Carleton University

Honour's Project

Towards an Implementation of Fast Algorithms for Approximate Fréchet Matching Queries in Geometric Trees

Author:
Chris Ermel

Supervisor:
Dr. Michiel Smid

April 18, 2018

Contents

1 Introduction **2**

2 Data Structures and Algorithms **3**

2.1 The Exponential Grid 4

2.1.1 Description 4

2.1.2 Implementation Considerations 5

2.2 The Fréchet Grid 5

2.2.1 Description 5

2.2.2 Implementation Considerations 7

2.3 The Curve Range Tree 8

2.3.1 Description 8

2.3.2 Implementation Considerations 9

2.4 The Fréchet Tree 10

2.4.1 Description 10

2.4.2 Implementation Considerations 10

3 Implementations **13**

3.1 The Exponential Grid 14

3.1.1 Challenges/Experiments 14

3.2 The Fréchet Grid 17

3.2.1 Challenges/Experiments 18

3.3 The Curve Range Tree 20

3.3.1 Challenges/Experiments 21

4 Conclusion **22**

1 Introduction

In the domain of Geographical Information Systems, we are often limited to inaccurate location data provided by satellites. This becomes problematic in modern applications when we attempt to match a person's location trajectory onto a graph network of roads. This problem is commonly known as map-matching, and problems of this type could be simplified if we had an efficient way to measure the similarity between two different polygonal curves.

We introduce the Fréchet Distance [12] as a measure of similarity between two curves P and Q. Mathematically the Fréchet Distance is defined as

$$\delta_F(P, Q) = \inf_{f} \max_{0 \le t \le 1} |P(f(t))Q(t)| \tag{1}$$

where f ranges among all orientation-preserving reparameterizations. Conceptually, the Fréchet Distance can be described as the minimum length leash required assuming that the path P represents the trajectory of a human walking their dog, and Q represents the path of the dog itself.

In 2015, Smid and Gudmundsson [1] published a paper providing a data structure that allows for efficient Fréchet Matching queries in geometric trees. The paper builds on a couple of results obtained by Anne Driemel [2] in order to provably demonstrate efficient time bounds for such queries. The primary result achieved in the paper is the following: We have a geometric tree T in $\mathbb{R}^d$ and we wish to preprocess it such that, given a polygonal query path P, we can efficiently determine whether or not T contains a path P' such that $\delta_F(P, P') \le 3(1+\epsilon)\Delta$, where Δ and ϵ are predetermined constants. This type of query is illustrated in Figure 1 and the result is Theorem 4 in [1].

The results obtained in the paper are a topic of interest due to the fact that determining the continuous Fréchet Distance is computationally difficult. In fact, to date, no sub-quadratic time algorithms exist that allow for computing $\delta_F(P, Q)$. As such, we are interested in attempting to implement the data structures and algorithms that are discussed in [1].

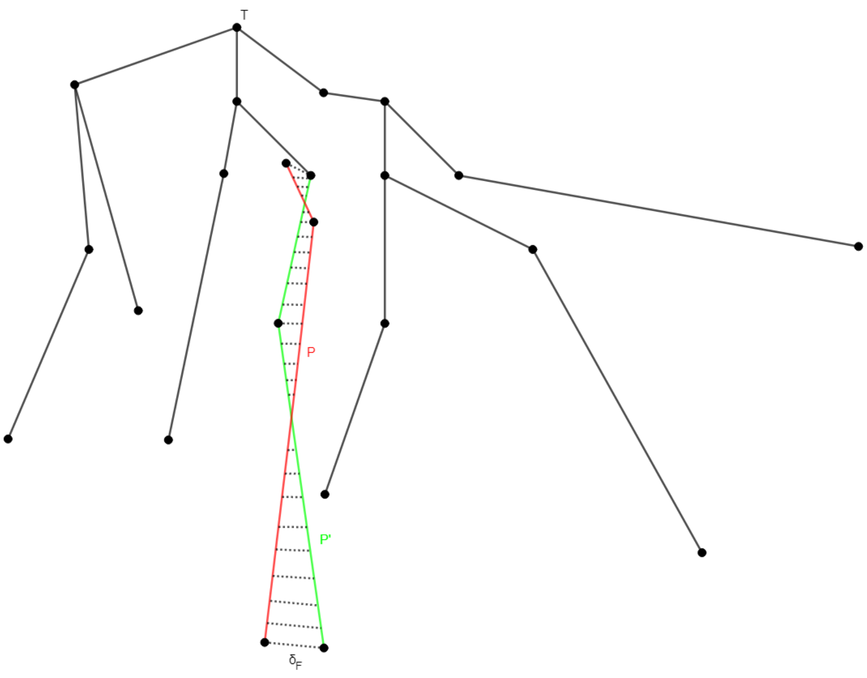

Figure 1: An illustration of a Fréchet Matching query. Given the query curve P, we find the subpath P' in T. Figure generated with [6].

This report is structured in such a way as to first describe the data structures and algorithms that were implemented over the course of this project. Then, we focus on a few of the more concrete implementation challenges and results derived through experimentation on the implementations. Finally, we conclude by summarizing major takeaways from the project.

2 Data Structures and Algorithms

In this section we will describe the major components discussed by Smid and Gudmundsson [1] that allow for efficient Fréchet Queries against geometric trees. This section will take a bottom-up approach to explaining the required data structures and algorithms and will attempt to explain what would be needed in order to realize a practical implementation. It is worth noting that the descriptions of data structures and algorithms will be from a high-level standpoint. References to their original papers will be provided for the case where more explicit details are sought.

Initially, the goal of this project was to determine the requirements for a full implementation of the aforementioned data structure, then attempt to implement it - if time allowed. However, after a great deal of time digesting the information in [1], the writer decided that the most practical way to understand what would be required for a full implementation would be to begin attempting to implement the data structure from the bottom up. In the end, not all the data structures were implemented due to time constraints. As such, only the data structures for which the writer has concrete implementation experience will be discussed.

2.1 The Exponential Grid

2.1.1 Description

The root element of the entire data structure is one created by Anne Driemel in her 2013 thesis 'Realistic Analysis for Algorithmic Problems on Geographical Data'. The data structure is rigorously described in Lemma 4.2.3 of [2].

The data structure functions by creating a set of axis-aligned hypercube-based grids around an input point $u \in \mathbb{R}^d$ such that the side length of each grid, and by consequence the side length of its grid cells, becomes exponentially larger as we increase the number of grids. Then, we take each of the four corner points of each grid cell in each grid and define this as the point set $G(u)$. An example of an exponential grid can be seen in Figure 2.

In essence, the grid is used to ensure that for a given region around u, all points can be estimated to a certain predetermined percentage of error. In other words, we are discretizing the space around u so that we can perform preprocessing operations on a finite set of data points which approximate the space. To this end, the data structure supports queries of the following type. For a given query point p, we can determine in $O(1)$ time a point $p' \in G(u)$ such that $||p-p'|| \leq (\epsilon/2)||p-u||$.

2.1.2 Implementation Considerations

The primary concern when implementing this data structure is ensuring that the set $G(u)$ does not contain overlapping points from the various grids. In essence, if we have a grid C_i, where the side length of C_i is larger than that of C_{i-1}, which contributes a number of points to the set $G(u)$, we want to ensure that we deduct the set of points in C_{i-1} from the set of points in C_i. If this statement is unclear, please note by looking at Figure 2 that larger grids overlap with smaller grids, since they all originate at the point u. If care is not taken to ensure that overlapping points aren't included in $G(u)$, then the magnitude of $G(u)$ will be much larger than necessary, which will hinder the preprocessing time when creating the Fréchet Grid, discussed later on.

Otherwise, implementing this data structure is rather simple. It is a matter of computing the number of grids along with their associated side lengths and grid cell side lengths, then using these to iteratively compute the set $G(u)$ of points within those grids. The constant time query access can be achieved using the geometric location of the query point p along with the floor or ceiling function used in conjunction with the side lengths of both the grids and grid cells in order to locate the cell in which p belongs. We then return p' as the corner point of the grid cell closest to p.

2.2 The Fréchet Grid

2.2.1 Description

The Fréchet Grid is another important data structure created by Anne Driemel which allows for constant time approximate Fréchet Distance queries from an edge to a polygonal curve. The data structure is rigorously described in Lemma 4.2.4 of [2].

The data structure functions by establishing two exponential grids $G(u)$ and $G(v)$, where u and v are the endpoints of the input polygonal curve, denoted as Z. Then, for each line segment $Q \in G(u) \times G(v)$, we precompute $\delta_F(Q, Z)$ and store it. We also precompute the Fréchet Distance from the spine of the curve (the line segment uv) to Z and store this value as L. An example of the Fréchet Grid can be seen in Figure 2.

The data structure's $O(1)$ time query algorithm computes a value

$$r = \max \left(||p - u||, ||q - v|| \right) \tag{2}$$

which is used in comparison with L in order to derive two potential cases for the Fréchet Distance approximation. If both of these cases fail, then we approximate the query segment pq using the exponential grid query algorithms of $G(u)$ and $G(v)$ to obtain the line segment $p'q'$. Then, we query the Fréchet Grid data structure using p' and q' in order to return the precomputed Fréchet Distance for the approximated line segment. The Fréchet Grid query algorithm and data structure provide us with the property that for any query segment pq, we can return in $O(1)$ time a $(1 + \epsilon)$-approximation of $\delta_F(pq, Z)$.

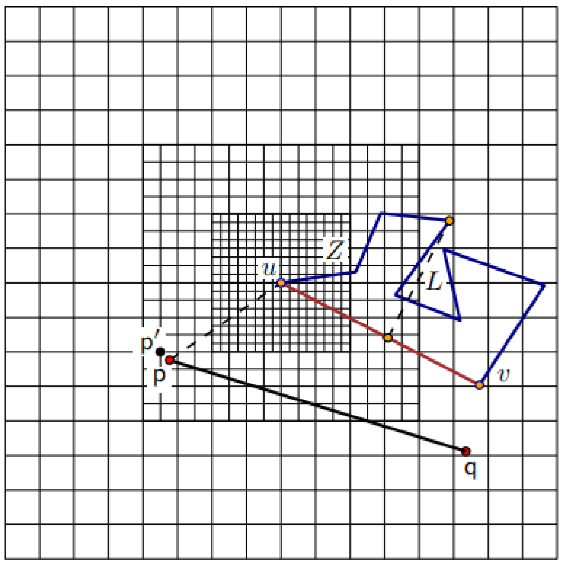

Figure 2: An illustrative example of the Fréchet and Exponential Grids taken from section 4.2.2.1 of [2].

2.2.2 Implementation Considerations

The Fréchet Grid data structure itself is actually much more simple to implement than the Exponential Grid. The biggest challenge lies in storing the precomputed Fréchet Distances in a way that can be retrieved in $O(1)$ time given the approximated line segment $p'q'$. This challenge is easily overcome once it is realized that all possible points in $G(u)$ and $G(v)$ are fixed for the polygonal curve Z. So-long as we have a way to uniquely represent an edge in a single-dimension, we can store all the precomputed values in an associative array (i.e., a **dict** in Python).

Since we can write the coordinates of a point in $\mathbb{R}^d$ in a one-dimensional string representation to obtain a unique one-dimensional mapping, we can also just write the string representation of both points comprising the endpoints of an edge in order to map it uniquely in one dimension. Therefore, we can store an associative array where the keys are the one-dimensional representation of each point $a \in G(u)$, and for each mapping we can store another associative array which in-turn maps each point $b \in G(v)$ to $\delta_F(ab, Z)$. Since the query points of $G(u)$ and $G(v)$ are fixed, we will always be able to key into this structure in constant time to retrieve the precomputed Fréchet Distance value.

It is worth noting that implementation of the Fréchet Grid requires an implementation for computing the continuous Fréchet distance in $O(nm \log nm)$ time, where n is the number of vertices in the first curve and m is the number of vertices in the second curve. Since we are only computing the Fréchet Distance between an edge - which obviously has $O(1)$ vertices - our required runtime would be only $O(n \log n)$. A well known technique, due to Alt and Godau [3] can be used to attain this goal using parametric search. However, it is noted that the algorithm contains large constants that result in too high of a runtime for practical usage.

Instead, Anne Driemel recommends to implement an algorithm published by Har-Peled and Raichel [4] which does not use parametric searching but still runs in $O(mn \log mn)$ time. To preserve implementation complexity, the writer implemented the Fréchet Grid using an implementation of the discrete Fréchet Distance due to Eiter and Mannila [5] which executes in $O(nm)$ time, and can be used to approximate the continuous Fréchet Distance by adding steiner points to the edges and polygonal curve used in the computation.

2.3 The Curve Range Tree

2.3.1 Description

The Curve Tree is the first data structure described by Smid and Gudmundsson and builds upon the Fréchet Grid described in the previous section. The ultimate goal of this data structure is to determine, given a query edge Q, whether or not a polygonal path P contains a subpath $P[x, y]$ such that $\delta_F(Q, P[x, y]) \leq (1 + \epsilon)\Delta$. The data structure is rigorously described in Lemma 2 of [1].

The data structure functions by partitioning the polygonal path P into a binary tree where the left child of a node contains the left half of the path stored at the parent, and analogously for the right child. An example of the aforementioned curve partitioning can be seen in Figure 3. Further, in each node of the tree, we store a Fréchet Grid for the subpath contained within that node. We also fix a constant Δ used later on in the query algorithm.

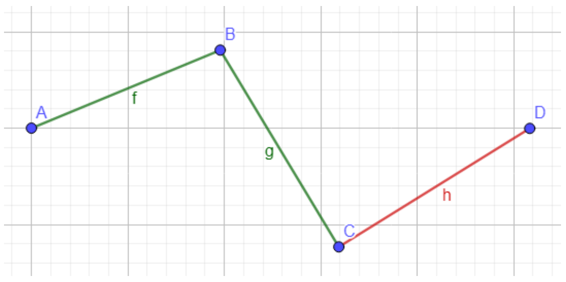

Figure 3: An example of the Binary Tree partitioning. The curve $ABCD$ will be contained within the parent node, while ABC will be in the left child and CD in the right. Figure generated with [6].

The query algorithm for the Curve Tree is a little bit more involved in comparison to the data structure itself. The details of the algorithm are omitted in this paper and instead can be seen in Section 2.1 of [1]. We are given a query segment Q and two points x and y along with the edges of P that contain them.

We use a range tree-style query on the binary tree to find $O(\log n)$ subpaths which we use to compute a Directed Acyclic Graph (DAG) whose edge weights are the the Fréchet Distance from various subdivisions of Q to P. We then compute the heaviest weight of the bottleneck path of the DAG, denoted as Δ', which is used to determine whether not P contains a subpath whose Fréchet Distance to Q is at most $(1+\epsilon)\Delta$.

2.3.2 Implementation Considerations

The data structure itself is rather simple to implement. Using the recursive construction illustrated in Figure 3, we can easily form a balanced binary tree from an input curve P. However, care is needed when initializing the Fréchet Grids stored at each node of the tree. As will be discussed later on, instantiating a Fréchet Grid requires a large amount of time, depending on the value ϵ applied to the grid itself. While Smid and Gudmundsson instantiate a Fréchet Grid applied with $\epsilon/2$ for each node in the tree, the writer found that applying ϵ achieved more practical performance for testing the implementation of the data structure.

An important requirement for the query algorithm is an implementation of a range tree-style query. This technique is rigorously described in [7]. However, the implementation of such a query in the scope of this data structure is much simpler. Firstly, we only need to execute the functional equivalent of a one-dimensional range tree query on the tree. And secondly, rather than return all values at the leaves of the binary tree, we simply return the root nodes of the selected subtrees - this provides us with our $O(\log n)$ subpaths.

Finally, an implementation of a DAG is required to implemented the query algorithm. This can be achieved by storing each vertex of the DAG in an adjacency list. For each vertex in the adjacency list, we store all vertices to which there is a directed edge. Then, we require the ability to compute a bottleneck path on this DAG representation, which is possible using dynamic programming and will be discussed later on.

2.4 The Fréchet Tree

2.4.1 Description

The Fréchet Tree is a data structure that builds upon the result achieved by the Curve Range Tree described in the previous section. The main difference is that it extends the query from polygonal curves to general geometric trees. The result achieved is that for any query edge Q and two points x and y on a geometric tree T along with the edges that contain them, we can determine in $O((\log^2 n)/\epsilon^2)$ time whether or not $\delta_F(Q, T[x, y]) \leq (1+\epsilon)\Delta$. The result is described in Lemma 4 of [1].

The data structure uses an important technique for decomposing general geometric trees in order to attain its efficiency. While a slight variation of the technique is described in section 2.3.2 of [8], the only difference in our case is that, for a node u, the $size(u)$ is equal to the number of vertices of the subtree rooted at u. This technique, called the path decomposition of T and denoted as $PD(T)$, gives us the property that, for any subpath $T[x, y] \in T$, $T[x, y]$ intersects with $O(\log n)$ subpaths in $PD(T)$. The construction of the data structure works as follows. First, we decompose the tree T in order to determine all subpaths in $PD(T)$. Then, for each path in $PD(T)$ we construct a Curve Range Tree.

Finally, the query algorithm for this data structure is fairly similar to that of the Curve Range tree. We use the $PD(T)$ to find $O(\log n)$ subpaths from T that intersect with $T[x, y]$. Then, for each of these subpaths, we perform the range-tree style query on their Curve Range Trees in order to decompose them further into $O(\log^2 n)$ subpaths. Finally, we re-use steps 2-5 of the query algorithm from the Curve Range Tree in order to determine whether or not $\delta_F(Q, T[x, y]) \leq (1+\epsilon)\Delta$.

2.4.2 Implementation Considerations

The most important consideration for a practical implementation of this data structure is determining how to compute $PD(T)$. Firstly, the sizes can be computed by executing a post-order traversal on T. For each node u visited during this search, if u is a leaf, then we set $size(u) = 1$. Otherwise, let $C(u)$ denote all children of the node u. We set

$$size(u) = \sum_{v \in C(u)} size(v) \tag{3}$$

Additionally, we must store a value $\ell(u)$ at each node such that

$$\ell(u) = \lfloor \log(size(u)) \rfloor. \tag{4}$$

Since the value of $\ell(u)$ is derived from $size(u)$, it is possible to compute all $\ell(u)$ values as well during this post-order traversal. Then, finally, one can compute the paths themselves by executing a depth first search over T.

Another implementation consideration is the question of how we go about storing general geometric trees. It is possible to represent these trees storing a list of children at each node. However, for simplicity, the writer recommends implementing the general trees by storing, at each node, a pointer to its left child and right sibling.

Further, it is important to consider how to go about storing the subpaths created in $PD(T)$. The input to the query algorithm receives the edge Q and two points x and y on T along with the edges of T that contain x and y. Now, in the representation of general trees described above, the tree itself will have no notion of an edge. It may be worth considering, rather than taking as input the edges of T that contain x and y, instead taking as input the nodes containing the endpoints of the edges containing x and y furthest from the root of the tree - we will denote these nodes as x_n and y_n. For an illustration of what is meant in the previous sentence, please refer to Figure 4. This way, we can easily find the edges that contain x and y by taking the parents of these input nodes, and we additionally gain a pointer into the tree T without needing to perform any searching. Unfortunately, this may cause additional overhead when attempting to call the query algorithm since we will now need to determine which nodes to pass it. Regardless, if we want to remain in the logarithmic time bounds of the query algorithm, it is not recommended to perform any search on an unstructured tree: in the worst case, T could be a polygonal path, which would cause the search time to be $O(n)$.

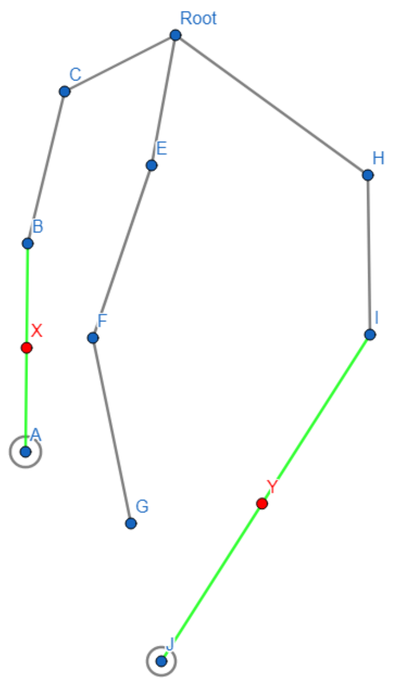

Figure 4: An example of the proposed modification to the query algorithm. Rather than providing edges AB and JI as input, it is proposed instead that nodes A and J be provided. Figure generated with [6].

Beyond that, it is important to store $PD(T)$ in such a way as to ensure efficient retrieval of the subpaths along $T[x, y]$ given the query algorithm input parameters. This can be achieved by storing, at each node, the subpaths in $PD(T)$ to which that particular node belongs. We can then compare the curves stored in a node to the curves stored in its group parent (the concept of a group parent is discussed in [8]) to find the matching subpath. Denote $lca(u, v)$ as the lowest common ancestor node of two nodes u and v. See [8] for the algorithm which computes $lca(u, v)$ given $PD(T)$. We perform this process from x_n up to $lca(x_n, y_n)$ and from y_n up to $lca(x_n, y_n)$ in order to determine all of the subpaths along $T[x, y]$. Since the path $T[x, y]$ only intersects with $O(\log n)$ many paths in $PD(T)$, we therefore only spend $O(\log n)$ time searching $T[x, y]$ for the decomposition's paths, as well as an additional $O(\log n)$ time in finding $lca(x_n, y_n)$.

This concludes the high-level discussion on the data structures and algorithms that were studied over the course of the project.

3 Implementations

In this section, we discuss the more fine-grained details corresponding to the concrete implementations of the aforementioned data structures and algorithms. In general, we will discuss the strategies applied to attain the implementations as well as challenges that arose during application of the theory. Note, in advance, that the implementation of the Fréchet Tree did not attain a testable state at the time of writing. As such, it has been omitted from this section entirely.

It was determined early on that Python 2.7 would be employed for the implementations. Although it is well known that better running times can be achieved using lower level programming languages (such as C++), it was the writer's belief that implementing prototypes of complex algorithms and data structures first in a higher level programming language could be beneficial in terms of ironing out difficult and sometimes unforeseen implementation details.

As an additional note, it was further determined by the writer that all implementations would be created from the ground up, without building on existing code. The reason for this is purely philosophical; while it is true that all practical software libraries leverage existing code, it is also true that developers often take for granted the concepts and details abstracted away by such libraries. In order to fully appreciate the complexity associated with each and every data structure explored, it was necessary to understand every component that contributed to the larger picture - to the finest grain of detail.

Although the philosophy behind the decision to go ground-up is quite debatable, there is no denying the fact that coding from a blank slate decreases the velocity at which implementations can be achieved. This is part of the reason why we leverage preexisting and pretested software libraries: we code faster and with more confidence in the robustness of our implementations. However, it should be noted that the implementations derived in the context of this project are merely prototypes. They should not be considered as fully functional library code.

Instead, they were developed solely for the purpose of deepening the writer's understanding of the subject matter. As such, they may contain bugs that a more robust solution otherwise would not have. Either way, in most cases the prototypes are sufficient to execute a fair number of test cases against them in order to assess their performance. A few of the more interesting performance results will be presented in the subsequent section.

All code associated with this project is available online through Github via [9].

3.1 The Exponential Grid

To implement the exponential grid we abstract a few different concepts into four Python objects. At the base level, we have a grid cell object - defined by the four corner points forming the square cell. Next, we define grid object which is composed of many grid cells. The grid is initialized with a hypercube object (very similar to the grid cell) which is used to define the bounds of the grid itself. We also pass the grid a cell width measurement, in order to determine the spacing between the four points in each grid cell.

Finally, we define the Exponential Grid object itself, which has a one-to-many relationship with the grid objects. Based on [2], the Exponential Grid takes as input parameters α and β, used in order to define the distance boundaries of the exponential grid. We then initialize $\lceil \log \alpha/\beta \rceil$ many hypercubes of exponentially larger side length and use these to initialize each of the $\lceil \log \alpha/\beta \rceil$ many grid objects.

3.1.1 Challenges/Experiments

While the above method provides an easily understandable way of finding all the grid points required to compute the set $G(u)$ of points, there are a few technicalities that must be addressed.

Firstly, in the implementation of the Fréchet Grid, we need a method of iterating over all points in $G(u)$. We therefore must provide, in each of the aforementioned objects, a mechanism for accessing all the points contained within. The grid cell is rather simple, since we can iterate over the four points that define it. For the grid structure however, we initially attempted to make use of the Python **yield** keyword in order to create a Python **Generator** object for which we could iterate over all grid cells and iterate over their four points. We then used the same technique at the Exponential Grid level in order to iterate over all of the Grid objects and return their grid cell points.

Unfortunately, performance implications deemed this technique impractical. The magnitude of the point set $G(u)$ is so large that we already begin to run into issues with Python's performance. As can be seen in Figure 5, performing a large scale operation such as iterating over a massive array can take a significant amount of time in Python as opposed to other techniques. However, we can save a execution time by using the **ndarray** object provided by the Numpy scientific computing library for Python [11]. We regrettably note the deviation this causes from the ground-up philisophy described earlier.

Regardless, by discarding use of the generator-based implementation, we save enough time for practical testability of the Fréchet Grid. Unfortunately, as was noted in Section 2.1.2, we must take care in ensuring that we do not store points that overlap with other grids. Further, each grid cell object shares either one or two points with its adjacent cells.

To rectify the grid cell overlap, we make use of a hash set at the Grid object level in order to discard any overlapping grid cells points. Ensuring that overlapping grids do not contain overlapping grid points, however, is slightly more complicated. In order to ensure that a grid C_i does not contain any grid points from the grid C_{i-1}, we must pass in the hypercube used for C_{i-1} when instantiating grid C_i. Then, as we iteratively create the grid cells, we check to see if the grid cell is entirely contained within C_{i-1}'s hypercube, and avoid creating the grid cell if this is true. It is worth noting that we only do not create the grid cell if the entire cell is contained within the hypercube because it is possible that a grid cell only partially overlaps with smaller grid. In such a case, we want to ensure the grid cell still exists, otherwise the Exponential Grid's point set $G(u)$ will contain areas for which points cannot be found.

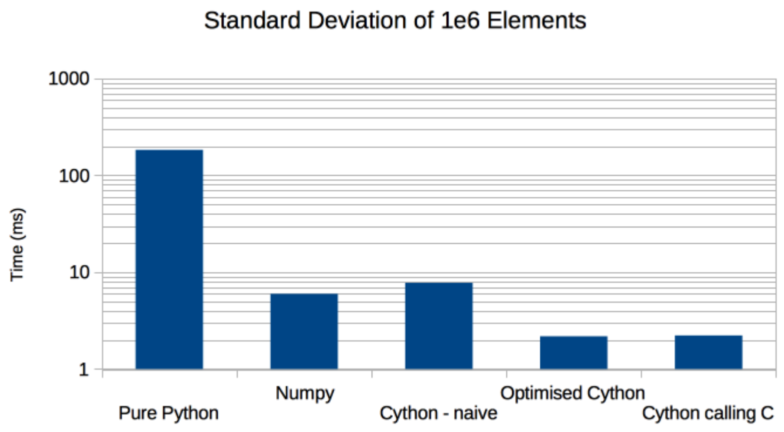

Figure 5: Experimental results of calculating the standard deviation of one million floats, taken from [10]. Note the logarithmic scale.

Finally, an additional implementation challenge associated with this data structure involved determining how to go about obtaining $O(1)$ grid point access. Note that we have $i = 0, ..., \lceil \log \alpha/\beta \rceil$ many grids, stored in an array. From [2], we define the side length of each grid as $2^{i+2}\alpha$. Recall that the Exponential Grid is centered around a point $u = (x_u, y_u)$. For a query point $v = (x_v, y_v)$, we can determine the index of its corresponding grid by computing

$$A = \left\lceil \log\left(\frac{|x_v - x_u|}{\alpha}\right) - 1 \right\rceil \tag{5}$$

$$B = \left\lceil \log\left(\frac{|y_v - y_u|}{\alpha}\right) - 1 \right\rceil \tag{6}$$

then finally doing

$$i = \max(A, B). \tag{7}$$

Note that we can now index in $O(1)$ time to the grid array to obtain the grid containing the query point v. Within this grid, we can use a similar technique by leveraging the side lengths of the grid cells in order find the grid cell containing v in $O(1)$ time as well.

Unfortunately, the above solution contains some minor degenerate cases which were revealed during testing of the data structure. The problem is that we run into a mathematical domain error if we ever attempt to compute $\log(0)$. Clearly, this occurs either when $x_v = x_u$, $y_v = y_u$, or $\alpha = 0$. We solve these problems as follows. Firstly, we restrict the data structure to ensure that $\alpha > 0$. Secondly, if either $x_v = x_u \oplus y_v = y_u$, then we determine i by using the coordinates in the other dimension. Finally, if $x_v = x_u$ and $y_v = y_u$ then we know that $v = u$ and so we simply return u as the approximation point.

3.2 The Fréchet Grid

While the implementation of the Exponential Grid required some creativity in abstracting geometric concepts into objects - the implementation of the Fréchet Grid was not quite as involved.

For starters, we required an abstraction of a polygonal curve. This can be represented in the form of an ordered list of two-dimensional points. We can further extend this definition to create an edge object, which is but a special case of a polygonal curve containing only two points.

Next, we required a method of computing the Fréchet Distance efficiently. As was noted in Section 2.2.2, we employed a dynamic programming-based solution [5] for computing the discrete Fréchet Distance in $O(nm)$ time. However, since the discrete Fréchet Distance is only an approximation of the continuous Fréchet Distance, it was necessary to add steiner points along all edges and polygonal curves used in the data structure. Through experimentation it was determined that each steiner point would be placed a Euclidean distance of 1 apart in order to maintain a healthy balance of execution speed and accuracy.

Otherwise, the major implementation concern associated with implementing this data structure has already been covered in Section 2.2.2. Upon initialization of the Fréchet Grid, we iterate over a doubly-nested for-loop which represents the set $G(u) \times G(v)$ of edges, where u and v are the endpoints of the input curve, Z. In doing so, we can create our associative array of precomputed Fréchet Distances from any edge in $G(u) \times G(v)$ to Z.

3.2.1 Challenges/Experiments

While we discussed earlier that the magnitude of the point sets contained in the Exponential Grids contributes significantly to the preprocessing time of the Fréchet Grid, we did not yet provide any empirical performance data on the initialization speed of the data structure. Please consult Figure 6 to see the results of initializing a Fréchet Grid over increasingly large values of ϵ.

Observe, in Figure 6, that when initializing a Fréchet Grid data structure with $\epsilon = 0.4$, we require time over $1000s \approx 16m$ in order to preprocess the data structure. For this reason, we have omitted ϵ values less than 0.4 as we believe that this chart is sufficient to illustrate the performance bottleneck caused by reducing the value of ϵ. It is also worth noting that the data in Figure 6 was obtained while using an implementation of an Exponential Grid that made use of the Numpy **ndarray** object.

There a few different conclusions that one may draw from these results. First, it is possible that the implementation of the Fréchet Grid is not optimal and thus is causing unnecessarily long execution times. While this is very possible, we took care to ensure that there did not exist unnecessary points in the implementation of the Exponential Grid, and thus we do not think that this is the major cause.

Secondly, it is possible that the selection of implementing the data structure in Python is to blame for these run times. While it is true that implementing the data structure in C would shave off a constant factor of the runtime, it is unclear how much time would be saved. Recall that this implementation makes use of Numpy **ndarray** objects - we see in Figure 5 that this already boosts the efficiency of the implementation a great deal. As such, it is likely that an identical implementation in C/C++ would experience the same execution time bottleneck, perhaps just for a smaller value of ϵ.

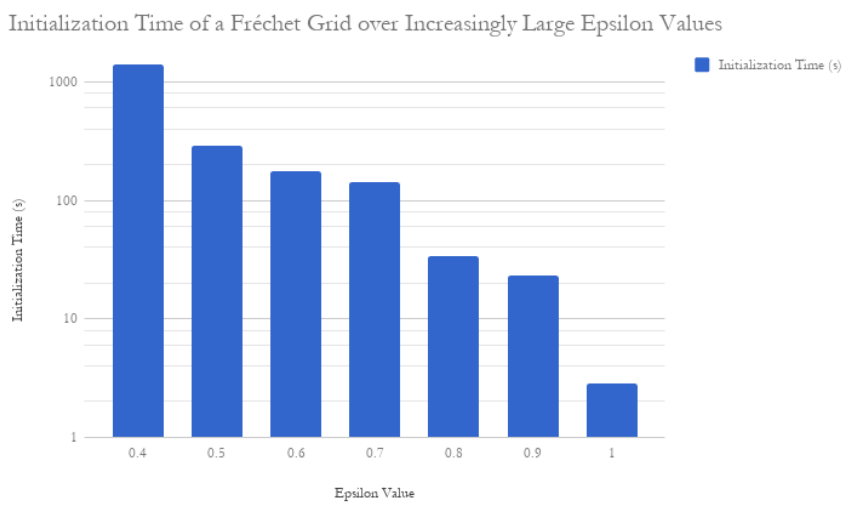

Figure 6: Execution time performance of initializing a Fréchet Grid over increasingly large values of epsilon. Note the logarithmic scale.

Finally, it is possible that the preprocessing time of the data structure itself contains constants that are impractically large as ϵ approaches 0. While this cannot be verified as true until the previous two hypotheses are proven incorrect, it is the hypothesis of the writer that this particular hypothesis is correct. Throughout experimentation with initializing the Fréchet Grid, it was noted that most of the preprocessing time is spent in the doubly nested for-loop while computing the Fréchet Distances of the set $G(u) \times G(v)$. Note that the asymptotic time complexity of this loop is quadratic in the number of points in $G(u)$ and $G(v)$, since $|G(u)| = |G(v)|$ as both Exponential Grids are initialized with the same parameters. As ϵ approaches 0, $|G(u)|$ and $|G(v)|$ grow much larger, and this growth is felt to a polynomial degree in the Fréchet Grid.

Regardless of these results, the initialization time for creating a Fréchet Grid with $\epsilon = 1$ was relatively fast at $\approx 3s$. If we consider that we gain a significant reduction in runtime for settling for a 2-approximation of the Fréchet Distance, the trade-off suddenly does not seem to be as bad.

3.3 The Curve Range Tree

The Curve Range Tree data structure was implemented as a standard binary tree. Initialization takes as input a polygonal curve, Z, as well the parameters ϵ and Δ. The preprocessing involves recursively building the tree, and performing the path decomposition on the tree. The requirement for performing the path decomposition on the tree stems from the fact that we need, in the query algorithm, to determine the lowest common ancestor of two nodes in the tree.

While the implementation described in [1] does not use the *lca* implementation discussed in [8], it was noted that implementing the *lca* algorithm cited in [1] would be a significant amount of work. Thus, we simply leverage the *lca* algorithm of [8] to save implementation time.

Using the recursive structure illustrated in Figure 3, we build the binary tree such that the leaves of the tree only store the edges of Z. Each node object within the tree maintains a pointer to its parent, its left child, its right child, its group parent (used in the decomposition), the subpath of Z at that particular node, and the Fréchet Grid for the subpath of Z stored in the node. Finally, the path decomposition is computed as discussed in section 2.4.2.

Most of the details of the query algorithm are omitted, as they follow quite closely to the description of the query algorithm in [1]. However, an important implementation detail worth considering is the implementation of the algorithm for computing the heaviest edge on the bottleneck path of the DAG. Please refer to Algorithm 1 for the pseudocode implementation.

It is important to note that Algorithm 1 does not compute the heaviest weight on the bottleneck path for all DAGs. Instead, it relies on the assumption that all paths from the start node terminate at the end node given as input to the algorithm.

The algorithm attempts to traverse all possible paths from s to e in the DAG, in-turn traversing subpaths from s_t to e. It maintains at each subpath the bottle-

Algorithm 1 Pseudocode for computing the heaviest edge on the bottleneck path from a node s to e. The algorithm assumes that nodes s and e belong to a DAG, and that all paths from s lead to e

```
1: function ComputeBottleneck($s$, $e$)
2:     function c($s_t$, $w$)
3:         $A \leftarrow$ {all nodes adjacent to node $s_t$}
4:         $W_A \leftarrow$ {all edge weights from $s_t$ to nodes adjacent to node $s_t$}
5:         if $s_t = e$ then
6:             return $w$
7:         $w = \max(\min_{t \in W}(t), w)$
8:         $Z = \langle\rangle$
9:         for $n \in A$ do
10:            $w_n$ = the weight of the edge from $s_t$ to $n$
11:            $Z.add(c(n, w_n))$
12:        return $\max(\min_{t \in Z}(t), w)$
13:    return $c(s, 0)$
```

neck edge weight for that particular subpath, comparing this weight to all other subpaths that have branched off at any point during the search.

3.3.1 Challenges/Experiments

It was mentioned in Section 2.3.2 that an important implementation consideration would be to initialize the Fréchet Grids stored at each node of the Curve Range tree with ϵ rather than $\epsilon/2$. Based on the preprocessing time performance illustrated in Figure 6, it should be a little bit clearer as to why this suggestion is made.

Let us assume that we initialize a Curve Range tree data structure with $\epsilon = 1$. We assume that the input curve Z contains 5 vertices. Therefore, the binary tree built on the polygonal curve Z will contain 7 nodes and will require initialization of 7 Fréchet Grid data structures, each initialized with $\epsilon = 0.5$. Based on Figure 6, let us assume that initializing each Fréchet Grid takes $100s$. We therefore have that initialization of the Curve Range tree structure would take at least $7 \times 100s = 700s \approx 11.5m$. Clearly, this initialization time is substantial for a curve of only 5 vertices. Based on this reasoning, it was determined that a more time-efficient approach would be to initialize the Fréchet Grids stored at each tree node with the full value of ϵ.

Beyond the challenge of determining the correct value of ϵ to even-out the preprocessing performance of this data structure due to the Fréchet Grid initialization, testing found that time efficiency for the query algorithm was not a problem. It is noted by the writer, however, that the initialization time for Fréchet Tree data structure could also suffer similar problems, since the Fréchet Tree data structure initializes a Curve Range tree data structure for each of the paths in the path decomposition $PD(T)$. In effect, the implementation of the Fréchet Tree would most likely be throttled even more by the Fréchet Grid instantiation, however there is no empirical evidence to support this claim since the implementation of the Fréchet Tree was never completed.

4 Conclusion

In summary, only a small portion of the data structures and algorithms discussed in [1] were implemented over the course of this project. For the implementations that were realized, it was found that we suffer from a preprocessing execution time bottleneck due to the magnitude of points contained in the Fréchet Grid data structure. It is believed that the constants associated with the preprocessing time of this data structure become too high as ϵ approaches 0. We have further seen how this bottleneck affects the implementation the Curve Range tree data structure, which maintains a one-to-many relationship with the Fréchet Grid.

Although concrete implementations and thus experiments are lacking for the other data structures described in [1], it is believed that they would also suffer from the preprocessing bottleneck caused by the Fréchet Grid, since each data structure builds upon the Fréchet Tree/Curve Range Tree structures.

However, it is also possible that the performance issues experienced could be attributed to the decision to implement the project using Python. Although the writer believes porting the code to a lower-level language such as C/C++ would save a constant factor of time, the writer still believes that computing the Fréchet Grid structure would constitute a bottleneck relative to each implementation's total preprocessing time.

This project also consisted of many other less tangible work items not necessarily discussed in this paper. Over the course of the term, much time was spent deciphering the literature in order to grasp the concepts being described. Furthermore, the writer developed a better understanding of the skill set required to translate theory to practical implementation. Sometimes, single line descriptions of algorithm steps in the literature would translate to hundreds of lines of code in practice. Regardless, the writer believes that having the ability to directly translate theory to implementation is a worthwhile skill to have, and is grateful for the opportunity to have spent time furthering his knowledge towards an open problem in the field this term.

References

[1] M. Smid and J. Gudmundsson. Fast Algorithms for Approximate Fréchet Matching Queries in Geometric Trees. *Computational Geometry - Theory and Applications*, 48:6:479–494, 2015.

[2] Anne Driemel. Realistic Analysis for Algorithmic Problems on Geographical Data. *Utrecht University*, 51–53, 2013.

[3] H. Alt and M. Godau. Computing the Fréchet distance between two polygonal curves. *International Journal of Computational Geometry & Applications*, 5:75–91, 1995.

[4] S. Har-Peled and B. Raichel. The Fréchet distance revisited and extended. *27th Annual Symposium on Computational Geometry*, 448–457, 2011.

[5] T. Eiter and H. Mannila. Computing Discrete Fréchet Distance. *Technische Universität Wien*, 1994.

[6] Geometry Tool. *GeoGebra - Free Online Geometry Tool*, Math10. `https://www.math10.com/en/geometry/geogebra/fullscreen.html`

[7] M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars. Computational Geometry - Algorithms and Applications (Third Edition). *Springer*, 105–109, 2008.

[8] M. Smid and G. Narasimhan. Geometric Spanner Networks. *Cambridge University Press*, 21–23, 2007.

[9] C. Ermel. *Frechet Matching Queries.* `https://github.com/ermel272/frechet-matching-queries`

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

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

[12] M. Fréchet. Sur quelques points du calcul fonctionnel. *Rendiconti del Circolo Mathematico di Palermo*, 22:1–74, 1906.