A GPU approach to subtrajectory clustering using the Fréchet distance

Joachim Gudmundsson[*]
University of Sydney and NICTA[†]
Sydney, Australia.
joachim.gudmundsson@sydney.edu.au

Nacho Valladares[‡]
IMA, Universitat de Girona
Girona, Spain
ivalladares@ima.udg.edu

ABSTRACT

Given a trajectory T we study the problem of reporting all subtrajectory clusters of T. To measure similarity between curves we choose the Fréchet distance. We show how the existing sequential algorithm can be modified exploiting parallel algorithms together with the GPU computational power showing substantial speed-ups.

This is to the best of our knowledge not only the first GPU implementation of a subtrajectory clustering algorithm but also the first implementation using the continuous Fréchet distance, instead of the discrete Fréchet distance.

Categories and Subject Descriptors

F.2.0 [**Analysis of algorithms and problem complexity**]: General; D.1.3 [**Parallel programming**]: Concurrent Programming

General Terms

GPU programming, Algorithms

Keywords

Clustering, movement patterns, GPU, parallel programming

1. INTRODUCTION

Recent technological advances of location-aware devices produce numerous opportunities to trace moving entities. Due to this the number of commercial and academic projects around the world tracking animals, people and vehicles has increased dramatically. For example, elephants are tagged to map their migration behaviors which is needed for urban planning, the movement of soldiers is monitored to analyze how they react to different conditions and how they behave in a group, vehicles are tracked to detect commuting behaviors with the aim of improving traffic conditions and, finally, the positions of football players are extracted to analyze their performance.

In this paper we focus on the problem of detecting subtrajectory clusters in a trajectory. That is, we search for similar subtrajectories along a single trajectory, as is illustrated in Fig. 1a. Depending on the application a subtrajectory cluster (SC) can have a different meaning. For example, for a person a SC might indicate a daily commuting behavior between home and work. When tracking birds a SC might indicate annual migration behavior or regular movement between the nest and foraging areas. A final example, which we will focus our attention on in this papers, is football where a subtrajectory cluster indicates frequent movements on the field performed by a player, see Fig. 1b for an example.

More formally, following the definition from [7], the input is a moving point object in d-dimensional Euclidean space, called *entity*, whose location is known at n consecutive time-steps. We assume that an entity moves between two consecutive time steps on a straight line. Given the trajectory T of a moving entity, an integer $m > 0$ and two positive real values ℓ and ε, we define a subtrajectory cluster C as a set of at least m subtrajectories of T, where the subtrajectories are within distance ε from each other, at least one subtrajectory has length ℓ and the time intervals of two subtrajectories in C overlap in at most one point.

To measure spatial similarity we choose the Fréchet distance between (sub)trajectories. In this paper we only consider the trajectory as a directed curve in 2D, thus invariant under differences in speed: for instance, in a transportation context, this allows us to detect a commuting pattern even in the presence of different traffic conditions and varying means of transport. However, speed can be taken into account by considering time to be the third dimension. The algorithm presented in this paper works for any dimension, so does the Fréchet metric, but for the application we consider the variance in speed is of minor importance.

The problem was considered in [7] in which the authors also developed approximation algorithms using both the discrete Fréchet distance and the continuous Fréchet distance. Their algorithms have a running time of $O(\ell n^2)$ and $O(n^3 m \cdot 2^{\alpha(n/m)}(\log n \log(n/m) + m))$ when using the discrete and continuous Fréchet distance (to be defined), respectively. In the case when at least one of the subtrajectories in a cluster

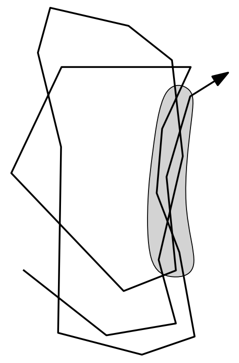
[*]Funded by ARC grant FT100100755.

[†]NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

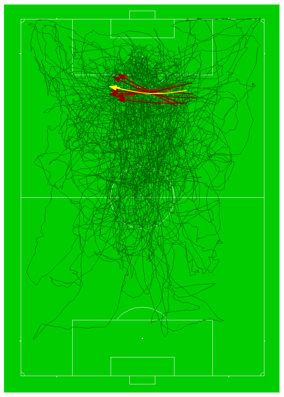
[‡]Partially supported by the Spanish MCI grant TIN2010-20590-C02-02.

has to start and end at vertices of the trajectory the algorithm requires $O(n^2\ell)$ time and $O(n\ell^2)$ space, which is also the version we will focus on in this paper. Their algorithm can be modified to return many different outputs depending on the settings, however, in general it reports a set of subtrajectory clusters where each cluster C in the set contains at least m subtrajectories of T, at least one of the subtrajectories has length ℓ and the Fréchet distance between any two subtrajectories in C is at most 2ε. Due to this the algorithm is said to be a 2-distance approximation algorithm.

For the discrete Fréchet distance only distances between vertices are computed. The discrete Fréchet distance works well in the case when the input data is very dense. In these cases there is only a small difference between the discrete and continuous Fréchet distance. However, in most cases the data is not dense, and even in the case when the input data is dense one often wants to compress the data by using a simplification algorithm [8, 14]. In most realistic settings the input data can be compressed to $5-10\%$ of its original size, thus developing a practical approach for trajectory clustering using the continuous Fréchet distance is a crucial open problem. Note that due to the high complexity only the algorithm using the discrete Fréchet distance was implemented and tested in [7].



(a)



(b)

Figure 1: (a) Illustrating a subtrajectory cluster. (b) An example of a subtrajectory cluster reported from a trajectory of a football player.

1.1 Related results

Vlachos et al. [22] state that in the area of spatio-temporal analysis it is important to detect commuting patterns of a single entity, or to find objects that move in a similar way. An efficient clustering algorithm for trajectories, or subtrajectories, is essential for such analysis tasks. Gaffney et al. [12, 13] proposed a model-based clustering algorithm for trajectories. In this algorithm, a set of trajectories is represented using a regression mixture model. Specifically, their algorithm is used to determine cluster memberships and it only clusters whole trajectories. Lee et al. [17] argued that clustering trajectories as a whole would not detect similar portions of the trajectories. Instead they suggested an approach that partitions a trajectory into a set of line segments and then group similar segments. The obvious drawback is that only segments are clustered, while a commuting pattern might be a complicated path whose shape may not be captured by a single segment. Mamoulis et al. [18] used a different approach to detect periodic patterns. They assumed that the trajectories are given as a sequence of spatial regions, for example ABC would denote that the entity started at region A and then moved to region C via region B. Using this model data mining tools such as association rule mining can be used. Vlachos et al. [22] also looked at discovering similar trajectories of moving objects. They mainly focused on formalising a similarity function based on the longest common subsequence which they claim is very robust to noise. However, their approach matches the vertices along the trajectories which requires that the vertices along the trajectories are synchronized, or almost synchronized, which is rarely the case. If the trajectories are simplified (compressed) in a preprocessing step then the data will also not be synchronized.

1.2 Problem setting

As mentioned in earlier work [5, 16], specifying exactly which of the patterns should be reported is often a subject for discussion. In this paper we focus on reporting all SCs whose length is above a given threshold ℓ containing at least m subtrajectories. A cluster C of (sub)trajectories has length at least ℓ if there exists a (sub)trajectory in C whose length along T between its start- and endpoint is at least ℓ.

Definition 1. *Given a trajectory T with n vertices, a* subtrajectory cluster $C_T(m,\ell,\varepsilon)$ *for T of length ℓ consists of at least m subtrajectories $\tau_1,\ldots,\tau_m$ of T such that the time intervals for any two subtrajectories in the cluster overlap in at most one point, the distance between the subtrajectories is at most ε, and at least one subtrajectory has length ℓ.*

In Definition 1 we omitted to define the distance metric. Many of the fundamental problems in this area of research have one issue in common, namely calculating the similarity between two trajectories (or subtrajectories). Simplified, the problem is as follows. Given two (polygonal) curves f and g, one would like to match them up in an optimal way (i.e., find a continuous mapping from the points of f to the points of g, so that the mapping maps the endpoints of one curve to the endpoints of the other curve). Such a mapping is especially useful if it is associated with an appropriate metric. The Fréchet metric is one of the most natural measures of this type: For completeness, we include the formal definitions of these distance measures here.

Definition 2. *Let $f: I=[l_I,r_I]\to\mathbb{R}^d$ and $g: J=[l_J,r_J]\to\mathbb{R}^d$ be two curves, and let $|\cdot|$ denote the Euclidean norm[1] in $\mathbb{R}^d$. Then the Fréchet distance $\delta_F(f,g)$ is defined as*

$$\delta_F(f,g)=\inf_{\substack{\alpha:[0,1]\to I\\ \beta:[0,1]\to J}}\max_{t\in[0,1]}|f(\alpha(t))-g(\beta(t))|,$$

where α and β range over continuous and increasing functions with $\alpha(0)=l_I$, $\alpha(1)=r_I$, $\beta(0)=l_J$ and $\beta(1)=r_J$.

This is also known as the person-dog metric: imagine a person walking on f and a dog walking on g. The Fréchet distance between f and g is the shortest leash that enables both the person and the dog to travel along f and g with a

[1]Other norms are possible as well, see Alt and Godau [3].

leash connecting them. Playing around with this metric, one realizes that because of the continuity requirement on the mapping f, this measure is in most cases much better suited than the classical Hausdorff distance between the curves [4], or many other proposed metrics (the Longest Common Subsequence model [22], a combination of parallel distance, perpendicular distance and angle distance [17], the average Euclidean distances between paths [19] and so on [15]). Unfortunately, in the continuous setting, computing the Fréchet distance between two curves requires quadratic time in the complexity of the curves [3]. It is currently an open question if it is possible to do better than quadratic even when one tries to approximate this quantity within a constant multiplicative factor. Alt [2] conjectured that the decision problem may be 3SUM-hard (i.e. a subquadratic time algorithm is unlikely to exist). The only subquadratic algorithms known are for restricted classes of curves such as for closed convex curves and when the input has low densityĄ(k-packed, k-straight, k-bounded) [10]. Very recently Agarwal et al. [1] showed that the discrete Fréchet distance can be computed in subquadratic time, namely in $O(n^2 \frac{\log\log n}{\log n})$ time.

For a set of $m > 2$ curves there is a natural extension due to Dumitrescu and Rote [11].

Definition 3. *For a set of m curves $\mathcal{F} = \{f_1, \ldots, f_m\}$, $f_i : [a_i, a'_i] \to \mathbb{R}^d$ we define the Fréchet distance as*

$$\delta_F(\mathcal{F}) = \inf_{\substack{\alpha_1:[0,1]\to[a_1,a'_1] \\ \vdots \\ \alpha_m:[0,1]\to[a_m,a'_m]}} \max_{t\in[0,1] 1\leq i,j\leq m} |f_i(\alpha_i(t)) - f_j(\alpha_j(t))|,$$

where $\alpha_1, \ldots, \alpha_m$ range over continuous and increasing functions with $\alpha_i(0) = a_i$ and $\alpha_i(1) = a'_i$, $i = 0, \ldots, m$.

1.3 GPU programing model

GPU manufacturers provide tools and software so users can exploit GPU parallel capabilities. Each manufacturer has its own technology and consequently its own software. For instance, the AMD/ATI technology is based in the AMD App acceleration model, while Nvidia GPU's are CUDA based. Despite the parallel algorithm presented in this paper can be applied to any parallel technology we use a NVidia GPU card for the implementation.

CUDA (Compute Unified Device Architecture) [20, 21] is the computing engine in NVidia GPUs which is designed to support various languages and application programming interfaces. It is accessible to software developers through variants of industry standard programming languages like *CUDA c* or *OpenCL*.

CUDA programming model.

The basis of the CUDA programming model are threads. Threads are lightweight processes which are easy to create and to synchronize. The user writes programs called kernels and its execution is scheduled according to the distribution of threads. Threads are organized into blocks and blocks are organized into a grid. The grid and the block dimensions are defined according to the needs of the problem. All threads of the same kernel are executed in parallel and, typically, each one computes a result element. The GPU contains a number of multiprocessors consisting of a set of independent processors and each processor is responsible for executing one thread.

Memory model.

CUDA allows users to access GPU memory concurrently. The user can choose different types of memory and various access patterns which significantly affects the performance.

Registers are the fastest type of memory. They are used to store local variables of a single thread and can only be accessed by that thread. Each thread is limited to a few registers per thread depending on the number of threads used and the GPU capabilities. **Shared** memory is a very fast memory which can be as fast as registers if accessed properly. It is accessible to all thread within a block, thus enabling cooperation. Its limited in size (up to 48 KB) which sometimes can makes it complicated to use. The last type of memory is the **global** memory which is the largest and slowest type of memory and is the only one visible from the CPU. Even though it has an impressive bandwidth, it has a high latency which is hidden by a large number of parallel accesses. Depending on the GPU capabilities this area can be up 8 GB size.

Memory access pattern is an important issue when programming. Each memory has its own correct access pattern which should be performed. Despite it being a difficult task one has to take it into account since the algorithm's efficiency is highly dependant on the way the memory is accessed.

Atomic operations.

CUDA concurrent memory access is a powerful feature which, in some situations, can entail simultaneous memory reads and writes causing unexpected results. This is given by the so-called thread 'race condition'. For a group of threads that are executed in parallel the thread order execution may differ between executions, even when executing exactly the same code. This follows from the fact that there is no guarantee that two or more threads reading and writing in the same position of memory will produce the same result for different executions. This is a typical scenario in parallel computing. CUDA provides a set of operations called 'atomic operations' to solve this issue.

Atomic operations are operations which are performed without interference from any other threads. If a thread is computing an atomic operation over a position of memory no other thread can read or write over this position until the atomic operation is finished. This ensures the correct computation but atomic operations are very slow compared to non-atomic operations.

2. THE DATA STRUCTURE

In the following sections we will show how to construct the data structure in the GPU needed to compute the subtrajectory clusters, then in Section 3 we show how it can be used to report the clusters. The algorithm builds upon the corresponding CPU algorithm by Buchin et al. [7], which we now will describe in more detail.

Their algorithm uses the *free space diagram* of two polygonal curves h and g, which is a geometric data structure introduced by Alt and Godau [3] for computing the Fréchet distance. Let h be a polygonal curve with n vertices $p_1, \ldots, p_n$ and let g be a polygonal curve with m vertices $q_1, \ldots, q_m$. We use ϕ_h to denote the following natural parameterization of h. The map $\phi_h \colon [1, n] \to \mathbb{R}^d$ maps $i \in \{1, \ldots, n\}$ to p_i and interpolates linearly in between vertices. The free space diagram of two polygonal curves h and g, with n and m

vertices, respectively, is the set

$$F_\varepsilon(h,g) = \{(s,t) \in [1,n] \times [1,m] : |\phi_h(s), \phi_g(t)| \leq \varepsilon\}$$

which describes all tuples (s,t) such that the points $\phi_h(s)$ and $\phi_g(t)$ have Euclidean distance at most ε, the so-called free space of the diagram. See Fig. 2 for an example of a free space diagram with $n-1$ columns and $m-1$ rows, the white region in the diagram is the free space. Alt and Godau [3] showed that the Fréchet distance between h and g is less than ε if and only if there exists an xy-monotone path in the free space of $F_\varepsilon(h,g)$ from $(0,0)$ to (n,m).

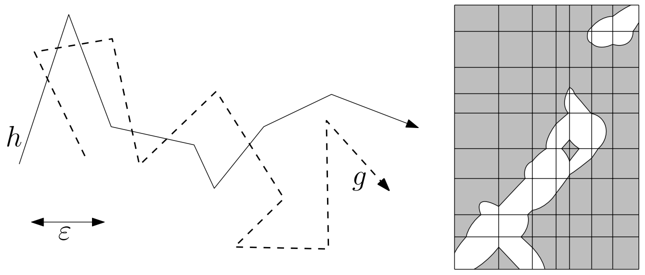


Figure 2: Two trajectories h and g and the corresponding free space diagram with respect to ε.

Consider a free space diagram $F_\varepsilon(T,T)$ of a polygonal curve T and itself. In [7] it was noted that if a subtrajectory τ of length ℓ within T belongs to a subtrajectory cluster $SC(m,\ell,2\varepsilon)$ then there exists m xy-monotone paths from the vertical line intersecting the x-axis at the start point of τ and the vertical line intersecting the x-axis at the endpoint of τ. Furthermore, the projections of the xy-monotone paths onto the y-axis are not allowed to overlap (otherwise the subtrajectories in the cluster would overlap along T). We call τ the reference subtrajectory. See Fig. 3 for an example. Thus the aim is to find all maximal length intervals along the x-axis where there are m xy-monotone paths from the left vertical boundary to the right vertical boundary of the interval.

We assume that the reference subtrajectory will always start and end at a vertex of T, while no restriction is made on the start and end points of the other members of the SC. In $F_\varepsilon(T,T)$ each *cell* corresponds to two line segments of T and the free space in one cell is the intersection of the cell with an ellipse, possibly degenerated to the space between two parallel lines [3]. There are at most eight intersection points of the boundary of the cell with the free space. We call these intersection points *critical points*, see Fig. 4. For each critical point, place a vertex on it in the free space diagram. Furthermore, all critical points in the free space of the corresponding row or column are propagated. That is, propagate critical points on vertical cell boundaries horizontally to the right, and critical points on horizontal cell boundaries vertically upwards. Consider a critical point p on a vertical boundary. Move p horizontally to the right in the diagram until it hits a non-free space boundary. Every time p moves over a cell boundary an intersection point is added. When p hits the free space boundary the propagation stops. Each critical point can generate at most $n-1$ propagated critical points. Figure 5 shows an example of horizontal propagation.

Following the idea presented in [7] we use a labeled graph with the critical points and the propagated points as vertices

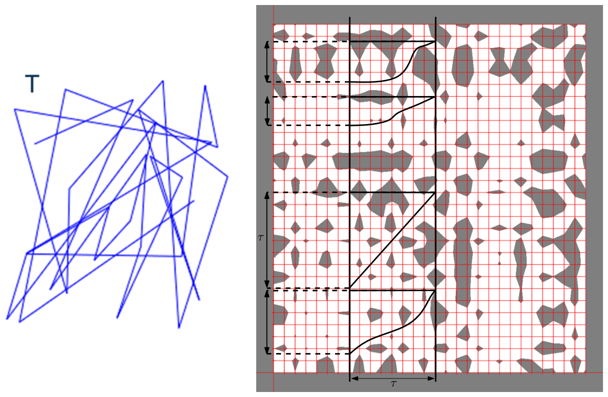


Figure 3: A free space diagram of a trajectory T with itself. The four xy-monotone curves represents a cluster with four subtrajectories of T within distance ε from τ, including τ itself.

so we can later navigate through the graph in order to find the clusters. Each critical and propagated point has a label that stores the smallest x-coordinate reachable by an xy-monotone path via this point.

These are the data structures needed to compute the clusters. In this section we will show how to construct these in the GPU. After the structure has been constructed one can easily find the subtrajectory clusters. In Section 3 we will show how this is done in the GPU. However, for the reader to have a full understanding of the approach we next describe the CPU approach.

Using the labeled graph, one can determine whether there are m cluster curves between two vertical lines l_s and l_t. This is done by greedily searching for monotone paths from l_s (start boundary) to l_t (end boundary). Let i be the x-coordinate of l_s. Start at the topmost vertex on l_t which has an outgoing edge whose label is at most i. In each vertex follow the topmost edge whose label is again at most i. This ends on a vertex (i,j) on l_s. Continue with the topmost vertex on l_t at height at most j which as before has an outgoing edge whose label is at most i. Stop when we have found m curves, or no more vertices on l_t with an outgoing edge whose label is at most i exist. Note that the edge labels prevent us from going into dead ends in the graph.

To obtain the labeled graph we need to show how to (1) compute the free space diagram (Section 2.1), (2) propagate the points (Section 2.2) and (3) label all the points of the free space diagram (Section 2.3) in the GPU model. The labeled graph is then used to report the clusters (Section 3.1).

In the following sections we will focus on a subproblem to the subtrajectory clustering problem. Given two trajectories h and g, and a positive constant ε we next show how to construct the free space diagram for h and g with respect to ε in the GPU.

2.1 Computing the free space diagram in the GPU

Inside the GPU it is not possible to use dynamic memory so all the memory required by a kernel must be allocated before it is executed. Furthermore the data structure must be optimized so that a maximum number of threads can

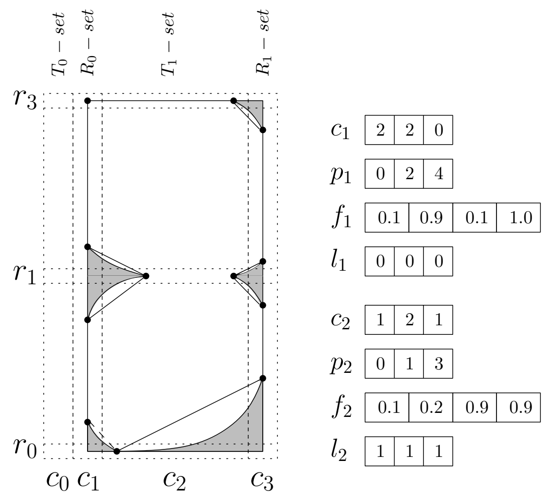


Figure 4: An example of $\mathcal{F}(h, g)$ where $n = 1$ and $m = 2$.

access any position in memory in constant time. Here we will discuss the structure used to compute the free space diagram.

2.1.1 *The free space diagram inside the GPU*

As input we are given two trajectories $h = \langle h_1, \ldots, h_n \rangle$, $g = \langle g_1, \ldots g_m \rangle$ and a positive constant ε. Next we will show how to construct the free space diagram in the GPU. Each cell in the free space diagram is a convex polygon, but instead of storing the whole polygon structure we just store the critical points (the intersections between the boundary of the cells with the free space), see Fig. 4. There are at most eight critical points per cell in the free space diagram.

The critical points of each cell is divided into two sets; the set of points on the top boundary (T-set), not including the ones in the corners of the cell, and the set of points on the right boundary (R-set). Thus each cell in the free space diagram will be stored as two sets. We store every critical point as floats in the interval $[0, 1]$ indicating the distance from the left or bottom boundary of the cell.

We create a matrix M with $2(n + 1)$ columns and $m + 1$ rows, $M[0..2(n + 1) - 1, 0..m]$. For each column in the free space diagram we have two columns in M; one for the T-set and one for the R-set. The first two columns of M correspond to the left boundary of the leftmost column in the free space diagram. Note that the T-set in this column ($T_0 - set$) is an empty set and it is only included to make the structure uniform. Finally, there is an extra row for the points on the bottom boundary of the bottommost row, Fig. 4.

Inside the GPU, each column i of the matrix M is stored in three arrays. The counting array $c_i[0..m]$ is an integer array such that $c_i[j]$ stores the number of critical points in cell (i, j) of M. The positioning array $p_i[0..m]$ is also an integer array and stores the prefix sum of c_i, that is

$$p_i[0] = 0 \quad and \quad p_i[j > 0] = \sum_{k=0}^{j-1} c_i[k] = p_i[j-1] + c_i[j-1].$$

Finally, the critical points array $f_i[1..k]$ is a float array, where k is the total number of critical points in column i of M. With this structure we can easily access any cell or critical point using the fact that the critical points of the cell (i, j) in the free space diagram starts at position $(p_{2i}[j] + 1)$ of f_{2i} and is stored in $c_{2i}[j]$ consecutive positions of f_{2i}. We will refer to this entire structure, (M, c, p, f), as $\mathcal{F}(h, g)$, and $\mathcal{F}_i(h, g)$ as the ith column of $\mathcal{F}(h, g)$. An example is shown in Fig. 4.

Finally an additional array l is initialized to store the label of each point (to be defined in Section 2.3).

In the rest of this section we assume that $\mathcal{F}_i(h, g)$ fits in the GPU memory. However this is not always possible and in Section 3.2.1 we will explain how to partition the free space diagram into smaller pieces where the data structure for each piece can fit in the GPU memory.

2.1.2 *Computing the critical points using the GPU*

An optimal scenario in parallel computation is when a single element can be computed and reported without the interference from other elements and without requiring the results from the computation of other elements. Luckily the computation of the critical points is an optimal scenario since each critical point is independent from the others, which allows us to compute them very efficiently in parallel. This process is performed in three steps:

Step 1:

Launch a parallel kernel with $(n+1) \times (m+1)$ threads, one thread for each cell (i, j) in the free space diagram. Here we imagine a 0th row and column. Each thread computes the number of critical points in the T- and R-points for that cell. The number of points in the T-set of (i, j) is stored in $c_{2i}[j]$ and the number of points in the R-set of (i, j) is stored in $c_{2i+1}[j]$.

Step 2

Perform a parallel scan [9] over c_i to obtain the values needed to populate the position array p_i.

Step 3:

For each i, $0 \leq i \leq 2n + 1$, allocate $p_i[m] + c_i[m]$ memory to f_i. Launch a kernel with $(n + 1) \times (m + 1)$ threads, one per cell in the free space diagram. A thread (i, j) stores the T_i and R_i critical points starting at position, $p_{2i}[j] + 1$ for the T-set and $p_{2i+1}[j] + 1$ for the R-set.

2.2 Propagate the critical points

We need to propagate the critical points of $\mathcal{F}(h, g)$ vertically and horizontally. This process is easy to parallelize since the propagation of a critical point does not affect the rest of the propagations. The process is performed in four steps, see Fig. 5.

Step 1:

For each i, $0 \leq i \leq 2n+1$, create an auxiliary array $\mathtt{accum}_i$ of size $m + 1$ which will be used to keep track of the number of propagated points in each cell. To populate `accum` we launch a kernel with as many threads as critical points in $\mathcal{F}(h, g)$, assigning to each thread a critical point and propagating that critical point horizontally or vertically, depending on if the critical point lies on a horizontal or vertical boundary.

Consider a critical point q in a cell (i, j) of $\mathcal{F}(h, g)$. If column i represents a T-set (i is even) then q is propagated vertically upward otherwise, if column i represents an R-set, it is propagated horizontally to the right. Assume without

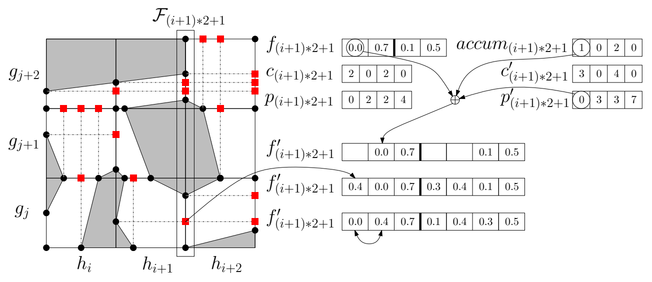

$\mathcal{F}_{(i+1)*2+1}$
g_{j+2}
g_{j+1}
g_j
h_i
h_{i+1}
h_{i+2}
$f_{(i+1)*2+1}$ 0.0 0.7 0.1 0.5
$c_{(i+1)*2+1}$ 2 0 2 0
$p_{(i+1)*2+1}$ 0 2 2 4
$accum_{(i+1)*2+1}$ 1 0 2 0
$c'_{(i+1)*2+1}$ 3 0 4 0
$p'_{(i+1)*2+1}$ 0 3 3 7
$f'_{(i+1)*2+1}$ 0.0 0.7 0.1 0.5
$f'_{(i+1)*2+1}$ 0.4 0.0 0.7 0.3 0.4 0.1 0.5
$f'_{(i+1)*2+1}$ 0.0 0.4 0.7 0.1 0.4 0.3 0.5

Figure 5: Critical points marked as black disks and the propagate points as red squares.

loss of generality that column i is even. Move q vertically upwards until it either hits a non-free space boundary or until it intersects a horizontal cell boundary. Every time q hits a cell boundary an intersection point is added. When q hits the free space boundary the thread immediately terminates.

Thus, if q belongs to a T-set then it might generate up to $m - i$ new propagated points and if it belongs to an R-set then it might generate up to $n-j$ new propagated points. At each iteration in the propagation the corresponding `accum` cell value is incremented by 1 indicating that there is a new propagated point in this cell. At the end of the process `accum` stores the number of propagated points in each cell.

A complication is that two different critical points can be propagated to the same cell boundary, and consequently to the same `accum` position at the same time in the process. This can lead to memory interference between threads. To avoid this situation we use atomic operations when writing to `accum`.

Step 2:

For each i, $0 \leq i \leq 2n + 1$, create an integer array $c'_i[0..m]$ and set $c'_i[j] = c_i[j]+\texttt{accum}_i[j]$, for $0 \leq j \leq m$. Also create an integer array $p'_i[1..m]$ and set $p'_i[j]$ to the sum of $c'_i[1..j]$.

Step 3:

For each i, $0 \leq i \leq 2n + 1$, create f'_i of size $p'_i[j] + c'_i[j]$ and then run a kernel with one thread per critical point in $\mathcal{F}(h, g)$. Note that we now repeat the propagating process but this time storing the propagated points in f'. Once this is done $\mathcal{F}'(h, g)$ stores the free space diagram including all the propagated points.

Note that the corner points never have to be propagated (either they already exist or they do not lie in the free space).

Step 4:

For each cell (i, j) in M, if it is a T-set then place the leftmost point at position $(p'_i[j] + 1)$ of f'_i and the rightmost point at position $p'_i[j + 1]$ of f'_i. Similarly, if it is an R-set then place the bottommost point at position $(p'_i[j]+1)$ of f'_i and the topmost point at position $p'_i[j + 1]$ of f'_i. This can be done in parallel with one thread per cell, Fig. 5. This step is needed to speed up the labeling step that will be describe next.

2.3 Parallel labeling

The labeling process is the most expensive part of the algorithm since there is no natural way to parallelize it. The reason for this is that the label of the points in cell (i, j) in the free space diagram depends on the labels of all the points to the left and below cell (i, j). We say that a point u is *reachable* from a point v if there is an xy-monotone path from u to v. Imagine the situation showed in Fig. 6 were the regions reachable from a are marked in stripes, the regions reachable from b in lightgray and the regions reachable from both a and b is shown as checkered.

In Fig. 6 the point v (f[43]) is reachable from both a and b. If we label all the points in parallel then $f'[43]$ could be labeled with any of the two labels a or b depending on the thread race condition. This is a problem since the algorithm requires information at each point in the free space diagram of the leftmost point it can reach with an xy-monotone path. This is crucial in deciding if there exists a subtrajectory cluster or not.

To get around this problem we label the points as follows. We use an integer array $l[1..k]$, where k is the total number of critical and propagated points in the free space diagram. For each critical and propagated point p we store a label in $l(p)$, where p is associated to an integer in $[1, k]$. Initially a critical point is labeled with the row index of the cell it belongs to and a propagated point is initially labeled with the same label as the label of the critical point that induced it.

In order to avoid excessive relabeling we compute the labeling in several steps. Recall that the labels in f'_0 and f'_1 of M do not have to be re-labeled. So first label all points in f'_2 (T-sets) with labels from the points in f'_1, then f'_3 with the labels from the points in f'_2 and f'_1, f'_4 with the labels from the points in f'_3, f'_5 with the points in f'_4 and f'_3, and so on. To do this efficiently in parallel we run a kernel with one thread per point (critical and propagated) in f'_i and each thread labels all its visible points. Navigating through $\mathcal{F}$ is easily done using the first and last point of each $f'_i[j]$.

Even though this avoids a large part of the unnecessary re-labeling we still have points that are labeled several times. Some of them can be avoided by defining a new stop condition based on the idea of which points are reachable by an xy-monotone path. Observe that all points reachable from a point u are reachable from a point v if u is reachable from v. Using this observation, we can force a thread to stop when it tries to re-label a point v with the same label, or a smaller label, since this implies that another thread already labeled it and will label all other points that are reachable from v.

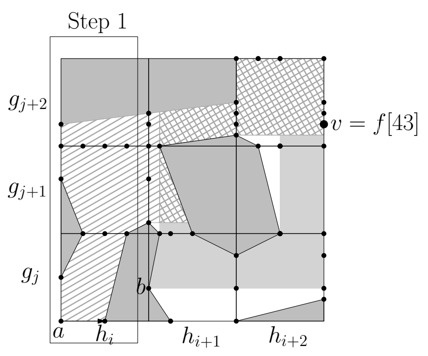
This concludes the construction of the data structure.

3. REPORTING SUBTRAJECTORY CLUSTERS

Which clusters that should be reported is often a topic for discussion, since it often depends on the application. For instance, is it necessary to report a subtrajectory which has already been reported in another cluster? Do we allow two subtrajectory clusters to overlap? Do we want the clusters of maximum length or the ones with a maximum number of subtrajectories?

Another major problem is the huge amount of clusters reported if no restrictions are made on the output. Especially if subtrajectories are allowed to overlap, or if one allow clusters that are very similar but have different representative subtrajectories. Therefore we made the following restrictions on the subtrajectory clusters.

1. Two subtrajectories in a single cluster may only over-



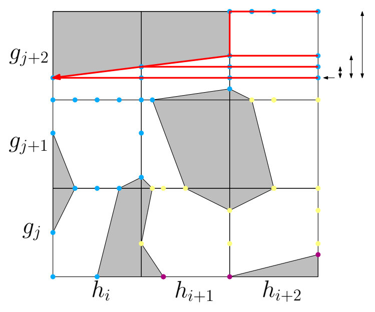


Figure 6: The visibility of points (left). Four potential clusters found (right).

lap along T in a single point. (Section 3.1)

2. A representative subtrajectory must start and end at a vertex of T. (Section 3.2)

3. A subtrajectory in a cluster must have length greater than zero. (Section 3.3)

4. Subtrajectories belonging to different clusters may only overlap along T in a single point. (Section 3.4)

5. The reported cluster should have maximal length in the following sense. No two clusters can be merged into a single cluster (Section 3.2). A representative subtrajectory cannot be extended in either direction without destroying the cluster or overlapping with another cluster (Section 3.4).

Consider a free space diagram $F_\varepsilon(T, T)$ of a polygonal curve T and itself. Recall that if a subtrajectory τ of length ℓ within T belongs to a subtrajectory cluster $SC(m, \ell, 2\varepsilon)$ then there exist m xy-monotone paths from the vertical line in the free space diagram intersecting the x-axis at the start point of τ, denoted l_s, and the vertical line intersecting the x-axis at the endpoint of τ, denoted l_t. Furthermore, according to restriction (2) the projections of any two xy-monotone paths onto the y-axis are not allowed to overlap in more than a single point (otherwise the subtrajectories in the cluster would overlap along T). We call τ the reference subtrajectory. See Fig. 3 for an example. Thus the aim is to find all maximal length intervals (restriction (5)) along the x-axis where there are m xy-monotone paths from the left vertical boundary to the right vertical boundary of the interval such that the projections onto the y-axis of any two xy-monotone paths overlap in at most one point.

3.1 Finding m xy-monotone paths in the free space diagram

Consider the case when we have a fixed representative subtrajectory, that is the left and right boundaries (l_s and l_t) are fixed. In [7] it was shown that a greedy approach can be used in the sequential setting to decide if there is m xy-monotone paths between the left and right boundary. That is, start at the topmost vertex on the right boundary which has an edge connected to a vertex whose label is l_s. In each vertex follow the topmost edge (decreasing along the y-coordinate) connected to a vertex whose label is again l_s. This ends on a vertex (l_s, j) on the left boundary. Continue with the topmost vertex on the right boundary at height at most j which, as before, has an edge connected to a vertex with label l_s. Stop when m curves have been found, or no more vertices on the right boundary with an edge connected to a vertex with label l_s exist. Note that this guarantees that restriction (1) and the first part of restriction (5) are fulfilled.

We propose a semi parallel approach of this process. We use a thread per point in $\mathcal{F}_{2t+2}$. Each thread, which point has label l_s, moves along adjacent cells of $\mathcal{F}$ until they reach a point in $\mathcal{F}_s$ with label l_s. This is done without any thread interference and at the same time. Because each thread has follow its independent path, the resulting paths may overlap. The path overlap removing process is difficult to parallelize because the removal of a path depends on the above paths which can be removed or not. This leads into several kernel executions to guarantee correct results. We tested this approach but it is slower than executing the sequential algorithm in the CPU, so we decided to merge our approach with the one presented in [7]. We found the xy-monotone paths in parallel using the GPU and then we remove the overlapped ones using sequential CPU algorithm.

3.2 Sweep the free space diagram

Above we showed how to find all xy-monotone paths in the free space diagram of two given trajectories h and g. We can use the same approach to report subtrajectory clusters within a trajectory T. This can be done by considering T as the trajectory h and also as the reference trajectory g, having T in the both dimensions of the free space diagram. We want to report all subtrajectory clusters while we consider all the possible reference subtrajectories within T. To this end we proceed as follows.

We define s and t as the start and end vertices of a possible representative subtrajectory $T(s, t)$ of T, respectively. Let l_s and l_t be the vertical lines in the free space diagram with x-coordinate s and t, respectively. The idea is to sweep l_s and l_t from left to right while maintaining the number of xy-monotone paths starting on l_s and ending on l_t while fulfilling the restrictions. The number of xy-monotone paths between l_s and l_t is denoted $\lambda(s, t)$ and is computed using the approach described in the previous section.

Initially set $s = 0$ and $t = 1$. If $\lambda(s, t) < m$ then increase both s and t by 1, until $\lambda(s, t) \geq m$. This implies that there is a subtrajectory cluster of size at least m and length $|T(s, t)|$ having $T(s, t)$ as its representative subtrajectory. Since we need to find the maximal length cluster (restriction (5)) we increase t until we reach the first value where $\lambda(s, t) < m$. Then report the subtrajectory cluster with $T(s, t - 1)$ as a representative subtrajectory.

The process starts all over again by setting $s = t - 1$.

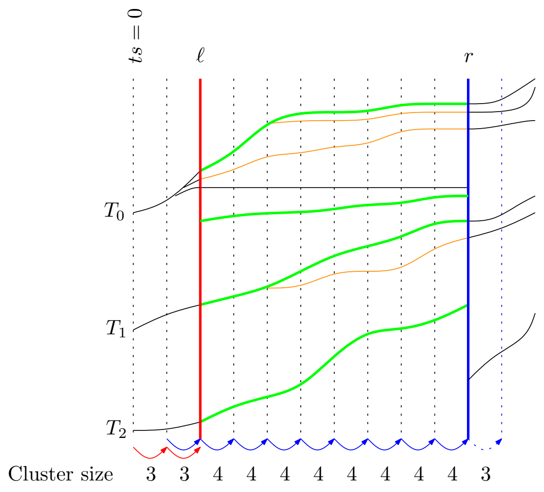


Figure 7: Left and right movement for a $m = 4$. The reported subtrajectories are drawn fat.

This is repeated until $t = n$. Since the event points of the sweepline is the integer coordinates this guarantees that restriction (2) is fulfilled. An example of the process is shown in Fig. 7.

It is important to note that when t moves one step to the right we only compute the xy-monotone paths from l_{t-1} to l_t, since all the necessary information is already stored in the vertices along l_{t-1}. That is, paths found are connected with the already stored paths on l_{t-1} obtaining xy-monotone paths from l_s to l_t.

3.2.1 *Super columns*

As mentioned in Section 2 the approach presented in previous sections assumes that the whole $\mathcal{F}(T,T)$ fits in the GPU memory. This is obviously unrealistic since the structure uses quadratic space and the trajectory can be very large, even after it has been simplified. To overcome this problem we compute $\mathcal{F}(T,T)$ in parts using what we call super columns, that is, a set of M consecutive columns that fit inside the GPU.

Initially compute and store $\mathcal{F}_0$ to $\mathcal{F}_{M-1}$ in the GPU. Super columns are stored as a CPU vector array $S[0..n]$, where each $S[i]$ contains a pointer to the position in the GPU memory storing $\mathcal{F}_i$. Next, search for subtrajectory clusters with s and t in this interval. When s is incremented the previous column is removed from the GPU memory.

The right boundary, t is incremented as described above until it reaches the end of the super column. The next super column is then computed and the corresponding pointers in the GPU memory is added to the CPU array S. Since all the necessary information is stored in the last column $\mathcal{F}_{M-1}$ (Section 3.2) we can free up the GPU memory storing all the earlier column of the super column (see also Section 3.4). This implies that even if a subtrajectory cluster is very long the algorithm will find it.

Note that this approach requires that the size of the GPU memory is $\Omega(n)$ since at least one column, $\mathcal{F}_i$ is required to fit in memory. We believe it is a reasonable assumption.

3.3 Paths of zero length

Another issue we have to deal with is when a monotone increasing path has length 0 (constraint (3)). Even though it fulfills the cluster condition and it is not overlapping any other subtrajectory, it makes no sense from a practical point of view. This usually happens when ε is very large with respect to ℓ, or when a representative trajectory has stayed within a small spatial region for a long time, which is not an uncommon situation in real data sets.

To handle this constraint we add an extra constraint to the xy-monotone paths between l_s and l_t, namely that no path is allowed to be a single horizontal segment. Note that any restriction on the length of the segments can easily be added.

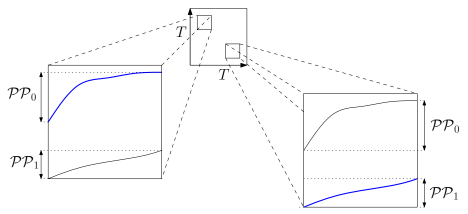


Figure 8: Two copies of the same two subtrajectories occurring at two places in the free space diagram.

3.4 Avoiding overlapping clusters

To avoid reporting very similar clusters the fourth constraint guarantees that a subtrajectory that is already included in a found cluster cannot be included (even in parts) in a different cluster. The problem occurs for two reasons; (1) we check all possible start and endpoints for the representative clusters during the sweep and (2) the free space diagram is symmetric we will find the same cluster with the reference subtrajectory swapped, see Fig. 8.

To handle this we create an array of size n initialized to 0. Each time a cluster is reported we set all the time steps of the subtrajectories involved in the cluster to 1. Then when we search for the xy-monotone paths we check if any of its time steps have been already been reported. If they have the corresponding subtrajectory is discarded.

This pruning step also has the additional effect that when a maximal length reference subtrajectory $T(s,t)$ for a cluster has been found we do not have to consider any part between s and t again. The result of this is that all the information needed during the sweep can be stored in column $t-1$, we do not have to keep any information in the columns between s and $t-1$, which gives a considerable speed-up.

4. EXPERIMENTAL RESULTS

The experimental results have been run using an i5-200 CPU with a Nvidia GTX 580. We studied the running times and the influence of the input parameters using three different data sets. The trajectories of two soccer players during one match (see the diagrams in Fig. 10 and Fig. 11) and a trajectory of a bird tracked for one month (Fig. 12). The soccer data sets have 28, 597 and 27, 894 time steps, respectively, and the bird trajectory has 4, 161 time steps.

4.1 Input parameters

Modifying the input parameters shows a relationship between the number of reported clusters and the length of the clusters. In the experiments we only measured the length of the longest cluster, however, in general the average cluster length increases when the length of the longest cluster increases. In Fig. 10 left column, we increase ε while m remains fixed. Naturally the length of the longest cluster increases. But the number of reported clusters also decreases, the main reason is that several clusters may merge into a single cluster while only a small number of new clusters is found.

In the right column of Fig. 10, we increase m while ε remains fixed. As m increases the constraint becomes more restrictive and the algorithm finds less clusters and shorter clusters. This is what we expected, however, a different behavior is seen in Fig. 11 when performing the same experiment. As mentioned above, both trajectories are generated by soccer players, however, the second trajectory was generated by a goalkeeper and, hence, the distribution of the points is very different. For the goal keeper the trajectory is enclosed in a very small region and when ε becomes large almost every point of the trajectory lies within distance ε from all other points. As a result the algorithm finds many clusters of length zero (Section 3.3) which are discarded and consequently, fewer clusters are reported. Note that, since T is normalized in the $[0, 1]^2$ space, a 0.15 ε-value represents 15% of the length of the soccer field, which is approximately the area where a goal keeper player spends most of the time. This does not happen in Fig. 10 since a "normal" player moves within a much larger area.

Finally, for the goal keeper trajectory we also tested the algorithm by fixing ε and varying m. Again, shorter clusters were reported while the number of clusters found increased. This is counter-intuitive and we believe it is a result of the large ε value (0.15), which is much too large to analyze the movement of a goal keeper.

4.2 Running times

The running times shown in Fig. 10-12 are divided into the three main parts of the algorithm; building the free space, label the graph and finding the clusters. The *total time* includes the time of computing the subtrajectory clusters and reporting them. As expected the most expensive process is the labeling (Section 2.3) which is the part of the algorithm that is not using the parallelism fully. Note that the running times depend on ε while ℓ's value has a minor effect.

Finally we compared the clustering method implemented in [7] that uses the discrete Fréchet distance for the CPU model with our method that uses the continuous Fréchet distance in the GPU model, see Fig. 9. Note that the theoretical bound on the running time of the CPU algorithm is $O(n^2\ell)$, which is also supported by the experimental results. The algorithm was able to handle data sets containing up to 39k points, anything larger caused it to crash.

Clearly the CPU algorithm grows much faster than the GPU algorithm presented in this paper. The GPU algorithm was tested with trajectories with up to 113k points without any problems.

5. CONCLUSIONS

In this paper we presented a parallel algorithm to compute subtrajectory clusters using the continuous Fréchet distance between curves as a measure of proximity. We show how the steps in the original algorithm by Buchin et al. [7] are mostly parallelizable, which results in a large improvement in running times. The running times strongly depend on the labeling since there is no natural way of parallelizin that

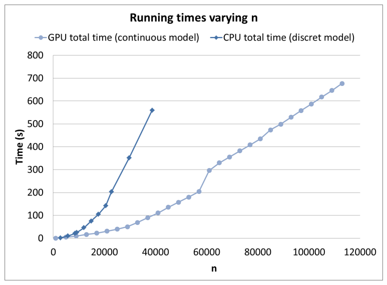


Figure 9: Clustering running times varying n. The results shown are, for the GPU, using the continuous Fréchet distance and, for the CPU, using the discrete Fréchet distance

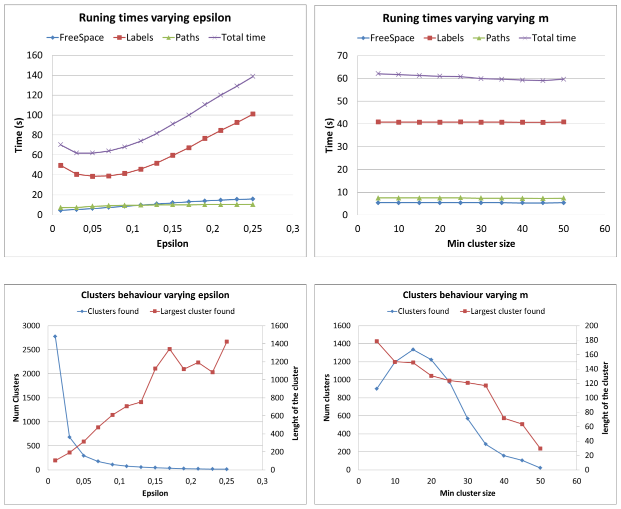


Figure 10: Soccer data set player. 28597 time steps.

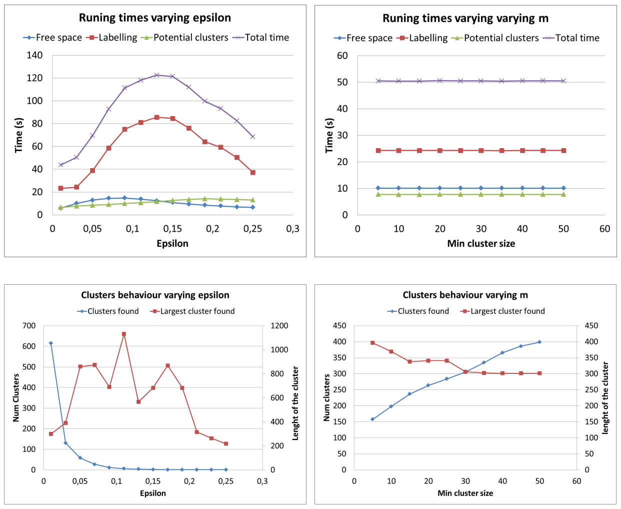


Figure 11: Soccer data set player. 27894 time steps.

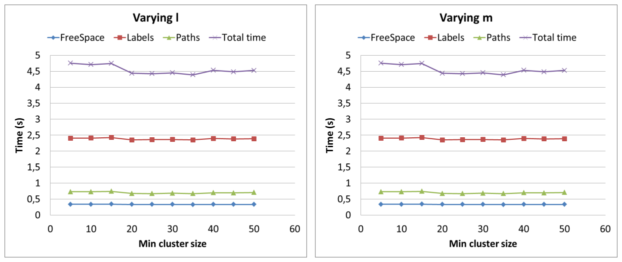


Figure 12: 1 **seagull data set.** 4161 **time steps.**

step.

Previous implementations of the subtrajectory clustering algorithm have all used the discrete Fréchet distance instead of the continuous. The main advantage with using the continuous is that it allows us to apply compressing techniques to the input data. Various studies have shown that a trajectory can be compressed to 5%-10% of its original size without loosing much information. Thus our implementation gives us the ability to handle much larger data sets than previously possible.

It is important to remark that there exists best practices and programming techniques to reach the maximum performance of the GPU [6]. In our paper, most of these guide lines and recommendations are very difficult to enforce due the complexity of the algorithm.

6. REFERENCES

[1] P.K. Agarwal, R.B. Avraham, H. Kaplan, and M. Sharir. Computing the discrete fréchet distance in subquadratic time. *CoRR*, abs/1204.5333, 2012.

[2] H. Alt. *In Efficient Algorithms: Essays Dedicated to Kurt Mehlhorn on the Occasion of His 60th Birthday*, chapter The computational geometry of comparing shapes, pages 235Ű–248. Springer-Verlag, 2009.

[3] H. Alt and M. Godau. Computing the fréchet distance between two polygonal curves. *Int. J. Comput. Geometry Appl.*, 5:75–91, 1995.

[4] H. Alt, C. Knauer, and C. Wenk. Comparison of distance measures for planar curves. *Algorithmica*, 38(1):45–58, 2003.

[5] M. Benkert, J. Gudmundsson, F. Hübner, and T. Wolle. Reporting flock patterns. *Computational Geometry*, 41(3):111–125, 2008.

[6] A.R. Brodtkorb, T.R. Hagen, and M.L. Sætra. Graphics processing unit (gpu) programming strategies and trends in gpu computing. *Journal of Parallel and Distributed Computing*, pages –, 2012.

[7] K. Buchin, M. Buchin, J. Gudmundsson, M. Löffler, and J. Luo. Detecting commuting patterns by clustering subtrajectories. *Int. J. Comput. Geometry Appl.*, 21(3):253–282, 2011.

[8] H. Cao, O. Wolfson, and G. Trajcevski. Spatio-temporal data reduction with deterministic error bounds. *The VLDB Journal*, 15(3):211–228, 2006.

[9] Nvidia Corporation. Nvidia CUDA 4.1 sdk samples. `http://developer.nvidia.com/gpu-computing-sdk`.

[10] A. Driemel, S. Har-Peled, and C. Wenk. Approximating the fréchet distance for realistic curves in near linear time. In *Symposium on Computational Geometry*, pages 365–374, 2010.

[11] A. Dumitrescu and G. Rote. On the Fréchet distance of a set of curves. In *CCCG*, pages 162–165, 2004.

[12] S. Gaffney, A. Robertson, P. Smyth, S. Camargo, and M. Ghil. Probabilistic clustering of extratropical cyclones using regression mixture models. *Climate Dynamic*, 29(4):423–440, 2007.

[13] S. Gaffney and P. Smyth. Trajectory clustering with mixtures of regression models. In *Proceedings of the Fifth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 63–72, 1999.

[14] J. Gudmundsson, J. Katajainen, D. Merrick, C. Ong, and T. Wolle. Compressing spatio-temporal trajectories. *Computational Geometry - Theory and Applications*, 42(9):825–841, 2009.

[15] J. Gudmundsson, P. Laube, and T. Wolle. Movement analysis. In W. Kresse and D. M. Danko, editors, *Handbook of Geographic Information*, chapter 22, pages 725–741. Springer, 2012.

[16] J. Gudmundsson and M.J. van Kreveld. Computing longest duration flocks in trajectory data. In Rolf A. de By and Silvia Nittel, editors, *GIS*, pages 35–42. ACM, 2006.

[17] J.G. Lee, J. Han, and K.-Y. Whang. Trajectory clustering: a partition-and-group framework. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 593–604, 2007.

[18] N. Mamoulis, H. Cao, G. Kollios, M. Hadjieleftheriou, Y. Tao, and D. Cheung. Mining, indexing, and querying historical spatiotemporal data. In *Proceedings of the 10th International ACM Conference On Knowledge Discovery and Data Mining*, pages 236–245. ACM, 2004.

[19] M. Nanni and D. Pedreschi. Time-focused clustering of trajectories of moving objects. *J. Intell. Inf. Syst.*, 27(3):267–289, 2006.

[20] CUDA Programing Guide 4.1. Technical report, Nvidia Corporation, 2011.

[21] CUDA C Programming Best Practices Guide 4.1. Technical report, Nvidia Corporation, 2011.

[22] M. Vlachos, D. Gunopoulos, and G. Kollios. Discovering similar multidimensional trajectories. In *Proceedings of the 18th International Conference on Data Engineering*, pages 673–682, 2002.