Carleton University

A Comparison of Grid to TIN Conversion Algorithms

Author:
Chris Ermel

Professor:
Dr. Jorg-Rudiger Sack

April 5, 2018

Contents

1 Introduction

In their most basic forms, maps can be represented as grids wherein each cell, a number is stored. The number may represent arbitrary data depending on the purpose for which the map was generated. For example, perhaps for a map of land elevation data, the number in each grid cell represents the land elevation at that particular geographical location. These grid-style map representations are known as raster data.

While raster data lends itself naturally to image renditions of maps - each pixel is mapped to a grid cell, and the RGB value can be derived from the number stored each grid cell - it can be quite cumbersome to store a fine grain raster image for a large geographic region. Further, it can be very computationally inefficient to operate on a two-dimensional array of data points. Rather, we wish to instead operate on the so-called Triangular Irregular Networks (TINs). TINs represent terrain through a graph of interconnected triangles, which can be generated by converting from a raster.

The objective of this paper is to contrast both the execution time performance as well as error performance of a few known grid to TIN conversion algorithms. The structure of this paper first describes both conversion algorithms, discusses implementation challenges, displays the experimental results of both algorithms, then draws conclusion from the experimental data.

2 Algorithms

We examine two different grid to TIN conversion algorithms, both of which take as input a fixed parameter $\epsilon : 0 < \epsilon \leq 1$ representing the maximum error value allowed by the generated TIN. In general, as ϵ decreases, more points from the grid will remain in the TIN, and vice versa.

2.1 Fjallstrom's Algorithm

The first algorithm implemented was published by Per-Olof Fjallstrom in 1991 [1] and implemented a bottom-up approach based on the Delaunay Triangulation. The algorithm works as follows.

First, we create a set $S = \emptyset$ and a set P containing all points in the raster. We then add the four corner points of the raster to S and set $P = P \setminus S$. We then compute the Delaunay Triangulation of the set S and distribute all of the points of P into the initial two triangles. Then, using barycentric interpolation on the points in the triangles, we compute the error of the points given the current triangulation. We take the point of P with the highest error, and, if it is larger than ϵ, we add it to S, remove it from P, retriangulate with respect to the new point in S, redistribute the points of P to the newly created triangles, and continue on to another iteration. We terminate execution once the point with the largest error is smaller than ϵ or if we have run out of points to add to the triangulation.

The efficient implementation of this algorithm relies on having the ability to incrementally create a Delaunay Triangulation over a set of points. We maintain a tree of point error values and at each iteration, and for each triangle that has been changed, remove the points that were affected from the tree, recompute their errors, then re-add them to the structure. In this way, if m points are affected by the new point in the triangulation, we only require $O(m \log n)$ time to recompute the errors. It can be shown that the runtime of this algorithm with realistic expectations is $O(n \log^2 n)$. We will describe what is meant by 'realistic expectations' in a subsequent section.

2.2 Lee's Algorithm

The second algorithm implemented was published by Jay Lee in 1989 [2] and implemented a top-down approach again based on the Delaunay Triangulation. The algorithm works as follows.

First, we create the Delaunay Triangulation of the raster by using every grid cell as a point in the triangulation. Then, for each vertex in this triangulation,

we compute the hypothetical error of its removal using barycentric interpolation. Among all hypothetical error values, we take the smallest and, if it is below ϵ, remove it from the triangulation and update the hypothetical error values of its neighbours. We terminate once the point with the smallest error has error larger than ϵ.

The efficient implementation of this algorithm relies on having the ability to incrementally create a Delaunay Triangulation by removing points from a pre-existing Delaunay Triangulation. We maintain a min-heap on the hypothetical error values computed for each point in the triangulation. To find the smallest, we pop from the heap in $O(1)$ time, retriangulate, then must remove the d neighbor error values from this heap in order to recompute and re-add them. It can be shown that the runtime of this algorithm is $O(n \log n)$.

3 Implementation

The Python programming language was selected in order to implement the aforementioned algorithms. This decision was not made with execution efficiency in mind - indeed, it can be shown that performing identical operations in the C programming language can execute over seventy fives times the speed of pure python code [4]. This decision was made, instead, to reduce complexity of the code itself. The writer believes it can be beneficial to prototype complex algorithms in a simpler language before porting them to a more efficient one for optimization and, as a result, legitimate application.

3.1 Challenges

A number of challenges arose during the implementation of both the Fjallstrom and Lee algorithms described in section 2. For the most part, the general structure of the algorithms remained the same as the efficient implementation was described. However, it was deemed that an efficient implementation of a Delaunay Triangulation algorithm was out of the scope of the experiment, in order to focus on the two algorithms for which this paper is concerned. As such, we were limited to the capabilities of the Delaunay Triangulation algorithm provided by

the Scipy Python library [3]. More specifically, the Scipy algorithm does not allow for the computation of a Delaunay Triangulation by removing points from a pre-existing one. This meant that, on each iteration of Lee's algorithm, it was necessary to retriangulate the entire point set, increasing the complexity of this step from $O(d \log d)$ to $O(n \log n)$, where d is the number of neighbouring points of the point with minimum error. In order to maintain an aspect of fairness, we imposed the same restriction on the Fjallstrom algorithm - on each iteration, the Delaunay Triangulation of the point set S was recomputed.

Another deviation from the implementation of the algorithms involved the removal of the error-tree structure in the Fjallstrom algorithm and the error min-heap structure in the Lee algorithm. This deviation actually saved execution time, rather than increasing it. It can be shown that executing an equivalent operation using the Python Numpy library can be thirty times faster [4] than native Python. Therefore, we observed an increase in time efficiency when the error values were stored in a Numpy **ndarray** object [5] and the min/max error values were maintained by sorting this array in $O(n \log n)$ time per iteration of each algorithm.

4 Experiments

The initial goal of the experiment was to compare the runtime and average error metrics of each algorithm on elevation data generated by satellite imaging. However, due to the reliance on the SciPy Delaunay Triangulation algorithm, we were unable to perform tests on data of this magnitude that could terminate in a reasonable period of time. To rectify this problem, we created an algorithm for randomly generating rasters that contain somewhat natural data-features. An example of a raster generated by this algorithm can be seen in Figure 1.

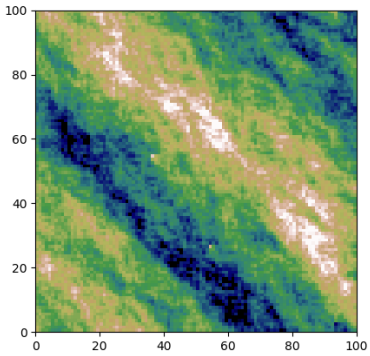

Figure 1: An example of a randomly generated raster used for the experimentation.

It was necessary to ensure that the generated raster were not completely random, as the analysis of the Fjallstrom algorithm's runtime operates with a set of natural assumptions. That is to say that we expect certain portions of land within the raster to have the same features. In a purely random raster, the Fjallstrom algorithm would potentially need to run until all points of the raster are in the set S, since the error of each point would be to large at any given iteration.

The experimental procedure was as follows. Both algorithms were executed with $\epsilon = 0.3$ and $\epsilon = 0.05$. At each stage we generated increasingly large raster images and used both algorithms on the same source raster. Since the source rasters were random, we generated 10 different rasters of the same size in order to ensure that an accurate runtime and error calculation could be discerned from the generated TIN. The results of the experimentation are as follows.

4.1 30% Maximum Error

The experiments on the runtime of both algorithms with $\epsilon = 0.3$ on increasingly large raster data revealed the behaviour in Figure 2.

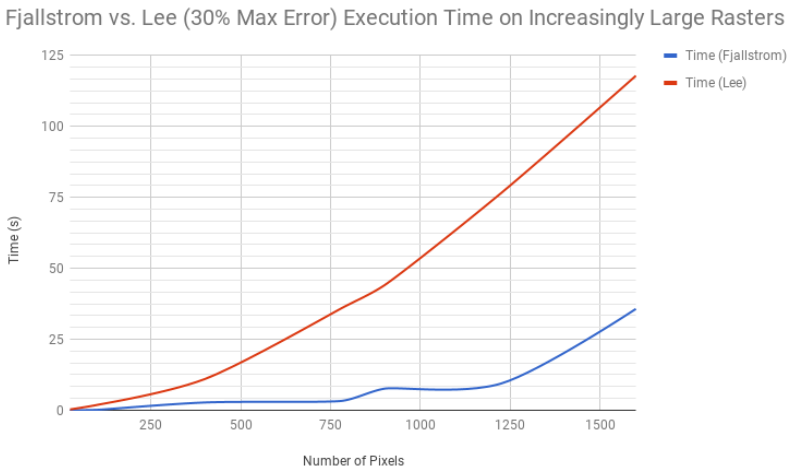

Figure 2: Runtime behaviour of Fjallstrom vs. Lee algorithms on increasingly large rasters.

It is clear from Figure 2 that in all cases for $\epsilon = 0.3$ the Fjallstrom algorithm provides more efficient runtime behaviour. This is unsurprising due to the bottom-up nature of the algorithm. Since the Fjallstrom algorithm begins with only 4 points in its triangulation, for higher ϵ values, the algorithm will need to retriangulate fewer times before terminating. Whereas for the Lee algorithm, for higher ϵ values more retriangulations must be performed in order to remove more points. It is noticeable that the Fjallstrom runtime curve in Figure 2 observes a slight bump around the 875 pixel mark. This is due to the fact that execution depends entirely on the input raster, and our experimental procedure involves creating random rasters rather than using the same one on each test. This bump could be rectified by executing more iterations of the experiment for each raster size in order to ensure that the computed runtime average is closer to its true value. We next examine the average error generated by both algorithms in Figure 3.

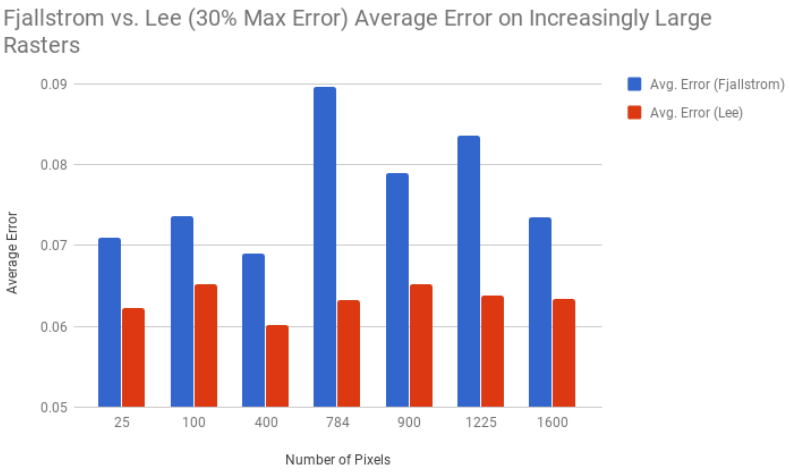

Figure 3: Average error of Fjallstrom vs. Lee algorithms on increasingly large rasters.

It is clear from Figure 3 that Lee's algorithm provides a lower average rate of error for all points than the Fjallstrom algorithm (for $\epsilon = 0.3$). However, regardless, the average errors observed in the worst case are only upwards of 9%.

4.2 5% Maximum Error

The experiments on the runtime of both algorithms with $\epsilon = 0.05$ on increasingly large raster data revealed the behaviour in Figure 4.

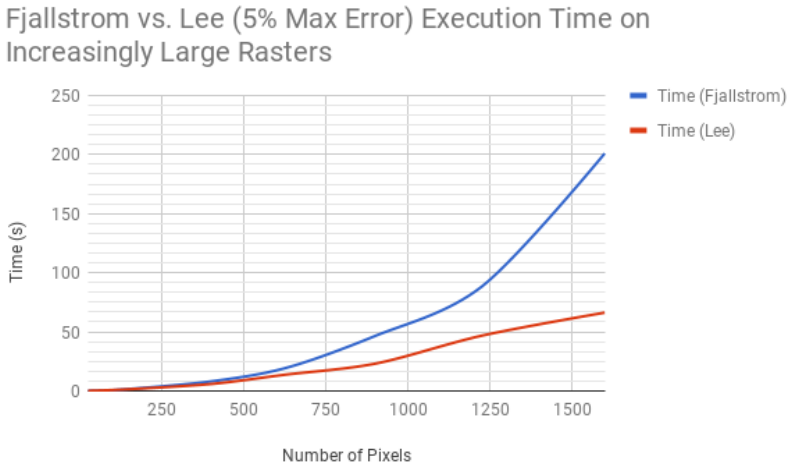

Figure 4: Runtime behaviour of Fjallstrom vs. Lee algorithms on increasingly large rasters.

It is clear from Figure 4 that eventually, Lee's algorithm becomes more efficient than the Fjallstrom algorithm once a certain raster size threshold is met. Again, this is not surprising given the top-down nature of Lee's algorithm. For lower ϵ values, we expect that more points will be added to the Delaunay Triangulation. As such, Lee's algorithm must retriangulate fewer times than the Fjallstrom algorithm, leading to a reduced run time. We next examine the average error generated by both algorithms in Figure 5.

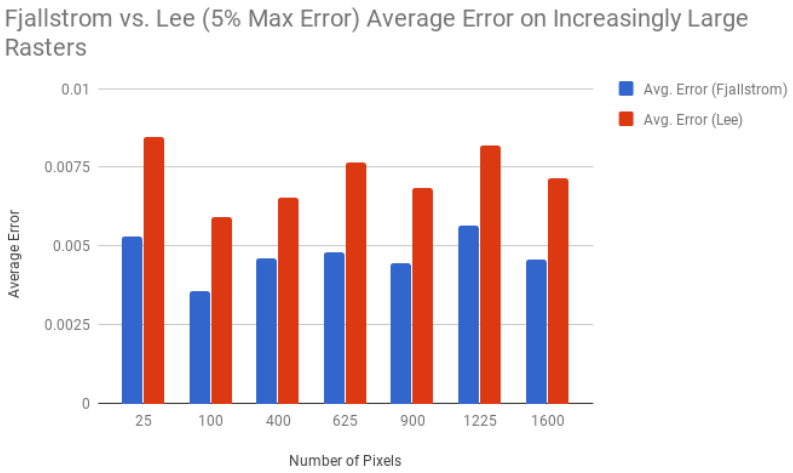

Figure 5: Average error of Fjallstrom vs. Lee algorithms on increasingly large rasters.

By Figures 5 and 3 it is clear that a particular algorithm always guarantees superior error performance, much like the runtime statistics. In this case, we see that Lee's algorithm executes fewer retriangulations and ends up with higher average error values than that of the Fjallstrom method. It is hypothesized by the writer that this phenomena occurs because for low ϵ values, the Fjallstrom algorithm ends up adding more points to its TIN triangulation. Whereas for high ϵ values, the Lee algorithm ends up adding more points to its TIN triangulation. In the next section, we draw conclusions from these experiments and reiterate the major takeaways from the project.

5 Conclusion

To conclude, we have shown empirically that depending on the maximum error parameter ϵ, both the Fjallstrom and Lee algorithms perform differently. In general, for lower epsilon values, it is suggested to convert a raster to a TIN using Lee's algorithm. For high epsilon values, it is suggested to convert a raster to a TIN using Fjallstrom's algorithm. The one stipulation is that the most time efficient solution in either scenario will have greater average error, although this increase is minimal.

While it is clear that both algorithms perform differently for different ϵ values, it is unclear for which values of ϵ the algorithms trade off in terms of their performance. As such, it is suggested by the writer that an optimal solution for converting a grid to a TIN would involve running both algorithms in parallel, and then returning the TIN generated by the first algorithm that terminates.

Finally, it was rather unfortunate that we were unable to test both algorithms against full-size satellite raster images. Not only would it have been more interesting, it would have been a more realistic test of the algorithms in their real-world applications. While both algorithms would exhibit similar behaviour with respect to various ϵ values, it would be more worthwhile to test performance under these real-world circumstances. As such, the writer believes that additional work is needed to implement an efficient Delaunay Triangulation algorithm that allows for efficient retriangulation based on point removal.

References

[1] Fjallstrom, P., Katajainen, J., and Petersson, J. (1991). Algorithms for the All-nearest-neighbors Problem. *Ikke angivet*, IFIP Int. Federation for Information Processing, 74–75.

[2] Lee, J. (1989). Coverage And Visibility Problems On Topographic Surfaces. *Digitized Theses*, 1818.

[3] Spatial algorithms and data structures. *SciPy v0.14.0 Reference Guide*, SciPy. `https://docs.scipy.org/doc/scipy-0.14.0/reference/generated/scipy.spatial.Delaunay.html`

[4] Ross, P. (2014). The Peformance of Python, Cython and C on a Vector. *Notes on Cython*. `http://notes-on-cython.readthedocs.io/en/latest/std_dev.html`

[5] The N-dimensional array. *NumPy v1.14 Manual*, SciPy. `https://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html`