An Experimental Study of Algorithms for Geodesic Shortest Paths in the Constant Workspace Model*

Jonas Cleve† Wolfgang Mulzer†

Abstract

We evaluate experimentally algorithms for finding shortest paths in polygons in the constant workspace model. In this model, the input resides in a read-only array that can be accessed at random. In addition, the algorithm may use a constant number of words for reading and writing. The constant workspace model has been studied extensively in recent years, and algorithms for geodesic shortest paths have received particular attention.

We have implemented three such algorithms and compare them to the classic algorithm by Lee and Preparata that uses linear time and space. We also clarify implementation details that were missing in the original descriptions. Our experiments show that all algorithms perform as advertised and according to the theoretical guarantees. However, the constant factors in the running times turn out to be rather large for the algorithms to be practical.

1 Introduction

In recent years, the *constant workspace model* has enjoyed increasing popularity in computational geometry. Motivated by the increasing deployment of small devices with limited memory capacities, the goal is to develop simple and efficient algorithms for the situation where little workspace is available. The model posits that the input resides in a read-only array that can be accessed at random. In addition, the algorithm may use a constant number of memory words for reading and writing. The output must be written to a write-only memory that cannot be accessed again. Following the initial work from 2011 [2], numerous results have been published for this model, leading to a solid theoretical foundation for dealing with geometric problems when memory is scarce.

But how do these theoretical results measure up in practice? To investigate this question, we have implemented three different algorithms for computing geodesic shortest paths in simple polygons. This is one of the first problems to be studied in the constant workspace model. Given that the general shortest path problem is unlikely to be amenable to constant workspace algorithms, it may be a surprise that a solution for the geodesic case exists at all. By now, several algorithms are known, for constant workspace as well as in the time-space-trade-off regime [1, 8].

Due to the wide variety of approaches and the fundamental nature of the problem, geodesic shortest paths are a natural candidate for an experimental study. Our experiments show that all three algorithms work well in practice and live up to their theoretical guarantees. However, the large running times make them ill-suited for large input sizes.

During our implementation, we also noticed some missing details in the original publications, and we explain below how we have dealt with them.

2 The algorithms

We provide a brief overview over all implemented algorithms; further details can be found in the references. Let P be the input polygon and let $s, t \in P$ be the endpoints of the desired shortest path.

The algorithm by Lee and Preparata. In the classic algorithm, we triangulate P and find the triangles containing s and t. Next, we find the unique path in the dual graph of the triangulation between these two triangles. This gives a sequence $e_1, \ldots, e_m$ of diagonals crossed by the geodesic shortest path. The algorithm walks along these diagonals while maintaining a *funnel*. The funnel has a *cusp* p, initialized to s, and two concave *chains* from p to the two endpoints of the current diagonal e_i. In each step i, there are two cases: (i) if e_{i+1} remains visible from p, we update the appropriate concave chain, using Graham's scan; (ii) if e_{i+1} is not visible from p, we proceed along the appropriate chain until we find the cusp for the next funnel, and we output the vertices encountered along the way as part of the shortest path. Implemented properly, this takes linear time and space [10].

Delaunay. The first constant-workspace-algorithm, called *Delaunay*, directly adapts the method of Lee and Preparata to the constant-workspace model. It was proposed by Asano, Mulzer, and Wang [3] in 2011.

Since we cannot explicitly compute and store a triangulation of P, we use instead a unique implicit triangulation, the *constrained Delaunay triangulation* of P [6]. This triangulation can be navigated efficiently using constant workspace: given a diagonal or a polygon

*Supported by DFG projects MU/3501-1 and RO/2338-6.

†Institut für Informatik, Freie Universität Berlin, Germany.
{jonascleve,mulzer}@inf.fu-berlin.de

edge, we can find the incident triangles in $O(n^2)$ time. Using an $O(n)$ time constant-workspace-algorithm for shortest paths in trees, we then enumerate all triangles in the dual graph between the two triangles for s and t in $O(n^3)$ time.

As in Lee and Preparata [10], we need to maintain the visibility funnel while walking along the triangles. Instead of the whole chains, we store only the two line segments that define the current visibility cone, and we recompute the two chains when necessary. The total running time of the algorithm is $O(n^3)$.

Trapezoid. This algorithm was also published by Asano, Mulzer, and Wang [3]. It is based on the same principle as *Delaunay*, but it uses the trapezoidal decomposition of P [5]. Instead of walking along triangles, in $O(n^2)$ time per step, we walk along trapezoids, which takes $O(n)$ time per step. Since there are $O(n)$ steps, the running time improves to $O(n^2)$.

Makestep. This algorithm was presented by Asano et al. [2], and it uses a different approach. We maintain a *current vertex* p of the shortest path together with a *visibility cone*, defined by two points a and b on the boundary of P. The segments pa and pb cut off a subpolygon $P' \subseteq P$. The invariant is that t lies in P'. We gradually shrink P' by advancing a and b, sometimes also relocating p. A charging argument shows that there are $O(n)$ shrinking steps. Each step takes $O(n)$ time, for a total running time of $O(n^2)$.

3 Implementation

We have implemented the algorithms in Python [11]. Graphical output and plots use the `matplotlib` library [9]. Even though there are some geometry packages available for Python, none of them seemed suitable for our needs. Thus, we decided to implement all geometric primitives on our own. The source code of the implementation is available online[1].

For *Lee–Preparata*, we need a triangulation of P. Since polygon triangulation is not the main objective of our study, we relied for this on the *Python Triangle* library by Rufat [12], a wrapper for Shewchuk's *Triangle* [13]. *Triangle* does not provide a linear-time algorithm, but it implements Fortune's sweep, randomized incremental construction, and a divide-and-conquer algorithm, all with a running time of $O(n \log n)$. We used the divide-and conquer algorithm, the default choice. The triangulation phase is not included in the time and memory measurement.

General implementation details. All three constant-workspace algorithms have a general position assumption: *Delaunay* and *Makestep* assume that no three vertices lie on a line. Our implementations also assume general position but throw exceptions if a non-recoverable general position violation is encountered. Most violations, however, can be recovered; e.g. when trying to find the delaunay triangle(s) for a diagonal we can simply ignore points collinear to this diagonal. *Trapezoid* on the other hand assumes that no two vertices have the same x-coordinate. As described by Asano, Mulzer, and Wang [3], this can be fixed by changing the x-coordinate of every vertex to $x + \varepsilon y$ for some small ε such that the x-order of all vertices is maintained. We apply this fix to every polygon in which two vertices share the same x-coordinate.

The coordinates are stored as 64 bit IEEE 754 floats, and the coordinates of randomly generated polygons are rounded to four decimal places. To prevent precision or rounding problems we take the following steps: We never explicitly calculate angles but rely on the three-point-orientation test, i.e. the position of a point c relative to the line through points a and b. Additionally, if points need to be placed somewhere on a polygon edge, an edge reference is stored to account for inaccuracies when calculating the point's coordinates.

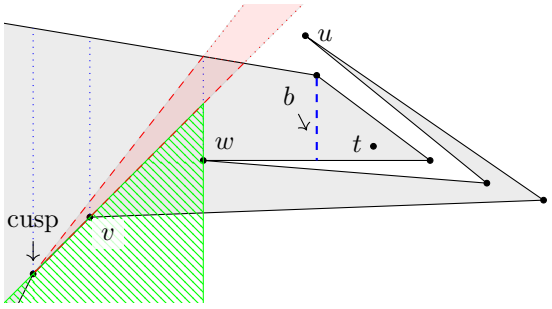

Figure 1: During the Jarvis march from the cusp to the diagonal b, the vertices need to be restricted to the shaded area. Otherwise, u would be considered part of the geodesic shortest path, as it is left of vw.

Delaunay and Trapezoid. In both algorithms, we need to find a piece of the shortest path as soon as the next diagonal is no longer visible from the current cusp. Asano, Mulzer, and Wang [3] only say that this should be done with a Jarvis march. During the implementation, we noticed that a naive Jarvis march with all vertices on P between the cusp and the next diagonal might include vertices that are not visible. Figure 1 shows an example: the vertex u would be included in the shortest path because it lies to the right of the cone and to the left of vw.

The solution for *Trapezoid* is to consider only vertices whose x-coordinate is between the cusp and the point where the visibility cone leaves P for the first time. For ease of implementation, one can also limit it to the x-coordinate of the last trapezoid boundary visible from the cusp. Figure 1 shows this region in

[1] `https://github.com/jonasc/constant-workspace-algos`

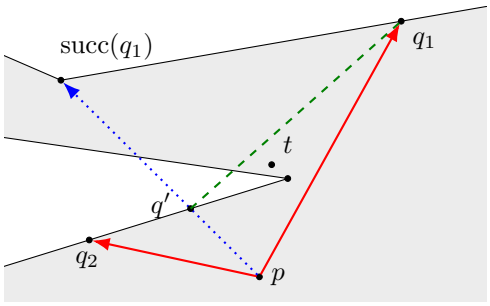

Figure 2: Asano et al. [2] state that one should check whether "t lies in the subpolygon from q' to q_1." Here, we should use q_1pq' to shrink the cutoff region.

green. For *Delaunay*, a similar approach can be used. The only difference is that the triangle boundaries are not all vertical lines.

Makestep. In our implementation of *Makestep*, we would like to point out an interesting detail; see Figure 2. The description by Asano et al. [2] says that to advance the visibiliy cone, we should check if "t lies in the subpolygon from q' to q_1." If so, the visibility cone should be shrunk to $q'pq_1$, otherwise to q_2pq'.

However, the "subpolygon from q' to q_1" is not clearly defined if the line segment $q'q_1$ is not contained in P. To avoid this difficulty, we consider pq' instead. This line segment is always in P, and it divides the cutoff region P' into two parts, a "subpolygon" between q' and q_1 and a "subpolygon" between q_2 and q'. Now we can easily choose the one containing t.

4 Experiments

Test set generation. Our experiments were conducted as follows: given a number of vertices n, we generate 4–10 random polygons, depending on n. For this, we use a tool developed in a software project at our department [7] which (among others) uses the *Space Partitioning* algorithm by Auer and Held [4].

For each edge e of each generated polygon, we find the incident triangle t_e of the constrained Delaunay triangulation. We add the barycenter of t_e to a point set S. Then, S has between $\lfloor n/2 \rfloor$ and $n-2$ points.

Test execution. For each pair of points from S, we find the shortest path using all of the implemented algorithms. Since the number of pairs grows quadratically, we restrict the tests to 1500 random pairs for all $n \geq 200$. We first run each algorithm once in order to determine the memory consumption. To obtain reproducible numbers, we disable the garbage collection functionality. After that, we run the algorithm between 5 and 20 times, depending on how long it takes. We measure the time for each run. We then take the median of the times as a representative for this point pair.

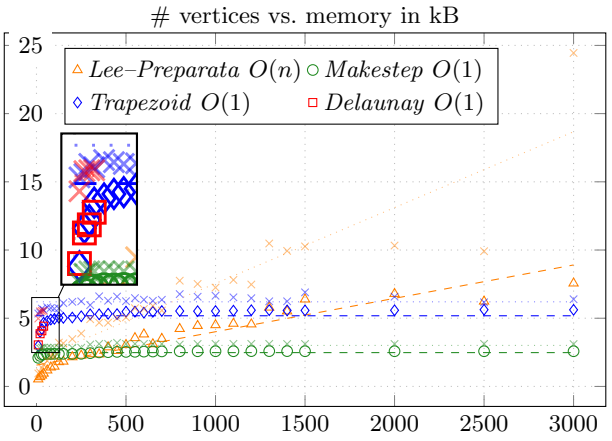

Figure 3: Memory consumption for random instances. The solid shapes are the median values; the transparent crosses are maximum values.

Test setup. Since we have a quadratic number of test cases, a lot of time is needed to run the tests. Thus the tests were distributed on multiple machines and on multiple cores. We had six computing machines, each with two quad-core CPUs. Three machines had Intel Xeon E5430 CPUs with 2.67 GHz; the other three had AMD Opteron 2376 CPUs with 2.3 GHz.

5 Results

The results of the experiments can be seen in the following plots. The plot in Figure 3 shows the median and maximum memory consumption as solid shapes and transparent crosses, respectively, for each algorithm and each input size. More precisely, the plot shows the median and the maximum over all polygons with a given size and over all pairs of points in each such polygon.

We observe that the memory consumption for *Trapezoid* and *Makestep* is always smaller than a certain constant. The shape of the median values might suggest logarithmic growth. However, a smaller number of vertices leads to a higher probability that s and t lie in the same triangle or can see each other. In this case, many geometric functions and subroutines, each of which requires an additional constant amount of memory, are not called. A large number of point pairs with only small memory consumption naturally entails a smaller median value.

The second plot in Figure 4 shows the median and the maximum running time in the same way as Figure 3. Not only does *Delaunay* have a cubic running time, but it also seems to have a quite large constant, as it grows much faster than the other algorithms.

In the lower part of Figure 4, we see the same x-domain, but with a much smaller y-domain. Here, we observe that *Trapezoid* and *Makestep* both have a quadratic running time; *Trapezoid* needs about two

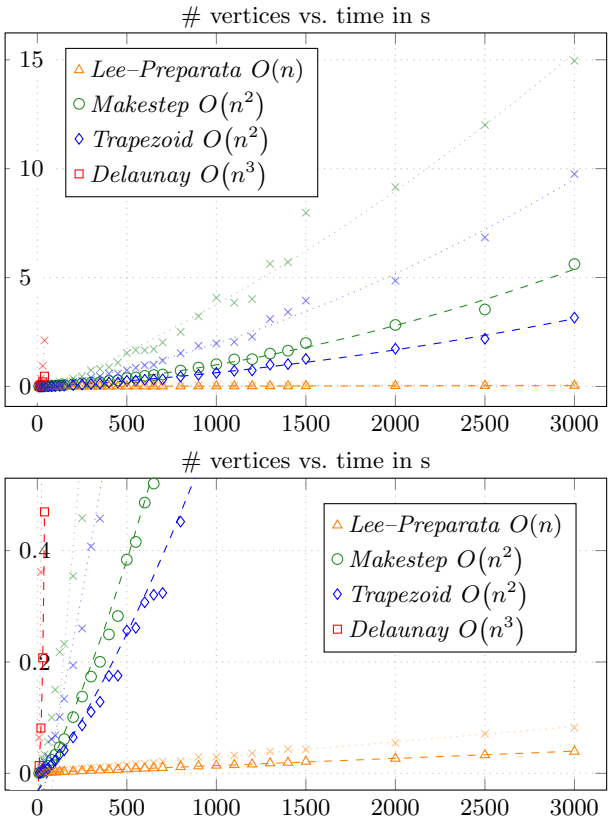

Figure 4: Runtime for random instances. Solid shapes are median values; transparent crosses are maximum values. The bottom plot is a scaled version of the top.

thirds of the time needed by *Makestep*. Finally, the linear-time behavior of *Lee–Preparata* is clearly visible.

We observed that the tests ran approximately 85 % slower on the AMD machines compared to the Intel servers. This reflects the difference between 2.3 GHz and 2.67 GHz. Since the tests were distributed equally on the machines it does not change the overall results.

6 Conclusion

We have implemented and experimented on three different constant-workspace algorithms for geodesic shortest paths in simple polygons. Not only did we observe the cubic worst-case running time of *Delaunay*, but we also noticed that the constant factor is rather large. This renders the algorithm useless already for polygons with a few hundred vertices, where the computation might, in the worst case, take several minutes.

As predicted by the theory, *Makestep* and *Trapezoid* exhibit the same asymptotic time and space consumption. *Trapezoid* has an advantage in the constant factor of the running time, while *Makestep* needs only about half as much memory. Since in both cases the memory requirement is bounded by a constant, *Trapezoid* would be our preferred algorithm.

We chose Python for the implementation mostly due to our experience, good debugging facilities, fast prototyping possibilities and the availability of numerous libraries. In hindsight, it might have been better to choose another programming language. Python's memory profiling and tracking abilities are limited, so that we cannot easily get a detailed view of the used memory with all the variables. Furthermore, a more detailed control of the memory management could be useful for performing more detailed experiments.

References

[1] T. Asano, K. Buchin, M. Buchin, M. Korman, W. Mulzer, G. Rote, and A. Schulz. "Memory-Constrained Algorithms for Simple Polygons". In: *CGTA* 46.8 (2013), pp. 959–969.

[2] T. Asano, W. Mulzer, G. Rote, and Y. Wang. "Constant-Work-Space Algorithms for Geometric Problems". In: *JoCG* 2.1 (2011), pp. 46–68.

[3] T. Asano, W. Mulzer, and Y. Wang. "Constant-Work-Space Algorithms for Shortest Paths in Trees and Simple Polygons". In: *JGAA* 15.5 (2011), pp. 569–586.

[4] T. Auer and M. Held. "Heuristics for the Generation of Random Polygons". In: *Proc. 8th Canada Conf. Comput. Geom.* Ottawa, 1996, pp. 38–43.

[5] M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars. *Computational Geometry*. Springer, 2008.

[6] L. P. Chew. "Constrained Delaunay Triangulations". In: *Algorithmica* 4 (1-4 1989), pp. 97–108.

[7] S. Dierker, M. Ehrhardt, J. Ihrig, M. Rohde, S. Thobe, and K. Tugan. *Abschlussbericht zum Softwareprojekt: Zufällige Polygone und kürzeste Wege*. Institut für Informatik, Freie Universität Berlin, Aug. 20, 2012. URL: `https://github.com/marehr/simple-polygon-generator`.

[8] S. Har-Peled. "Shortest Path in a Polygon Using Sublinear Space". In: *JoCG* 7.2 (2016), pp. 19–45.

[9] J. D. Hunter. "Matplotlib: A 2D Graphics Environment". In: *Computing In Science & Engineering* 9.3 (2007), pp. 90–95.

[10] D. T. Lee and F. P. Preparata. "Euclidean Shortest Paths in the Presence of Rectilinear Barriers". In: *Networks* 14.3 (Aut. 1984), pp. 393–410.

[11] Python Software Foundation. *Python*. Version 3.5. URL: `https://www.python.org/`.

[12] D. Rufat. *Python Triangle*. Version 20160203. 2016. URL: `http://dzhelil.info/triangle/` (visited on 12/05/2016).

[13] J. R. Shewchuk. "Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator". In: *Applied Computational Geometry towards Geometric Engineering*. Springer, 1996, pp. 203–222.