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

Jonas Cleve and Wolfgang Mulzer

Institut für Informatik, Freie Universität Berlin, Takustr. 9, 14195 Berlin, Germany
[jonascleve, mulzer]@inf.fu-berlin.de

Abstract. We perform an experimental evaluation of algorithms for finding geodesic shortest paths between two points inside a simple polygon 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 for 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 in Python, and we compare them to the classic algorithm by Lee and Preparata that uses linear time and linear space. We also clarify a few implementation details that were missing in the original description of the algorithms. Our experiments show that all algorithms perform as advertised in the original works 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 fully useful in practice.

Keywords: simple polygon · geodesic shortest path · constant workspace · experimental evaluation

1 Introduction

In recent years, the constant-workspace model has enjoyed growing popularity in the computational geometry community [6]. 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 for writing. The output must be written to a write-only memory that cannot be accessed again for reading. Following the initial work by Asano et al. from 2011 [2], numerous results have been published for this model, leading to a solid theoretical foundation for dealing with geometric problems when the working memory is scarce. The recent survey by Banyassady et

* Supported in part by DFG projects MU/3501-1 and RO/2338-6 and ERC StG 757609.
al.

But how do these theoretical results measure up in practice, particularly in view of the original motivation? To investigate this question, we have implemented three different constant-workspace algorithms for computing geodesic shortest paths in simple polygons. This is one of the first problems to be studied in the constant-workspace model [23]. Given that the general shortest path problem is unlikely to be amenable to constant-workspace algorithms (it is NL-complete [18]), it may come as a surprise that a solution for the geodesic case exists at all. By now, several algorithms are known, both for constant workspace as well as in the time-space-trade-off regime, where the number of available cells of working memory may range from constant to linear [1,12].

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

As far as we know, our study constitutes the first large-scale comparative evaluation of geometric algorithms in the constant-workspace model. A previous implementation study, by Baffier et al. [5], focused on time-space trade-offs for stack-based algorithms and was centered on different applications of a powerful algorithmic technique. Given the practical motivation and wide applicability of constant-workspace algorithms for geometric problems, we hope that our work will lead to further experimental studies in this direction.

2 The Four Shortest-Path Algorithms

We provide a brief summary for each of the four algorithms in our implementation; further details can be found in the original papers [3,2,14]. In each case, we use $P$ to denote a simple input polygon in the plane with $n$ vertices. We consider $P$ to be a closed, connected subset of the plane. Given two points $s, t \in P$, our goal is to compute a shortest path from $s$ to $t$ (with respect to the Euclidean length) that lies completely inside $P$.

2.1 The Classic Algorithm by Lee and Preparata

This is the classic linear-space algorithm for the geodesic shortest path problem that can be found in textbooks [14,11]. It works as follows: we triangulate $P$, and we find the triangle that contains $s$ and the triangle that contains $t$. Next, we determine the unique path between these two triangles in the dual graph of the triangulation. The path is unique since the dual graph of a triangulation of a simple polygon is a tree [7]. We obtain a sequence $e_1, \ldots, e_m$ of diagonals (incident to pairs of consecutive triangles on the dual path) crossed by the geodesic shortest
path between \( s \) and \( t \), in that order. The algorithm walks along these diagonals, while maintaining a *funnel*. The funnel consists of a *cusp* \( p \), initialized to be \( s \), and two concave *chains* from \( p \) to the two endpoints of the current diagonal \( e_i \). An example of these funnels can be found in Fig. 1. In each step \( i \) of the algorithm, \( i = 1, \ldots, m - 1 \), we update the funnel for \( e_i \) to the funnel for \( e_{i+1} \).

There are two cases: (i) if \( e_{i+1} \) remains visible from the cusp \( p \), we update the appropriate concave chain, using a variant of Graham’s scan; (ii) if \( e_{i+1} \) is no longer visible from \( p \), we proceed along the appropriate chain until we find the cusp for the next funnel. We output the vertices encountered along the way as part of the shortest path. Implemented in the right way, this procedure takes linear time and space.

### 2.2 Using Constrained Delaunay-Triangulations

The first constant-workspace-algorithm for geodesic shortest paths in simple polygons was presented by Asano et al. [3] in 2011. It is called *Delaunay*, and it constitutes a relatively direct adaptation of the method of Lee and Preparata to the constant-workspace model.

In the constant-workspace model, we cannot explicitly compute and store a triangulation of \( P \). Instead, we use a uniquely defined implicit triangulation of \( P \), namely the *constrained Delaunay triangulation* of \( P \) [9]. In this variant of the classic Delaunay triangulation, we prescribe the edges of \( P \) to be part of the desired triangulation. Then, the additional triangulation edges cannot cross the prescribed edges. Thus, unlike in the original Delaunay triangulation, the circumcircle of a triangle may contain other vertices of \( P \), as long as the line segment from a triangle endpoint to the vertex crosses a prescribed polygon edge, see Fig. 2 for an example.

\(^1\) If a triangulation of \( P \) is already available, the implementation is relatively straightforward. If not, a linear-time implementation of the triangulation procedure constitutes a significant challenge [8]. Simpler methods are available, albeit at the cost of a slightly increased running time of \( O(n \log n) \) [7].
The constrained Delaunay triangulation of \( P \) can be navigated efficiently using constant workspace: given a diagonal or a polygon edge, we can find the two incident triangles in \( O(n^2) \) time \[3\]. Using an \( O(n) \) time constant-workspace-algorithm for finding shortest paths in trees, also given by Asano et al. \[3\], we can thus enumerate all triangles in the dual path between the constrained Delaunay triangle that contains \( s \) and the constrained Delaunay triangle that contains \( t \) in \( O(n^3) \) time.

As in the algorithm by Lee and Preparata, we need to maintain the visibility funnel while walking along the dual path of the constrained Delaunay triangulation. Instead of the complete chains, we store only the two line segments that define the current visibility cone (essentially the cusp together with the first vertex of each chain). We recompute the two chains whenever it becomes necessary. The total running time of the algorithm is \( O(n^3) \). More details can be found in the paper by Asano et al. \[3\].

### 2.3 Using Trapezoidal Decompositions

This algorithm was also proposed by Asano et al. \[3\], as a faster alternative to the algorithm that uses constrained Delaunay triangulations. It is based on the same principle as Delaunay, but it uses the trapezoidal decomposition of \( P \) instead of the Delaunay triangulation \[7\]. See Fig. 3 for a depiction of the decomposition and the symbolic perturbation method to avoid a general position assumption. In the algorithm, we compute a trapezoidal decomposition of \( P \), and we follow the dual path between the trapezoid that contains \( s \) and the trapezoid that contains \( t \), while maintaining a funnel and outputting the new vertices of the geodesic shortest path as they are discovered. Assuming general position, we can find all incident trapezoids of the current trapezoid and determine how to continue on the way to \( t \) in \( O(n) \) time (instead of \( O(n^2) \) time in the case of the Delaunay algorithm). Since there are still \( O(n) \) steps, the running time improves to \( O(n^2) \).
2.4 The Makestep Algorithm

This algorithm was presented by Asano et al. [2]. It uses a direct approach to the geodesic shortest path problem and unlike the two previous algorithms, it does not try to mimic the algorithm by Lee and Preparata. In the traditional model, this approach would be deemed too inefficient, but in the constant-workspace world, its simplicity turns out to be beneficial. The main idea is as follows: we maintain a current vertex $p$ of the geodesic shortest path, together with a visibility cone, defined by two points $q_1$ and $q_2$ on the boundary of $P$. The segments $pq_1$ and $pq_2$ cut off a subpolygon $P' \subseteq P$. We maintain the invariant that the target $t$ lies in $P'$. In each step, we gradually shrink $P'$ by advancing $q_1$ and $q_2$, sometimes also relocating $p$ and outputting a new vertex of the geodesic shortest path. These steps are illustrated in Fig. 4. It is possible to realize the shrinking steps in such a way that there are only $O(n)$ of them. Each shrinking step takes $O(n)$ time, so the total running time of the MakeStep algorithm is $O(n^2)$.

3 Our Implementation

We have implemented the four algorithms from Section 2 in Python [15]. For graphical output and for plots, we use the matplotlib library [13]. Even though there are some packages for Python that provide geometric objects such as line segments, circles, etc., 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 in a Git-repository [2].

In order to apply the algorithm Lee-Preparata, we must be able to triangulate the simple input polygon $P$ efficiently. Since implementing an efficient polygon

---

2 https://github.com/jonasc/constant-workspace-algos
triangulation algorithm can be challenging and since this is not the main objective of our study, we relied for this on the Python Triangle library by Rufat [16], a Python wrapper for Shewchuk’s Triangle, which was written in C [17]. We note that Triangle does not provide a linear-time triangulation algorithm, which would be needed to achieve the theoretically possible linear running time for the shortest path algorithm. Instead, it contains three different implementations, namely Fortune’s sweep line algorithm, a randomized incremental construction, and a divide-and-conquer method. All three implementations give a running time of $O(n \log n)$. For our study, we used the divide-and-conquer algorithm, the default choice. In the evaluation, we did not include the triangulation phase in the time and memory measurement for running the algorithm by Lee and Preparata.

3.1 General Implementation Details

All three constant-workspace algorithms have been presented with a general position assumption: Delaunay and Makestep assume that no three vertices lie on a line, while Trapezoid assumes that no two vertices have the same $x$-coordinate. Our implementations of Delaunay and Makestep also assume general position, but they throw exceptions if a non-recoverable general position violation is encountered. Most violations, however, can be dealt with easily in our code; e.g. when trying to find the constrained Delaunay triangle(s) for a diagonal, we can simply ignore points collinear to this diagonal. For the case of Trapezoid, Asano et al. [3] described how to enforce the general position assumption by changing the $x$-coordinate of every vertex to $x + \varepsilon y$ for some small enough $\varepsilon > 0$ such that the $x$-order of all vertices is maintained. In our implementation, we apply this method to every polygon in which two vertices share the same $x$-coordinate.
The coordinates are stored as 64 bit IEEE 754 floats. In order to prevent problems with floating point precision or rounding, we take the following steps: first, we never explicitly calculate angles, but we rely on the usual three-point-orientation test, i.e., the computation of a determinant to find the position of a point $c$ relative to the directed line through to points $a$ and $b$ [7]. Second, if an algorithm needs to place a point somewhere in the relative interior of a polygon edge, we store an additional edge reference to account for inaccuracies when calculating the new point’s coordinates.

### 3.2 Implementing the Algorithm by Lee and Preparata

The algorithm by Lee and Preparata can be implemented easily, in a straightforward fashion. There are no particular edge cases or details that we need to take care of. Disregarding the code for the geometric primitives, the algorithm needs less than half as many lines of code than the other algorithms.

### 3.3 Implementing Delaunay and Trapezoid

In both constant-workspace adaptations of the algorithm by Lee and Preparata, we encounter the following problem: whenever the cusp of the current funnel changes, we need to find the cusp of the new funnel, and we need to find the piece of the geodesic shortest path that connects the former cusp to the new cusp. In their description of the algorithm, Asano et al. [3] only say that this should be done with an application of gift wrapping (Jarvis’ march) [7]. While implementing these two algorithms, we noticed that a naive gift wrapping step that considers all the vertices on $P$ between the cusp of the current funnel and the next diagonal might include vertices that are not visible inside the polygon. Figure 5 shows an example: here $b$ is the next diagonal, and naively we would look at all vertices along the polygon boundary between $v$ and $w$. Hence, $u$ would be...
Asano et al. [2] state that one should check whether “t lies in the subpolygon from q’ to q_1.” This subpolygon, however, is not clearly defined as the line segment q’q_1 does not lie inside P. Considering pq’ instead and using q_1pq’ to shrink the cutoff region gives the correct result on the right.

considered as a gift wrapping candidate, and since it forms the largest angle with the cusp and v (in particular, an angle that is larger than the angle formed by w) it would be chosen as the next point, even though w should be the cusp of the next funnel. A simple fix for this problem would be an explicit check for visibility in each gift-wrapping step. Unfortunately, the resulting increase in the running time would be too expensive for a realistic implementation of the algorithms.

Our solution for Trapezoid is to consider only vertices whose x-coordinate is between the cusp of the current vertex and the point where the current visibility cone crosses the boundary of 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 5 shows this as the dotted green region. For Delaunay, a similar approach can be used. The only difference is that the triangle boundaries in general are not vertical lines.

3.4 Implementing Makestep

Our implementation of the Makestep algorithm is also relatively straightforward. Nonetheless, we would like to point out one interesting detail; see Fig. The description by Asano et al. [2] says that to advance the visibility 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 for the case that the line segment q’q_1 is not contained in P. To avoid this difficulty, we instead consider the line segment pq’. This line segment is always contained 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 Experimental Setup

We now describe how we conducted the experimental evaluation of our four implementations for geodesic shortest path algorithms.

4.1 Generating the Test Instances

Our experimental approach is as follows: given a desired number of vertices $n$, we generate 4–10 (pseudo)random polygons with $n$ vertices. For this, we use a tool developed in a software project carried out under the supervision of Günter Rote at the Institute of Computer Science at Freie Universität Berlin [10]. Among others, the tool provides an implementation of the Space Partitioning algorithm for generating random simple polygons presented by Auer and Held [4].

Next, we generate the set $S$ of desired endpoints for the geodesic shortest paths. This is done as follows: for each edge $e$ of each generated polygon, we find the incident triangle $t_e$ of $e$ in the constrained Delaunay triangulation of the polygon. We add the barycenter of $t_e$ to $S$. In the end, the set $S$ will have between $\lfloor n/2 \rfloor$ and $n - 2$ points. We will compute the geodesic shortest path for each pair of distinct points in $S$.

4.2 Executing the Tests

For each pair of points $s, t \in S$, we find the geodesic shortest path between $s$ and $t$ using each of the four implemented algorithms. Since the number of pairs grows quadratically in $n$, we restrict the tests to 1500 random pairs for all $n \geq 200$.

First, we run each algorithm once in order to assess the memory consumption. This is done by using the get_traced_memory function of the built-in tracemalloc module which returns the peak and current memory consumption—the difference tells us how much memory was used by the algorithm. Starting the memory tracing just before running the algorithm gives the correct values for the peak memory consumption. In order to obtain reproducible numbers we also disable Python’s garbage collection functionality using the built-in gc.disable and gc.enable functions.

After that, we run the algorithm between 5 and 20 times, depending on how long it takes. We measure the processor time for each run with the process_time function of the time module which gives the time during which the process was active on the processor in user and in system mode. We then take the median of the times as a representative running time for this point pair.

4.3 Test Environment

Since we have a quadratic number of test cases for each instance, our experiments take a lot of time. Thus, the tests were distributed on multiple machines and on multiple cores. We had six computing machines at our disposal, 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. All machines had 32 GB RAM, even though, as can be seen in the next section, memory was never an issue. The operating system was a Debian 8 and we used version 3.5 of the Python interpreter to implement the algorithms and to execute the tests.

5 Experimental Results

The results of the experiments can be seen in the following plots. The plot in Figure 7 shows the median and maximum memory consumption as solid shapes and transparent crosses, respectively, for each algorithm and for 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 for Makestep is always smaller than a certain constant. At first glance, the shape of the median values might suggest logarithmic growth. However, a smaller number of vertices leads to a higher probability that $s$ and $t$ are directly visible to 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
Fig. 8: Runtime for random instances. Outlined shapes are median values; semi-transparent crosses are maximum values. The bottom plot is a scaled version of the top.
value. We can observe a very similar effect in the memory consumption of the Lee-Preparata algorithm for small values of \( n \). However, as \( n \) grows, we can see that the memory requirement begins to grow linearly with \( n \).

The second plot in Figure 8 shows the median and the maximum running time in the same way as Figure 7. Not only does Delaunay have a cubic running time, but it also seems to exhibit a quite large constant: it grows much faster than the other algorithms.

In the lower part of Figure 8, 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 thirds of the time required by Makestep. Finally, the linear-time behavior of Lee-Preparata can clearly be discerned.

Additionally, we observed that the tests ran approximately 85\% slower on the AMD machines than on the Intel servers. This reflects the difference between the clock speeds of 2.3 GHz and 2.67 GHz. Since the tests were distributed equally on the machines, this does not change the overall qualitative results and the comparison between the algorithms.

6 Conclusion

We have implemented and experimented with 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 virtually useless already for polygons with a few hundred vertices, where the shortest path computation might, in the worst case, take several minutes.

As predicted by the theory, Makestep and Trapezoid exhibit the same asymptotic running 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 previous programming 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 that allows for more low-level control of the underlying hardware. 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. Asano, T., Buchin, K., Buchin, M., Korman, M., Mulzer, W., Rote, G., Schulz, A.: Memory-constrained algorithms for simple polygons. Comput. Geom. Theory Appl. 46(8), 959–969 (2013)
2. Asano, T., Mulzer, W., Rote, G., Wang, Y.: Constant-work-space algorithms for geometric problems. J. of Computational Geometry 2(1), 46–68 (2011)
3. Asano, T., Mulzer, W., Wang, Y.: Constant-work-space algorithms for shortest paths in trees and simple polygons. J. Graph Algorithms Appl. 15(5), 569–586 (2011)
4. Auer, T., Held, M.: Heuristics for the generation of random polygons. In: Proc. 8th Canad. Conf. Comput. Geom. (CCCG). pp. 38–43 (1996)
5. Buffier, J.F., Diez, Y., Korman, M.: Experimental study of compressed stack algorithms in limited memory environments. In: Proc. 17th Inter. Symp. Experimental Algorithms, (SEA). pp. 19:1–19:13 (2018)
6. Banyassady, B., Korman, M., Mulzer, W.: Computational geometry column 67. SIGACT News 49(2), 77–94 (2018)
7. de Berg, M., Cheong, O., van Kreveld, M., Overmars, M.: Computational Geometry. Theory and Applications. Springer-Verlag, 3rd edn. (2008)
8. Chazelle, B.: Triangulating a simple polygon in linear time. Discrete Comput. Geom. 6, 485–524 (1991)
9. Chew, L.P.: Constrained Delaunay triangulations. Algorithmica 4, 97–108 (1989)
10. Dierker, S., Ehrhardt, M., Ihrig, J., Rohde, M., Thobe, S., Tugan, K.: Abschlussbericht zum Softwareprojekt: Zufällige Polygone und kürzeste Wege (2012), https://github.com/marehr/simple-polygon-generator Institut für Informatik, Freie Universität Berlin
11. Ghosh, S.K.: Visibility algorithms in the plane. Cambridge University Press (2007)
12. Har-Peled, S.: Shortest path in a polygon using sublinear space. J. of Computational Geometry 7(2), 19–45 (2016)
13. Hunter, J.D.: Matplotlib: A 2D graphics environment. Computing in Science & Engineering 9(3), 90–95 (2007)
14. Lee, D.T., Preparata, F.P.: Euclidean shortest paths in the presence of rectilinear barriers. Networks 14(3), 393–410 (1984)
15. Python Software Foundation: Python, https://www.python.org/, version 3.5
16. Rufat, D.: Python Triangle (2016), http://dzhelil.info/triangle/, version 20160203
17. Shewchuk, J.R.: Triangle: Engineering a 2D quality mesh generator and Delaunay triangulator. In: Workshop on Applied Computational Geometry, Towards Geometric Engineering (WACG), pp. 203–222 (1996)
18. Tantau, T.: Logspace optimization problems and their approximability properties. Theoret. Comput. Sci. 411(2), 327–350 (2007)

A Tables of Experimental Results

Here we list the experimental results shown in Figs. 7 and 8.
Table 1: The median and maximum memory usage in bytes for all runs with a specific number of vertices $n$.

| $n$ | Delaunay median | Delaunay max | Lee-Preparata median | Lee-Preparata max | Makestep median | Makestep max | Trapezoid median | Trapezoid max |
|-----|-----------------|--------------|----------------------|-------------------|-----------------|--------------|-----------------|---------------|
| 10  | 3048            | 4976         | 528                  | 952               | 2096           | 2552         | 2976            | 5344          |
| 20  | 3864            | 5512         | 696                  | 1032              | 2240           | 2776         | 3992            | 5432          |
| 30  | 4080            | 5536         | 808                  | 1360              | 2344           | 2840         | 4208            | 5704          |
| 40  | 4416            | 5536         | 952                  | 1592              | 2344           | 2840         | 4672            | 5616          |
| 60  | 1184            | 1872         | 2384                 | 2840              | 2976           | 5344         | 4904            | 5752          |
| 80  | 1464            | 2200         | 2392                 | 2840              | 4952           | 5704         | 4952            | 5704          |
| 100 | 1792            | 3216         | 2384                 | 2840              | 5040           | 5720         | 5040            | 5720          |
| 125 | 1832            | 3160         | 2392                 | 2840              | 5024           | 6104         | 5024            | 6104          |
| 150 | 2152            | 3472         | 2384                 | 2840              | 5048           | 6200         | 5048            | 6200          |
| 200 | 2264            | 3880         | 2392                 | 2840              | 5144           | 6240         | 5144            | 6240          |
| 250 | 2376            | 4928         | 2440                 | 3072              | 5284           | 5964         | 5284            | 5964          |
| 300 | 2360            | 4672         | 2496                 | 3120              | 5288           | 6608         | 5288            | 6608          |
| 350 | 2880            | 4672         | 2512                 | 3120              | 5328           | 6284         | 5328            | 6284          |
| 400 | 2616            | 5008         | 2532                 | 3200              | 5304           | 6588         | 5304            | 6588          |
| 450 | 3048            | 5064         | 2532                 | 3120              | 5484           | 6248         | 5484            | 6248          |
| 500 | 3552            | 5736         | 2540                 | 3120              | 5480           | 6476         | 5480            | 6476          |
| 550 | 3824            | 5680         | 2552                 | 3120              | 5404           | 6360         | 5404            | 6360          |
| 600 | 3104            | 5904         | 2560                 | 3120              | 5472           | 6276         | 5472            | 6276          |
| 650 | 3496            | 5568         | 2568                 | 3120              | 5528           | 6352         | 5528            | 6352          |
| 700 | 4224            | 7752         | 2580                 | 3072              | 5528           | 6768         | 5528            | 6768          |
| 800 | 4448            | 7512         | 2580                 | 3112              | 5516           | 6576         | 5516            | 6576          |
| 900 | 4504            | 7248         | 2580                 | 3120              | 5528           | 6648         | 5528            | 6648          |
| 1000| 4608            | 7808         | 2580                 | 3120              | 5556           | 6532         | 5556            | 6532          |
| 1100| 4560            | 7472         | 2588                 | 3120              | 5588           | 6468         | 5588            | 6468          |
| 1200| 5792            | 10480        | 2588                 | 3120              | 5592           | 6240         | 5592            | 6240          |
| 1300| 5512            | 9936         | 2588                 | 3112              | 5572           | 6240         | 5572            | 6240          |
| 1400| 6384            | 10264        | 2580                 | 3112              | 5572           | 6896         | 5572            | 6896          |
| 1500| 6792            | 10328        | 2580                 | 3112              | 5584           | 6580         | 5584            | 6580          |
| 2000| 6232            | 9912         | 2580                 | 3120              | 5624           | 6168         | 5624            | 6168          |
| 2500| 7560            | 24448        | 2580                 | 3104              | 5616           | 6392         | 5616            | 6392          |
Table 2: The median and maximum running times in seconds for all runs with a specific number of vertices $n$.

| $n$  | Delaunay median | 0.014 | max | 0.064 | 309 | 0.000 | 367 | 0.000 | 910 | 0.000 | 519 | 0.004 | 162 | 0.000 | 820 | 0.002 | 603 |
|------|-----------------|-------|-----|-------|-----|-------|-----|-------|-----|-------|-----|-------|-----|-------|-----|-------|
| 10   |                 | 0.081 | 0.347 | 0.361 | 0.370 | 0.000 | 616 | 0.001 | 613 | 0.001 | 156 | 0.011 | 853 | 0.002 | 010 | 0.007 | 143 |
| 20   |                 | 0.207 | 516 | 0.943 | 375 | 0.000 | 830 | 0.002 | 879 | 0.003 | 319 | 0.026 | 406 | 0.003 | 655 | 0.014 | 552 |
| 40   |                 | 0.469 | 530 | 2.112 | 217 | 0.001 | 045 | 0.002 | 166 | 0.006 | 867 | 0.033 | 851 | 0.005 | 716 | 0.020 | 334 |
| 60   |                 |       |     |       |     | 0.001 | 399 | 0.002 | 918 | 0.013 | 428 | 0.056 | 691 | 0.009 | 516 | 0.030 | 625 |
| 80   |                 |       |     |       |     | 0.001 | 756 | 0.003 | 444 | 0.024 | 055 | 0.100 | 309 | 0.016 | 658 | 0.061 | 132 |
| 100  |                 |       |     |       |     | 0.002 | 056 | 0.004 | 030 | 0.033 | 428 | 0.150 | 279 | 0.022 | 560 | 0.068 | 170 |
| 125  |                 |       |     |       |     | 0.002 | 501 | 0.005 | 372 | 0.046 | 976 | 0.217 | 762 | 0.033 | 954 | 0.101 | 315 |
| 150  |                 |       |     |       |     | 0.002 | 888 | 0.005 | 534 | 0.061 | 505 | 0.232 | 505 | 0.041 | 352 | 0.133 | 888 |
| 200  |                 |       |     |       |     | 0.003 | 576 | 0.007 | 240 | 0.100 | 989 | 0.354 | 232 | 0.064 | 532 | 0.193 | 956 |
| 250  |                 |       |     |       |     | 0.004 | 321 | 0.008 | 537 | 0.137 | 829 | 0.458 | 281 | 0.086 | 141 | 0.260 | 132 |
| 300  |                 |       |     |       |     | 0.005 | 073 | 0.009 | 685 | 0.173 | 749 | 0.739 | 960 | 0.110 | 249 | 0.407 | 216 |
| 350  |                 |       |     |       |     | 0.005 | 579 | 0.010 | 597 | 0.200 | 256 | 0.808 | 604 | 0.128 | 425 | 0.457 | 386 |
| 400  |                 |       |     |       |     | 0.006 | 372 | 0.011 | 761 | 0.249 | 399 | 0.887 | 698 | 0.175 | 070 | 0.589 | 235 |
| 450  |                 |       |     |       |     | 0.006 | 710 | 0.013 | 537 | 0.282 | 497 | 1.096 | 251 | 0.175 | 412 | 0.587 | 010 |
| 500  |                 |       |     |       |     | 0.007 | 469 | 0.015 | 005 | 0.383 | 682 | 1.541 | 501 | 0.256 | 470 | 0.746 | 144 |
| 550  |                 |       |     |       |     | 0.008 | 528 | 0.016 | 190 | 0.415 | 579 | 1.666 | 938 | 0.261 | 306 | 0.859 | 130 |
| 600  |                 |       |     |       |     | 0.008 | 899 | 0.017 | 127 | 0.486 | 157 | 1.660 | 158 | 0.307 | 261 | 0.969 | 043 |
| 650  |                 |       |     |       |     | 0.009 | 350 | 0.018 | 987 | 0.520 | 370 | 1.707 | 651 | 0.320 | 547 | 0.991 | 330 |
| 700  |                 |       |     |       |     | 0.010 | 033 | 0.021 | 339 | 0.548 | 668 | 2.018 | 272 | 0.323 | 926 | 1.197 | 180 |
| 800  |                 |       |     |       |     | 0.011 | 583 | 0.021 | 906 | 0.729 | 638 | 2.502 | 922 | 0.452 | 032 | 1.526 | 597 |
| 900  |                 |       |     |       |     | 0.012 | 974 | 0.029 | 481 | 0.866 | 354 | 3.218 | 503 | 0.536 | 978 | 1.866 | 497 |
| 1000 |                 |       |     |       |     | 0.014 | 204 | 0.029 | 770 | 1.019 | 635 | 4.070 | 813 | 0.623 | 762 | 1.969 | 603 |
| 1100 |                 |       |     |       |     | 0.015 | 121 | 0.032 | 160 | 1.239 | 577 | 5.846 | 221 | 0.717 | 239 | 2.040 | 144 |
| 1200 |                 |       |     |       |     | 0.016 | 401 | 0.035 | 842 | 1.251 | 472 | 4.010 | 515 | 0.733 | 767 | 2.282 | 640 |
| 1300 |                 |       |     |       |     | 0.018 | 357 | 0.039 | 272 | 1.506 | 918 | 5.627 | 138 | 1.001 | 474 | 3.095 | 028 |
| 1400 |                 |       |     |       |     | 0.019 | 354 | 0.043 | 886 | 1.641 | 150 | 5.707 | 774 | 1.026 | 240 | 3.415 | 236 |
| 1500 |                 |       |     |       |     | 0.021 | 279 | 0.043 | 013 | 1.990 | 088 | 7.978 | 124 | 1.261 | 539 | 3.941 | 024 |
| 2000 |                 |       |     |       |     | 0.026 | 627 | 0.054 | 653 | 2.821 | 684 | 9.151 | 548 | 1.731 | 935 | 4.854 | 338 |
| 2500 |                 |       |     |       |     | 0.032 | 861 | 0.070 | 760 | 3.533 | 656 | 12.003 | 607 | 2.187 | 277 | 6.840 | 824 |
| 3000 |                 |       |     |       |     | 0.039 | 188 | 0.081 | 773 | 5.616 | 593 | 14.949 | 720 | 3.159 | 590 | 9.751 | 315 |