j(R)$, $K_j(G)$, and $K_j(B)$ be the number of red, green, and blue points, respectively, in E_j. Let t be the number of triples recorded in the creation of ξ. Then $M_\xi(R)$ is the number of red labels we have seen that are not used in a full triangle, and t counts the red labels that have been used in a triangle. Thus, their sum is the total number of red labels we have seen. So we have

$$\begin{aligned} K_j(R) &= t + M_\xi(R) \\ K_j(G) &= t + M_\xi(G) \\ K_j(B) &= t + M_\xi(B). \end{aligned}$$

This gives us

$$\begin{aligned} M_\xi(B) - M_\xi(R) &= K_j(B) - K_j(R) \\ M_\xi(G) - M_\xi(R) &= K_j(G) - K_j(R). \end{aligned}$$

These two independent, linear relations imply that we can store the configurations with only 10 components, giving us a $O(n^{11})$ time algorithm. Note that we can modify the $O(n^{13})$ algorithm by removing two of its components and recalculating their value when needed, but each time we would create a new configuration in the $O(n^{13})$ version, we would create one in the $O(n^{11})$ version as well. This shows that the algorithm actually runs in $O(n^{11})$ time.

6. Further Analysis

The analysis of the CTP algorithm described above is rather simple. Hence, we attempted to provide a more thorough and accurate runtime bound. We went about this in two majors ways: combinatorially and finding the worst case runtime.

From the original analysis, we know that the algorithm runs in $O(nm)$, where m is the maximum number of configurations examined by the algorithm for a given point. Therefore, if we can find a bound for the number of configurations better than $O(n^{10})$, then we have improved on previous work.

The bound $O(n^{10})$ comes from the fact that we can save the configurations as 10-tuples with $O(n)$ possible values for each component. This simple analysis does not take into account many limitations placed on the configurations, such as the fact that the 10 values cannot sum to more than $3n$. If we forget for a moment the interpretations of each of the values in the 10-tuple, we can abstract this to a combinatorial problem. We have 10 bins, and we have a total of $3n$ points to distribute. More generally, we wish to determine the number of ways of distributing x identical objects into y identical bins. We can think of this as having the x objects in a line and placing $y-1$ dividers between them like so

$$\bullet\bullet\bullet|\,\bullet\,|\,\bullet\bullet\bullet\bullet\,|\,\bullet\bullet$$

The objects before the first divider go into the first bin, the objects between the $(i-1)$th and ith divider go into the ith bin. There are $x+y-1$ symbols being placed, and we are choosing $y-1$ of them to be dividers. Thus, the number of ways to do this is

$$\binom{x+y-1}{y-1}$$

The number of ways the values of the components of a configuration can sum to $3n$ is given by

$$\binom{3n+10-1}{10-1} = \frac{(3n+9)!}{(3n)!9!} = O(n^9).$$

Each time a triple is encountered, we decrease certain components of a configuration. Thus, when examining the final point of the input, the components of a configuration can sum to any value between 0 and $3n$. Thus, this gives the bound

$$\sum_{i=0}^{3n} \binom{i+10-1}{10-1}.$$

Unfortunately, this is still $O(n^{10})$. Thus, the simple restriction that the sum of the components of the configurations must be less than or equal to $3n$ does not provide an asymptotic improvement in the bounds.

The more promising approach we took to analyzing the CTP algorithm began as empirical investigation. We wrote an implementation of the algorithm which kept

n	n^9	Actual Number
1	1	4
2	512	31
3	19683	227
4	262144	1290
5	1953125	4505
6	10077696	14989
7	40353607	39159
8	134217728	101941
9	387420489	225957

TABLE 2. A comparison of the actual number of configurations with the growth of n^9.

track of the number of configurations produced while examining an arrangement of points, which were input as an ordered sequence of labels as described above. We ran the program with every possible permutation of labels to see how close the maximum number of configurations comes to the theoretical upper bound. Unfortunately, the number of permutations grows quickly, and we could only do this for small values of n. However, we discovered two interesting patterns.

First, the arrangement of points that produces the maximum number of configurations seems to follow the same clear pattern, independent of the number of points. In this pattern, the points alternate in color and sign. For $n = 3$, the arrangement is

`R+ G- B+ R- G+ B- R+ G- B+`.

Intuitively this pattern makes sense. Each point the algorithm considers can always be used as a singleton prefix, so we need not worry about those. We can get up to two doubleton prefixes, but only if we have points of opposite color and sign to work with. And we can get triples if there are enough pairs of points with opposite color and sign that we have already examined. This pattern seems to maximize the likelihood that each point will be able to be used as two doubletons and a triple.

The second point to note is the number of configurations this pattern produces compared to the theoretical bound. The chart in Table 2 shows the number of configurations produced for the first few values of n, along with n^9, which is asymptotically dominated by the theoretical bound on the number of configurations. Clearly, the actual number of configurations generated is significantly smaller than n^9, and thus less than the theoretical bound. Moreover, the rate of growth (i.e. the ratio of consecutive values) is significantly smaller for the actual number than for n^9.

Assuming this pattern is the worst case scenario for the algorithm, this data suggests that the trivial bound can be improved, perhaps significantly. Unfortunately, proving that this pattern is the worst case scenario and finding an expression for the number of configurations needed to examine this pattern are both still open problems. These problems are complicated by the fact that repetitions are not allowed in the set of configurations, which makes analysis quite difficult.

References

[1] P.K. Agarwal, M. Sharir, and E. Welzl, Algorithms for center and Tverberg points, *Proc. 20th Symp. Computational Geometry*, 2004, 61–67.

[2] M. de Berg, et al, *Computational Geometry: Algorithms and Applications, Second Edition*, Springer Verlag, Heidelberg, 2000.

[3] R.L. Graham, An efficient algorithm for determining the convex hull of a finite planar set. *Inform. Process. Lett.*, 1972, 1:132–133.

[4] S. Jadhav and A. Mukhopadhyay, Computing a centerpoint of a finite planar set of points in linear time, *Discrete Computational Geometry* 12, 1994, 291–312.

[5] J. Matoušek, Computing the center of a planar point set. *Discrete and Computational Geometry*, American Mathematical Society, 1991, 221–230.

[6] J. O'Rourke, *Computational Geometry in C, Second Edition*, Cambridge University Press, 1998.

[7] F.P. Preparata and S.J. Hong, Convex hulls of finite sets of points in two and three dimensions, *Commun. ACM*, 1997, 20:87–93.