Speeding Up the Douglas-Peucker Line-Simplification Algorithm

John Hershberger
DEC Systems Research Center
130 Lytton Ave
Palo Alto, CA 94305 USA
johnh@src.dec.com

Jack Snoeyink
Department of Computer Science
University of British Columbia
Vancouver, BC V6T 1Z2 Canada
snoeyink@cs.ubc.ca

Abstract

We analyze the line simplification algorithm reported by Douglas and Peucker and show that its worst case is quadratic in n, the number of input points. Then we give a algorithm, based on *path hulls*, that uses the geometric structure of the problem to attain a worst-case running time proportional to $n \log_2 n$, which is the best case of the Douglas algorithm. We give complete C code and compare the two algorithms theoretically, by operation counts, and practically, by machine timings.

1 Introduction

An important task of the cartographer's art is extracting features from detailed data and representing them on a simple and readable map. As computers become increasingly involved in automated cartography, efficient algorithms are needed for the tasks of extraction and simplification. Cartographers [1, 10] have identified the *line simplification problem* as an important part of representing linear features. An ordered set of $n + 1$ points in the plane, $\{V_0, V_1, \ldots, V_n\}$, forms a *polygonal chain*—the sequence of line segments $\overline{V_0V_1}$, $\ldots$, $\overline{V_iV_{i+1}}$, $\ldots$, $\overline{V_{n-1}V_n}$. Given a chain C with n segments, the line simplification problem asks for a chain C' with fewer segments that "represents C well."

We further assume that our given chain C is simple—that is, C has no self-intersections. In cartographic applications, self-intersections often indicate errors in digitization [10]. Although we would like to require that the approximation C' should also be simple, there is some indication that it is computationally infeasible do to so [5].

"Representing well" has many possible meanings. For example, it may require that C and C' are close in distance, that the area between C and C' is small, that the critical points of C are incorporated into C', or that other measures of curve discrepancy are small. McMaster [9] gives a detailed study of mathematical similarity and discrepancy measures. He also ranks the method reported by Douglas and Peucker [4] as "mathematically superior." White [14] performed a study of three simplification algorithms, based on Marino's work [8] on critical points as a psychological measure of curve similarity. She showed that the Douglas-Peucker method was best at choosing critical points and also reports, "The generalizations produced by the Douglas algorithm were overwhelmingly—by 86 percent of all sample subjects—deemed the best perceptual representations of the original lines."

The Douglas method also has other advantages. First, it is easy to program; Whyatt and Wade [15] give complete FORTRAN code in their paper. Second, the simplification it produces has a hierarchical structure that has been exploited by Cromley [2], following the work of Jones and Abraham [6] on scale-independent cartographic databases. It should be no surprise that this method has been independently proposed in other contexts. Two examples are papers of Ramer [12] in image processing and Rote [13] in computational geometry.

In this paper we analyze two algorithms that implement the Douglas-Peucker simplification method using different data structures. In the next section we give the basic simplification method and discuss how to compare algorithms. In section 3, we give a theoretical analysis of Douglas and Peucker's implementation and show that its worst-case running time shows quadratic growth as the number of input segments increases. In section 4 we give a new algorithm based on the *path hull* data structure whose worst-case running time asymptotically the same as the best case of the Douglas-Peucker algorithm. Subsections describe the geometric structure of the problem, define the path hull data structure to exploit it, and analyzes the new implementation that results. Section 5 gives timings of sample runs.

2 The simplification method of Douglas and Peucker

The method given in Douglas and Peucker [4] is best described recursively: To approximate the chain from V_i to V_j, start with the line segment $\overline{V_iV_j}$. If the farthest vertex from this segment has distance at most ϵ, then accept this approximation. Otherwise, split the chain at this vertex and recursively approximate the two pieces. Algorithm 1 makes this more precise.

Given an array of vertices V, the call `DPbasic`(V, i, j) simplifies the subchain from V_i to V_j.

Procedure `DPbasic`(V, i, j)

1. Find the vertex V_f farthest from the line $\overleftrightarrow{V_iV_j}$.
  Let *dist* be its distance.
2. if *dist* $> \epsilon$ then
  3a. `DPbasic`(V, i, f) /* *Split at* V_f *and approximate* */
  3b. `DPbasic`(V, f, j) /* *recursively* */
else
  4. `Output`$(\overline{V_iV_j})$ /* *Use* $\overline{V_iV_j}$ *in the approximation* */

Algorithm 1: The basic line simplification method

For those familiar with the original paper of Douglas and Peucker [4], this recursive procedure is equivalent to their iterative "Method 2," which was a heuristic improvement by stacking "anchors." Stated as a recursive procedure, one can see the justification for their "anchor stacking."

Notice that the algorithm uses a subset of the original vertices $\{V_0, V_1, \ldots, V_n\}$ to form the approximate chain. The statement of line 1 can be seen as a way to choose a vertex as the splitting vertex. The primary difference between the Douglas-Peucker algorithm and the path hull algorithm is how they choose the splitting vertex. Notice also, for future use, that the calls 3a and 3b can be made in either order.

Before we describe the two algorithms in the next two sections, let us say a few words about how to compare them. Of course, we can compare implementations by timing them on sample data—and we will do so in section 5. But such timings tell only about a given, specific problem instance unless we know how to extrapolate from the timing to instances that we encounter in practice. If we understand the capabilities of the algorithms—knowing what is the best and worst case data—then we can design better test sets and interpret the results of the tests more accurately.

In the theoretical analysis of algorithms [7], one counts the number of operations that the algorithm uses as a function of the number of input segments. (Those familiar with such analysis can safely skip the remainder of this section.) Because of differences between machines and compilers, the qualitative behavior of such counts can be more revealing than the quantitative behavior. Let us define the commonly used big-O notation and then illustrate with examples. For integer functions f and g, we say that $f(n) = O(g(n))$ if there are constants a and N such that $f(i) \leq ag(i)$ for all $i \geq N$. We say that $f(n) = \Theta(g(n))$ if $f(n) = O(g(n))$ and there is a positive constant c such that $f(i) \geq cg(i)$ for infinitely many integers i.

A statement involving big-O gives an upper limit on the increase of the running time; a statement involving big-Θ gives an upper and lower limit. For example, since each call to `DPbasic`() either finds a splitting vertex or outputs a line segment, we know there will be $2k - 1$ calls to compute an approximate chain C' having k segments. We could give weaker qualitative bounds as $\Theta(k)$ calls or as $O(n)$ calls, since $k \leq n$. Since different computer systems have different costs associated with making a call, the actual count is often only slightly more informative than an asymptotic count—and it can be much more difficult to obtain.

In the next two sections we will see that the running times of Douglas and Peucker's algorithm and of our path hull algorithm have worst cases of $\Theta(n^2)$ steps and $\Theta(n \log_2 n)$ steps, respectively. This qualitative difference implies that as the size of the problem increases by 4, the worst case time of the former increases 16-fold. The latter increases by much less—a factor of 5 to go from 64 to 256 and less than 4.5 to go from 5000 to 20000. Furthermore, if technology improves 16-fold, the former can handle a problem 4 times greater, while the latter allows jumps from 64 to 656 or from 5000 to 61768.

The best case for both algorithms is $\Theta(n)$ when the chain C can be approximated by a single segment. If C can only be approximated by itself, i.e. $k = n$, then the best cases are $\Theta(n \log_2 n)$ and $\Theta(n)$, respectively. By way of remark, Cromley's data structure [2] gives a reason to look at the case $k = n$. His key idea is to run a recursive simplification algorithm with a tolerance of $\epsilon = 0$, so that no simplification occurs, and to keep track of the splitting points found and their distances. Then, when a scale-dependent tolerance ϵ is given and the algorithm is rerun, it need not search for the splitting vertex.

Of course, when it comes to the practical implementation, factors of two or ten in the constants matter. For example, the Douglas algorithm records less geometric structure so it can be expected to have a lower constant than our algorithm. Thus, theoretical analysis is not a substitute for running test cases and discovering (machine dependent) quantitative behavior. The qualitative behavior that it reveals, however, is important for understanding the algorithm, designing test cases, and interpreting results.

3 A simple implementation of step 1

An obvious way to find the splitting vertex in step 1 of algorithm 1, *i.e.,* to find the point farthest from the line $\overleftrightarrow{V_iV_j}$, is to compute the distance of each point of $\{V_i, \ldots, V_j\}$ and keep the maximum. Algorithm 2 implements this idea.

Find the splitting vertex V_f, which is farthest from the line $\overleftrightarrow{V_iV_j}$, by evaluating all distances and keeping the maximum. Return the distance *dist* of V_f.

Procedure `FindSplit`$(V, i, j, f, dist)$
    set $dist = 0$
    for $k = i + 1$ to $j - 1$ do

$$distVk = \left(\frac{\begin{vmatrix} 1 & V_i.x & V_i.y \\ 1 & V_j.x & V_j.y \\ 1 & V_k.x & V_k.y \end{vmatrix}}{(V_i.x - V_j.x)^2 + (V_i.y - V_j.y)^2} \right)^{1/2}$$

        if $distVk \geq dist$ then
            set $dist = distVk$
            set $f = k$

Algorithm 2: Douglas and Peucker's algorithm for finding the splitting vertex

The most time-consuming part of this procedure is the evaluation of the distance formula, so we count how often that occurs. If the original chain C has n line segments (points $V_0, V_1, \ldots, V_n$) and the approximate chain C' has k segments, then we know that step one is executed k times and each execution has at most n distance evaluations. Thus, an upper bound is $O(kn)$.

To make a careful analysis of the worst case easier, let us assume that no actual simplification occurs; i.e., that $k = n$. Then, if we know the splitting behaviour, the total number of distance evaluations can be counted as a function of the number of input segments n. The best outcome would be for every split to lie in the middle of its chain. The number of distance evaluations then satisfies a recurrence:

$$\begin{aligned} D(1) &= 0 \\ D(n) &= n - 1 + D(\lfloor \tfrac{n}{2} \rfloor) + D(\lceil \tfrac{n}{2} \rceil), \end{aligned}$$

which has a solution $(n-2)\log_2 n \leq D(n) \leq n \log_2(n+1)$. Thus, $D(n) = \Theta(n \log_2 n)$.

In the worst case, however, a split could take just one segment off either end of the chain:

$$\begin{aligned} D(1) &= 0 \\ D(n) &= n - 1 + D(1) + D(n-1) \end{aligned}$$

This gives the worst case performance $D(n) = (n-1)(n-2)/2 = \Theta(n^2)$. This qualitative difference becomes significant as larger and larger problem instances are considered—we will see concrete evidence of this in section 5. Furthermore, the vast difference between the best and worst case can be a problem in parallel applications, which are only as fast as the slowest task, and in interactive applications, in which users want predictable response time.

The reason for the inefficiency of the worst case is easy to see—one must look at all the vertices to find the splitting vertex, but then the resulting split leaves a problem nearly as large as the original. In the next section, we show how to exploit information about the geometric structure of the problem to find a splitting vertex by inspecting at most $\log_2 n$ vertices. We also see how to maintain this structural information while performing the n splits. This allows us to improve the worst case to $O(n \log_2 n)$ time.

On a single-user batch system, one is less concerned with the worst-case performance and more concerned with the "expected" running time on "typical" data. Mathematically, one could define a probability distribution that certain data would actually occur and then compute the expected time by summing the running time times the probability of each possible instance. The only way to exactly define a distribution, however, is to look at all data sets ever or ever to be simplified. To compare two or more algorithms on these data sets would involve running each algorithm on each data set to find out which is the fastest; but if the solutions are known, the fastest algorithm is one that simply outputs the known solution. To define a probability distribution that allows one to predict "typical" running times without solving all possible problems is practically impossible.

To consider problems that may be more typical than the best or worst case, however, we can look at the number of distance calculations when a random vertex is chosen as the splitting vertex. This would give the recurrence relation

$$D(1) = 0 \tag{1}$$

$$D(n) = n - 1 + \frac{1}{n-1} \sum_{1 \le i \le n-1} (D(i) + D(n-i)) \tag{2}$$

which has a solution of $D(n) = \Theta(n \log_2 n)$. A carefull analysis shows that, if splits occur at random vertices, the average running time is a factor of $2 \log_2 e \approx 2.885$ greater than the best.

A programmer concerned with efficiency should work with squares of the distance and of ϵ (as do Whyatt and Wade [15]) because computing square roots is time consuming. Also, the portions of the determinant computation that do not depend on k can be taken out of the loop. These changes are reflected in the timings for the `DPimproved` algorithm in section 5.

4 The path hull algorithm

In this section, we give a new algorithm for the method of Douglas and Peucker with $\Theta(n \log_2 n)$ worst-case running time. In subsection 4.1, we show that the splitting vertices for a chain must be found on its *convex hull*. We then define the *path hull* data structure and give the path hull algorithm for line simplification, momentarily assuming the existence of procedures to build and split path hulls and to find extreme vertices. In subsection 4.2, we outline the structure of the path hulls and the procedures that operate on them. Finally, we analyze the asymptotic running time of the new implementation in subsection 4.3.

4.1 Splitting vertices and convex hulls

We begin with some definitions. A set C is *convex* if whenever points p and q are in C, the segment $\overline{pq} \subset C$. A line t is *tangent* to C if C intersects t and lies entirely on and to one side of t. A closed set C has two tangents parallel to a given direction, as illustrated in figure 1. These tangents touch extreme points of C.

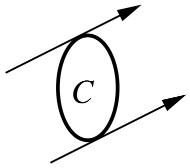

Figure 1: Tangents to a convex set

Lemma 4.1 *Given a convex set C and a line ℓ, the points of C at max distance from ℓ lie on the two tangents to C parallel to ℓ.*

Proof: The points at equal distance from ℓ lie on lines parallel to ℓ. If a point $p \in C$ achieves the maximum distance, then consider the line ℓ' through p and parallel to ℓ. No point of C can lie on the opposite side of ℓ' from ℓ, so ℓ' is a tangent to ℓ. ∎

The *convex hull* of a set of points P is the smallest convex set containing P. The boundary of the hull, which we denote $CH(P)$, is a polygon consisting of points of P and segments joining them. According to lemma 4.1, only the vertices of $CH(P)$ need be considered as farthest points from ℓ.

So, our central task will be to maintain and search convex hulls of subchains efficiently. Dobkin et al. [3] developed the path hull data structure for a similar hull maintenance problem; here, we modify their structure slightly. We define the *path hull* of the chain from V_i to V_j to consist of a *tag vertex* V_m and a pair of boundaries of convex hulls for the subchains ending at that tag vertex, namely $CH(V_i, \ldots, V_m)$ and $CH(V_m, \ldots, V_j)$. We assume three operations on path hulls.

`Build`(V, i, j, PH) Build the path hull PH of $V_i, \ldots, V_j$ by choosing the middle vertex $V_{\lfloor (i+j)/2 \rfloor}$ as the tag and computing the convex hulls of two subchains.

`Split`(PH, V, k) Given the path hull PH of the chain $V_i, \ldots, V_j$, split the chain at V_k and return a path hull of the subchain containing the tag vertex. (The path hull for the other subchain will be rebuilt later.)

`FindFarthest`(PH, ℓ) Find the farthest vertex of PH from a given line ℓ.

Algorithm 3 uses these three operations to compute the same simplification as that found by the Douglas-Peucker algorithm. The next section fleshes out the path hull data structure and its operations.

4.2 Implementing path hulls

A path hull consists of two arrays storing the vertices of the two convex hulls: the *left* stores $CH(V_i, \ldots, V_m)$ and the *right* stores $CH(V_m, \ldots, V_j)$. With each array is a *history stack.* In this subsection, we outline efficient implementations of the path hull operations used in algorithm 3. Interested readers can compare with the path hulls in Dobkin et al. [3].

We begin with the implementation of the operation `FindFarthest`(PH, ℓ). To find the farthest vertex from ℓ, we can separately find the farthest in the left and right hull and select the correct vertex. Thus, we concentrate on finding the farthest vertex on a convex polygon $CH(P)$.

If PH contains a path hull of the subchain $V_i, \ldots, V_{j+1}$, then the call `DPhull`(V, i, j, PH) simplifies the subchain.

Procedure `DPhull`(V, i, j, PH)

1. $V_f =$ `FindFarthest`$(PH, \overleftrightarrow{V_iV_j})$
2. if `Distance`$(V_f, \ell) \leq \epsilon$ then
  3. `Output`(V, i, j) /* *Accept approximation* */
else
  4. if V_f is less than the tag vertex then
    5. `Split`(PH, V, f) /* *Form PH for* $V_f, \ldots, V_j$ */
    6. `DPhull`(V, f, j, PH)
    7. `Build`(V, i, f, PH) /* *Build PH for* $V_i, \ldots, V_f$ */
    8. `DPhull`(V, i, f, PH)
  else
    9. `Split`(PH, V, f) /* *Form PH for* $V_i, \ldots, V_f$ */
    10. `DPhull`(V, i, f, PH)
    11. `Build`(V, f, j, PH) /* *Build PH for* $V_f, \ldots, V_j$ */
    12. `DPhull`(V, f, j, PH)

Algorithm 3: The path hull algorithm for line simplification

Suppose the edges of $CH(P)$ are listed in counter-clockwise (ccw) order and that ℓ is oriented from left to right as illustrated in figure 2. A tangent to the convex hull of P touches a vertex of $CH(P)$. If we rotate the tangent counter-clockwise, then it pivots around a vertex until it encounters the outgoing segment of $CH(P)$ and switches its pivot to the next vertex in ccw order. This implies that the vertices with tangents parallel to ℓ separate $CH(P)$ into two chains, one whose edges form positive angles less than 180° with ℓ and one whose edges form negative angles. As an abbreviation, we call these *positive* and *negative* edges.

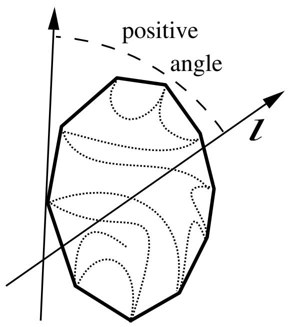

Figure 2: Postive angle from ℓ to tangent

We can find the farthest vertex by a three step process: First, find a positive and a negative edge, which separate $CH(P)$ into two pieces that each contain one extreme point. Second, use binary searches to locate a vertex on each piece where the edge angles change sign. Third, compute and compare the distances to these extreme points to determine which is greater.

To perform the first step, choose an arbitrary edge e of $CH(P)$ as a base edge. Let us assume that e is positive, as in figure 3, so that our task is to find a negative edge. Choose the edge e' that splits $CH(P)$ into two equal pieces. If e' is negative, we are done with the first step. If e' is positive, then look at a segment s (drawn dashed) from an endpoint of e to an endpoint of e'. If s is also positive, then we can discard the hull edges before e' because they are all positive. If s is negative, then we can discard the portion after e'. Then we choose a new e' that divides the remaining portion in equal halves and repeat. We can perform this halving at most $\log_2 n$ times before finding a negative edge e'.

For the second step, we have a positive edge e and a negative edge e' and we must search the portion between them for the vertex adjacent to both positive and negative edges. As before, we look at a middle edge e''. If e'' is positive, then we replace e with e''; if e'' is negative, we replace e' with e''. After testing at most $\log_2 n$ edges, we find the vertex with tangent parallel to ℓ.

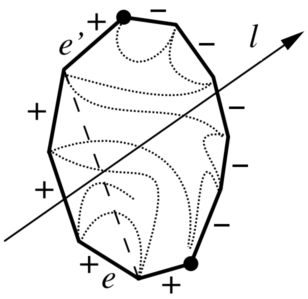

Figure 3: Searching for a negative edge

The third step is the easiest—compute the two distances (or their squares) and compare them. Thus, after $O(\log_2 n)$ operations, we can report the farthest point from ℓ if we have $CH(P)$ available in an array.

For the `Build()` operation, we use a modification of Melkman's convex hull algorithm [3, 11]. This algorithm (and therefore our algorithm as well) is valid only if the underlying path has no self intersections, which is usually the desired case in cartograpic and vision applications.

Melkman's algorithm computes the right convex hull $CH(V_m, \ldots, V_j)$ incrementally, by successively adding vertices V_m through V_j to a double-ended queue, which is implemented in the right array. Initially, the queue contains, from bottom to top, V_{m+1}, V_m, and V_{m+1}. To insert V_l into the queue storing the hull $CH(V_m, \ldots, V_{l-1})$ we do the following: While V_l is not to the right of the hull edge from the top of queue, pop the top vertex from the queue. And, while V_l is not to the right of the hull edge into the bottom of queue, pop the bottom vertex. If any vertex has been popped, then push V_l onto the top and bottom of the queue. Thus, Melkman's algorithm computes not only the desired convex hull, but also all intermediate hulls.

We modify his algorithm to store the sequence of pushes and pops, and the vertices involved, in the right history stack. We also construct the left convex hull $CH(V_i, \ldots, V_m)$ by adding vertices from V_m down to V_i and record the pushes and pops in the left history stack. This completes the `Build()` operation.

The `Split`(PH, V, k) operation is now quite easy. We play back the history of the hull that contains the splitting vertex V_k and undo the operations until we reach the operation that pushes V_k.

4.3 Analyzing the running time

In this section we analyze the running time of the path hull algorithm, Algorithm 3. First we consider the total cost of calls to `FindFarthest()`, then calls to `Split()` and `Build()`.

Lemma 4.2 *The total cost of calls to* `FindFarthest()` *is* $O(n \log_2 n)$

Proof: We have already noticed in section 2 that there are $O(n)$ calls to `DPhull()` and thus to `FindFarthest()`. Each call performs $O(\log_2 n)$ tests and halving operations, so the total cost of all calls is $O(n \log_2 n)$. ∎

We can relate the cost of `Split()`s to the cost of `Build()`s.

Lemma 4.3 *The cost of all calls to* `Split()` *is at most the cost of all calls to* `Build()`.

Proof: Since there are only $O(n)$ calls to `Split()`, there are at most $O(n)$ operations that do not pop the history stack. Any operation that does pop the history stack is simply undoing the effect of some `Build()`. ∎

Now we can account for the work done by `Build()` operations by a "credit" scheme, in which each call to `DPhull()` is given some number of credits to pay for the `Build()`s in its body and its recursive calls. We give $O(n \log_2 n)$ credits to the first call to `DPhull()`, then show that all calls have enough credits to pay for their own work and that of their recursive calls.

Lemma 4.4 *The total cost of all* `Build()` *operations is* $O(n \log_2 n)$.

Proof: Let l and r be the number of internal vertices to the left and right of the tag vertex in a given call to `DPhull()`. "Internal" means that we neglect the endpoints, because a chain with only one edge cannot be split further. We show that if the call is given at least $(l + r)\log_2(\max\{l, r\})$ credits, then it can pay for its own `Build()` operation and give the correct number of credits to its recursive calls of `DPhull()` to pay for their `Build()` operations.

Credits are spent as follows. Let the index of the splitting vertex be $f = i + s$. Since the algorithm and the credit function are both symmetric with respect to left and right, let us assume that v_f is to the left of the tag. In line 6 of Algorithm 3 the call to `DPhull()` must be given

$$(l - s + r)\log_2 \max\{l - s, r\} \leq (l - s + r)\log_2 \max\{l, r\} \qquad *$$

credits to pay for its calls to `Build()`. In line 7 we spend s credits in the call to `Build()`, which constructs a path hull with at most $\lfloor s/2 \rfloor$ internal vertices on each side of the tag. Therefore, in line 8, we give

$$s \log_2 \lfloor s/2 \rfloor \leq s(\log_2 s - 1) \leq s \log_2 \max\{l, r\} - s \qquad **$$

credits to the recursive call to `DPhull()`.

Summing $(*)$, $(**)$ and the s credits spent on `Build()`, we find that the total expenditure is at most $(l + r)\log_2(\max\{l, r\})$ credits. Therefore, if we give the first call to `DPhull()` a total of $n \log_2 n$ credits, it has sufficient credits to pay for all calls to `Build()`. ■

With this lemma, we have shown that the work done by each line of the algorithm can be bounded by $O(n \log_2 n)$. This completes the analysis.

5 Comparing running times

In this section we compare machine timings of three implementations of line simplification algorithms on a SUN 3/50. The first, `DPbasic`, is the basic Douglas-Peucker implementation as described in section 3. The second, `DPimproved`, incorporates the suggestions given at the end of that section for speeding up the search for a splitting vertex—squared distances are compared and loop invariants are moved outside the loop. The third, `DPhull`, uses path hulls to find a splitting vertex.

Based on the analysis that we have done in sections 3 and 4, we can run these algorithms on inputs that reveal their best and worst cases. The first case is the example of Douglas and Peucker with points evenly spaced on a circle. This is readily seen to be the best case for the standard implementation, since all splits occur in the middles of chains, and the worst case for the path hull algorithm, since the convex hull of any subchain includes all the points. Since the path hull implementation records structural information not considered by the standard implementation, it is not surprising that it is slower.

	Time (secs)		
Algorithm	1000 pts	5000 pts	10000 pts
`DPbasic`	0.350	1.883	4.950
`DPimproved`	0.067	0.300	0.767
`DPhull`	0.200	1.050	2.600

The worst case for the standard implementation is a zig-zag or spiral where every split chops off only a single segment. Here the improvement given by the path hull implementation is dramatic.

	Time (secs)		
Algorithm	1000 pts	5000 pts	10000 pts
`DPbasic`	19.700	407.067	1993.817
`DPimproved`	2.150	43.883	235.500
`DPhull`	0.150	0.650	1.550

As mentioned in section 2 it is difficult to test algorithms on "typical" data. The last test set is a monotone chain with random y coordinates. On this input the `DPimproved` and `DPhull` algorithms have comparable performances.

	Time (secs)		
Algorithm	1000 pts	5000 pts	10000 pts
`DPbasic`	1.050	7.717	25.800
`DPimproved`	0.133	0.833	2.883
`DPhull`	0.167	0.800	2.000

6 Conclusions

A mathematical analysis of the line simplification algorithm reported by Douglas and Peucker [4] shows that its worst-case running time is quadratic. That is, $\Theta(n^2)$ time can be required to simplify a polygonal line with n input points. Investigation into the geometry of their simplification method reveals how one can maintain enough structure to guarantee a running time of $O(n \log_2 n)$, which asymptotically matches the best case of the standard algorithm. Implementation shows the new algorithm to be a factor of 2.5 to 3 slower than the best case of the standard algorithm, due to the extra structural information that it maintains, but to be far faster (e.g. a factor of 150 on 10000 points) in the worst case of the standard algorithm. The fact that the variance of the running time of the new algorithm is small may make it suitable for interactive and/or parallel applications.

References

[1] B. Buttenfield. Treatment of the cartographic line. *Cartographica*, 22:1–26, 1985.

[2] R. G. Cromley. Hierarchical methods of line simplification. *Cartography and Geographic Information Systems*, 18(2):125–131, 1991.

[3] D. Dobkin, L. Guibas, J. Hershberger, and J. Snoeyink. An efficient algorithm for finding the CSG representation of a simple polygon. *Computer Graphics*, 22(4):31–40, 1988. Proceedings of SIGGRAPH '88.

[4] D. H. Douglas and T. K. Peucker. Algorithms for the reduction of the number of points required to represent a line or its caricature. *The Canadian Cartographer*, 10(2):112–122, 1973.

[5] L. J. Guibas, J. E. Hershberger, J. S. B. Mitchell, and J. S. Snoeyink. Minimum link approximation of polygons and subdivisions. In W. L. Hsu and R. C. T. Lee, editors, *ISA '91 Algorithms*, number 557 in LNCS, pages 151–162. Springer-Verlag, 1991.

[6] C. Jones and I. Abraham. Line generalisation in a global cartographic database. *Cartographica*, 24(3):32–45, 1987.

[7] D. E. Knuth. Mathematical analysis of algorithms. In *Proceedings of the IFIP Congress 1971*, pages 19–27. North-Holland, Amsterdam, 1972.

[8] J. S. Marino. Identification of characteristic points along naturally occurring lines: An empirical study. *Can. Cartog.*, 16:70–80, 1979.

[9] R. B. McMaster. A statistical analysis of mathematical measures for linear simplification. *Amer. Cartog.*, 13:103–116, 1986.

[10] R. B. McMaster. Automated line generalization. *Cartographica*, 24(2):74–111, 1987.

[11] A. A. Melkman. On-line construction of the convex hull of a simple polyline. *Info. Proc. Let.*, 25:11–12, 1987.

[12] U. Ramer. An iterative procedure for the polygonal approximation of plane curves. *Comp. Vis. Graph. Image Proc.*, 1:244–256, 1972.

[13] G. Rote. Quadratic convergence of the sandwich algorithm for approximating convex functions and convex figures in the plane. In *Proc. Second Can. Conf. Comp. Geom.*, pages 120–124, Ottawa, Ontario, 1990.

[14] E. R. White. Assessment of line-generalization algorithms using characteristic points. *Amer. Cartog.*, 12(1):17–27, 1985.

[15] J. D. Whyatt and P. R. Wade. The Douglas-Peucker line simplification algorithm. *Bulletin of the Society of University Cartographers*, 22(1):17–27, 1988.

Appendix A: C code for DPhull()

```
#define EPSILON         0.00000000000          /* Error tolerance */
#define EPSILON_SQ      0.00000000000000000    /* Error tolerance squared */

#define HULL_MAX        10000
#define TWICE_HULL_MAX  20000
#define THRICE_HULL_MAX 30000

#define PUSH_OP 0                              /* Operation names saved in history stack */
#define TOP_OP  1
#define BOT_OP  2

#define XX 0
#define YY 1
#define WW 2

typedef double POINT[2];
typedef double HOMOG[3];

typedef struct {                               /* Half of a Path Hull: elt is a double ended queue
  int top, bot,                                   storing a convex hull, top and bot are the two ends.
  hp, op[THRICE_HULL_MAX];                        The history stack is helt for points and op for opera-
  POINT *elt[TWICE_HULL_MAX],                     tions, hp is the stack pointer. */
        *helt[THRICE_HULL_MAX];
} PATH_HULL;



#define MIN(a,b) ( a < b ? a : b)
#define MAX(a,b) ( a > b ? a : b)
#define SGN(a) (a >= 0)

#define CROSSPROD_2CCH(p, q, r)                /* 2-d cartesian to homog cross product */
 (r)[WW] = (p)[XX] * (q)[YY] - (p)[YY] * (q)[XX];
 (r)[XX] = - (q)[YY] + (p)[YY];
 (r)[YY] =   (q)[XX] - (p)[XX];

#define DOTPROD_2CH(p, q)                      /* 2-d cartesian to homog dot product */
 (q)[WW] + (p)[XX]*(q)[XX] + (p)[YY]*(q)[YY]

#define LEFT_OF(a, b, c)                       /* Determine if point c is left of line a to b */
     (((*a)[XX] - (*c)[XX])*((*b)[YY] - (*c)[YY])
      >= ((*b)[XX] - (*c)[XX])*((*a)[YY] - (*c)[YY]))

#define SLOPE_SIGN(h, p, q, l)                 /* Return the sign of the projection of h[q] - h[p]
  SGN((l[XX])*((*h->elt[q])[XX] -                 onto the normal to line l */
      (*h->elt[p])[XX]) + (l[YY])*((*h->elt[q])[YY] - (*h->elt[p])[YY]))
```

```
#define Hull_Push(h, e)                          /* Push element e onto path hull h    */
  (h)->elt[++(h)->top] = (h)->elt[--(h)->bot] = (h)->helt[++(h)->hp] = e;
  (h)->op[(h)->hp] = PUSH_OP
#define Hull_Pop_Top(h)                          /* Pop from top                       */
  (h)->helt[++(h)->hp] = (h)->elt[(h)->top--];
  (h)->op[(h)->hp] = TOP_OP
#define Hull_Pop_Bot(h)                          /* Pop from bottom                    */
  (h)->helt[++(h)->hp] = (h)->elt[(h)->bot++];
  (h)->op[(h)->hp] = BOT_OP
#define Hull_Init(h, e1, e2)                     /* Initialize path hull and history   */
  (h)->elt[HULL_MAX] = e1;
  (h)->elt[(h)->top = HULL_MAX+1] = (h)->elt[(h)->bot = HULL_MAX-1]
        = (h)->helt[(h)->hp = 0] = e2;
  (h)->op[0] = PUSH_OP;



POINT *V;                                        /* Vertices of the input chain        */
int n;                                           /* Number of vertices in V            */

PATH_HULL *left, *right;                         /* The Path Hull: pointers to left hull, right hull,
POINT *PHtag;                                    and tag vertex PHtag.                 */



void Find_Extreme(h, line, e, dist)              /* Return e, the extreme vertex of the hull h with
     register PATH_HULL *h;                      respect to line. Return also its distance dist.  */
     HOMOG line;
     POINT **e;
     register double *dist;
{
  int sbase, sbrk, mid,
    lo, m1, brk, m2, hi;
  double d1, d2;

  if ((h->top - h->bot) > 6)                     /* If there are > 6 points on the hull (otherwise we
    {                                            will just look at them all.)          */
      lo = h->bot; hi = h->top - 1;
      sbase = SLOPE_SIGN(h, hi, lo, line);       /* The sign of the base edge          */
      do
        {
          brk = (lo + hi) / 2;                   /* Binary search for an edge with opposite sign  */
          if (sbase == (sbrk = SLOPE_SIGN(h, brk, brk+1, line)))
            if (sbase == (SLOPE_SIGN(h, lo, brk+1, line))) lo = brk + 1;
            else hi = brk;
        }
      while (sbase == sbrk);

      m1 = brk;                                  /* Now, the sign changes between the base edge and
      while (lo < m1)                            brk + 1. Binary search for the extreme point.    */
        {
          mid = (lo + m1) / 2;
          if (sbase == (SLOPE_SIGN(h, mid, mid+1, line))) lo = mid + 1;
          else m1 = mid;
        }
```

```
      m2 = brk;                                          /* The sign also changes between brk and the base.
      while (m2 < hi)                                    Binary search again.                          */
        {
          mid = (m2 + hi) / 2;
          if (sbase == (SLOPE_SIGN(h, mid, mid+1, line))) hi = mid;
          else m2 = mid + 1;
        }                                                /* Compute distances to extreme points found    */
      if ((d1 = DOTPROD_2CH(*h->elt[lo], line)) < 0) d1 = - d1;
      if ((d2 = DOTPROD_2CH(*h->elt[m2], line)) < 0) d2 = - d2;
      *dist = (d1 > d2 ? (*e = h->elt[lo], d1) : (*e = h->elt[m2], d2));
    }
  else                                                   /* Few points in hull—search by brute force     */
    {
      *dist = 0.0;
      for (mid = h->bot; mid < h->top; mid++)
        {
          if ((d1 = DOTPROD_2CH(*h->elt[mid], line)) < 0) d1 = - d1;
          if (d1 > *dist)
            { *dist = d1; *e = h->elt[mid]; }
        }
    }
}


void Hull_Add(h, p)                                      /* Add p to the hull h. Implements Melkman's convex
     register PATH_HULL *h;                              hull algorithm.                               */
     POINT *p;
{
  register int topflag, botflag;

  topflag = LEFT_OF(h->elt[h->top], h->elt[h->top-1], p);
  botflag = LEFT_OF(h->elt[h->bot+1], h->elt[h->bot], p);

  if (topflag || botflag)                                /* If the new point is outside the hull         */
    {
      while (topflag)                                    /* Pop points until convexity is ensured        */
        {
          Hull_Pop_Top(h);
          topflag = LEFT_OF(h->elt[h->top], h->elt[h->top-1], p);
        }
      while (botflag)
        {
          Hull_Pop_Bot(h);
          botflag = LEFT_OF(h->elt[h->bot+1], h->elt[h->bot], p);
        }
      Hull_Push(h, p);                                   /* Then push the new point on the top and bottom
    }                                                    of the deque                                  */
}
```

```
void Build(i, j)                        /* Build the Path Hull for the chain from vertex i
     POINT *i, *j;                      to vertex j.                                   */
{
  register POINT *k;

  PHtag = i + (j - i) / 2;              /* Make the middle vertex the tag              */
  Hull_Init(left, PHtag, PHtag - 1);    /* Build left hull                             */
  for (k = PHtag - 2; k >= i; k--) Hull_Add(left, k);
  Hull_Init(right, PHtag, PHtag + 1);   /* Build right hull                            */
  for (k = PHtag + 2; k <= j; k++) Hull_Add(right, k);
}



void Split(h, e)                        /* Split the hull h at the point e, leaving the hull
     register PATH_HULL *h;             from the tag to e.                             */
     POINT *e;
{
  register POINT *tmpe;
  register int tmpo;

  while ((h->hp >= 0)                   /* Loop until we reach the PUSH_OP for e.      */
         && ((tmpo = h->op[h->hp]), ((tmpe = h->helt[h->hp]) != e) || (tmpo != PUSH_OP)))
    {
      h->hp--;
      switch (tmpo)                     /* Undo the operation                          */
        {
        case PUSH_OP: h->top--; h->bot++; break;
        case TOP_OP: h->elt[++h->top] = tmpe; break;
        case BOT_OP: h->elt[--h->bot] = tmpe; break;
        }
    }
}



POINT *DPhull(i, j)                     /* Recursively simplify the chain from vertex i to j.
     POINT *i, *j;                      Returns the last point on the simplified chain. */
{
  POINT *lextr, *rextr;                 /* Extrema on left and right hulls and their dis-
  double ldist, rdist;                  tances                                         */
  HOMOG line;
  double len_sq;
  POINT * tmp;

  CROSSPROD_2CCH(*i, *j, line);         /* Compute line through i and j.               */
  len_sq = line[XX] * line[XX] + line[YY] * line[YY];

  if (j - i <= 1)                       /* If no vertices between i and j, then simplification
    return(j);                          is complete.                                   */
  else
    {
      Find_Extreme(left, line, &lextr, &ldist);
      Find_Extreme(right, line, &rextr, &rdist);
      if (ldist <= rdist)               /* See if a split occurs at the right or left  */
```

```
        if (rdist * rdist <= EPSILON_SQ * len_sq)
          return(j);                                    /* No split needed if within tolerance */
        else
          {
            if (PHtag == rextr) Build(i, rextr);        /* If the split occurs at the tag, rebuild otherwise
            else Split(right, rextr);                      split the end off the right. */
            OutputVertex(DPhull(i, rextr));             /* Simplify before the split. */
            Build(rextr, j);                            /* Rebuild path hull and simplify after split. */
            return(DPhull(rextr, j));
          }
      else
        if (ldist * ldist <= EPSILON_SQ * len_sq)
          return(j);                                    /* No split needed if within tolerance */
        else
          {
            Split(left, lextr);                         /* Split the end off the left. Simplify after the split,
            tmp = DPhull(lextr, j);                        and save the output for later. */
            Build(i, lextr);                            /* Rebuild path hull and simplify after split. */
            OutputVertex(DPhull(i, lextr));
            return(tmp);
          }
    }
}


void main()                                             /* Read in the points, build the first path hull, and
{                                                          call DPhull() to simplify. */
  n = Get_Points(V);                                    /* Get points and allocate memory */
  Init(V);
  Build(V, V + n - 1);                                  /* Build the initial path hull */
  OutputVertex(V);
  OutputVertex(DP(V, V + n - 1));                       /* Simplify */
}
```