On the bit complexity of minimum link paths:
Superquadratic algorithms for problems solvable in linear time

Simon Kahan
Tera Computer Company
Seattle, Washington, USA

Jack Snoeyink*
UBC Dept. of Comp. Sci.
Vancouver, BC, Canada

September 1, 1998

Abstract

All of the linear-time algorithms that have been developed for minimum-link paths use the real RAM model of computation. If one considers bit complexity, however, merely representing a minimum-link path may require a superquadratic number of bits. This paper considers bounds on the number of links (segments) needed by limited-precision approximations of minimum-link paths: When vertices are restricted to "first-derived" points, the number of links can increase by a constant factor; when they are restricted to points of an $N \times N$ grid, the number of links can increase by $\Theta(\log N)$.

1 Introduction

Researchers in computational geometry often use the real RAM model of computation [22] when developing algorithms. This can be forgiven when it leads to algorithms in which any derived value can be expressed as a constant-degree computation on a constant number of the inputs—such algorithms can be carefully implemented in some finite precision.

In other cases, however, the real RAM can obscure a problem's true complexity. Consider the problem of computing minimum-link paths: Given a simple polygon P containing two points s and t, compute a piecewise-linear path from s to t that does not intersect the exterior of P and has the minimum number of line segments. Suri [24, 25], Ghosh [6], and Hershberger and Snoeyink [10] are all guilty of "linear-time" algorithms for this problem. In Section 2 however, we give a simple instance in which representing path vertices with rational coordinates requires $\Theta(n^2 \log n)$ bits, even though the input polygon uses $\Theta(n \log n)$ bits. The same problem can be observed in work on link-centers [5, 14, 16], and minimum-link paths in polygons with holes [1, 9, 21].

*Supported in part by an NSERC Research Grant and a fellowship from the B.C. Advanced Systems Institute.

As a partial atonement for the second author, we investigate the cost (in the number of additional links) when the precision of vertices is limited. In Section 3, we show that by using only "first-derived points" one gets a path from s to t with at most twice the minimum number of links. In Section 4, we show that restricting the path to use only grid points increases its complexity by a logarithmic factor, which is tight.

We also look briefly at the algorithmic issues involved. These are closely related to "rounding" techniques for robust handling of arrangements of line segments, although we ask slightly different questions. See papers by Hobby [11, 12, 13], Milenkovic [17, 18, 19, 20], and Guibas et al. [7, 8] for more on arrangements of segments.

2 Exact minimum-link paths

Under the real RAM model of computation, a minimum-link path from s to t in P can be computed by a simple greedy algorithm. We let $\mu(P, s, t)$ denote the number of links in such a path. One can phrase the computation of $\mu(P, s, t)$ in terms of illumination: Imagine that P is a walled room and let region R_0 be the point s. Then, for $i \geq 1$, place lights at every point in R_{i-1} and let R_i be the region of P illuminated. Stop when t is found in R_i. Region R_i consists of all points of P that can be reached by an i-link path from s. To make the computation more efficient, if $P \setminus R_{i-1}$ is not connected, then we compute R_i only in the component of $P \setminus R_{i-1}$ that contains the destination t.

One difficulty in implementing such an algorithm is that extra precision can be required for the boundary of each region. The spiral in figure 1 is a simple example. The polygon is formed by two rectilinear spirals from origin that make right turns: an *inner* spiral with segments of Euclidean length 1, 1, 2, 3, ... and an *outer* spiral with segments of length 1, 2, 3, ...

The path drawn in figure 1 contains all chords that bound regions $R_1, R_2, \ldots, R_n$. Each non-grid endpoint of a chord must be computed from the previous, and each requires an additional $\Theta(\log n)$ bits on the average.

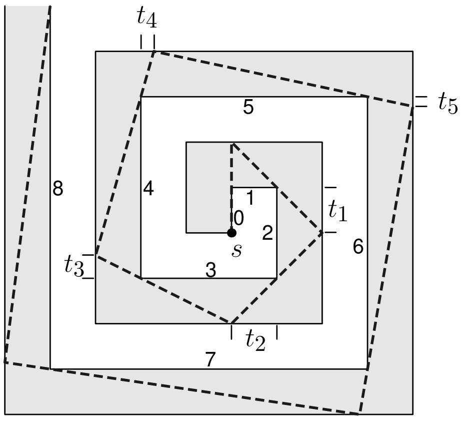

Figure 1: A spiraling path

Theorem 2.1 *Representing the regions $R_1, R_2, \ldots, R_n$ as polygons with rational coordinates requires $\Theta(n^2 \log n)$ bits.*

Proof: Number the first edge of the inner spiral with 0 and all later ones by their Euclidean length, as in figure 1. Let ℓ_k be the extension of the edge k inside P.

Starting illuminating from s. The first illuminated region R_1 is bounded by a chord that coincides with ℓ_0. For each subsequent R_{k+1}, the chord that bounds the illuminated region extends beyond the segment ℓ_k to a point on the outer spiral; Let t_k be the distance along the outer spiral from from ℓ_k to that last point of R_{k+1}. Because the kth chord, the outer spiral, and ℓ_k define similar triangles with the ratio $t_k/1 = 1/(k - t_{k-1})$, where $t_0 = 0$. Thus, $t_1 = t_2 = 1$, $t_3 = 1/2$, and $t_4 = 2/7$.

Expressing t_k as a ratio of two integers,

$$t_k = \frac{a_k}{b_k} = \frac{1}{k - \frac{a_{k-1}}{b_{k-1}}} = \frac{b_{k-1}}{kb_{k-1} - a_{k-1}.}$$

If we don't worry about common factors, we can equate numerators $a_k = b_{k-1}$ and obtain a recurrence for the denominators $b_0 = 1$, $b_1 = 1$, and, for $k \geq 2$, $b_k = kb_{k-1} - b_{k-2}$.

In fact, by induction on k, a_k and b_k have no common factors: For the base case, $a_1 = b_0 = 1$ and $b_1 = 1$ have 1 as the greatest common divisor. If $a_{k-1} = b_{k-2}$ and $b_k m$ are relatively prime, then so are $b_{k-1} = a_k$ and $kb_{k-1} - a_k m = b_k$. Therefore, $t_k = a_k/b_k$ is in lowest terms.

To give a lower bound on b_k, we note that $b_{k-1} \leq b_k$ by induction. Thus, b_k is at least $c_k = (k-1)c_{k-1} = (k-1)!$ for $k \geq 1$. It is easy to bound b_k from above by $k!$. The number of bits required to represent the denominator b_k is therefore $\Theta(k \log k)$. The total number of bits $\sum_{2 \leq k \leq n} \Theta(k \log k) = \Theta(n^2 \log n)$. ∎

3 Using first-derived points

The problem with representing the illuminated regions in the previous section is that the chord endpoints of R_n may by derived from the input data by an n-step process of deriving lines from points and then intersecting lines to form new points. (At least one algorithm [2] actually performs such an n step process to decompose a polygon for link-path queries.) One way to limit precision without explicitly considering which coordinates are representable is to limit ourselves to one step of this process.

We say that a *first-derived point* is an intersection point of two lines defined by original input data. Recall that $\mu(P, s, t)$ denotes the number of segments in a minimum-link path in P from s to t, and define $\mu_F(P, s, t)$ be the corresponding number when the vertices are also restricted to be original data points or first-derived points.

Theorem 3.1 *When path vertices are restricted to input data and first-derived points, $\mu_F(P, s, t) \leq 2\mu(P, s, t)$. Given P, we can compute, in $O(|P|)$ time, a path from s to t with at most $2\mu(P, s, t)$ segments.*

Proof: The Euclidean shortest path α from s to t in P can be obtained in linear time [3, 10, 15]. We can traverse the path α and label each vertex as a left or right turn. We call an edge $\overline{qu}$ of α an *inflection edge* if the labels of q and u differ; edges incident to s and to t are also called inflection edges. Because α turns only at reflex vertices of P, the line through $\overline{qu}$ is derived from input data.

Ghosh [6] (see also [10]) observed that a minimum-link path can always be made to have a link containing each inflection edge. Thus, we can reduce our problem to one of finding a minimum-link path between two segments, $\overline{uv}$ and $\overline{u'v'}$, that are extensions of inflection edges such that the shortest path α from u to u' is concave, as in figure 2.

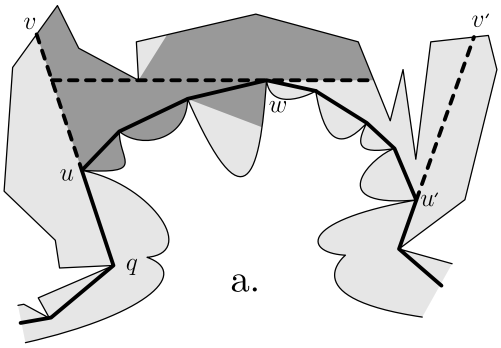

Figure 2: Greedily computing a minimum link path between inflection edges

We might have one of two easy cases: First, if the extension segments $\overline{uv}$ and $\overline{u'v'}$ intersect in P, then no additional segments are needed. Second, if the extension of some edge of α in P intersects $\overline{uv}$ and $\overline{u'v'}$, then one additional segment is necessary and sufficient. In both cases the vertices are first-derived points.

Otherwise, consider $\overline{uv}$ as a fluorescent light. The chord that bounds the region illuminated by $\overline{uv}$ is a tangent to α at a vertex w; it starts at either a vertex of P (figure 2a) or at v (drawn dotted in figure 2b). In the former case, the line through the chord is derived from input data, so it can be used as a new extension. In the latter, we extend the two edges of α incident to w to obtain a first-derived point on $\overline{uv}$ and a new extension based on input data. This replaces one link by two, as shown in dashed lines in figure 2b.

If the polygon P is triangulated, then this path can be computed in linear time using simple data structures [6, 10]. ■

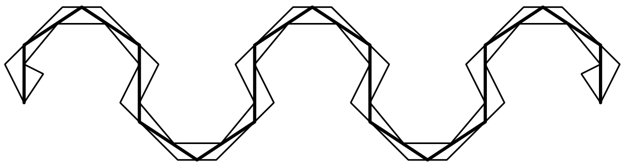

Figure 3: Using first-derived points requires 33% more links

Figure 3 shows an example polygon in which $\mu_F(P,s,t) = (4/3)\mu(P,s,t)$. The bold path shown uses two first-derived and one second-derived vertex between each inflection edge. To eliminate the second-derived vertex one must increase the number of edges. It is easy to construct this example on a polynomial-size grid; we have found it difficult to construct examples that more closely approach the upper bound's factor of two.

4 Using grid points

The first-derived points used in the previous section still require more precision than is present in the input data. In this section, we restrict the input and output coordinates to lie on an $N \times N$ grid. This restriction may increase the number of links in the minimum-link path by a factor of $\Theta(\log N)$.

Suppose that we are given a polygon P, whose vertices lie on an $N \times N$ grid, and grid points s and t in P. Recall that $\mu(P,s,t)$ denotes the number of segments in a minimum-link path in P from s to t. Let $\mu_G(P,s,t)$ be the number of segments in a minimum-link path whose vertices are also restricted to lie on the grid.

Theorem 4.1 *When path vertices are restricted to the $N \times N$ grid, $\mu_G(P,s,t) \leq \Theta(\log N)\mu(P,s,t)$, and there are polygons that achieve the lower bound. Given P we can compute a path from s to t with at most $\Theta(\log N)\mu(P,s,t)$ segments in time proportional to $O(|P| + (\log N)\mu(P,s,t))$.*

Our algorithm is based on computing convex hulls of grid points near the path. In the next four subsections, we go into some detail on the basic steps because they suggest the lower-bound construction and an efficient implementation. Section 4.1 reviews a well-known geometric interpretation of Euclid's GCD algorithm: *ApproxRay*() finds the best approximations of a line segment $\overline{ov}$ by rays through grid points that are in the bounding box of $\overline{ov}$. Section 4.2 describes a family of polygons P that establishes the lower bound $\mu_G(P,s,t) = \Omega(\log N)\mu(P,s,t)$. Section 4.3 details an efficient algorithm to identify all convex hull vertices below segment $\overline{ov}$. Finally, Section 4.4 approximates a minimum-link path.

4.1 Approximating a Segment by a Ray

Given a segment $\overline{ov}$ that starts from the origin (a grid point) and goes to a point v in the upper right quadrant, we will compute the convex hull of the grid points below $\overline{ov}$. To begin, algorithm *ApproxRay*(o,v) computes the directions of edges incident on o for the convex hulls above and below $\overline{ov}$. It uses a classical geometric construction based on continued fractions [4, ch. 4].

ApproxRay(o, v) computes the directions a and b of the rays from grid point o that best approximate $\overline{ov}$ from above and below. Rays a and b must be defined by grid points in the bounding box of $\overline{ov}$.

Procedure *ApproxRay*(o, v)
    $a \leftarrow (0,1)$; $b \leftarrow (1,0)$; $p \leftarrow a + b + o$;
    while $p.x < v.x$ and $p.y < v.y$ **do begin**
        if p is on $\overline{ov}$ **then** $a \leftarrow b \leftarrow a + b$; **break**;
        if p is above $\overline{ov}$ **then** $a \leftarrow a + b$;
        if p is below $\overline{ov}$ **then** $b \leftarrow a + b$;
        $p \leftarrow a + b + o$;
    end

Algorithm 1: Identifying rays that approximate $\overline{ov}$

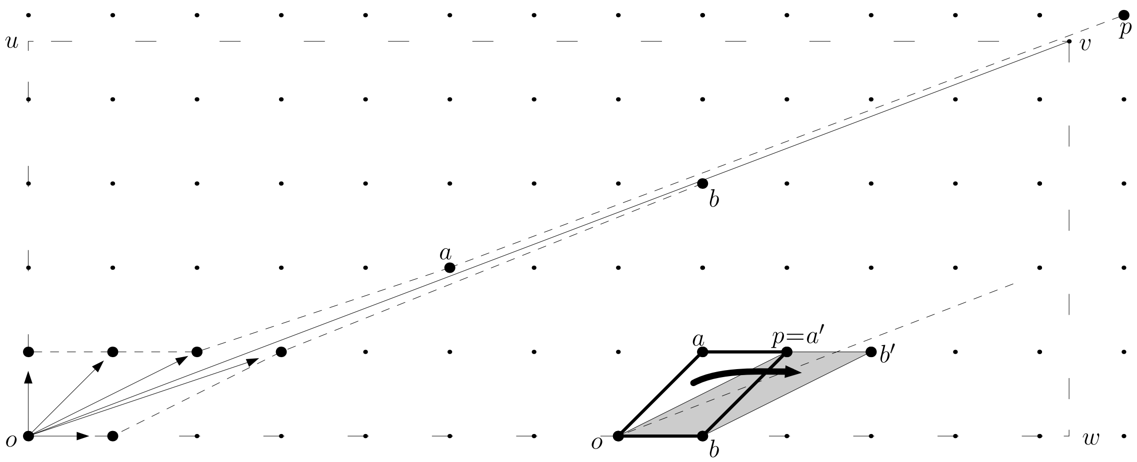

Figure 4: Candidates in $\Box ouvw$ computed by *ApproxRay*(o, v). At right, empty parallelograms.

Lemma 4.2 *In the 'while' loop of ApproxRay(o, v), the closed parallelogram with vertices o, $o + a$, p, and $o + b$ contains no grid points other than its vertices.*

Proof: We prove this by induction on the number of times through the 'while' loop. Initially, parallelogram G is the unit square whose lower left corner is o. If $p \in G$ is not on $\overline{ov}$, then G' is constructed by shifting triangle $\triangle o(o + a)p$ by b or $\triangle o(o + b)p$ by a. (See figure 4.) Since these triangles were empty in G, they remain so in G'. ■

Corollary 4.3 *On termination of ApproxRay(o, v), the angle formed by rays from o in directions a and b contains no grid points of $\Box ouvw$ in its interior.*

Proof: On termination of *ApproxRay*(o, v), p is outside of $\Box ouvw$. The parallelogram G, with vertices o, $o + a$, p, and $o + b$, is empty and translates of G by multiples of a or b cover the angle

$\angle(o+a)o(o+b)$ in $\Box ouvw$ with empty parallelograms whose vertices are outside $\Box ouvw$ or on the rays defining the angle. ■

Lemma 4.4 *In ApproxRay(o, v), the number of alternations between p being above and below $\overline{ov}$ is bounded by $\lceil \log_\phi M \rceil$, where M is the length of $\overline{ov}$.*

Proof: Assume that $o = (0,0)$. If p alternated between being above and below $\overline{ov}$ each iteration of the while loop, then we would alternate between assigning the sum $(a+b)$ to a and to b. The sequence of "p"s would therefore satisfy the Fibonacci recurrence: $p_{i+1} = p_i + p_{i-1}$. Since the coordinates of p are monotonically increasing between alternations as well, after k alternations the coordinates of p sum to at least the kth Fibonacci number $F_k = \lfloor \phi^k/\sqrt{5} \rfloor$, where $\phi = (\sqrt{5}+1)/2$ is the Golden Ratio. Thus, there are at most $\lceil \log_\phi M \rceil$ alternations. ■

4.2 A Lower Bound for Paths on the Grid

The observation that the alternations are related to the Fibonacci numbers suggests an example polygon that requires many links when restricted to a grid. (The grid has been rotated in Figure 5.)

Lemma 4.5 *There is a family of polygons for which P, on an $N \times N$ grid, satisfies $\mu_G(P, s, t) = \Theta(\log N)\mu(P, s, t)$.*

Proof: Define $F_0 = (0, 1)$ and $F_{i+1} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} F_i = F_i + F_{i-1}$. In words, F_{i+1} is the vector of the ith and $(i+1)$st Fibonacci numbers.

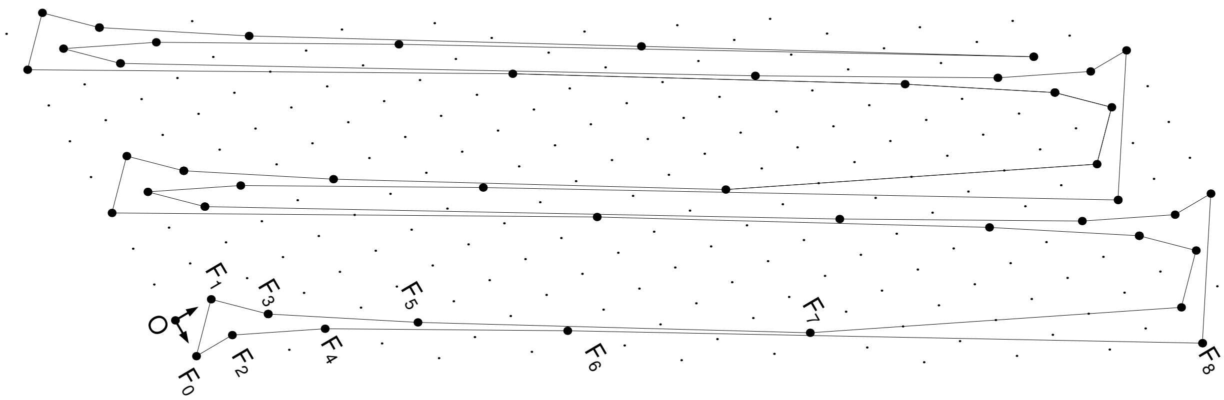

Figure 5: Using grid points increases the number of links by a factor of $\Theta(\log N)$

The basic element of Figure 5 is a polygon P_k in which the vertices of one side are the even F vectors up to F_k and the other the odd F vectors up to F_k. Note that the algorithm *ApproxRay*(o, v) generates the vectors F_i for $i < k$ when looking for candidates to approximate the vector F_k. Thus, P contains no grid points and has concave sides. If desired, we can connect basic elements as shown in the figure so that these properties are preserved. On an $N \times N$ grid, one can choose $k = \Theta(\log N)$. The unrestricted (or the first-derived) minimum-link path can traverse a basic element and get to the next using two line segments.

Vertices F_i, F_{i+2}, and F_{i+3} are co-linear: $F_{i+3} - F_{i+2} = F_{i+1}$ and $F_{i+2} - F_i = F_{i+1}$. Since F_{i+4} is not co-linear, the only vertices visible from F_i in P are F_{i-3} through F_{i+3}. Therefore, when the path is restricted to a grid—that is, to the vectors $F_0, \ldots, F_k$—it must use $\Theta(k) = \Theta(\log N)$ segments. ■

4.3 Efficiently Computing the Convex Hull of Grid Points Below $\overline{ov}$

Euclid's GCD algorithm, as embodied in *ApproxRay*(o, v), can be extended to compute the vertices of the convex hull of grid points below $\overline{ov}$ in logarithmic time. This extension requires us to correctly handle three things, which appear in Algorithms 2 and 3.

1. *ApproxRay*(o, v) finds the direction of the first hull vertex. The vertex itself must be found by a binary search as in Algorithm 2, since there could be many co-linear vertices in the indicated direction.
2. The running time of *ApproxRay*(o, v) is determined by the number of candidate points generated; unfortunately, this number can also be linear in the length of $\overline{ov}$. For example, if $v = (M, 1)$ and o the origin then all but three of the $(M + 3)$ candidates lie above $\overline{ov}$. We again use binary search to generate only the candidates that alternate sides of $\overline{ov}$; this is the idea of the Left-Shift Binary GCD algorithm. (See Sorenson [23] for a summary of GCD algorithms.)
3. The candidate directions for the second and subsequent vertices are a prefix of the candidate directions generated by *ApproxRay*(o, v) for the first hull vertex. By pushing these candidates on a stack in Algorithm 3, we can reuse them to generate all hull vertices.

Lemma 4.6 *HullVertices*(o, v) *computes the vertices of the convex hull of the grid points below* $\overline{ov}$ *and runs in* $O(\log M)$ *time, where* M *is the length of* $\overline{ov}$*.*

Proof: To establish correctness, we prove that we can reuse the candidates generated by *ApproxRay*(o, v) to obtain all vertices of the hull below $\overline{ov}$. Then we argue that *HullVertices*(o, v) obtains the same vertices despite its use of binary search to avoid inspecting all candidates. We end by establishing the running time.

According to Corollary 4.3, *ApproxRay*(o, v) computes directions a and b that form an angle whose interior is empty of grid points of $\Box ouvw$. The first half of *HullVertices*(o, v) computes the same directions—the only difference is that *FindLongest*() is used to skip over candidates on the same side of $\overline{ov}$ so that candidates alternate sides of $\overline{ov}$ on each iteration of the while

FindLongest(o, a, t) takes an origin o, a vector a, and a triangle t that contains o. It computes the greatest integer m such that $o + ma$ is also in t and returns ma. It takes $O(\log m)$ time.

```
Function FindLongest(o, a, t) returns vector;
    ℓ ← 0; h ← 1;
    while o + ha is not outside of △t do
        h ← 2h;                               /* Double to find search range */
    while ℓ + 1 < h do                        /* Invariant: (o + ℓa) ∈ t, */
        begin                                 /* and (o + ha) ∉ t */
            m ← ⌊(ℓ + h)/2⌋;
            if o + ma is outside of △t then h ← m
            else ℓ ← m
        end                                   /* end while ℓ + 1 < h */
    return ℓa;
```

Algorithm 2: Find farthest grid point (hull vertex) in particular direction.

loop. Thus, the first hull vertex, p_1, is correctly chosen as the farthest vertex in $\square ouvw$ in direction b.

Because there are no grid points of $\square ouvw$ in the angle $\angle vop$, running *ApproxRay*(p, v) in $\square ouvw$ gives a prefix of the sequence of candidates as running *ApproxRay*$(p, p + v)$ in $\square ouvw$, which is the same sequence as *ApproxRay*(o, v). (See figure 6.) If we had saved on a stack all the candidate directions, then to determine the next hull vertex p_{i+1} we could simply pop candidate directions until both $(p_i + a)$ and $(p_i + b)$ were in $\square ouvw$. The next hull vertex is the farthest vertex in $\square ouvw$ in direction b.

In *HullVertices*(o, v), we push only the candidate directions that follow an alternation; the other candidates can be recovered using *FindLongest*$()$. Specifically, we pop candidate pairs, a and b, until the x-coordinates of $(p_i + a)$ and $(p_i + b)$ are at most $v.x$—the pair obtained is the pair that would be generated by *ApproxRay*(p_i, v). The direction of the next hull vertex will be b if $(p_i + a + b)$ is either out of $\square ouvw$ or above $\overline{ov}$; we choose the farthest vertex in direction b in these cases. Otherwise, the first half of *HullVertices*$()$ skipped over a sequence of consecutive candidates of the form $b + ka$. We use *FindLongest*$()$ to determine the largest k such that $p_{i+1} = (p_i + b + ka)$ is still in $\square ouvw$. We know that p_{i+1} be below $\overline{ov}$ because the candidate above $\overline{ov}$ was on the stack but was rejected. We therefore know that $p_{i+1} + a$ will be outside $\square ouvw$ so that the current pair of candidates, a and b, need not be considered again. Finally, $p_i + 2(b + ka)$ is also outside $\square ouvw$ since $k \geq 1$. This proves that we have set the next hull vertex p_{i+1} correctly.

HullVertices(o,v) determines grid points in the bounding box $\Box ouvw$ that are vertices of the hull below $\overline{ov}$. It takes $O(\log \|ov\|)$ time.

```
Function HullVertices(o, v) returns list;
    a ← (0,1); b ← (1,0);
    p ← a + b + o;                        /* Run accelerated ApproxRay(o,v) */
    while p.x < v.x and p.y < v.y do begin
        Push(a, b, Stack);                          /* Save cand. for reuse */
        if p is on ov then a ← b ← (a + b); break;
        if p is above ov then
            a ← a + FindLongest(o + a, b, △ouv);
        if p is below ov then
            b ← b + FindLongest(o + b, a, △ovw);
        p ← a + b + o;
    end                                               /* end while p in □ */
    p0 ← o; i ← 1; p1 = o + FindLongest(o, b, △ovw);
    do                                       /* Find next convex hull vertex */
        Pop(a, b, Stack);
        if pi.x + a.x ≤ v.x and pi.x + b.x ≤ v.x then
            if (pi + b + a) is outside △ovw then
                pi+1 ← pi + FindLongest(pi, b, △ovw);
            else pi+1 ← pi + b + FindLongest(pi + b, a, △ovw);
        i ← i + 1;
until Stack is empty;
return p1, ..., pi;
```

Algorithm 3: Identify the grid points of $\triangle ovw$ that are vertices of the upper convex hull

Now, consider the time spent computing the first hull vertex, p_1. Because the number of iterations of the while loop is the number of alternations, $O(\log M)$, we need only to bound the time spent in *FindLongest*() up to a constant. From two iterations of the while loop that update a then b, we obtain a new $a' = a + m_1 b$ and new $b' = b + m_2 a' = m_2 a + (1 + m_2 m_1)b$, where m_1 and m_2 are the values computed in the two calls to *FindLongest*(). Thus, $|b'|/|b| > m_2 m_1$. If the call in the first iteration computes m_0, then $|b'| > m_0$, even if a is not updated. And if call to find p_1 computes m_k, then $|p_1 - o|/|b| = m_k$. The non-constant cost of the $O(\log M)$ calls to *FindLongest*() is therefore bounded by $\log |p - o| < O(\log M)$.

The time spent finding the rest of the hull vertices is proportional to the time spent finding the first: Each candidate pair of directions is considered once, and if a *FindLongest*() is performed, then it takes time proportional to $\log(|b'|/|b|)$ or $\log(|a'|/|a|)$, where the primed values were the most recently popped candidate vectors and the unprimed are current candidate vectors. The initial $|a| < M$ and $|b| < M$, so the cost of calls to *FindLongest*() is bounded by $O(\log M)$. ∎

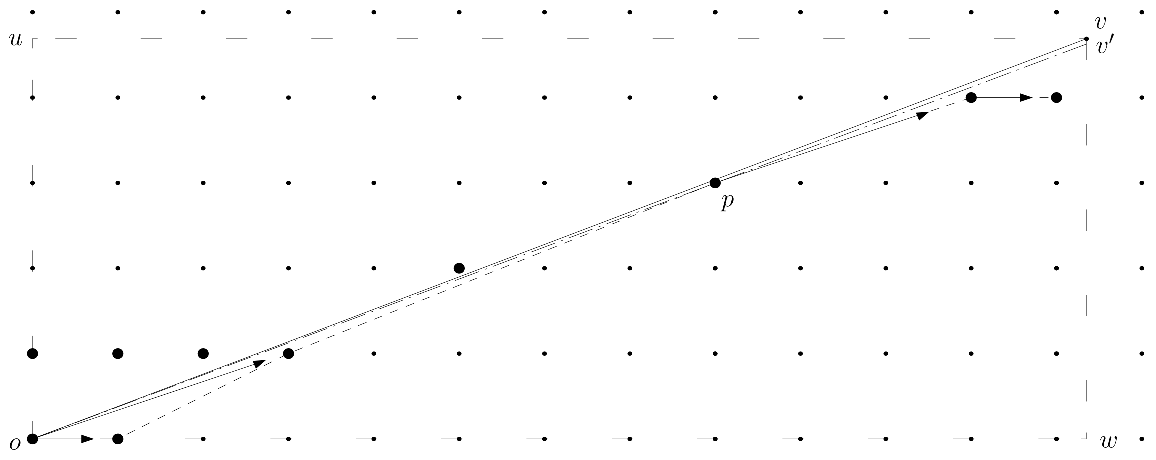

Figure 6: Reusing candidates

4.4 Approximating a Minimum-Link Path

Finally, we sketch how to approximate an entire minimum-link path from s to t in P. An interesting open question is the complexity of finding a minimum-link path among all paths whose vertices lie on grid points. While Theorem 4.1 says that a logarithmic increase is sufficient and may be necessary, we expect that the increase would typically be smaller.

For ease of exposition, we start with the approximation by "first-derived" points that is given in Section 3. This uses twice the minimum number of segments but ensures that every segment contains a polygon vertex. Thus, we have a corner $\angle ovq$ formed by two adjacent links with o and q at grid points, and we need to replace it with the convex hull of the grid points inside $\angle ovq$.

To approximate $\overline{ov}$, we run the algorithm of the previous section with the modification that whenever we test against $\square ouvw$, we also require our grid points to be in the halfspace of $\overleftrightarrow{vq}$ that contains o. Similarly restrict the approximation of $\overline{qv}$ to the halfspace of $\overleftrightarrow{ov}$ containing q. These approximations end with the same vertex or same (horizontal or vertical) line segment, where they can be joined. Each original segment is approximated by at most logarithmically-many segments, which shows that the lower bound of Section 4.2 is tight up to a multiplicative constant.

This construction completes the proof of Theorem 4.1.

5 Conclusion and open problems

We study limited-precision versions of the problem of computing a minimum-link path from s to t inside a simple polygon. We give upper and lower bounds on how the complexity of the solution (measured in number of line segments) can increase when vertices of the path are restricted to be first-derived points or grid points. We also discuss how to modify minimum-link path algorithms to produce paths whose increase in complexity is worst-case optimal. We found it interesting to note, in the problems we studied, that the solution complexity did increase, and that the increase could be bounded.

The paths that our limited-precision algorithms compute may not be minimum-link paths. Consider a polygon on an $N \times N$ grid that may have a k-link unrestricted path, for an example. Our algorithm guarantees to produces a path, restricted to the grid, that has at most $O(k \log N)$ links. However, there may be a restricted path with $2k$ or even $k+1$ links. We would find it interesting to know the time required to compute the minimum-link restricted path.

One can also look for more precise estimates for the increase in path complexity, especially for first-derived paths, or consider other precision limitations for other geometric problems.

Acknowledgement

We thank Michael Goodrich, John Hobby, Tom Shermer, and others for discussions on this problem; Preston Briggs, Michael McAllister, and Makoto Kobayashi for their comments on drafts of this paper.

References

[1] A. Aggarwal, H. Booth, J. O'Rourke, S. Suri, and C. K. Yap. Finding minimal convex nested polygons. *Information and Computation*, 83(1):98–110, Oct. 1989.

[2] E. M. Arkin, J. S. B. Mitchell, and S. Suri. Optimal link path queries in a simple polygon. In *Proceedings of the Third Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 269–279, 1992.

[3] B. Chazelle. A theorem on polygon cutting with applications. In *Proceedings of the 23rd IEEE Symposium on Foundations of Computer Science*, pages 339–349, 1982.

[4] H. Davenport. *The Higer Arithmetic: An Introduction to the Theory of Numbers.* Dover Publications, New York, 1983.

[5] H. N. Djidjev, A. Lingas, and J. Sack. An $O(n \log n)$ algorithm for computing the link center of a simple polygon. *Discrete & Computational Geometry*, 8(2):131–152, 1992.

[6] S. K. Ghosh. Computing the visibility polygon from a convex set and related problems. *Journal of Algorithms*, 12:75–95, 1991.

[7] M. Goodrich, L. J. Guibas, J. Hershberger, and P. Tanenbaum. Snap rounding line segments efficiently in two and three dimensions. In *Proceedings of the Thirteenth Annual ACM Symposium on Computational Geometry*, pages 284–293, 1997.

[8] L. Guibas and D. Marimont. Rounding arrangements dynamically. In *Proceedings of the Eleventh Annual ACM Symposium on Computational Geometry*, pages 190–199, 1995.

[9] L. J. Guibas, J. E. Hershberger, J. S. B. Mitchell, and J. S. Snoeyink. Approximating polygons and subdivisions with minimum-link paths. *International Journal of Computational Geometry & Applications*, 3(4):383–415, 1993.

[10] J. Hershberger and J. Snoeyink. Computing minimum length paths of a given homotopy class. *Computational Geometry: Theory and Applications*, 4:63–97, 1994.

[11] J. Hobby. Practical segment intersection with finite precision output. Manuscript, 1994.

[12] J. D. Hobby. Polygonal approximations that minimize the number of inflections. In *Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 93–102, 1993.

[13] J. D. Hobby. Space-efficient outlines from image data via vertex minimization and grid constraints. *CVGIP: Graphical Models and Image Processing*, 59(2), 1997.

[14] Y. Ke. An efficient algorithm for link-distance problems. In *Proceedings of the Fifth Annual ACM Symposium on Computational Geometry*, pages 69–78, 1989.

[15] D. T. Lee and F. P. Preparata. Euclidean shortest paths in the presence of rectilinear barriers. *Networks*, 14(3):393–410, 1984.

[16] W. Lenhart, R. Pollack, J.-R. Sack, R. Seidel, M. Sharir, S. Suri, G. T. Toussaint, S. Whitesides, and C. K. Yap. Computing the link center of a simple polygon. *Discrete & Computational Geometry*, 3:281–293, 1988.

[17] V. Milenkovic. Calculating approximate curve arrangements using rounded arithmetic. In *Proceedings of the Fifth Annual ACM Symposium on Computational Geometry*, pages 197–207, 1989.

[18] V. Milenkovic. Double precision geometry: a general technique for calculating line and segment intersections using rounded arithmetic. In *Proceedings of the 30th IEEE Symposium on Foundations of Computer Science*, pages 500–505, 1989.

[19] V. J. Milenkovic. *Verifiable Implementations of Geometric Algorithms Using Finite Precision Arithmetic*. PhD thesis, Carnegie–Mellon University, Pittsburg, Penn., 1988.

[20] V. J. Milenkovic. Practical methods for set operations on polygons using exact arithmetic. In *Proceedings of the Seventh Canadian Conference on Computational Geometry*, pages 55–60, Université Laval, Quebec, 1995.

[21] J. S. B. Mitchell, G. Rote, and G. Woeginger. Minimum-link paths among obstacles in the plane. *Algorithmica*, 8:431–459, 1992.

[22] F. P. Preparata and M. I. Shamos. *Computational Geometry—An Introduction*. Springer-Verlag, New York, 1985.

[23] J. Sorenson. Two fast GCD algorithms. *Journal of Algorithms*, 16:110–144, 1994.

[24] S. Suri. A linear time algorithm for minimum link paths inside a simple polygon. *Computer Vision, Graphics, and Image Processing*, 35:99–110, 1986.

[25] S. Suri. *Minimum link paths in polygons and related problems*. Ph.D. thesis, Dept. Comput. Sci., Johns Hopkins Univ., Baltimore, MD, 1987.

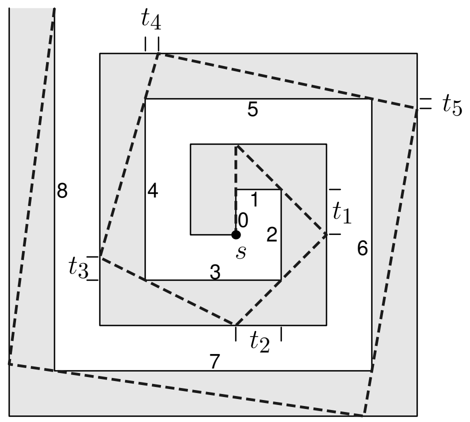

Figure 1: A spiraling path

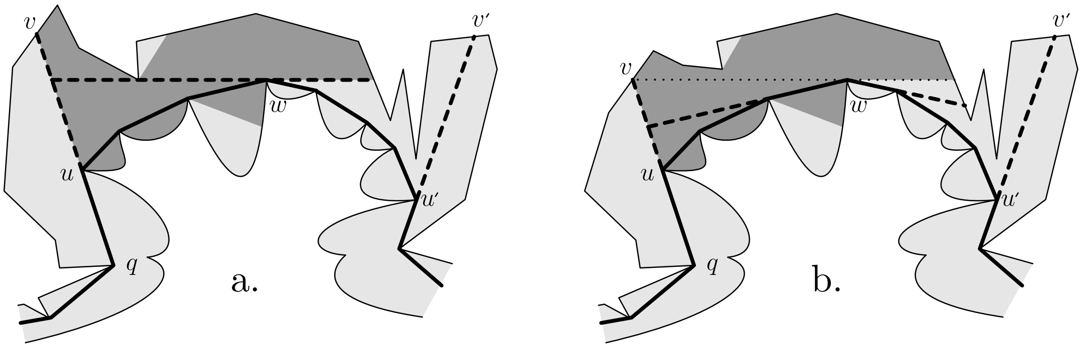

Figure 2: Greedily computing a minimum link path between inflection edges

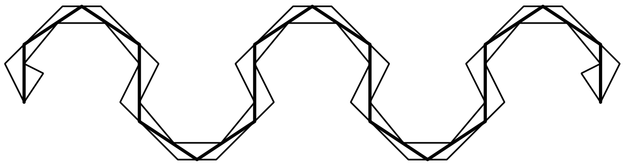

Figure 3: Using first-derived points requires 33% more links

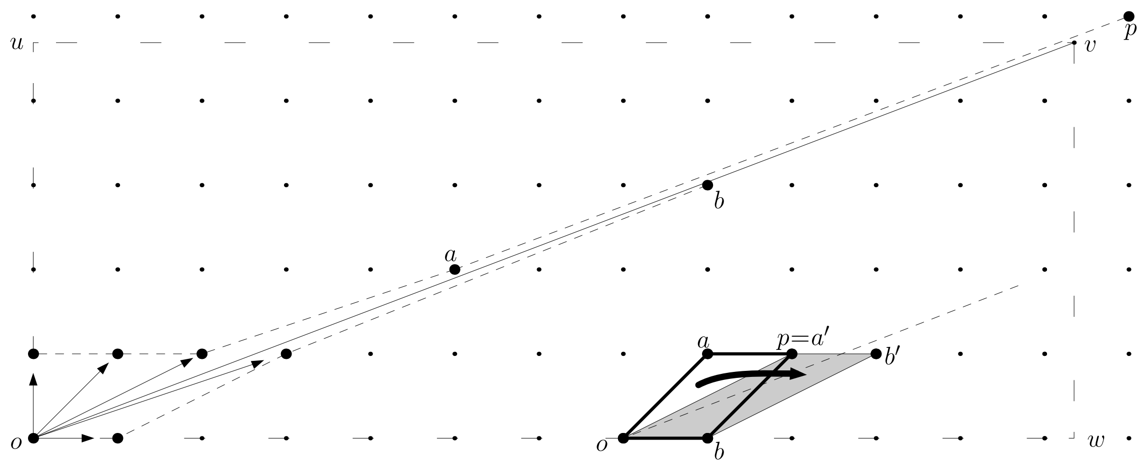

Figure 4: Candidates in $\square ouvw$ computed by $ApproxRay(o, v)$. At right, empty parallelograms.

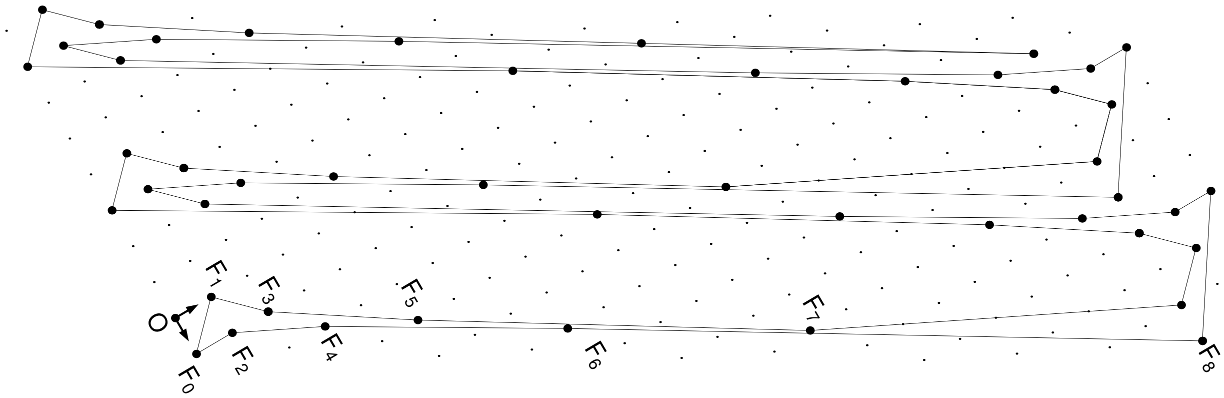

Figure 5: Using grid points increases the number of links by a factor of $\Theta(\log N)$

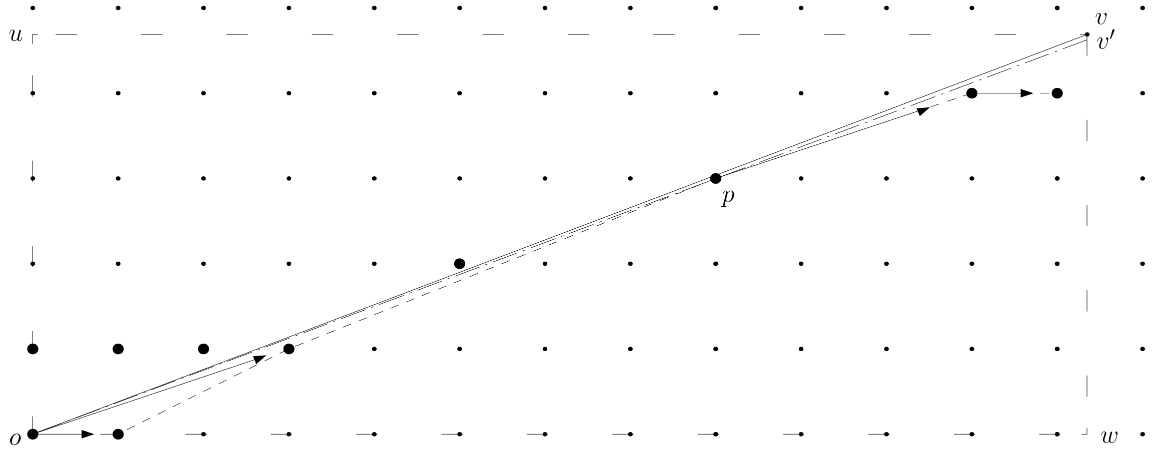

Figure 6: Reusing candidates